# Documentation PostgreSQL 8.0.5

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Documentation PostgreSQL 8.0.5

## Le Groupe de Développement Global de PostgreSQL





---

**Table des matières**

**Liste des tableaux**

**Liste des illustrations**



**Liste des exemples**

# Préface

Ce livre est la documentation officielle de PostgreSQL. Elle est écrite par les développeurs de PostgreSQL ainsi que par d'autres volontaires en parallèle du développement du logiciel PostgreSQL. Elle décrit toutes les fonctionnalités supportées officiellement par la version actuelle de PostgreSQL.

Pour rendre gérable la grande quantité d'informations sur PostgreSQL, ce livre a été organisé en plusieurs parties. Chacune a pour cible une classe différente d'utilisateurs ou dépend du niveau et de l'expérience de chaque utilisateur sur PostgreSQL :

- La Partie I est une introduction informelle pour les nouveaux utilisateurs.
- La Partie II documente l'environnement du langage de requêtes SQL, ceci incluant les types de données et les fonctions ainsi que le réglage des performances au niveau de l'utilisateur. Chaque utilisateur PostgreSQL devrait l'avoir lu.
- La Partie III décrit l'installation et l'administration du serveur. Tous ceux qui gèrent un serveur PostgreSQL, que ce soit pour une utilisation personnelle ou dans d'autres cas, devraient lire cette partie.
- La Partie IV décrit les interfaces de programmation pour les programmes clients PostgreSQL.
- La Partie V contient des informations sur les capacités d'extensions du serveur pour les utilisateurs avancés. Les thèmes concernent, par exemple, les types de données définis par l'utilisateur et les fonctions.
- La Partie VI contient des informations de référence sur les commandes SQL, sur les programmes clients et serveurs. Cette partie aide les autres parties en apportant une information structurée par commande ou programme.
- La Partie VII contient des informations triées, utiles pour les développeurs PostgreSQL.

---

## Définition de PostgreSQL

PostgreSQL est un système de gestion de bases de données relationnelles objet (ORDBMS) basé sur POSTGRES, Version 4.2, développé à l'université de Californie au département des sciences informatiques de Berkeley. POSTGRES a lancé de nombreux concepts rendus ensuite disponibles dans plusieurs systèmes de bases de données commerciales.

PostgreSQL est un descendant open−source du code original de Berkeley. Il supporte une grande partie du standard SQL:2003 tout en offrant de nombreuses fonctionnalités modernes :

- requêtes complexes ;
- clés étrangères ;
- déclencheurs (triggers) ;
- vues ;
- intégrité des transactions ;
- contrôle des accès simultanés (MVCC ou multiversion concurrency control).

De plus, PostgreSQL peut être étendu de plusieurs façons par l'utilisateur, par exemple en ajoutant de nouveaux

- types de données ;
- fonctions ;
- opérateurs ;

- fonctions d'agrégat ;
- méthodes d'indexation ;
- langages de procédure.

Et grâce à sa licence libérale, PostgreSQL peut être utilisé, modifié et distribué par tout le monde gratuitement quelque soit le but visé, qu'il soit privé, commercial ou académique.

# Bref historique de PostgreSQL

Le système de bases de données relationnel objet PostgreSQL est issu de POSTGRES, programme écrit à l'université de Californie à Berkeley. Avec plus d'une dizaine d'années de développement, PostgreSQL est le système de bases de données libre le plus avancé.

## Le projet POSTGRES de Berkeley

Le projet POSTGRES, mené par le professeur Michael Stonebraker, était sponsorisé par l'agence des projets avancés de la Défense (DARPA, acronyme de *Advanced Research Projects Agency*), le bureau des recherches de l'armée (ARO, acronyme de *Army Research Office*), le NSF (acronyme de National Science Foundation) ainsi que ESL, Inc. L'implémentation de POSTGRES a commencé en 1986. Les concepts initiaux du système ont été présentés dans *The design of POSTGRES* et la définition du modèle de données initial est apparu dans *The POSTGRES data model*. Le concept du système de règles à ce moment était décrit dans *The design of the POSTGRES rules system*. L'architecture du gestionnaire de stockage était détaillée dans *The design of the POSTGRES storage system*.

Depuis, POSTGRES a connu plusieurs versions majeures. La première << démo >> devint opérationnel en 1987 et fut présenté en 1988 à la conférence ACM−SIGMOD. La version 1, décrite dans *The implementation of POSTGRES*, fut livrée à quelques utilisateurs externes en juin 1989. En réponse à une critique du premier mécanisme de règles (*A commentary on the POSTGRES rules system*), celui−ci fut réécrit (*On Rules, Procedures, Caching and Views in Database Systems*) dans la version 2, présentée en juin 1990. La version 3 apparut en 1991. Elle ajoutait le support de plusieurs gestionnaires de stockage, un exécuteur de requêtes amélioré et un gestionnaire de règles réécrit. La plupart des versions suivantes jusqu'à Postgres95 (voir plus loin) portèrent sur la portabilité et la fiabilité.

POSTGRES fut utilisé pour réaliser différentes applications de recherche et de production. Par exemple : un système d'analyse de données financières, un programme de suivi des performances d'un moteur à réaction, une base de données de suivi d'astéroïdes, une base de données médicale et plusieurs systèmes d'informations géographiques. POSTGRES a aussi été utilisé comme outil de formation dans plusieurs universités. Enfin, Illustra Information Technologies (devenu Informix, maintenant détenu par IBM) a repris le code et l'a commercialisé. Fin 1992, POSTGRES devint le gestionnaire de données principal du projet de calcul scientifique Sequoia 2000 fin 1992.

La taille de la communauté d'utilisateurs doubla pratiquement durant l'année 1993. Il devint de plus en plus évident que la maintenance du code et le support nécessitaient de plus en plus de temps et d'énergie, qui auraient dûs être employés à des recherches sur les bases de données. Afin de réduire le poids du support, le projet POSTGRES de Berkeley se termina officiellement avec la version 4.2.

## Postgres95

En 1994, Andrew Yu et Jolly Chen ajoutèrent un interpréteur de langage SQL à POSTGRES. Sous un nouveau nom, Postgres95 fut publié par la suite sur le Web, afin de devenir un descendant libre (open−source) du code source initial de POSTGRES, version Berkeley.

Le code de Postgres95 était complètement compatible avec le C ANSI et 25% moins gros. De nombreux changements internes amélioraient les performances et la maintenabilité. Les versions 1.0.x de Postgres95 étaient 30 à 50% plus rapides que POSTGRES, version 4.2, pour le test Wisconsin Benchmark. Mis à part les corrections de bogues, les principales améliorations étaient :

- Le langage PostQUEL était remplacé par SQL (exécuté côté serveur). Les requêtes imbriquées ne furent pas supportées avant PostgreSQL (voir plus loin) mais elles pouvaient être imitées dans Postgres95 avec des fonctions SQL définies par l'utilisateur. Les agrégats furent reprogrammés, l'utilisation de la clause GROUP BY ajoutée.
- Un nouveau programme (psql) permettait d'exécuter des requêtes SQL interactives, en utilisant GNU Readline. Il a complètement remplacé le programme monitor.
- Une nouvelle bibliothèque cliente, `libpgtcl`, supportait les programmes écrits en Tcl. Un shell exemple, `pgtclsh`, fournissait de nouvelles commandes Tcl pour utiliser l'interface des applications Tcl avec le serveur Postgres95.
- L'interface pour les gros objets était revue. Les gros objets Inversion étaient le seul mécanisme pour stocker de tels objets, le système de fichiers Inversion étant supprimé.
- Le système de règles de niveau instance était supprimé. Les règles étaient toujours disponibles en tant que règles de réécriture.
- Un bref tutoriel présentant les possibilités de SQL ainsi que celles spécifiques à Postgres95 était distribué avec le code source.
- La version GNU de make était utilisée pour la construction à la place de la version BSD. Par ailleurs, Postgres95 pouvait être compilé avec GCC sans correctif (l'alignement des nombres était corrigé).

## PostgreSQL

En 1996, il devint évident que le nom << Postgres95 >> vieillissait mal. Nous avons choisi un nouveau nom, PostgreSQL, pour mettre en avant la relation entre le POSTGRES original et les capacités SQL des versions plus récentes. En même temps, nous avons positionné le numéro de version à 6.0 pour reprendre la numérotation originale du projet POSTGRES de Berkeley.

Durant le développement de Postgres95, un effort particulier avait été fourni pour identifier et comprendre les problèmes existants dans le code. Avec PostgreSQL, la priorité put être mise sur l'augmentation des caractéristiques et des possibilités, même si le travail a continué dans tous les domaines.

Les détails sur ce qui est arrivé dans PostgreSQL à partir de ce moment est disponible dans l'Annexe E.

# Conventions

Ce livre utilise les conventions typographiques suivantes pour marquer certaines portions du texte : les nouveaux termes, les phrases en langue étrangère et d'autres passages importants sont tous affichés en *italique*. Tout ce qui représente une entrée ou une sortie d'un programme est affiché avec une police à espacement fixe (`exemple`). À l'intérieur de tels passages, l'italique (*`exemple`*) indique des emplacements ; vous devez insérer une valeur particulière au lieu de cet emplacement. En d'autres occasions, des bouts de

code de programme sont affichés en gras (**`exemple`**) s'ils ont été ajoutés ou modifiés depuis l'exemple précédent.

Les conventions suivantes sont utilisées dans le synopsis d'une commande : les crochets ( `[` et `]` ) indiquent des parties optionnelles (dans le synopsis d'une commande Tcl, les points d'interrogation (`?`) sont utilisés à la place, comme c'est le cas habituellement en Tcl). Les accolades ( `{` et `}` ) et les lignes verticales ( `|` ) indiquent que vous devez faire un choix entre plusieurs options. Les points ( `. . .` ) signifient que l'élément précédant peut être répété.

Pour améliorer la clarté, certaines commandes SQL sont précédées par une invite << => >> alors que les commandes shell le sont avec << `$` >>. Malgré tout, dans le cadre général, les invites ne sont pas indiquées.

Un *administrateur* est généralement une personne en charge de l'installation et de la bonne marche du serveur. Un *utilisateur* peut être toute personne qui utilise ou veut utiliser n'importe quelle partie du système PostgreSQL. Ces termes ne doivent pas être pris trop à la lettre ; ce livre n'a pas d'avis fixe sur les procédures d'administration système.

## Pour plus d'informations

En dehors de la documentation, c'est−à−dire en dehors de ce livre, il existe d'autres ressources sur PostgreSQL :

FAQ
: La FAQ contient des réponses continuellement mises à jours pour les questions les plus fréquentes.

README
: Les fichiers `README` sont disponibles pour la plupart des paquets de contribution.

Site web
: Le site web de PostgreSQL apporte des détails sur la dernière version et sur d'autres informations pour rendre votre travail ou votre investissement personnel sur PostgreSQL plus productif.

Listes de discussion
: Les listes de discussion sont un bon endroit pour avoir une réponse à vos questions, pour partager vos expériences avec celles d'autres utilisateurs et pour contacter les développeurs. Consultez le site web de PostgreSQL pour plus de détails.

Vous−même !
: PostgreSQL est un projet open−source. En tant que tel, il dépend de la communauté d'utilisateurs pour son support. En débutant sur PostgreSQL, vous dépendrez des autres pour une aide, soit au travers de la documentation soit via les listes de discussion. Pensez à contribuer en apportant vos connaissances à votre tour. Lisez les listes de discussion et répondez aux questions. Si vous apprenez quelque chose qui n'est pas dans la documentation, écrivez−le et donnez−le en contribution. Si vous ajoutez des fonctionnalités au code, faites de même.

## Lignes de conduite pour les rapports de bogues

Lorsque vous trouvez un bogue dans PostgreSQL, nous voulons en entendre parler. Vos rapports de bogues jouent un rôle important pour rendre PostgreSQL plus fiable car même avec la plus grande attention, nous ne pouvons pas garantir que chaque partie de PostgreSQL fonctionnera sur toutes les plates−formes et dans toutes les circonstances.

Les suggestions suivantes ont pour but de vous former à la saisie d'un rapport de bogue qui pourra ensuite être géré de façon efficace. Il n'est pas requis de les suivre mais ce serait à l'avantage de tous.

Nous ne pouvons pas promettre de corriger tous les bogues immédiatement. Si le bogue est évident, critique ou affecte un grand nombre d'utilisateurs, il y a de grandes chances pour que quelqu'un s'en charge. Il peut aussi arriver que nous vous demandons d'utiliser une version plus récente pour voir si le bogue est toujours présent. Ou nous pourrions décider que le bogue ne peut être corrigé avant qu'une réécriture massive, que nous avions planifiée, ne soit réalisée. Ou peut−être est−il trop difficile et que des choses plus importantes nous attendent. Si vous avez besoin d'aide immédiatement, considérez l'obtention d'un contrat de support commercial.

## Identifier les bogues

Avant de rapporter un bogue, merci de lire et re−lire la documentation pour vérifier que vous pouvez réellement faire ce que vous essayez de faire. Si ce n'est pas clair, rapportez−le aussi ; c'est un bogue dans la documentation. S'il s'avère que le programme fait différemment de ce qu'indique la documentation, c'est un bogue. Ceci peut inclure les circonstances suivantes, sans s'y limiter :

- Un programme se terminant avec un signal fatal ou un message d'erreur du système d'exploitation qui indiquerait un problème avec le programme (un contre−exemple pourrait être le message << disk full >>, disque plein, car vous devez le régler vous−même).
- Un programme produit une mauvaise sortie pour une entrée donnée.
- Un programme refuse d'accepter une entrée valide (c'est−à−dire telle que définie dans la documentation).
- Un programme accepte une entrée invalide sans information ou message d'erreur. Mais gardez en tête que votre idée d'entrée invalide pourrait être notre idée d'une extension ou d'une compatibilité avec les pratiques traditionnelles.
- PostgreSQL échoue à la compilation, à la construction ou à l'installation suivant les instructions des plateformes supportées.

Ici, << programme >> fait référence à un exécutable, pas seulement au moteur du serveur.

Une lenteur ou une absorption des ressources n'est pas nécessairement un bogue. Lisez la documentation ou demandez sur une des listes de discussion pour de l'aide concernant l'optimisation de vos applications. Ne pas se conformer au standard SQL n'est pas nécessairement un bogue sauf si une telle conformité est indiquée explicitement.

Avant de continuer, vérifiez sur la liste des choses à faire ainsi que dans la FAQ pour voir si votre bogue n'est pas déjà connu. Si vous n'arrivez pas à décoder les informations sur la liste des choses à faire, écrivez un rapport. Le minimum que nous puissions faire est de rendre cette liste plus claire.

## Que rapporter ?

Le plus important point à se rappeler avec les rapports de bogues est de donner tous les faits et seulement les faits. Ne spéculez pas sur ce que vous pensez qui ne va pas, sur ce qu'<< il semble faire >> ou sur quelle partie le programme contient une erreur. Si vous n'êtes pas familier avec l'implémentation, vous vous tromperez probablement et vous ne nous aiderez pas. Et même si vous avez raison, des explications complètes sont un bon supplément mais elles ne doivent pas se substituer aux faits. Si nous pensons corriger le bogue, nous devons toujours le reproduire nous−même. Rapporter les faits stricts est relativement simple (vous pouvez probablement copier/coller à partir de l'écran) mais, trop souvent, des détails importants sont oubliés

parce que quelqu'un a pensé qu'ils n'avaient pas d'importance ou que le rapport serait compris.

Les éléments suivants devraient être fournis avec chaque rapport de bogue :

- La séquence exacte des étapes nécessaires pour reproduire le problème *à partir du lancement du programme*. Ceci devrait se suffire ; il n'est pas suffisant d'envoyer une simple instruction `SELECT` sans les commandes `CREATE TABLE` et `INSERT` qui ont précédé, si la sortie dépend des données contenues dans les tables. Nous n'avons pas le temps de comprendre le schéma de votre base de données. Si nous sommes supposés créer nos propres données, nous allons probablement ne pas voir le problème.

  Le meilleur format pour un test suite à un problème relatif à SQL est un fichier qui peut être lancé via l'interface psql et qui montrera le problème. Assurez−vous de ne rien avoir dans votre fichier de lancement `~/.psqlrc`. Pour commencer facilement la création de ce fichier, utilisez pg_dump en récupérant les déclarations des tables ainsi que les données nécessaires pour mettre en place la scène. Il ne reste plus qu'à ajouter la requête posant problème. Vous êtes encouragé à minimiser la taille de votre exemple mais ce n'est pas une obligation. Si le bogue est reproductible, nous le trouverons de toute façon.

  Si votre application utilise une autre interface client, telle que PHP, alors essayez d'isoler le problème aux requêtes erronées. Nous n'allons certainement pas mettre en place un serveur web pour reproduire votre problème. Dans tous les cas, rappelez−vous d'apporter les fichiers d'entrée exacts ; n'essayez pas de deviner que le problème se pose pour les << gros fichiers >> ou pour les << bases de données de moyenne taille >>, etc. car cette information est trop inexacte et subjective pour être utile.
- La sortie que vous obtenez. Merci de ne pas dire que cela << ne fonctionne pas >> ou que cela s'est << arrêté brutalement >>. S'il existe un message d'erreur, montrez−le même si vous ne le comprenez pas. Si le programme se termine avec une erreur du système d'exploitation, dites−le. Même si le résultat de votre test est un arrêt brutal du programme ou un autre soucis évident, il peut ne pas survenir sur notre plateforme. Le plus simple est de copier directement la sortie du terminal, si possible.

  > **Note :** Si vous rapportez un message d'erreur, merci d'obtenir la forme la plus verbeuse de ce message. Avec psql, exécutez `\set VERBOSITY verbose` avant tout. Si vous récupérez le message des traces du serveur, initialisez la variable d'exécution log_error_verbosity avec `verbose` pour que tous les détails soient tracés.
  >
  > **Note :** Dans le cas d'erreurs fatales, le message d'erreur rapporté par le client pourrait ne pas contenir toutes les informations disponibles. Jetez aussi un œil aux traces du serveur de la base de données. Si vous ne conservez pas les traces de votre serveur, c'est le bon moment pour commencer à le faire.
- La sortie que vous attendez est une information très importante à donner. Si vous écrivez uniquement << Cette commande m'a donné cette réponse. >> ou << Ce n'est pas ce que j'attendais. >>, nous pourrions le lancer nous−même, analyser la sortie et penser que tout est correct car cela correspond exactement à ce que nous attendions. Nous ne devrions pas avoir à passer du temps pour décoder la sémantique exacte de vos commandes. Tout spécialement, ne vous contentez pas de dire que << Ce n'est pas ce que SQL spécifie/Oracle fait. >> Rechercher le comportement correct à partir de SQL n'est pas amusant et nous ne connaissons pas le comportement de tous les autres serveurs de base de données relationnels (si votre problème est un arrêt brutal du serveur, vous pouvez évidemment omettre cet élément).

- Toutes les options en ligne de commande ainsi que les autres options de lancement incluant les variables d'environnement ou les fichiers de configuration que vous avez modifié. Encore une fois, soyez exact. Si vous utilisez une distribution pré−packagée qui lance le serveur au démarrage, vous devriez essayer de retrouver ce que cette distribution fait.
- Tout ce que vous avez fait de différent à partir des instructions d'installation.
- La version de PostgreSQL. Vous pouvez lancer la commande `SELECT version();` pour trouver la version du serveur sur lequel vous êtes connecté. La plupart des exécutables dispose aussi d'une option `--version` ; `postmaster --version` et `psql --version` devraient au moins fonctionner. Si la fonction ou les options n'existent pas, alors votre version est bien trop ancienne et vous devez mettre à jour. Si vous avez lancé une version préparée sous forme de paquets, tel que les RPM, dites−le en incluant la sous−version que le paquet pourrait avoir. Si vous êtes sur une version CVS, mentionnez−le en indiquant la date et l'heure.

  Si votre version est antérieure à la 8.0.5, nous allons certainement vous demander de mettre à jour. Beaucoup de corrections de bogues et d'améliorations sont apportées dans chaque nouvelle version, donc il est bien possible qu'un bogue rencontré dans une ancienne version de PostgreSQL soit déjà corrigée. Nous ne fournissons qu'un support limité pour les sites utilisant d'anciennes versions de PostgreSQL ; si vous avez besoin de plus de support que ce que nous fournissons, considérez l'acquisition d'un contrat de support commercial.
- Informations sur la plateforme. Ceci inclut le nom du noyau et sa version, la bibliothèque C, le processeur, la mémoire et ainsi de suite. Dans la plupart des cas, il est suffisant de rapporter le vendeur et la version mais ne supposez pas que tout le monde sait ce que << Debian >> contient ou que tout le monde utilise des Pentiums. Si vous avez des problèmes à l'installation, des informations sur l'ensemble des outils de votre machine (compilateurs, `make`, etc.) sont aussi nécessaires.

N'ayez pas peur si votre rapport de bogue devient assez long. C'est un fait. Il est mieux de tout rapporter la première fois plutôt que nous ayons à vous demander tous les faits. D'un autre côté, si vos fichiers d'entrée sont trop gros, il est préférable de demander si quelqu'un souhaite s'y plonger.

Ne passez pas tout votre temps à vous demander quelles modifications apportées pour que le problème s'en aille. Ceci ne nous aidera probablement pas à le résoudre. S'il arrive que le bogue ne peut pas être corrigé immédiatement, vous aurez toujours l'opportunité de chercher ceci et de partager vos trouvailles. De même, encore une fois, ne perdez pas votre temps à deviner pourquoi le bogue existe. Nous le trouverons assez rapidement.

Lors de la rédaction d'un rapport de bogue, merci de choisir une terminologie qui ne laisse pas place aux confusions. Le paquet logiciel en totalité est appelé << PostgreSQL >>, quelque fois << Postgres >> en court. Si vous parlez spécifiquement du serveur, mentionnez−le mais ne dites pas seulement << PostgreSQL a planté >>. Un arrêt brutal d'un seul processus serveur est assez différent de l'arrêt brutal du << postmaster >> père ; merci de ne pas dire que << le postmaster a planté >> lorsque vous voulez dire qu'un seul processus s'est arrêté, et non pas vice versa. De plus, les programmes clients tels que l'interface interactive << psql >> sont complètement séparés du moteur. Essayez d'être précis sur la provenance du problème : client ou serveur.

## Où rapporter des bogues ?

En général, envoyez vos rapports de bogue à la liste de discussion des rapports de bogue (<pgsql-bogues@postgresql.org>). Nous vous demandons d'utiliser un sujet descriptif pour votre courrier électronique, par exemple une partie du message d'erreur.

Une autre méthode consiste à remplir le formulaire web disponible sur le site web du projet http://www.postgresql.org/. Saisir un rapport de bogue de cette façon fait que celui–ci est envoyé à la liste de discussion <pgsql-bogues@postgresql.org>.

Si votre rapport de bogue a des implications sur la sécurité et que vous préfériez qu'il ne soit pas immédiatement visible dans les archives publiques, ne l'envoyez pas sur `pgsql-bugs`. Les problèmes de sécurité peuvent être rapportés de façon privé sur <security@postgresql.org>.

N'envoyez pas de rapports de bogue aux listes de discussion des utilisateurs, comme <pgsql-sql@postgresql.org> ou <pgsql-general@postgresql.org>. Ces listes de discussion servent à répondre aux questions des utilisateurs et les abonnés ne souhaitent pas recevoir de rapports de bogues. Plus important, ils ont peu de chance de les corriger.

De même, n'envoyez *pas* vos rapports de bogue à la liste de discussion des développeurs <pgsql-hackers@postgresql.org>. Cette liste sert aux discussions concernant le développement de PostgreSQL et il serait bon de conserver les rapports de bogue séparément. Nous pourrions choisir de discuter de votre rapport de bogue sur `pgsql-hackers` si le problème nécessite que plus de personnes s'en occupent.

Si vous avez un problème avec la documentation, le meilleur endroit pour le rapporter est la liste de discussion pour la documentation <pgsql-docs@postgresql.org>. Soyez précis sur la partie de la documentation qui vous déplaît.

Si votre bogue concerne un problème de portabilité sur une plateforme non supportée, envoyez un courrier électronique à <pgsql-ports@postgresql.org>, pour que nous puissions travailler sur le portage de PostgreSQL sur votre plateforme.

> **Note :** Dû, malheureusement, au grand nombre de pourriels (spam), toutes les adresses de courrier électronique ci–dessus appartiennent à des listes de discussion fermées. C'est–à–dire que vous devez être abonné pour être autorisé à y envoyer un courrier. Néanmoins, vous n'avez pas besoin de vous abonner pour utiliser le formulaire web de rapports de bogue. Si vous souhaitez envoyer des courriers mais ne pas recevoir le trafic de la liste, vous pouvez vous abonner et configurer l'option `nomail`. Pour plus d'information, envoyez un courrier à <majordomo@postgresql.org> avec le seul mot `help` dans le corps du message.

# I. Tutoriel

Bienvenue dans le tutoriel de PostgreSQL. Les quelques chapitres suivants sont destinés à présenter une courte introduction à PostgreSQL, aux concepts des bases de données relationnelles et au langage SQL à ceux qui sont nouveaux dans l'un de ces domaines. Nous supposons seulement des connaissances générales sur l'utilisation des ordinateurs. Aucune expérience particulière d'Unix ou de programmation n'est requise. Cette partie est principalement destinée à vous donner une expérience pratique avec les aspects importants du système PostgreSQL. Ce tutoriel ne cherche pas à être un tour complet ou approfondi des sujets qu'il couvre.

Après avoir parcouru ce tutoriel, vous voudrez peut−être continuer en lisant la Partie II pour acquérir une connaissance plus conventionnelle du langage SQL ou la Partie IV pour des informations sur le développement d'applications pour PostgreSQL. Ceux qui veulent configurer et gérer leur propre serveur devraient aussi lire la Partie III.

**Table des matières**

# Chapitre 1. Démarrage

## 1.1. Installation

Avant de pouvoir utiliser PostgreSQL vous devez l'installer, bien sûr. Il est possible que PostgreSQL soit déjà installé dans votre environnement, soit parce qu'il est inclus dans votre distribution, soit parce que votre administrateur système l'a déjà installé. Si cela est le cas, vous devriez obtenir, dans la documentation de votre distribution ou de la part de votre administrateur, les informations nécessaires pour accéder à PostgreSQL.

Si vous n'êtes pas sûr que PostgreSQL soit déjà disponible ou que vous puissiez l'utiliser pour vos tests, vous avez la possibilité de l'installer vous−même. Le faire n'est pas difficile et peut être un bon exercice. PostgreSQL peut s'installer par n'importe quel utilisateur sans privilège ; aucun accès super−utilisateur (root) n'est requis.

Si vous installez PostgreSQL vous−même, référez−vous au Chapitre 14 pour les instructions sur l'installation, puis revenez à ce guide quand l'installation est terminée. Nous vous conseillons de suivre attentivement la section sur la configuration des variables d'environnement appropriées.

Si votre administrateur n'a pas fait une installation par défaut, vous pouvez avoir du travail supplémentaire à faire. Par exemple, si le serveur de bases de données est une machine distante, vous aurez besoin de configurer la variable d'environnement `PGHOST` avec le nom du serveur de bases de données. La variable d'environnement `PGPORT` pourra aussi être configurée. La démarche est la suivante : si vous essayez de démarrer un programme et qu'il se plaint de ne pas pouvoir se connecter à la base de données, vous devez consulter votre administrateur ou, si c'est vous, la documentation pour être sûr que votre environnement est correctement paramétré. Si vous n'avez pas compris le paragraphe précédent, lisez donc la prochaine section.

---

## 1.2. Concepts architecturaux de base

Avant de continuer, vous devriez connaître les bases de l'architecture système de PostgreSQL. Comprendre comment les parties de PostgreSQL interagissent entre elles rendra ce chapitre légèrement plus clair.

Dans le jargon des bases de données, PostgreSQL utilise un modèle client/serveur. Une session PostgreSQL est le résultat de la coopération des processus (programmes) suivants :

- Un processus serveur, qui gère les fichiers de la base de données, accepte les connexions à la base de la part des applications clientes et effectue sur la base les actions des clients. Le programme serveur est appelé `postmaster`.
- L'application cliente des utilisateurs qui veulent effectuer des opérations sur la base de données. Les applications clientes peuvent être de nature très différentes : un client peut être un outil texte, une application graphique, un serveur web qui accède à la base de données pour afficher des pages web ou un outil spécialisé dans la maintenance de bases de données. Certaines applications clientes sont fournies avec PostgreSQL ; la plupart sont développées par les utilisateurs.

Comme souvent avec les applications client/serveur, le client et le serveur peuvent être sur des hôtes différents. Dans ce cas, ils communiquent à travers une connexion réseau TCP/IP. Vous devriez garder cela à l'esprit car les fichiers qui sont accessibles sur la machine cliente peuvent ne pas l'être (ou l'être seulement en utilisant des noms de fichiers différents) sur la machine serveur de bases de données.

Le serveur PostgreSQL peut traiter de multiples connexions simultanées depuis les clients. Dans ce but, il démarre (<< fork >>) un nouveau processus pour chaque connexion. À ce moment, le client et le nouveau processus serveur communiquent sans intervention de la part du processus `postmaster` original. Ainsi, le `postmaster` tourne toujours, attendant de nouvelles connexions clientes, tandis que le client et les processus serveurs associés vont et viennent. (Tout ceci est bien sûr invisible pour l'utilisateur. Nous le mentionnons ici seulement par exhaustivité.)

# 1.3. Création d'une base de données

Le premier test pour voir si vous pouvez accéder au serveur de bases de données consiste à essayer de créer une base. Un serveur PostgreSQL peut gérer plusieurs bases de données. Généralement, une base de données distincte est utilisée pour chaque projet ou pour chaque utilisateur.

Il est possible que votre administrateur ait déjà créé une base pour votre utilisation. Il devrait vous avoir dit quel est le nom de celle−ci. Dans ce cas, vous pouvez omettre cette étape et aller directement à la prochaine section.

Pour créer une nouvelle base, dans cet exemple nommée `ma_base`, utilisez la commande suivante :

```
$ createdb ma_base
```

Cela devrait renvoyer la réponse :

```
CREATE DATABASE
```

Si oui, cette étape est réussie et vous pouvez sauter le reste de cette section.

Si vous voyez un message similaire à :

```
createdb: command not found
```

alors PostgreSQL n'a pas été installé correctement. Soit il n'a pas été installé du tout, soit le chemin n'a pas été configuré correctement. Essayez d'appeler la commande avec le chemin absolu :

```
$ /usr/local/pgsql/bin/createdb ma_base
```

Le chemin sur votre site peut être différent. Contactez votre administrateur ou vérifiez en arrière dans les instructions d'installation pour corriger la situation.

Une autre réponse peut être :

```
createdb: could not connect to database template1: could not connect to server:
No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

Cela signifie que le serveur n'était pas démarré, ou qu'il n'était pas démarré où `createdb` l'attendait. Une fois encore, vérifiez les instructions d'installation ou consultez votre administrateur.

Une autre réponse pourrait être ceci :

```
createdb: could not connect to database template1: FATAL:  user "joe" does not
exist
```

où votre nom de connexion est mentionné. Ceci survient si l'administrateur n'a pas créé de compte utilisateur PostgreSQL pour vous. (Les comptes utilisateurs PostgreSQL sont distincts de ceux du système d'exploitation.) Si vous êtes l'administrateur, la lecture du Chapitre 17 vous expliquera comment créer de tels comptes. Vous aurez besoin de devenir l'utilisateur du système d'exploitation sous lequel PostgreSQL a été installé (généralement `postgres`) pour créer le compte du premier utilisateur. Cela pourrait aussi signifier que vous vous êtes affecté un nom d'utilisateur PostgreSQL qui est différent de celui de votre compte utilisateur du système d'exploitation ; dans ce cas, vous avez besoin d'utiliser l'option `-U` ou de configurer la variable d'environnement `PGUSER` pour spécifier votre nom d'utilisateur PostgreSQL.

Si vous n'avez pas les droits requis pour créer une base, vous verrez le message suivant :

```
createdb: database creation failed: ERROR:  permission denied to create database
```

Tous les utilisateurs n'ont pas l'autorisation de créer de nouvelles bases de données. Si PostgreSQL refuse de créer des bases pour vous, alors l'administrateur a besoin de vous accorder le droit pour cela. Consultez votre administrateur si cela arrive. Si vous avez installé vous−même PostgreSQL, alors vous devriez ouvrir une session sous le compte utilisateur que vous avez utilisé pour démarrer le serveur. [1]

Vous pouvez aussi créer des bases de données avec d'autres noms. PostgreSQL vous permet de créer un nombre quelconque de bases sur un site donné. Le nom des bases doit avoir comme premier caractère un caractère alphabétique et est limité à 63 caractères de longueur. Un choix pratique est de créer une base avec le même nom que votre nom d'utilisateur courant. Beaucoup d'outils utilisent ce nom comme nom par défaut pour la base ; cela permet de gagner du temps en saisie. Pour créer cette base, tapez simplement :

```
$ createdb
```

Si vous ne voulez plus utiliser votre base, vous pouvez la supprimer. Par exemple, si vous êtes le propriétaire (créateur) de la base `ma_base`, vous pouvez la détruire en utilisant la commande suivante :

```
$ dropdb ma_base
```

(Pour cette commande, le nom de la base n'est pas par défaut le nom du compte utilisateur. Vous devez toujours en spécifier un.) Cette action supprime physiquement tous les fichiers associés avec la base de données et elle ne peut pas être annulée, donc cela doit être fait avec beaucoup de prévoyance.

Plus d'informations sur `createdb` et `dropdb` peuvent être trouvées respectivement dans createdb et dropdb.

## 1.4. Accéder à une base

Une fois que vous avez créé la base, vous pouvez y accéder :

- Démarrez le programme en ligne de commande de PostgreSQL, appelé *psql*, qui vous permet d'entrer, d'éditer et d'exécuter de manière interactive les commandes SQL.
- Utilisez un outil existant avec une interface graphique comme PgAccess ou une suite bureautique avec un support ODBC pour créer et manipuler une base. Ces possibilités ne sont pas couvertes dans ce tutoriel.

- Écrivez une application personnalisée en utilisant un des nombreux langages disponibles. Ces possibilités sont davantage examinées dans la Partie IV.

Vous voulez probablement lancer `psql` pour essayer les exemples de ce tutoriel. Pour cela, saisissez la commande suivante :

```
$ psql ma_base
```

Si vous n'indiquez pas le nom de la base alors cela utilisera par défaut le nom de votre compte utilisateur. Vous avez déjà découvert ce schéma dans la section précédente.

Dans `psql`, vous serez salué avec le message suivant :

```
Welcome to psql 8.0.5, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

ma_base=>
```

La dernière ligne peut aussi être :

```
ma_base=#
```

Cela voudra dire que vous êtes le super−utilisateur de la base de données ce qui est souvent le cas si vous avez installé PostgreSQL vous−même. Être super−utilisateur signifie que vous n'êtes pas sujet aux contrôles d'accès. Concernant ce tutoriel, cela n'a pas d'importance.

Si vous rencontrez des problèmes en démarrant `psql`, alors retournez à la section précédente. Le diagnostique de `psql` et de `createdb` sont semblables, et si le dernier fonctionnait, alors le premier devrait fonctionner également.

La dernière ligne affichée par `psql` est l'invite. Cela indique que `psql` est à l'écoute et que vous pouvez saisir des requêtes SQL dans l'espace de travail maintenu par `psql`. Essayez ces commandes :

```
ma_base=> SELECT version();
                          version
------------------------------------------------------------
 PostgreSQL 8.0.5 on i586-pc-linux-gnu, compiled by GCC 2.96
(1 row)

ma_base=> SELECT current_date;
    date
------------
 2002-08-31
(1 row)

ma_base=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

Le programme psql a un certain nombre de commandes internes qui ne sont pas des commandes SQL. Elles commencent avec le caractère barre oblique inverse, << \ >>. Certaines de ces commandes sont listées dans le message de bienvenue. Par exemple, vous pouvez obtenir de l'aide sur la syntaxe de nombreuses commandes SQL de PostgreSQL en tapant :

```
ma_base=> \h
```

Pour sortir de psql, saisissez

```
ma_base=> \q
```

et psql se terminera et vous ramènera à votre shell. (Pour plus de commandes internes, tapez \? à l'invite de psql.) Les possibilités complètes de psql sont documentées dans psql. Si PostgreSQL est installé correctement vous pouvez aussi tapez man psql à l'invite du shell système pour voir la documentation. Dans ce tutoriel nous ne verrons pas ces caractéristiques explicitement, mais vous pouvez les utiliser vous−même quand bon vous semblera.

---

# Chapitre 2. Le langage SQL

## 2.1. Introduction

Ce chapitre fournit un panorama sur la façon d'utiliser SQL pour exécuter des opérations simples. Ce tutoriel est seulement prévu pour vous donner une introduction et n'est, en aucun cas, un tutoriel complet sur SQL. De nombreux livres ont été écrits sur SQL, incluant *Understanding the New SQL* et *A Guide to the SQL Standard*. Vous devez être averti que certaines caractéristiques du langage de PostgreSQL sont des extensions de la norme.

Dans les exemples qui suivent, nous supposons que vous avez créé une base de données appelée `ma_base`, comme cela a été décrit dans le chapitre précédent et que vous avez lancé psql.

Les exemples dans ce manuel peuvent aussi être trouvés dans le répertoire `src/tutorial/` de la distribution source de PostgreSQL. Pour utiliser ces fichiers, commencez par changer de répertoire et lancez make :

```
$ cd ..../src/tutorial
$ make
```

Ceci crée les scripts et compile les fichiers C contenant des fonctions et types définis par l'utilisateur. (Vous devez utiliser GNU make pour ceci — il pourrait être nommé différemment sur votre système, souvent gmake.) Puis, pour lancer le tutoriel, faites ce qui suit :

```
$ cd
..../src/tutorial
$ psql -s ma_base
...

ma_base=> \i basics.sql
```

La commande `\i` lit les commandes depuis le fichier spécifié. L'option `-s` vous place dans un mode pas à pas qui fait une pause avant d'envoyer chaque instruction au serveur. Les commandes utilisés dans cette section sont dans le fichier `basics.sql`.

---

## 2.2. Concepts

PostgreSQL est un *système de gestion de bases de données relationnelles* (SGBDR). Cela signifie que c'est un système pour gérer les données stockées dans des *relations*. Relation est essentiellement un terme mathématique pour *table*. La notion de stockage de données dans des tables est si commune aujourd'hui que cela peut sembler en soi évident mais il y a de nombreuses autres manières d'organiser des bases de données. Les fichiers et répertoires dans les systèmes d'exploitation de type Unix forment un exemple de base de données hiérarchique. Un développement plus moderne est une base de données orientée objets.

Chaque table est un ensemble de *lignes*. Chaque ligne d'une table donnée a le même ensemble de *colonnes* et chaque colonne est d'un type de données particulier. Tandis que les colonnes ont un ordre fixé dans chaque ligne, il est important de se rappeler que SQL ne garantit, d'aucune façon, l'ordre des lignes à l'intérieur de la table (bien qu'elles puissent être explicitement triées pour l'affichage).

Les tables sont groupées dans des bases de données et un ensemble de bases gérées par une instance unique du serveur PostgreSQL constitue un *groupe* de bases.

## 2.3. Créer une nouvelle table

Vous pouvez créer une nouvelle table en spécifiant le nom de la table, suivi de tous les noms de colonnes et de leur type :

```
CREATE TABLE temps (
    ville           varchar(80),
    temp_basse      int,           -- température basse
    temp_haute      int,           -- température haute
    prcp            real,          -- précipitation
    date            date
);
```

Vous pouvez entrer cela dans `psql` avec les sauts de lignes. `psql` reconnaîtra que la commande n'est pas terminée jusqu'à ce qu'il arrive au point−virgule.

Les espaces blancs (c'est−à−dire les espaces, les tabulations et les nouvelles lignes) peuvent librement être utilisés dans les commandes SQL. Cela signifie que vous pouvez saisir la commande ci−dessus alignée différemment ou même tout sur une seule ligne. Deux tirets (<< −− >>) introduisent des commentaires. Ce qui les suit est ignoré jusqu'à la fin de la ligne. SQL est insensible à la casse pour les mots−clé et les identifiants excepté quand les identifiants sont entre double guillemets pour préserver leur casse (non fait ci−dessus).

`varchar(80)` spécifie un type de données pouvant contenir une chaîne de caractères arbitraires de 80 caractères au maximum. `int` est le type entier normal. `real` est un type pour les nombres décimaux en simple précision. `date` devrait s'expliquer de lui−même. (Oui, la colonne de type `date` est aussi nommée `date`. Cela peut être commode ou porter à confusion, à vous de choisir.)

PostgreSQL prend en charge les types SQL standards `int`, `smallint`, `real`, `double precision`, `char(`*N*`)`, `varchar(`*N*`)`, `date`, `time`, `timestamp` et `interval` ainsi que d'autres types d'utilité générale et un riche ensemble de types géométriques. PostgreSQL peut être personnalisé avec un nombre arbitraire de types de données définis par l'utilisateur. En conséquence, les noms des types ne sont pas des mots−clé syntaxiques sauf lorsqu'il est requis de supporter des cas particuliers dans la norme SQL.

Le second exemple stockera des villes et leur emplacement géographique associé :

```
CREATE TABLE villes (
    nom             varchar(80),
    emplacement     point
);
```

Le type `point` est un exemple d'un type de données spécifique à PostgreSQL.

Pour finir, il devrait être mentionné que si vous n'avez plus besoin d'une table ou que vous voulez la recréer différemment, vous pouvez l'enlever en utilisant la commande suivante :

```
DROP TABLE nom_table;
```

## 2.4. Remplir une table avec des lignes

L'instruction `INSERT` est utilisée pour remplir une table avec des lignes :

```
INSERT INTO temps VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Notez que tous les types utilisent des formats d'entrées plutôt évident. Les constantes qui ne sont pas des valeurs numériques simples doivent être habituellement entourées par des guillemets simples (`'`) comme dans l'exemple. Le type `date` est en réalité tout à fait flexible dans ce qu'il accepte mais, pour ce tutoriel, nous collerons au format non ambigu montré ici.

Le type `point` demande une paire de coordonnées en entrée comme cela est montré ici :

```
INSERT INTO villes VALUES ('San Francisco', '(-194.0, 53.0)');
```

La syntaxe utilisée jusqu'à maintenant nécessite de se rappeler l'ordre des colonnes. Une syntaxe alternative vous autorise à lister les colonnes explicitement :

```
INSERT INTO temps (ville, temp_basse, temp_haute, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

Vous pouvez lister les colonnes dans un ordre différent si vous le souhaitez ou même omettre certaines colonnes ; par exemple, si la précipitation est inconnue :

```
INSERT INTO temps (date, ville, temp_haute, temp_basse)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

De nombreux développeurs considèrent que le listage explicite des colonnes est un meilleur style que de compter sur l'ordre implicite.

S'il vous plaît, entrez toutes les commandes vues ci–dessus de façon à avoir des données sur lesquelles travailler dans les prochaines sections.

Vous auriez pû aussi utiliser `COPY` pour charger de larges quantités de données depuis des fichiers textes. Cela est habituellement plus rapide car la commande `COPY` est optimisée pour cet emploi mais elle est moins flexible que `INSERT`. Exemple :

```
COPY temps FROM '/home/user/temps.txt';
```

où le nom pour le fichier source doit être disponible sur la machine serveur et non pas sur le client puisque le serveur lit le fichier directement. Vous pouvez en lire plus sur la commande `COPY` dans COPY.

## 2.5. Interroger une table

Pour retrouver les données d'une table, elle est *interrogée*. Une instruction SQL `SELECT` est utilisée pour faire cela. L'instruction est divisée en liste de sélection (la partie qui liste les colonnes à retourner), une liste de tables (la partie qui liste les tables à partir desquelles les données seront retrouvées) et une qualification optionnelle (la partie qui spécifie les restrictions). Par exemple, pour retrouver toutes les lignes de la table `temps`, tapez :

```
SELECT * FROM temps;
```

Ici, * est un raccourci pour << toutes les colonnes >>. [2] Donc, le même résultat pourrait être :

```
SELECT ville, temp_basse, temp_haute, prcp, date FROM temps;
```

le résultat devrait être :

```
     ville     | temp_basse | temp_haute | prcp |    date
---------------+------------+------------+------+------------
 San Francisco |         46 |      50    | 0.25 | 1994-11-27
 San Francisco |         43 |      57    |    0 | 1994-11-29
 Hayward       |         37 |      54    |      | 1994-11-29
(3 rows)
```

Vous pouvez écrire des expressions, pas seulement des références à de simples colonnes, dans la liste de sélection. Par exemple, vous pouvez faire :

```
SELECT ville, (temp_haute+temp_basse)/2 AS temp_moy, date FROM temps;
```

Cela devrait donné :

```
     ville     | temp_moy |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

Notez comment la clause `AS` est utilisée pour renommer la sortie d'une colonne. (La clause `AS` est optionnelle.)

Une requête peut être << qualifiée >> en ajoutant une clause `WHERE` qui spécifie quelles lignes sont souhaitées. La clause `WHERE` contient une expression booléenne (vraie) et seules les lignes pour lesquelles l'expression booléenne est vraie sont renvoyées. Les opérateurs booléens habituels (`AND`, `OR` et `NOT`) sont autorisés dans la qualification. Par exemple, ce qui suit recherche le temps à San Francisco les jours pluvieux :

```
SELECT * FROM temps
    WHERE ville = 'San Francisco' AND prcp > 0.0;
```

Résultat :

```
     ville     | temp_basse | temp_haute | prcp |    date
---------------+------------+------------+------+------------
 San Francisco |         46 |         50 | 0.25 | 1994-11-27
(1 row)
```

Vous pouvez demander que les résultats d'une requêtes soient renvoyés dans un ordre trié :

```
SELECT * FROM temps
ORDER BY ville;
```

```
 ville         | temp_basse | temp_haute | prcp |    date
---------------+------------+------------+------+------------
Hayward        |         37 |         54 |      | 1994-11-29
San Francisco  |         43 |         57 |    0 | 1994-11-29
```

```
San Francisco |         46 |         50 | 0.25 | 1994-11-27
```

Dans cet exemple, l'ordre de tri n'est pas spécifié complètement, donc vous pouvez obtenir les lignes San Francisco dans n'importe quel ordre. Mais, vous auriez toujours obtenu les résultats affichés ci−dessus si vous aviez fait

```
SELECT * FROM temps
ORDER BY ville, temp_basse;
```

Vous pouvez demander que les lignes dupliquées soient supprimées du résultat d'une requête :

```
SELECT DISTINCT ville
    FROM temps;

     ville
---------------
 Hayward
 San Francisco
(2 rows)
```

De nouveau, l'ordre des lignes résultats pourrait varier. Vous pouvez vous assurer des résultats cohérents en utilisant `DISTINCT` et `ORDER BY` ensemble : [3]

```
SELECT DISTINCT ville
FROM temps
ORDER BY ville;
```

## 2.6. Jointures entre les tables

Jusqu'ici, nos requêtes avaient seulement consulté une table à la fois. Les requêtes peuvent accéder à plusieurs tables en même temps ou accéder à la même table de façon à ce que plusieurs lignes de la table soient traitées en même temps. Une requête qui consulte plusieurs lignes de la même ou de différentes tables en même temps est appelée requête de *jointure*. Comme exemple, dites−vous que vous souhaitez lister toutes les entrées du temps avec la colonne des noms de toutes les lignes de la table des villes et que vous choisissez les paires de lignes où ces valeurs correspondent.

> **Note :** Ceci est uniquement un modèle conceptuel. La jointure est habituellement exécutée d'une manière plus efficace qu'en comparant chaque paire de lignes mais c'est invisible pour l'utilisateur.

Ceci sera accompli avec la requête suivante :

```
SELECT *
    FROM temps, villes
    WHERE ville = nom;

     ville     | temp_basse | temp_haute | prcp |    date    |     nom       | emplacement
---------------+------------+------------+------+------------+---------------+-------------
 San Francisco |         46 |         50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |         43 |         57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

Remarquez deux choses à propos du résultat :

- Il n'y a pas de lignes pour la ville de Hayward dans le résultat. C'est parce qu'il n'y a aucune entrée correspondante dans la table `villes` pour Hayward, donc la jointure ignore les lignes n'ayant pas de correspondance avec la table temps. Nous verrons rapidement comment cela peut être résolu.
- Il y a deux colonnes contenant le nom des villes. Cela est correct car les listes de colonnes des tables `temps` et `villes` sont concaténées. En pratique, ceci est indésirable, vous voudrez probablement lister les colonnes explicitement plutôt que d'utiliser `*` :

```
SELECT ville, temp_basse, temp_haute, prcp, date, emplacement
    FROM temps, villes
    WHERE ville = nom;
```

**Exercice :** Essayez de trouver la sémantique de cette requête quand la clause `WHERE` est omise.

Puisque toutes les colonnes ont un nom différent, l'analyseur a automatiquement trouvé à quelle table elles appartiennent, mais c'est mieux de qualifier pleinement les noms des colonnes dans les requêtes de jointure :

```
SELECT temps.ville, temps.temp_basse, temps.temp_haute,
       temps.prcp, temps.date, villes.emplacement
    FROM temps, villes
    WHERE villes.nom = temps.ville;
```

Les requêtes de jointure vues jusqu'ici peuvent aussi être écrites dans un format alternatif :

```
SELECT *
    FROM temps INNER JOIN villes ON (temps.ville = villes.nom);
```

Cette syntaxe n'est pas aussi couramment utilisée que les précédentes mais nous la montrons ici pour vous aider à comprendre les sujets suivants.

Maintenant, nous allons essayer de comprendre comment nous pouvons avoir les entrées de Hayward. Nous voulons que la requête balaie la table `temps` et que, pour chaque ligne, elle trouve la ligne de `villes` correspondante. Si aucune ligne correspondante n'est trouvée, nous voulons que les valeurs des colonnes de la table `villes` soient remplacées par des << valeurs vides >>. Ce genre de requêtes est appelé *jointure externe* (outer join). (Les jointures que nous avons vus jusqu'ici sont des jointures internes −− inner joins.) La commande ressemble à cela :

```
SELECT *
    FROM temps LEFT OUTER JOIN villes ON (temps.ville = villes.nom);

     ville     | temp_basse | temp_haute | prcp |    date    |     nom       | emplacement
---------------+------------+------------+------+------------+---------------+-------------
 Hayward       |         37 |         54 |      | 1994-11-29 |               |
 San Francisco |         46 |         50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |         43 |         57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

Cette requête est appelée une *jointure externe à gauche* (left outer join) parce que la table mentionnée à la gauche de l'opérateur de jointure aura au moins une fois ses lignes dans le résultat tandis que la table sur la droite aura seulement les lignes qui correspondent à des lignes de la table de gauche. Lors de l'affichage d'une ligne de la table de gauche pour laquelle il n'y a pas de correspondance dans la table de droite, des valeurs vides (NULL) sont mises pour les colonnes de la table de droite.

**Exercice :** Il existe aussi des jointures externes à droite et des jointures externes complètes. Essayez de trouvez ce qu'elles font.

Nous pouvons également joindre une table avec elle–même. Ceci est appelé une *jointure réflexive*. Comme exemple, supposons que nous voulons trouver toutes les entrées de temps qui sont dans un intervalle de température d'autres entrées de temps. Nous avons donc besoin de comparer les colonnes `temp_basse` et `temp_haute` de chaque ligne de `temps` aux colonnes `temp_basse` et `temp_haute` de toutes les autres lignes de `temps`. Nous pouvons faire cela avec la requête suivante :

```
SELECT W1.ville, W1.temp_basse AS low, W1.temp_haute AS high,
    W2.ville, W2.temp_basse AS low, W2.temp_haute AS high
    FROM temps W1, temps W2
    WHERE W1.temp_basse < W2.temp_basse
    AND W1.temp_haute > W2.temp_haute;

     ville      | low | high |     ville     | low | high
----------------+-----+------+---------------+-----+------
  San Francisco |  43 |   57 | San Francisco |  46 |   50
  Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

Ici, nous avons renommé la table temps en `W1` et en `W2` pour être capable de distinguer le côté gauche et droit de la jointure. Vous pouvez aussi utiliser ce genre d'alias dans d'autres requêtes pour économiser de la frappe, c'est–à–dire :

```
SELECT *
    FROM temps w, villes c
    WHERE w.ville = c.nom;
```

Vous rencontrerez ce genre d'abréviation assez fréquemment.

## 2.7. Fonctions d'agrégat

Comme la plupart des autres produits de bases de données relationnelles, PostgreSQL supporte les fonctions d'agrégat. Une fonction d'agrégat calcule un seul résultat à partir de plusieurs lignes en entrée. Par exemple, il y a des agrégats pour calculer le compte (`count`), la somme (`sum`), la moyenne (`avg`), le maximum (`max`) et le minimum (`min`) d'un ensemble de lignes.

Comme exemple, nous pouvons trouvez la température la plus haute parmi les températures les plus faibles avec :

```
SELECT max(temp_basse) FROM temps;

 max
-----
  46
(1 row)
```

Si nous voulons connaître dans quelle ville (ou villes) ces lectures se sont produites, nous pouvons essayer

```
SELECT ville FROM temps WHERE temp_basse = max(temp_basse);
FAUX
```

mais cela ne marchera pas puisque l'agrégat `max` ne peut pas être utilisé dans une clause `WHERE`. (Cette restriction existe parce que la clause `WHERE` détermine les lignes qui seront traitées par l'agrégat ; donc les lignes doivent être évaluées avant que les fonctions d'agrégat calculent.) Cependant, comme cela est souvent

le cas, la requête peut être répétée pour arriver au résultat attendu, ici en utilisant une *sous−requête* :

```
SELECT ville FROM temps
    WHERE temp_basse = (SELECT max(temp_basse) FROM temps);

     ville
---------------
 San Francisco
(1 row)
```

Ceci est bon car la sous−requête est un calcul indépendant qui traite son propre agrégat séparément à partir de ce qui se passe dans la requête externe.

Les agrégats sont également très utiles s'ils sont combinés avec les clauses `GROUP BY`. Par exemple, nous pouvons obtenir la plus haute température parmi les températures les plus faibles observées dans chaque ville avec

```
SELECT ville, max(temp_basse)
    FROM temps
    GROUP BY ville;

     ville     | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

ce qui nous donnes une ligne par ville dans le résultat. Chaque résultat d'agrégat est calculé avec les lignes de la tables correspondant à la ville. Nous pouvons filtrer ces lignes groupées en utilisant `HAVING` :

```
SELECT ville, max(temp_basse)
    FROM temps
    GROUP BY ville
    HAVING max(temp_basse) < 40;

  ville  | max
---------+-----
 Hayward |  37
(1 row)
```

ce qui nous donnes le même résultat uniquement pour les villes qui ont toutes leurs valeurs de `temp_basse` en−dessous de 40. Pour finir, si nous nous préoccupons seulement des villes dont le nom commence par << `S` >>, nous pouvons faire

```
SELECT ville, max(temp_basse)
    FROM temps
    WHERE ville LIKE 'S%'(1)
    GROUP BY ville
    HAVING max(temp_basse) < 40;
```

**(1)**

L'opérateur `LIKE` fait la correspondance avec un modèle ; cela est expliqué dans la Section 9.7.

Il est important de comprendre l'interaction entre les agrégats et les clauses SQL `WHERE` et `HAVING`. La différence fondamentale entre `WHERE` et `HAVING` est que `WHERE` sélectionne les lignes en entrée avant que les groupes et les agrégats ne soient traités (donc, cette clause contrôle quelles lignes vont dans le calcul de

l'agrégat) tandis que HAVING sélectionne les lignes groupées après que les groupes et les agrégats aient été traités. Donc, la clause WHERE ne doit pas contenir des fonctions d'agrégat ; cela n'a aucun sens d'essayer d'utiliser un agrégat pour déterminer quelles lignes seront en entrée des agrégats. D'un autre côté, la clause HAVING contient toujours des fonctions d'agrégat. (Pour être précis, vous êtes autorisés à écrire une clause HAVING qui n'utilise pas d'agrégats, mais c'est inutile. La même condition pourra être utilisée plus efficacement par un WHERE.)

Dans l'exemple précédent, nous pouvons appliquer la restriction sur le nom de la ville dans WHERE puisque cela n'a besoin d'aucun agrégat. C'est plus efficace que d'ajouter la restriction dans HAVING parce que nous évitons le groupement et les calculs d'agrégat pour toutes les lignes qui ont échoué lors du contrôle fait par WHERE.

## 2.8. Mises à jour

Vous pouvez mettre à jour une ligne existante en utilisant la commande UPDATE. Supposez que vous découvrez que les températures sont toutes excédentes de 2 degrés à partir du 28 novembre. Vous pouvez mettre à jour les données de la manière suivante :

```
UPDATE temps
    SET temp_haute = temp_haute - 2,  temp_basse = temp_basse - 2
    WHERE date > '1994-11-28';
```

Regardez le nouvel état des données :

```
SELECT * FROM temps;

     ville      | temp_basse | temp_haute | prcp |    date
----------------+------------+------------+------+------------
 San Francisco  |         46 |         50 | 0.25 | 1994-11-27
 San Francisco  |         41 |         55 |    0 | 1994-11-29
 Hayward        |         35 |         52 |      | 1994-11-29
(3 rows)
```

## 2.9. Suppressions

Les lignes peuvent être supprimées de la table avec la commande DELETE. Supposez que vous n'êtes plus intéressé par le temps de Hayward. Vous pouvez faire ce qui suit pour supprimer ses lignes de la table :

```
DELETE FROM temps WHERE ville = 'Hayward';
```

Toutes les entrées de temps pour Hayward sont supprimées.

```
SELECT * FROM temps;

     ville     | temp_basse | temp_haute | prcp |    date
---------------+------------+------------+------+------------
 San Francisco |         46 |         50 | 0.25 | 1994-11-27
 San Francisco |         41 |         55 |    0 | 1994-11-29
(2 rows)
```

Faire très attention aux instructions de la forme

```
DELETE FROM nom_table;
```

Sans une qualification, `DELETE` supprimera *toutes* les lignes de la table donnée, la laissant vide. Le système le fera sans demander de confirmation !

---

# Chapitre 3. Fonctionnalités avancées

## 3.1. Introduction

Dans le chapitre précédent, nous avons couvert les bases de l'utilisation de SQL pour stocker et accéder à vos données avec PostgreSQL. Nous allons maintenant aborder quelques fonctionnalités avancées de SQL simplifiant la gestion et empêchant la perte ou la corruption de vos données. Enfin, nous regarderons quelques extensions de PostgreSQL.

Ce chapitre se référera occasionnellement aux exemples disponibles dans le Chapitre 2 pour les modifier ou les améliorer, donc il serait avantageux d'avoir lu ce chapitre. Quelques exemples de ce chapitre sont aussi disponibles dans `advanced.sql` placé dans le répertoire du tutoriel. Ce fichier contient aussi quelques données à charger pour l'exemple, ce qui ne sera pas répété ici. (Référez−vous à la Section 2.1 pour savoir comment utiliser ce fichier.)

---

## 3.2. Vues

Référez−vous aux requêtes de la Section 2.6. Supposons que la liste des enregistrements du temps et des villes soit d'un intérêt particulier pour votre application mais que vous ne voulez pas saisir la requête à chaque fois que vous en avez besoin. Vous pouvez créer une *vue* avec la requête, ce qui donne un nom à la requête à laquelle vous pouvez vous référer comme dans le cas d'une table ordinaire.

```
CREATE VIEW mavue AS
    SELECT ville, temp_basse, temp_haute, prcp, date, emplacement
        FROM temps, villes
        WHERE ville = nom;

SELECT * FROM mavue;
```

Avoir une utilisation libérale des vues est un aspect clé d'une bonne conception des bases de données avec SQL. Les vues vous permettent d'encapsuler les détails de la structure de vos tables, qui pourraient changer lors de l'évolution de votre application, tout en restant consistant au niveau de l'interface.

Les vues peuvent être utilisées pratiquement partout où une vraie table est utilisable. Construire des vues basées sur d'autres vues n'est pas inhabituel.

---

## 3.3. Clés secondaires

Souvenez−vous des tables `temps` et `villes` du Chapitre 2. Considérez le problème suivant : vous voulez vous assurer que personne n'insère de lignes dans la table `temps` qui ne correspondraient pas à une entrée dans la table `villes`. Ceci maintient l'*intégrité référentielle* de vos données. Dans les systèmes de bases de données simples, ceci serait implémenté (si possible) en vérifiant en premier lieu que la table `villes` dispose bien d'un enregistrement correspondant, puis en insérant ou en empêchant l'insertion du nouvel enregistrement dans `temps`. Cette approche présente un certain nombre de problèmes et n'est pas très pratique, donc PostgreSQL peut s'en charger pour vous.

La nouvelle déclaration des tables ressemblerait à ceci :

```
CREATE TABLE villes (
        ville       varchar(80) primary key,
        emplacement point
);

CREATE TABLE temps (
        ville      varchar(80) references villes,
        temp_haute int,
        temp_basse int,
        prcp       real,
        date       date
);
```

Maintenant, essayons d'insérer un enregistrement non valide :

```
INSERT INTO temps VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');

ERROR:  insert or update on table "temps" violates foreign key constraint "temps_ville_fkey"
DETAIL:  Key (ville)=(Berkeley) is not present in table "villes".
```

Le comportement des clés secondaires peut être précisé très finement pour votre application. Nous n'irons pas plus loin que cet exemple simple dans ce tutoriel mais référez−vous simplement au Chapitre 5 pour plus d'informations. Utiliser correctement les clés secondaires améliore la qualité de vos applications de bases de données, donc vous êtes fortement encouragé à les connaître.

## 3.4. Transactions

Les *transactions* sont un concept fondamental de tous les systèmes de bases de données. Le point essentiel d'une transaction est qu'il assemble plusieurs étapes en une seule opération tout−ou−rien. Les états intermédiaires entre les étapes ne sont pas visibles pour les autres transactions concurrentes, et si un échec survient empêchant la transaction de bien se terminer, alors aucune des étapes n'affecte la base de données.

Par exemple, considérez la base de données d'une banque qui contiendrait la balance pour différents comptes clients, ainsi que les balances du total du dépôt par branches. Supposez que nous voulons enregistrer un virement de 100 euros du compte d'Alice vers celui de Bob. En simplifiant énormément, les commandes SQL pour ceci ressembleraient à ça

```
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE nom = (SELECT nom_branche FROM comptes WHERE nom = 'Alice');
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE nom = (SELECT nom_branche FROM comptes WHERE nom = 'Bob');
```

Les détails de ces commandes ne sont pas importants ici ; le point important est que cela nécessite plusieurs mises à jour séparées pour accomplir cette opération assez simple. Les employés de la banque voudront être assurés que soit toutes les commandes sont effectuées soit aucune ne l'est. Il ne serait pas acceptable que, suite à une erreur du système, Bob reçoive 100 euros qui n'ont pas été débités du compte d'Alice. De la même façon, Alice ne restera pas longtemps une cliente si elle est débitée du montant sans que celui−ci ne soit crédité sur le compte de Bob. Nous avons besoin d'une garantie comme quoi si quelque chose se passe mal, aucune des étapes déjà exécutées ne prendra effet. Grouper les mises à jour en une *transaction* nous donne

cette garantie. Une transaction est dite *atomique* : du point de vue des autres transactions, cela se passe complètement ou pas du tout.

Nous voulons aussi la garantie qu'une fois une transaction terminée et validée par le système de base de données, les modifications seront enregistrées de façon permanente et ne seront pas perdues même si un arrêt brutal arrive peu après. Par exemple, si nous enregistrons un retrait d'argent par Bob, nous ne voulons surtout pas que le débit de son compte disparaisse lors d'un crash à sa sortie de la banque. Une base de données transactionnelle garantit que toutes les mises à jour faites lors d'une transaction sont enregistrées dans un stockage permanent (c'est−à−dire sur disque) avant que la transaction ne soit validée.

Une autre propriété importante des bases de données transactionnelles est en relation étroite avec la notion de mises à jour atomiques : quand de multiples transactions sont lancées en parallèle, chacune d'entre elles ne doit pas être capable de voir les modifications incomplètes faites par les autres. Par exemple, si une transaction est occupée à calculer le total de toutes les branches, il ne serait pas bon d'inclure le débit de la branche d'Alice sans le crédit de la branche de Bob, ou vice−versa. Donc, les transactions doivent être tout−ou−rien non seulement pour leur effet permanent sur la base de données, mais aussi pour leur visibilité au moment de leur exécution. Les mises à jour faites ainsi par une transaction ouverte sont invisibles aux autres transactions jusqu'à la fin de celle−ci, moment qui rendra visible toutes les mises à jours simultanément.

Avec PostgreSQL, une transaction est réalisée en entourant les commandes SQL de la transaction avec les commandes `BEGIN` et `COMMIT`. Donc, notre transaction pour la banque ressemblera à ceci

```
BEGIN;
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
-- etc etc
COMMIT;
```

Si, au cours de la transaction, nous décidons que nous ne voulons pas valider (peut−être nous sommes−nous aperçu que la balance d'Alice devenait négative), nous pouvons envoyer la commande `ROLLBACK` au lieu de `COMMIT`, et toutes nos mises à jour jusqu'à maintenant seront annulées.

En fait, PostgreSQL traite chaque instruction SQL comme étant exécutée dans une transaction. Si vous ne lancez pas une commande `BEGIN`, alors chaque instruction individuelle se trouve enveloppée avec un `BEGIN` et (en cas de succès) un `COMMIT` implicites. Un groupe d'instructions entouré par un `BEGIN` et un `COMMIT` est quelque fois appelé un *bloc transactionnel*.

> **Note :** Quelques bibliothèques clients lancent les commandes `BEGIN` et `COMMIT` automatiquement, de façon à ce que vous bénéficiiez des effets des blocs transactionnels sans les demander. Vérifiez la documentation de l'interface que vous utilisez.

Il est possible de contrôler les instructions dans une transaction d'une façon plus granulaire avec l'utilisation des *points de sauvegarde*. Les points de sauvegarde vous permettent d'annuler des parties de la transaction tout en validant le reste. Après avoir défini un point de sauvegarde avec `SAVEPOINT`, vous pouvez, si nécessaire, annuler jusqu'au point de sauvegarde avec `ROLLBACK TO`. Toutes les modifications de la transaction dans la base de données entre le moment où le point de sauvegarde est défini et celui où l'annulation est demandée sont annulées mais les modifications antérieures au point de sauvegarde sont conservées.

Après avoir annulé jusqu'à un point de sauvegarde, il reste défini, donc vous pouvez de nouveau annuler plusieurs fois et rester au même point. Par contre, si vous êtes sûr de ne plus avoir besoin d'annuler jusqu'à un

point de sauvegarde particulier, il peut être libéré pour que le système puisse récupérer quelques ressources. Gardez à l'esprit que libérer un point de sauvegarde ou annuler les opérations jusqu'à ce point de sauvegarde libérera tous les points de sauvegarde définis après lui.

Tout ceci survient à l'intérieur du bloc de transaction, donc ce n'est pas visible par les autres sessions de la base de données. Quand et si vous validez le bloc de transaction, les actions validées deviennent visibles en un seul coup aux autres sessions, alors que les actions annulées ne deviendront jamais visibles.

Rappelez−vous la base de données de la banque. Supposons que nous débitons le compte d'Alice de $100.00, somme que nous créditons au compte de Bob, pour trouver plus tard que nous aurions dû créditer le compte de Wally. Nous pouvons le faire en utilisant des points de sauvegarde comme ceci :

```
BEGIN;
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
SAVEPOINT mon_pointdesauvegarde;
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Bob';
-- oups ... oublions ça et créditons le compte de Wally
ROLLBACK TO mon_pointdesauvegarde;
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Wally';
COMMIT;
```

Cet exemple est bien sûr très simplifié mais il y a beaucoup de contrôle possible dans un bloc de transaction grâce à l'utilisation des points de sauvegarde. De plus, `ROLLBACK TO` est le seul moyen pour regagner le contrôle d'un bloc de transaction qui a été placé dans un état d'annulation par le système à cause d'une erreur, plutôt que de tout annuler et de tout recommencer.

## 3.5. Héritage

L'héritage est un concept provenant des bases de données orientées objet. Il ouvre de nouvelles possibilités intéressantes dans la conception de bases de données.

Créons deux tables : une table `villes` et une table `capitales`. Naturellement, les capitales sont aussi des villes, donc vous voulez un moyen pour afficher implicitement les capitales lorsque vous listez les villes. Si vous êtes réellement intelligent, vous pourriez inventer ceci :

```
CREATE TABLE capitales (
  nom        text,
  population real,
  altitude   int,    -- (en pied)
  etat       char(2)
);

CREATE TABLE non_capitales (
  nom        text,
  population real,
  altitude   int     -- (en pied)
);

CREATE VIEW villes AS
  SELECT nom, population, altitude FROM capitales
    UNION
```

```
  SELECT nom, population, altitude FROM non_capitales;
```

Ceci fonctionne bien pour les requêtes, mais c'est horrible lorsque vous avez besoin de mettre à jour plusieurs lignes par exemple.

Voici une meilleure solution :

```
CREATE TABLE villes (
  nom        text,
  population real,
  altitude   int     -- (en pied)
);

CREATE TABLE capitales (
  etat      char(2)
) INHERITS (villes);
```

Dans ce cas, une ligne de `capitales` *hérite* de toutes les colonnes (`nom`, `population` et `altitude`) de son *parent*, `villes`. Le type de la colonne `nom` est `text`, un type natif de PostgreSQL pour les chaînes de caractères à longueur variable. Les capitales d'état ont une colonne supplémentaire, etat, qui affiche leur état. Dans PostgreSQL, une table peut hériter d'aucune ou de plusieurs autres tables.

Par exemple, la requête suivante trouve les noms de toutes les villes, en incluant les capitales des états, situées à une altitude de plus de 500 pieds :

```
SELECT nom, altitude
  FROM villes
  WHERE altitude > 500;
```

ce qui renvoie :

```
   nom     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

Autrement, la requête suivante trouve toutes les villes qui ne sont pas des capitales et qui sont situées à une altitude d'au moins 500 pieds :

```
SELECT nom, altitude
    FROM ONLY villes
    WHERE altitude > 500;

   nom     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

Ici, `ONLY` avant `villes` indique que la requête ne doit être lancée que sur la table `villes`, et non pas sur les tables sous `villes` suivant la hiérarchie des héritages. La plupart des commandes dont nous avons déjà discutées — `SELECT`, `UPDATE` et `DELETE` — supportent cette notation (`ONLY`).

**Note :** Bien que l'héritage soit fréquemment utile, il n'a pas été intégré avec les contraintes uniques ou les clés étrangères, ce qui limite leur utilité. Voir Section 5.5 pour plus de détails.

## 3.6. Conclusion

PostgreSQL a bien plus de fonctionnalités que celles aperçues lors de ce tutoriel d'introduction, qui a été orienté vers les nouveaux utilisateurs de SQL. Ces fonctionnalités sont discutées avec plus de détails dans le reste de ce livre.

Si vous sentez que vous avez besoin d'une introduction plus approfondie, merci de visiter le site web PostgreSQL pour des liens vers d'autres ressources.

# II. Le langage SQL

Cette partie décrit l'utilisation du langage SQL dans PostgreSQL. Nous démarrons en décrivant la syntaxe générale de SQL, puis expliquons comment créer les structures pour contenir les données, comment peupler la base et comment l'interroger. La partie centrale liste les types de données et les fonctions disponibles ainsi que leur utilisation dans les requêtes SQL. Le reste traite de plusieurs aspects qui sont importants pour l'optimisation d'une base de données dans le but d'obtenir des performances idéales.

L'information dans cette partie est arrangée de façon à ce qu'un utilisateur novice puisse la suivre du début à la fin et avoir ainsi une compréhension complète des sujets sans avoir à s'y référer trop souvent. Les chapitres sont prévus pour être indépendants, de sorte que les utilisateurs avancés puissent lire individuellement les chapitres qu'ils ont choisis. L'information dans cette partie est présentée dans un style narratif en unités thématiques. Les lecteurs qui cherchent une description complète d'une commande particulière devraient regarder dans la Partie VI.

Les lecteurs de cette partie devraient savoir comment se connecter à une base PostgreSQL et éditer des commandes SQL. Les lecteurs qui ne sont pas familiarisés avec ces points sont encouragés à lire d'abord la Partie I. Les commandes SQL sont généralement saisies en utilisant le terminal interactif de PostgreSQL psql, mais d'autres programmes qui ont des fonctionnalités similaires peuvent aussi bien être utilisés.

**Table des matières**

# Chapitre 4. Syntaxe SQL

Ce chapitre décrit la syntaxe de SQL. Il donne les fondements pour comprendre les chapitres suivants qui iront plus en détail sur la façon dont les commandes SQL sont appliquées pour définir et modifier des données.

Nous avertissons aussi nos utilisateurs, déjà familiers avec le SQL, qu'ils doivent lire ce chapitre très attentivement car il existe plusieurs règles et concepts implémentés différemment suivant les bases de données SQL ou spécifiques à PostgreSQL.

## 4.1. Structure lexicale

Une entrée SQL consiste en une séquence de *commandes*. Une commande est composée d'une séquence de *jetons*, terminés par un point−virgule (<< ; >>). La fin du flux en entrée se termine aussi par une commande. Les jetons valides dépendent de la syntaxe particulière de la commande.

Un jeton peut être un *mot clé*, un *identifiant*, un *identifiant entre guillemets*, un *littéral* (ou une constante) ou un symbole de caractère spécial. Les jetons sont normalement séparés par des espaces blancs (espace, tabulation, nouvelle ligne) mais n'ont pas besoin de l'être s'il n'y a pas d'ambiguïté (ce qui est seulement le cas si un caractère spécial est adjacent à des jetons d'autres types).

De plus, des *commentaires* peuvent se trouver dans l'entrée SQL. Ce ne sont pas des jetons, ils sont réellement équivalents à un espace blanc.

Par exemple, ce qui suit est (syntaxiquement) valide pour une entrée SQL :

```
SELECT * FROM MA_TABLE;
UPDATE MA_TABLE SET A = 5;
INSERT INTO MA_TABLE VALUES (3, 'salut ici');
```

C'est une séquence de trois commandes, une par ligne (bien que cela ne soit pas requis ; plusieurs commandes pourraient se trouver sur une même ligne et les commandes peuvent être placées sur plusieurs lignes).

La syntaxe SQL n'est pas très consistante en ce qui concerne les jetons identifiants des commandes et lesquels sont des opérandes ou des paramètres. Les premiers jetons sont généralement le nom de la commande. Dans l'exemple ci−dessus, nous parlons d'une commande << SELECT >>, d'une commande << UPDATE >> et d'une commande << INSERT >>. Mais en fait, la commande `UPDATE` requiert toujours un jeton `SET` apparaissant dans une certaine position, et cette variante particulière de `INSERT` requiert aussi un `VALUES` pour être complète. Les règles de syntaxe précises pour chaque commande sont décrites dans la Partie VI.

### 4.1.1. Identifieurs et mots clés

Les jetons tels que `SELECT`, `UPDATE` ou `VALUES` dans l'exemple ci−dessus sont des exemples de *mots clés*, c'est−à−dire des mots qui ont une signification dans le langage SQL. Les jetons `MA_TABLE` et `A` sont des exemples d'*identifieurs*. Ils identifient des noms de tables, colonnes ou d'autres objets de la base de données suivant la commande qui a été utilisée. Du coup, ils sont quelques fois simplement nommés des << noms >>.

Les mots clés et les identifieurs ont la même structure lexicale, signifiant que quelqu'un ne peut pas savoir si un jeton est un identifiant ou un mot clé sans connaître le langage. Une liste complète des mots clé est disponible dans l'Annexe C.

Les identifieurs et les mots clés SQL doivent commencer avec une lettre (a−z, mais aussi des lettres de marques diacritiques différentes et des lettres non latines) ou un tiret bas (_). Les caractères suivants dans un identifiant ou dans un mot clé peuvent être des lettres, des tirets−bas, des chiffres (0−9) ou des signes dollar ($). Notez que les signes dollar ne sont pas autorisés en tant qu'identifiant suivant le standard SQL, donc leur utilisation pourrait rendre les applications moins portables. Le standard SQL ne définira pas un mot clé contenant des chiffres ou commençant ou finissant par un tiret bas, donc les identifieurs de cette forme sont sûr de passer les conflits possibles avec les futures extensions du standard.

Le système utilise pas plus de NAMEDATALEN−1 caractères d'un identifiant ; les noms longs peuvent être écrits dans des commandes mais ils seront tronqués. Par défaut, NAMEDATALEN vaut 64. Du coup, la taille maximum de l'identifiant est de 63. Si cette limite est problématique, elle peut être élevée en modifiant NAMEDATALEN dans src/include/postgres_ext.h.

L'identifiant et les noms de mots clés sont insensibles à la casse. Du coup,

```
UPDATE MA_TABLE SET A = 5;
```

peut aussi s'écrire de cette façon

```
uPDaTE ma_TabLE SeT a = 5;
```

Une convention souvent utilisée est d'écrire les mots clés en majuscule et les noms en minuscule, c'est−à−dire

```
UPDATE ma_table SET a = 5;
```

Voici un deuxième type d'identifiant : l'*identifiant délimité* ou l'*identifiant entre guillemets*. Il est formé en englobant une séquence arbitraire de caractères entre des guillemets doubles ("). Un identifiant délimité est toujours un identifiant, jamais un mot clé. Donc, "select" pourrait être utilisé pour faire référence à une colonne ou à une table nommée << select >>, alors qu'un select sans guillemets sera pris pour un mot clé et du coup, pourrait provoquer une erreur d'analyse lorsqu'il est utilisé alors qu'un nom de table ou de colonne est attendu. L'exemple peut être écrit avec des identifieurs entre guillemets comme ceci :

```
UPDATE "ma_table" SET "a" = 5;
```

Les identifieurs entre guillemets peuvent contenir tout caractère autre qu'un guillemet double. (Pour inclure un guillemet double, écrivez deux guillemets doubles.) Ceci permet la construction de noms de tables et de colonnes qui ne seraient pas possibles autrement, comme des noms contenant des espaces ou des arobases. La limitation de la longueur s'applique toujours.

Mettre un identifiant entre guillemets le rend sensible à la casse alors que les noms sans guillemets sont toujours convertis en minuscules. Par exemple, les identifieurs FOO, foo et "foo" sont considérés identiques par PostgreSQL mais "Foo" et "FOO" sont différents des trois autres et entre eux. (La mise en minuscule des noms sans guillemets avec PostgreSQL n'est pas compatible avec le standard SQL qui indique que les noms sans guillemets devraient être mis en majuscule. Du coup, foo devrait être équivalent à "FOO" et non pas à "foo" en respect avec le standard. Si vous voulez écrire des applications portables, nous vous conseillons de toujours mettre entre guillemets un nom particulier ou de ne jamais le mettre.)

## 4.1.2. Constantes

Il existe trois *types implicites de constantes* dans PostgreSQL : les chaînes, les chaînes de bits et les nombres. Les constantes peuvent aussi être spécifiées avec des types explicites, ce qui peut activer des représentations plus précises et gérées plus efficacement par le système. Les constantes implicites sont décrites ci−dessous ; ces constantes sont discutées dans les sous−sections suivantes.

### 4.1.2.1. Constantes de chaînes

Une constante de type chaîne en SQL est une séquence arbitraire de caractères entourée par des guillemets simples (`'`), c'est−à−dire `'Ceci est une chaîne'`. La façon standard d'écrire un guillemet simple dans une chaîne constante est d'écrire deux guillemets simples adjacents, par exemple `'Le cheval d''Anne'`. PostgreSQL autorise aussi l'échappement des guillemets simples avec un antislash (`\`), par exemple `'Le cheval d\'Anne'`.

Une autre extension de PostgreSQL fait que les échappements style C sont disponibles : `\b` est un retour arrière, `\f` est un retour chariot, `\n` est une nouvelle ligne, `\r` est un retour chariot, `\t` est une tabulation et `\`*`xxx`*, où *`xxx`* est un nombre en octal, est un octet avec la code correspondant. (Il est de votre responsabilité que les séquences d'octets créées soient composées de caractères valides dans le codage de l'ensemble des caractères.) Tout autre caractère suivant un antislash est pris littéralement. Du coup, pour inclure un antislash dans une constante de type chaîne, saisissez deux antislashs.

Le caractère de code zéro ne peut pas être dans une constante de type chaîne.

Deux constantes de type chaîne séparées par un espace blanc *avec au moins une nouvelle ligne* sont concaténées et traitées réellement comme si la chaîne avait été écrite dans une constante. Par exemple :

```
SELECT 'foo'
'bar';
```

est équivalent à

```
SELECT 'foobar';
```

mais

```
SELECT 'foo'      'bar';
```

n'a pas une syntaxe valide. (Ce comportement légèrement bizarre est spécifié par le standard SQL ; PostgreSQL suit le standard.)

### 4.1.2.2. Constantes de chaînes avec guillemet dollar

Alors que la syntaxe standard pour la spécification des constantes de chaînes est généralement agréable, elle peut être difficile à comprendre quand la chaîne désirée contient un grand nombre de guillemets ou d'antislashs car chacun d'entre eux doit être doublé. Pour permettre la saisie de requêtes plus lisibles dans de telles situations, PostgreSQL fournit une autre façon, appelée << guillemet dollar >>, pour écrire des constantes de chaînes. Une constante de chaîne avec guillemet dollar consiste en un signe dollar (`$`), une

<< balise >> optionnelle de zéro ou plus de caractères, un autre signe dollar, une séquence arbitraire de caractères qui constitue le contenu de la chaîne, un signe dollar, la même balise et un signe dollar. Par exemple, voici deux façons de spécifier la chaîne << Le cheval d'Anne >> en utilisant les guillemets dollar :

```
$$Le cheval d'Anne$$
$UneBalise$Le cheval d'Anne$UneBalise$
```

Notez qu'à l'intérieur de la chaîne avec guillemet dollar, les guillemets simples peuvent être utilisés sont devoir être échappés. En fait, aucun caractère à l'intérieur d'une chaîne avec guillemet dollar n'a besoin d'être échappé : le contenu est toujours écrit littéralement. Les antislashs ne sont pas spéciaux, pas plus que les signes dollar, sauf s'ils font partie d'une séquence correspondant à la balise ouvrante.

Il est possible d'imbriquer les constantes de chaînes avec guillemets dollar en utilisant différentes balises pour chaque niveau d'imbrication. Ceci est habituellement utilisé lors de l'écriture de définition de fonctions. Par exemple :

```
$function$
BEGIN
RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Ici, la séquence `$q$[\t\r\n\v\\]$q$` représente une chaîne littérale avec guillemet dollar `[\t\r\n\v\\]`, qui sera reconnu quand le corps de la fonction est exécuté par PostgreSQL. Mais comme la séquence ne correspond pas au délimiteur `$function$`, il s'agit juste de quelques caractères à l'intérieur de la constante pour autant que ne le sait la chaîne externe.

La balise d'une chaîne avec guillemets dollar, si elle existe, suit les mêmes règles qu'un identificateur sans guillemets, sauf qu'il ne peut pas contenir de signes dollar. Les balises sont sensibles à la casse, du coup `$balise$Contenu de la chaîne$balise$` est correct mais `$BALISE$Contenu de la chaîne$balise$` ne l'est pas.

Une chaîne avec guillemets dollar suivant un mot clé ou un identifiant doit en être séparé par un espace blanc ; sinon, le délimiteur du guillemet dollar serait pris comme faisant parti de l'identifiant précédent.

Le guillemet dollar ne fait pas partie du standard SQL mais c'est un moyen bien plus agréable pour écrire des chaînes littérales que d'utiliser la syntaxe des guillemets simples, bien que compatible avec le standard. Elle est particulièrement utile pour représenter des constantes de type chaîne à l'intérieur d'autres constantes, comme cela est souvent le cas avec les définitions de fonctions. Avec la syntaxe des guillemets simples, chaque antislash dans l'exemple précédent devrait avoir été écrit avec quatre antislashs, ce qui sera réduit à deux antislashs dans l'analyse de la constante originale, puis à un lorsque la constante interne est ré−analysée lors de l'exécution de la fonction.

#### 4.1.2.3. Constantes de chaînes de bits

Les constantes de chaînes de bits ressemblent aux constantes de chaînes standards avec un `B` (majuscule ou minuscule) juste avant le guillemet du début (sans espace blanc), c'est−à−dire `B'1001'`. Les seuls caractères autorisés dans les constantes de type chaîne de bits sont `0` et `1`.

Autrement, les constantes de chaînes de bits peuvent être spécifiées en notation hexadécimale en utilisant un `X` avant (minuscule ou majuscule), c'est–à–dire `X'1FF'`. Cette notation est équivalente à une constante de chaîne de bits avec quatre chiffres binaires pour chaque chiffre hexadécimal.

Les deux formes de constantes de chaînes de bits peuvent être continuées sur plusieurs lignes de la même façon que les constantes de chaînes habituelles. Le guillemet dollar ne peut pas être utilisé dans une constante de chaîne de bits.

### 4.1.2.4. Constantes numériques

Les constantes numériques sont acceptées dans ces formes générales :

```
chiffres
chiffres.
[chiffres]
[e[+-]chiffres]
[chiffres].
chiffres[e[+-]chiffres]
chiffres
e[+-]chiffres
```

où *chiffres* est un ou plusieurs chiffres décimaux (de 0 à 9). Au moins un chiffre doit être avant ou après le point décimal, s'il est utilisé. Au moins un chiffre doit suivre l'indicateur d'exponentiel (`e`), s'il est présent. Il pourrait ne pas y avoir d'espaces ou d'autres caractères imbriqués dans la constante. Notez que tout signe plus ou moins en avant n'est pas forcément considéré comme faisant part de la constante ; il est un opérateur appliqué à la constante.

Il existe des exemples de constantes numériques valides :

42
3.5
4.
.001
5e2
1.925e–3

Une constante numérique contenant soit un point décimal soit un exposant est tout d'abord présumée du type `integer` si sa valeur est contenue dans le type `integer` (32 bits) ; sinon, il est présumé de type `bigint` si sa valeur entre dans un type `bigint` (64 bits) ; sinon, il est pris pour un type `numeric`. Les constantes contenant des poins décimaux et/ou des exposants sont toujours présumées de type `numeric`.

Le type de données affecté initialement à une constante numérique est seulement un point de départ pour les algorithmes de résolution de types. Dans la plupart des cas, la constante sera automatiquement convertie dans le type le plus approprié suivant le contexte. Si nécessaire, vous pouvez forcer l'interprétation d'une valeur numérique sur un type de données spécifiques en la convertissant. Par exemple, vous pouvez forcer une valeur numérique à être traitée comme un type `real` (`float4`) en écrivant

```
REAL '1.23'  -- style chaîne
1.23::REAL   -- style PostgreSQL (historique)
```

Ce sont en fait des cas spéciaux des notations de conversion générales discutés après.

#### 4.1.2.5. Constantes d'autres types

Une constante de type *arbitrary* peut être saisie en utilisant une des notations suivantes :

```
type 'chaîne'
'chaîne'::type
CAST ( 'chaîne' AS type )
```

Le texte de la chaîne constante est passé dans la routine de conversion pour le type appelé *type*. Le résultat est une constante du type indiqué. La conversion explicite de type pourrait être omise s'il n'y a pas d'ambiguïté sur le type de la constante (par exemple, lorsqu'elle est affectée directement à une colonne de la table), auquel cas elle est convertie automatiquement.

La constante chaîne peut être écrite en utilisant soit la notation SQL standard soit les guillemets dollar.

Il est aussi possible de spécifier une conversion de type en utilisant une syntaxe style fonction :

```
nom_type ( 'chaîne' )
```

mais tous les noms de type ne peuvent pas être utilisés ainsi ; voir la Section 4.2.8 pour plus de détails.

Les syntaxes `::`, `CAST()` et d'appels de fonctions peuvent aussi être utilisées pour spécifier les conversions de type à l'exécution d'expressions arbitraires, comme discuté dans la Section 4.2.8. Mais, la forme *type* `'`*chaîne*`'` peut seulement être utilisée pour spécifier le type d'une constante littérale. Une autre restriction sur *type* `'`*chaîne*`'` est qu'il ne fonctionne pas pour les types de tableau ; utilisez `::` ou `CAST()` pour spécifier le type d'une constante de type tableau.

### 4.1.3. Opérateurs

Un nom d'opérateur est une séquence d'au plus `NAMEDATALEN`−1 (63 par défaut) caractères provenant de la liste suivante :

+ − * / < > = ~ ! @ # % ^ & | ` ?

Néanmoins, il existe quelques restrictions sur les noms d'opérateurs :

- `--` et `/*` ne peuvent pas apparaître quelque part dans un nom d'opérateur car ils seront pris comme le début d'un commentaire.
- Un nom d'opérateur à plusieurs caractères ne peut pas finir avec + ou −, sauf si le nom contient aussi un de ces trois caractères :

  ~ ! @ # % ^ & | ` ?

  Par exemple, `@-` est un nom d'opérateur autorisé, mais `*-` ne l'est pas. Cette restriction permet à PostgreSQL d'analyser des requêtes compatibles avec SQL sans requérir des espaces entre les jetons.

Lors d'un travail avec des noms d'opérateurs ne faisant pas partie du standard SQL, vous aurez habituellement besoin de séparer les opérateurs adjacents avec des espaces pour éviter toute ambiguïté. Par exemple, si vous

avez défini un opérateur unaire gauche nommé `@`, vous ne pouvez pas écrire `X*@Y` ; vous devez écrire `X* @Y` pour vous assurer que PostgreSQL le lit comme deux noms d'opérateurs, et non pas comme un seul.

## 4.1.4. Caractères spéciaux

Quelques caractères non alphanumériques ont une signification spéciale, différente de celui d'un opérateur. Les détails sur leur utilisation sont disponibles à l'endroit où l'élément de syntaxe respectif est décrit. Cette section existe seulement pour avertir de leur existence et pour résumer le but de ces caractères.

- Un signe dollar (`$`) suivi de chiffres est utilisé pour représenter un paramètre de position dans le corps de la définition d'une fonction ou d'une instruction préparée. Dans d'autres contextes, le signe dollar pourrait faire partie d'un identifiant ou d'une constante chaîne utilisant le dollar comme guillemet.
- Les parenthèses (`()`) ont leur signification habituelle pour grouper leurs expressions et renforcer la précédence. Dans certains cas, les parenthèses sont requises car faisant partie de la syntaxe fixée d'une commande SQL particulière.
- Les crochets (`[]`) sont utilisés pour sélectionner les éléments d'un tableau. Voir la Section 8.10 pour plus d'informations sur les tableaux.
- Les virgules (`,`) sont utilisées dans quelques constructions syntaxiques pour séparer les éléments d'une liste.
- Le point–virgule (`;`) termine une commande SQL. Il ne peut pas apparaître quelque part dans une commande, sauf à l'intérieur d'une constante de type chaîne ou d'un identifiant entre guillemets.
- Le caractère deux points (`:`) est utilisé pour sélectionner des << morceaux >> de tableaux. (Voir la Section 8.10.) Dans certains dialectes SQL (tel que le SQL embarqué), il est utilisé pour préfixer les noms de variable.
- L'astérisque (`*`) est utilisé dans certains contextes pour indiquer tous les champs de la ligne d'une table ou d'une valeur composite. Elle a aussi une signification spéciale lorsqu'elle est utilisée comme argument de la fonction d'agrégat `COUNT`.
- Le point (`.`) est utilisé dans les constantes numériques et pour séparer les noms de schéma, table et colonne.

## 4.1.5. Commentaires

Un commentaire est une séquence arbitraire de caractères commençant avec deux tirets et s'étendant jusqu'à la fin de la ligne, par exemple :

```
-- Ceci est un commentaire standard en SQL
```

Autrement, les blocs de commentaires style C peuvent être utilisés :

```
/* commentaires multilignes
 * et imbriqués: /* bloc de commentaire imbriqué */
 */
```

où le commentaire commence avec `/*` et s'étend jusqu'à l'occurrence de `*/`. Ces blocs de commentaires s'imbriquent, comme spécifié dans le standard SQL mais pas comme dans le langage C. De ce fait, vous pouvez commenter des blocs importants de code pouvant contenir des blocs de commentaires déjà existants.

Un commentaire est supprimé du flux en entrée avant une analyse plus poussée de la syntaxe et est remplacé par un espace blanc.

## 4.1.6. Précédence lexicale

Le Tableau 4−1 affiche la précédence et l'associativité des opérateurs dans PostgreSQL. La plupart des opérateurs ont la même précédence et sont associatifs par la gauche. La précédence et l'associativité des opérateurs sont codées en dur dans l'analyseur. Ceci pourrait conduire à un comportement non intuitif ; par exemple, les opérateurs booléens < et > ont une précédence différente des opérateurs booléens <= et >=. De même, vous aurez quelque fois besoin d'ajouter des parenthèses lors de l'utilisation de combinaisons d'opérateurs binaires et unaires. Par exemple :

```
SELECT 5 ! - 6;
```

sera analysé comme

```
SELECT 5 ! (- 6);
```

parce que l'analyseur n'a aucune idée, jusqu'à ce qu'il soit trop tard que ! est défini comme un opérateur suffixe, et non pas préfixe. Pour obtenir le comportement désiré dans ce cas, vous devez écrire :

```
SELECT (5 !) - 6;
```

C'est le prix à payer pour l'extensibilité.

**Tableau 4−1. Précédence des opérateurs (en ordre décroissant)**

| Opérateur/Élément | Associativité | Description |
|---|---|---|
| . | gauche | séparateur de noms de table et de colonne |
| :: | gauche | conversion de type, style PostgreSQL |
| [ ] | gauche | sélection d'un élément d'un tableau |
| − | droite | négation unaire |
| ^ | gauche | exponentiel |
| * / % | gauche | multiplication, division, modulo |
| + − | gauche | addition, soustraction |
| IS | | IS TRUE, IS FALSE, IS UNKNOWN, IS NULL |
| ISNULL | | test pour NULL |
| NOTNULL | | test pour non NULL |
| (autres) | gauche | tout autre opérateur natif et défini par l'utilisateur |
| IN | | appartenance à un ensemble |
| BETWEEN | | compris entre |
| OVERLAPS | | surcharge un intervalle de temps |
| LIKE ILIKE SIMILAR | | correspondance de modèles de chaînes |
| < > | | inférieur, supérieur à |
| = | droite | égalité, affectation |
| NOT | droite | négation logique |

| | | |
|---|---|---|
| AND | gauche | conjonction logique |
| OR | gauche | disjonction logique |

Notez que les règles de précédence des opérateurs s'appliquent aussi aux opérateurs définis par l'utilisateur qui ont le même nom que les opérateurs internes mentionnés ici. Par exemple, si vous définissez un opérateur << + >> pour un type de données personnalisé, il aura la même précédence que l'opérateur interne << + >>, peu importe ce que fait le votre.

Lorsqu'un nom d'opérateur qualifié par un schéma est utilisé dans la syntaxe OPERATOR, comme par exemple dans

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

la construction OPERATOR est prise pour avoir la précédence par défaut affichée dans le Tableau 4–1 pour les opérateurs << autres >>. Ceci est vrai quelque soit le nom spécifique de l'opérateur apparaissant à l'intérieur de OPERATOR().

# 4.2. Expressions de valeurs

Les expressions de valeurs sont utilisées dans une grande variété de contextes, tels que dans la liste cible d'une commande SELECT, dans les nouvelles valeurs de colonnes d'une commande INSERT ou UPDATE, ou dans les conditions de recherche d'un certain nombre de commandes. Le résultat d'une expression de valeurs est quelque fois appelé *scalaire*, pour le distinguer du résultat d'une expression de table (qui est une table). Les expressions de valeurs sont aussi appelées des *expressions scalaires* (voire même simplement des *expressions*). La syntaxe d'expression permet le calcul des valeurs à partir de morceaux primitifs en utilisant les opérations arithmétiques, logiques, d'ensemble et autres.

Une expression de valeur fait partie des suivantes :

- Une constante ou une valeur littérale.
- Une référence de colonne.
- Une référence de paramètre de position, dans le corps d'une définition de fonction ou d'instruction préparée.
- Une expression sous–scriptée.
- Une expression de sélection de champs.
- Un appel d'opérateur.
- Un appel de fonction.
- Une expression d'agrégat.
- Une conversion de type.
- Une sous–requête scalaire.
- Un constructeur de tableau.
- Un constructeur de ligne.
- Toute expression de tableau entre parenthèses, utile pour grouper des sous–expressions et surcharger la précédence.

En plus de cette liste, il existe un certain nombre de constructions pouvant être classées comme une expression mais ne suivant aucune règle de syntaxe générale. Elles ont généralement la sémantique d'une fonction ou d'un opérateur et sont expliquées à l'emplacement approprié dans le Chapitre 9. Un exemple est la clause IS NULL.

Nous avons déjà discuté des constantes dans la Section 4.1.2. Les sections suivantes discutent des options restantes.

### 4.2.1. Références de colonnes

Une colonne peut être référencée avec la forme

*correlation.nom_colonne*

*correlation* est le nom d'une table (parfois qualifié par son nom de schéma) ou un alias d'une table définie au moyen de la clause FROM ou un des mots clés NEW ou OLD. (NEW et OLD peuvent seulement apparaître dans les règles de réécriture alors que les autres noms de corrélation peuvent être utilisés dans toute instruction SQL.) Le nom de corrélation et le point de séparation peuvent être omis si le nom de colonne est unique dans les tables utilisées par la requête courante. (Voir aussi Chapitre 7.)

### 4.2.2. Paramètres de position

Un paramètre de position est utilisé pour indiquer une valeur fournie en externe par une instruction SQL. Les paramètres sont utilisés dans des définitions de fonction SQL et dans les requêtes préparées. Quelques bibliothèques clients supportent aussi la spécification de valeurs de données séparément de la chaîne de commande SQL, auquel cas les paramètres sont utilisés pour référencer les valeurs de données en dehors. Le format d'une référence de paramètre est :

*$numéro*

Par exemple, considérez la définition d'une fonction dept comme

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE nom = $1 $$
    LANGUAGE SQL;
```

Ici, $1 sera remplacé par le premier argument de fonction lorsque la commande sera appelée.

### 4.2.3. Indices

Si une expression récupère une valeur de type tableau, alors un élément spécifique du tableau peut être extrait en écrivant

*expression*[*indice*]

ou des éléments adjacents (un << morceau de tableau >>) peuvent être extrait en écrivant

*expression*[*indice_bas*:
*indice_haut*]

(Ici, les crochets [ ] doivent apparaître littéralement.) Chaque *indice* est lui–même une expression, qui doit renvoyer une valeur entière.

En général, l'*expression* de type tableau doit être entre parenthèses mais celles−ci peuvent être omises lorsque l'expression à indicer est seulement une référence de colonne ou une position de paramètre. De plus, les indices multiples peuvent être concaténés lorsque le tableau original est multi−dimensionnel. Par exemple,

```
matable.colonnetableau[4]
matable.colonnes_deux_d[17][34]
$1[10:42]
(fonctiontableau(a,b))[42]
```

Les parenthèses dans ce dernier exemple sont requises. Voir la Section 8.10 pour plus d'informations sur les tableaux.

## 4.2.4. Sélection de champs

Si une expression récupère une valeur de type composé (type row), alors un champ spécifique de la ligne est extrait en écrivant

*expression*.*nom_champ*

En général, l'*expression* de ligne doit être entre parenthèses mais les parenthèses peuvent être omises lorsque l'expression à partir de laquelle se fait la sélection est seulement une référence de table ou un paramètre de position. Par exemple,

```
matable.macolonne
$1.unecolonne
(fonctionligne(a,b)).col3
```

(Donc, une référence de colonne qualifiée est réellement un cas spécial de syntaxe de sélection de champ.)

## 4.2.5. Appels d'opérateurs

Il existe trois syntaxes possibles pour l'appel d'un opérateur :

*expression* *opérateur* *expression* (opérateur binaire préfixe)

*opérateur* *expression* (opérateur unaire préfixe)

*expression* *opérateur* (opérateur unaire suffixe)

où le jeton *opérateur* suit les règles de syntaxe de la Section 4.1.3, ou est un des mots clés `AND`, `OR` et `NOT`, ou est un nom d'opérateur qualifié de la forme

`OPERATOR(`*schema*`.`*nom_operateur*`)`

Quel opérateur particulier existe et est−il unaire ou binaire dépend des opérateurs définis par le système ou l'utilisateur. Le Chapitre 9 décrit les opérateurs internes.

## 4.2.6. Appels de fonctions

La syntaxe pour un appel de fonction est le nom d'une fonction (qualifié ou non du nom du schéma) suivi par sa liste d'arguments entre parenthèses :

```
fonction
([expression [,
expression ... ]] )
```

Par exemple, ce qui suit calcule la racine carré de 2 :

```
sqrt(2)
```

La liste des fonctions intégrées est dans le Chapitre 9. D'autres fonctions pourraient être ajoutées par l'utilisateur.

---

## 4.2.7. Expressions d'agrégat

Une *expression d'agrégat* représente l'application d'une fonction d'agrégat à travers les lignes sélectionnées par une requête. Une fonction d'agrégat réduit les nombres entrés en une seule valeur de sortie, comme la somme ou la moyenne des valeurs en entrée. La syntaxe d'une expression d'agrégat est une des suivantes :

```
nom_agregat (expression)
nom_agregat (ALL
expression)
nom_agregat (DISTINCT
expression)
nom_agregat ( * )
```

où *nom_agregat* est un agrégat précédemment défini (parfois qualifié d'un nom de schéma) et *expression* est toute expression de valeur qui ne contient pas lui−même une expression d'agrégat.

La première forme d'expression d'agrégat appelle l'agrégat pour toutes les lignes en entrée pour lesquelles l'expression donnée ne trouve pas une valeur NULL. (En fait, c'est à la fonction d'agrégat de savoir si elle doit ignorer ou non les valeurs NULL... mais toutes les fonctions standards le font.) La seconde forme est identique à la première car `ALL` est par défaut. La troisième forme implique l'agrégat pour toutes les valeurs de l'expression non NULL et distinctes trouvées dans les lignes en entrée. La dernière forme appelle l'agrégat une fois pour chaque ligne en entrée qu'elle soit NULL ou non ; comme aucune valeur particulière en entrée n'est spécifiée, c'est généralement utile pour la fonction d'agrégat `count()`.

Par exemple, `count(*)` trouve le nombre total de lignes en entrée ; `count(f1)` récupère le nombre de lignes en entrée pour lesquelles `f1` n'est pas NULL ; `count(distinct f1)` retrouve le nombre de valeurs distinctes non NULL de `f1`.

Les fonctions d'agrégat prédéfinies sont décrites dans la Section 9.15. D'autres fonctions d'agrégat pourraient être ajoutées par l'utilisateur.

Une expression d'agrégat pourrait apparaître dans la liste de résultat ou dans la clause `HAVING` d'une commande `SELECT`. Elle est interdite dans d'autres clauses, telles que `WHERE`, parce que ces clauses sont logiquement évaluées avant que les résultats des agrégats soient formés.

Lorsqu'une expression d'agrégat apparaît dans une sous−requête (voir la Section 4.2.9 et la Section 9.16), l'agrégat est normalement évalué sur les lignes de la sous−requête. Mais, une exception arrive si l'argument de l'agrégat contient seulement des niveaux externes de variables : ensuite, l'agrégat appartient au niveau externe le plus proche et est évalué sur les lignes de cette requête. L'expression de l'agrégat en un tout est une référence externe pour la sous−requête dans laquelle il apparaît et agit comme une constante sur toute évaluation de cette requête. La restriction apparaissant seulement dans la liste de résultat ou dans la clause `HAVING` s'applique avec respect du niveau de requête auquel appartient l'agrégat.

## 4.2.8. Conversions de type

Une conversion de type spécifie une conversion à partir d'un type de données en un autre. PostgreSQL accepte deux syntaxes équivalentes pour les conversions de type :

```
CAST ( expression AS type )
expression::type
```

La syntaxe `CAST` est conforme à SQL ; la syntaxe avec **::** est historique dans PostgreSQL usage.

Lorsqu'une conversion est appliquée à une expression de valeur pour un type connu, il représente une conversion de type à l'exécution. Cette conversion réussira seulement si une opération convenable de conversion de type a été définie. Notez que ceci est subtilement différent de l'utilisation de conversion avec des constantes, comme indiqué dans la Section 4.1.2.5. Une conversion appliquée à une chaîne littérale représente l'affectation initiale d'un type pour une valeur constante littérale, et donc cela réussira pour tout type (si le contenu de la chaîne littérale est une syntaxe acceptée en entrée pour le type de donnée).

Une conversion de type explicite pourrait être habituellement omise s'il n'y a pas d'ambiguïté sur le type qu'une expression de valeur pourrait produire (par exemple, lorsqu'elle est affectée à une colonne de table) ; le système appliquera automatiquement une conversion de type dans de tels cas. Néanmoins, la conversion automatique est réalisée seulement pour les conversions marquées << OK pour application implicite >> dans les catalogues système. D'autres conversions peuvent être appelées avec la syntaxe de conversion explicite. Cette restriction a pour but d'empêcher l'application silencieuse de conversions surprenantes.

Il est aussi possible de spécifier une conversion de type en utilisant une syntaxe de type fonction :

```
nom_type ( expression )
```

Néanmoins, ceci fonctionne seulement pour les types dont les noms sont aussi valides en tant que noms de fonctions. Par exemple, `double precision` ne peut pas être utilisé de cette façon mais son équivalent `float8` le peut. De même, les noms `interval`, `time` et `timestamp` peuvent seulement être utilisés de cette façon s'ils sont entre des guillemets doubles à cause des conflits de syntaxe. Du coup, l'utilisation de la syntaxe de conversion du style fonction amène à des inconsistances et devrait probablement être évitée dans les nouvelles applications. (La syntaxe style fonction est en fait seulement un appel de fonction. Quand un des deux standards de syntaxe de conversion est utilisé pour faire une conversion à l'exécution, elle appellera en interne une fonction enregistrée pour réaliser la conversion. Par convention, ces fonctions de conversion ont le même nom que leur type de sortie et, du coup, la << syntaxe style fonction >> n'est rien de plus qu'un appel direct à la fonction de conversion sous−jacente. Évidemment, une application portable ne devrait pas s'y fier.)

## 4.2.9. Sous−requêtes scalaires

Une sous−requête scalaire est une requête `SELECT` ordinaire entre parenthèses renvoyant exactement une ligne avec une colonne. (Voir le Chapitre 7 pour plus d'informations sur l'écriture des requêtes.) La requête `SELECT` est exécutée et la seule valeur renvoyée est utilisée dans l'expression de valeur englobante. C'est une erreur d'utiliser une requête qui renvoie plus d'une ligne ou plus d'une colonne comme requête scalaire. (Mais si, lors d'une exécution particulière, la sous−requête ne renvoie pas de lignes, alors il n'y a pas d'erreur ; le résultat scalaire est supposé NULL.) La sous−requête peut référencer des variables de la requête englobante, qui agiront comme des constantes durant toute évaluation de la sous−requête. Voir aussi Section 9.16 pour d'autres expressions impliquant des sous−requêtes.

Par exemple, ce qui suit trouve la ville disposant de la population la plus importante dans chaque état :

```
SELECT nom, (SELECT max(pop) FROM villes WHERE villes.etat = etat.nom)
    FROM etats;
```

## 4.2.10. Constructeurs de tableaux

Un constructeur de tableau est une expression qui construit une valeur de tableau à partir de valeurs de ses membres. Un constructeur de tableau simple utilise le mot clé `ARRAY`, un crochet ouvrant `[`, une ou plusieurs expressions (séparées par des virgules) pour les valeurs des éléments du tableau et finalement un crochet fermant `]`. Par exemple :

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

Le type d'élément du tableau est le type commun des expressions des membres, déterminé en utilisant les mêmes règles que pour les constructions `UNION` ou `CASE` (voir Section 10.5).

Les valeurs de tableaux multidimensionnels peuvent être construits par des constructeurs de tableaux imbriqués. Pour les constructeurs internes, le mot clé `ARRAY` pourrait être omis. Par exemple, ces expressions produisent le même résultat :

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

Comme les tableaux multidimensionnels doivent être rectangulaires, les constructeurs internes du même niveau doivent produire des sous−tableaux de dimensions identiques.

Les éléments d'un constructeur de tableau multidimensionnel peuvent être tout ce qui récupère un tableau du bon type, pas seulement une construction de sous−ARRAY. Par exemple :

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
------------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

Il est aussi possible de construire un tableau à partir des résultats d'une sous−requête. Avec cette forme, le constructeur de tableau est écrit avec le mot clé ARRAY suivi par une sous−requête entre parenthèses (et non pas des crochets). Par exemple :

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                           ?column?
------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31}
(1 row)
```

La sous−requête doit renvoyer une seule colonne. Le tableau à une dimension résultant aura un élément pour chaque ligne dans le résultat de la sous−requête, avec un type élément correspondant à celui de la colonne en sortie de la sous−requête.

Les indices d'une valeur de tableau construit avec ARRAY commencent toujours à un. Pour plus d'informations sur les tableaux, voir la Section 8.10.

## 4.2.11. Constructeurs de lignes

Un constructeur de ligne est une expression qui construit une valeur de ligne (aussi appelée une valeur composite) à partir des valeurs de ses membres. Un constructeur de ligne consiste en un mot clé ROW, une parenthèse gauche, zéro ou plus d'une expression (séparées par des virgules) pour les valeurs des champs de la ligne, et finalement une parenthèse droite. Par exemple,

```
SELECT ROW(1,2.5,'ceci est un test');
```

Le mot clé ROW est optionnel lorsqu'il y a plus d'une expression dans la liste.

Par défaut, la valeur créée par une expression ROW est d'un type d'enregistrement anonyme. Si nécessaire, il peut être converti en un type composite nommé — soit le type de ligne d'une table soit un type composite créé avec CREATE TYPE AS. Une conversion explicite pourrait être nécessaire pour éviter une ambiguïté. Par exemple :

```
CREATE TABLE matable(f1 int, f2 float, f3 text);

CREATE FUNCTION recup_f1(matable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Aucune conversion nécessaire parce que seul un recup_f1() existe
SELECT recup_f1(ROW(1,2.5,'ceci est un test'));
recup_f1
```

```
--------
1
(1 row)

CREATE TYPE montypeligne AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION recup_f1(montypeligne) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Maintenant, nous avons besoin d'une conversion pour indiquer la fonction à appeler:
SELECT recup_f1(ROW(1,2.5,'ceci est un test'));
ERROR:  function recup_f1(record) is not unique

SELECT recup_f1(ROW(1,2.5,'ceci est un test')::matable);
getf1
-------
1
(1 row)

SELECT recup_f1(CAST(ROW(11,'ceci est un test',2.5) AS montypeligne));
getf1
-------
11
(1 row)
```

Les constructeurs de lignes peuvent être utilisés pour construire des valeurs composites à stocker dans une colonne de table de type composite ou pour être passé à un fonction qui accepte un paramètre composite. De plus, il est possible de comparer deux valeurs de lignes ou pour tester une ligne avec `IS NULL` ou `IS NOT NULL`, par exemple

```
SELECT ROW(1,2.5,'ceci est un test') = ROW(1, 3, 'pas le même');

SELECT ROW(a, b, c) IS NOT NULL FROM table;
```

Pour plus de détails, voir Section 9.17. Les constructeurs de lignes peuvent aussi être utilisés en connexion avec des sous−requêtes, comme discuté dans Section 9.16.

## 4.2.12. Règles d'évaluation des expressions

L'ordre d'évaluation des sous−expressions n'est pas défini. En particulier, les entrées d'un opérateur ou d'une fonction ne sont pas nécessairement évaluées de la gauche vers la droite ou dans un autre ordre fixé.

De plus, si le résultat d'une expression peut être déterminé par l'évaluation de quelques parties de celui−ci, alors d'autres sous−expressions devraient ne pas être évaluées du tout. Par exemple, si vous écrivez :

```
SELECT true OR une_fonction();
```

alors `une_fonction()` pourrait (probablement) ne pas être appelée du tout. Pareil dans le cas suivant :

```
SELECT somefunc() OR true;
```

Notez que ceci n'est pas identique au << court−circuitage >> de gauche à droite des opérateurs booléens utilisé par certains langages de programmation.

En conséquence, il est déconseillé d'utiliser des fonctions ayant des effets de bord dans une partie des

expressions complexes. Il est particulièrement dangereux de se fier aux effets de bord ou à l'ordre d'évaluation dans les clauses `WHERE` et `HAVING`, car ces clauses sont reproduites de nombreuses fois lors du développement du plan d'exécution. Les expressions booléennes (combinaisons `AND`/`OR`/`NOT`) dans ces clauses pourraient être réorganisées de toute autre façon que celles autorisées dans l'algèbre booléenne.

Quand il est essentiel de forcer l'ordre d'évaluation, une construction `CASE` (voir Section 9.13) pourrait être utilisée. Par exemple, c'est une façon, non sûre, d'essayer d'éviter une division par zéro dans une clause `WHERE` :

```
SELECT ... WHERE x <> 0 AND y/x > 1.5;
```

Mais ceci est sûr :

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

Une construction `CASE` utilisée de cette façon déjouera les tentatives d'optimisation, donc cela ne sera fait que si nécessaire. (Dans cet exemple particulier, il serait sans doute mieux de contourner le problème en écrivant `y > 1.5*x`.)

# Chapitre 5. Définition des données

Ce chapitre couvre la création de structures de données qui contiendront les données de quelqu'un. Dans une base relationnelle, les données brutes sont stockées dans des tables. Du coup, la plus grande partie de ce chapitre sera consacrée à l'explication de la création et de la modification des tables et des fonctions disponibles pour contrôler les données stockées dans les tables. Ensuite, nous discuterons de l'organisation de tables en schémas, et comment des droits peuvent être attribués aux tables. Enfin, nous verrons brièvement d'autres fonctionnalités, tel que les vues, les fonctions et les déclencheurs.

## 5.1. Bases sur les tables

Une table dans une base relationnelle ressemble beaucoup à un tableau sur papier : elle est constituée de rangées et de colonnes. Le nombre et l'ordre des colonnes sont fixés et chaque colonne a un nom. Le nombre de rangées est variable −− il représente la quantité de données stockées à un moment donné. SQL n'apporte aucune garantie sur l'ordre des rangées dans une table. Quand une table est lue, les rangées apparaîtront dans un ordre aléatoire sauf si un tri est demandé explicitement. Ceci est couvert dans Chapitre 7. De plus, SQL n'attribue pas d'identifiant unique aux rangées. Du coup, il est possible d'avoir plusieurs rangées complètement identiques dans une table. Ceci est une conséquence du modèle mathématique sur lequel repose SQL mais n'est habituellement pas désiré. Plus tard dans ce chapitre, nous verrons comment traiter ce problème.

Chaque colonne a un type de donnée. Ce type de donnée restreint la série de valeurs possibles qui peuvent être attribuées à une colonne et attribue des sémantiques à la donnée stockée dans la colonne pour qu'elles puissent être utilisées pour des calculs. Par exemple, une colonne déclarée comme étant d'un type numérique n'acceptera pas une chaîne arbitraire de texte, et les données stockées dans une telle table peuvent être utilisées dans des calculs mathématiques. Par opposition, une colonne déclarée comme étant de type chaîne de caractères acceptera pratiquement n'importe quel type de donnée mais ne se prêtera pas à des calculs mathématiques bien que d'autres opérations tel que la concaténation des chaînes sont disponibles.

PostgreSQL inclut une série conséquente de types de données intégrés qui correspondent à plusieurs applications. Les utilisateurs peuvent aussi définir leurs propres types de données. La plupart des types de données intégrés ont des noms et des sémantiques évidents alors nous reportons une explication détaillée à Chapitre 8. Quelques−uns des types les plus utilisés sont `integer` pour les entiers, `numeric` pour les nombres pouvant être fractionnels, `text` pour les chaînes de caractères, `date` pour les dates, `time` pour les valeurs de type heure et `timestamp` pour les valeurs contenant et une date et une heure.

Pour créer une table, il faut utiliser la commande bien nommée `CREATE TABLE`. Dans cette commande, vous devez spécifier au moins le nom de la nouvelle table, les noms des colonnes et le type de données pour chacune des colonnes. Par exemple :

```
CREATE TABLE ma_premiere_table (
    premiere_colonne text,
    deuxieme_colonne integer
);
```

Ceci crée une table nommée `ma_premiere_table` avec deux colonnes. La première colonne est nommée `premiere_colonne` et a un type de données `text` ; la seconde colonne porte le nom `deuxieme_colonne` et le type `integer`. Les noms de table et colonnes suivent la syntaxe d'identification expliquée dans Section 4.1.1. Les noms des types sont souvent aussi des identifiants mais il y a des

exceptions. Notez que la liste des colonnes est séparée par des virgules et entourée par des parenthèses.

Bien sur, l'exemple précédant est un peu tiré par les cheveux. Normalement, on donne aux tables et aux colonnes des noms indiquant quels types de données ils stockent. Alors voyons un exemple plus réaliste :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric
);
```

(Le type `numeric` peut stocker des composants fractionnels comme on pourrait s'y attendre de montants monétaires.)

> **Astuce :** Quand vous créez des tables liées entre elles, il est prudent de choisir des règles de nommage pour les tables et les colonnes. Par exemple, il peut y avoir le choix d'utiliser des noms au pluriel ou au singulier pour les noms de table, chaque choix ayant les faveurs d'un théoricien ou d'un autre.

Il y a une limite sur le nombre de colonnes qu'une table peut contenir. Suivant le type de colonne, elle peut être entre 250 et 1600. Par contre, définir une table avec un nombre de colonnes proche de ceux−ci est très inhabituel et est souvent la preuve d'une conception douteuse.

Si vous n'avez plus besoin d'une table, vous pouvez la retirer en utilisant la commande `DROP TABLE`. Par exemple :

```
DROP TABLE ma_premiere_table;
DROP TABLE produits;
```

Tenter de supprimer une table qui n'existe pas est une erreur. Malgré cela, il est habituel dans des fichiers de scripts SQL d'essayer de supprimer chaque table avant de la créer, tout en ignorant les messages d'erreur.

Si vous avez besoin de modifier une table qui existe déjà, regardez Section 5.6 plus loin dans ce chapitre.

Avec les outils dont nous avons déjà discuté, vous pouvez créer des tables fonctionnelles. Le reste de ce chapitre est consacré à l'ajout de fonctionnalités, à la définition de tables pour garantir l'intégrité des données, la sécurité ou la facilité. Si vous êtes impatients de remplir vos tables avec des données tout de suite, vous pouvez sauter au Chapitre 6 et lire le reste de ce chapitre plus tard.

## 5.2. Valeurs par défaut

On peut attribuer une valeur par défaut à une colonne. Quant une nouvelle rangée est créée et aucune valeur n'est spécifiée pour certaines de ses colonnes, celles−ci sont remplies avec leur valeur par défaut respective. Une commande de manipulation de données peut aussi demander explicitement qu'une colonne soit mise à sa valeur par défaut sans avoir à connaître la valeur en question. (Les détails sur les commandes de manipulation de données sont dans Chapitre 6.)

Si aucune valeur par défaut n'est déclarée explicitement, la valeur par défaut est la valeur NULL. Ceci est d'habitude cohérent car on peut considérer que la valeur NULL représente des données inconnues.

Dans une définition de table, les valeurs par défaut sont listées après le type de donnée de la colonne. Par exemple:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

La valeur par défaut peut être une expression, qui sera évaluée à l'insertion de la valeur par défaut (*pas* à la création de la table.) Un exemple habituel est qu'une colonne de type timestamp pourrait avoir une valeur par défaut `now()`, de façon à ce qu'il obtienne la valeur de l'heure au moment de l'insertion. Un autre exemple habituel est la génération d'un << numéro de série >> pour chaque ligne. Dans PostgreSQL, ceci se fait habituellement par quelque chose comme

```
CREATE TABLE products (
product_no DEFAULT nextval('products_product_no_seq'),
...
);
```

où la fonction `nextval()` fournit des valeurs successives à partir d'une *objet séquence* (voir Section 9.12). Cet arrangement est suffisamment commun pour qu'il y ait un raccourci pour lui :

```
CREATE TABLE products (
pro SERIAL,
...
);
```

Le raccourci `SERIAL` est discuté plus tard dans Section 8.1.4.

# 5.3. Contraintes

Les types de données sont un moyen de limiter ce qui peut être stocké dans une table. Pour beaucoup d'applications, par contre, la contrainte qu'elles appliquent sont trop fortes. Par exemple, une colonne qui contient le prix d'un produit ne devrait accepter que des valeurs positives. Mais il n'y a pas de type de données qui n'acceptent que des valeurs positives. Un autre problème est le fait de vouloir limiter les données d'une colonne par rapport à d'autres colonnes ou rangées. Par exemple, dans une table contenant des informations de produit, il ne devrait y avoir qu'une rangée pour chaque numéro de produit.

Dans ce but, SQL vous permet de définir les contraintes sur les colonnes et les tables. Les contraintes vous donnent autant de contrôle sur les données de vos tables que vous désirez. Si un utilisateur tente de stocker des données dans une colonne qui violerait un contrainte, une erreur est soulevée. Ceci s'applique même si la valeur vient de la définition de la valeur par défaut.

## 5.3.1. Contraintes de Vérification

Une contrainte de vérification est le type de contrainte le plus générique qui soit. Elle vous permet de spécifier l'expression d'une certaine colonne doit satisfaire une expression booléenne. Par exemple, pour obliger des prix de produits positifs, on pourrait utiliser :

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

Comme vous pouvez le voir, la définition de contrainte vient après le type de données comme les définitions de valeur par défaut. Les valeurs par défaut et les contraintes peuvent être données dans n'importe quel ordre. Une contrainte de vérification s'utilise avec le mot clé `CHECK` suivi d'une expression entre parenthèses. L'expression de contrainte de vérification peut impliquer la colonne ainsi contrainte, sinon la contrainte n'aurait pas beaucoup de sens.

Vous pouvez aussi donner à la contrainte un nom diffèrent. Ceci clarifie les messages d'erreur et vous permet de faire référence à la contrainte lorsque vous avez besoin de la modifier. La syntaxe est:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

Alors, pour spécifier une contrainte nommée, utilisez le mot–clé `CONSTRAINT` suivi d'un identifiant et de la définition de contrainte. (Si vous ne donnez pas de nom à la contrainte, le système choisira un nom pour vous.)

Une contrainte de vérification peut faire référence à plusieurs colonnes. Admettons que vous souhaitez stocker un prix normal et un prix de promotion et, être sur que le prix de promotion soit inférieur au prix normal.

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

Les deux premières contraintes devrait vous être familières. La troisième utilise une nouvelle syntaxe. Elle n'est pas attachée à une colonne particulière, elle apparaît comme un élément distinct dans la liste de colonnes séparées par des virgules. Les définitions de colonnes et ces définitions de contraintes peut être définies dans un ordre quelconque.

On dit que les deux premières contraintes sont des contraintes de colonnes tandis que la troisième est une contrainte de table parce qu'elle est écrite séparément de toute définition de colonne tandis que l'inverse n'est pas forcément possible car une contrainte de colonne est supposé faire uniquement référence à la colonne à laquelle elle est attachée. (PostgreSQL ne force pas cette règle mais vous devriez la suivre si vous voulez que les définitions de votre table fonctionnent avec d'autres systèmes de bases de données.) L'exemple ci–dessus aurait pu s'écrire :

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
```

```
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

ou même

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

C'est une question de goût.

Des noms peuvent être affectés à des contraintes de table de la même façon que les contraintes de colonne :

```
CREATE TABLE products (
product_no integer,
name text,
price numeric,
CHECK (price > 0),
discounted_price numeric,
CHECK (discounted_price > 0),
CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

Il faut noter qu'une contrainte de vérification est satisfaite si l'expression est évaluée à vrai ou la valeur NULL. Puisque la plupart des expressions seront évaluées à la valeur NULL si l'un des opérandes est NULL, elles n'empêchent pas les valeurs NULL dans les colonnes contraintes. Pour s'assurer qu'une colonne ne contient pas de valeurs NULL, la contrainte non−NULL décrite dans la section suivante peut être utilisée.

## 5.3.2. Contraintes Non NULL

Une contrainte non NULL dit simplement qu'une colonne ne peut pas prendre la valeur NULL. Un exemple de syntaxe:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

Une contrainte non NULL est toujours écrite comme une contrainte de colonne. Une contrainte non NULL est l'équivalente fonctionnelle de créer une contrainte `CHECK (`*`nom_colonne`*` IS NOT NULL)`, mais dans PostgreSQL, créer une contrainte explicitement non NULL est plus efficace. L'inconvénient est que vous ne pouvez pas donner de noms explicites à des contraintes non NULL créées de cette manière.

Bien sur, une colonne peut avoir plus d'une contrainte. Écrivez juste les contraintes les unes après les autres:

```
CREATE TABLE products (
```

```
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

L'ordre n'importe pas. Il ne détermine pas dans quel ordre les contraintes seront vérifiées.

La contrainte `NOT NULL` a un opposé; la contrainte `NULL` . Ceci ne veut pas dire que la colonne doit être NULL, ce qui serait inutile. À la place, ceci sélectionne le comportement par défaut que la colonne doit être NULL. La contrainte `NULL` n'est pas définie dans le standard SQL et ne devrait pas être utilisé dans des applications portables. (Elle n'a été ajoutée dans PostgreSQL que pour assurer la compatibilité avec d'autres bases de données.) Certains utilisateurs l'apprécient car elle facilite le fait d'activer une contrainte dans un fichier de script. Par exemple, vous pourriez commencer avec:

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

et puis insérer le mot−clé `NOT` suivant vos besoins.

> **Astuce :** Dans beaucoup de conceptions de bases de données, la majorité des colonnes devraient être marquées non NULL.

## 5.3.3. Contraintes Uniques

Les contraintes uniques garantissent que les données contenues dans la colonne ou un groupe de colonnes est unique par rapport à toutes les rangées dans la table. La syntaxe est:

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

est écrit comme contrainte de colonne et

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

est écrit comme contrainte de table.

Si une contrainte unique fait référence à un groupe de colonnes, celles−ci sont listées séparées par des virgules:

```
CREATE TABLE exemple (
    a integer,
    b integer,
    c integer,
```

```
    UNIQUE (a, c)
);
```

Ceci spécifie que la combinaison de valeurs dans les colonnes indiquées est unique pour toute la table bien qu'une seule des colonnes puisse ne pas être (et habituellement n'est pas) unique.

Vous pouvez affecter votre propre nom pour une contrainte unique, de la façon habituelle :

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

En général, une contrainte unique est violée lorsqu'il y a au moins deux rangées dans une table ou la valeur de toutes les colonnes inclus dans la contrainte sont égales. Par contre, les valeurs NULL ne sont pas assimilées à une égalités dans cette comparaison. Ceci veut dire qu'il est possible de stocker un nombre illimité de rangées qui contiennent une valeur NULL dans au moins l'une des colonnes contraintes. Ce comportement est conforme au standard SQL mais nous avons été informé que d'autres bases SQL ne suivent pas cette règle. Alors, soyez prudents en développant des applications qui sont prévues pour être portable.

## 5.3.4. Clés Primaires

Techniquement, une contrainte de clé primaire est tout simplement une combinaison d'une contrainte unique et d'une contrainte non NULL. Donc, les définitions de tables suivantes accepteront les mêmes données:

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);

CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Les clés primaires peuvent contraindre sur plus d'une colonne; la syntaxe est semblable aux contraintes uniques:

```
CREATE TABLE exemple (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

Une clé primaire indique qu'une colonne ou un groupe de colonnes peuvent être utilisés comme identifiant unique pour les rangées de la table. (Ceci est une conséquence directe de la définition d'une clé primaire. Notez qu'une contrainte unique ne donne pas par elle−même, un identifiant unique car elle n'exclut pas les valeurs NULL.) Ceci est pratique à la fois pour des raisons de documentation et pour les applications clientes.

Par exemple, une application graphique qui permet de modifier les valeurs de rangées a probablement besoin de connaître la clé primaire d'une table pour pouvoir identifier les rangées de manière unique correctement.

Une table peut avoir au mieux une clé primaire (tandis qu'elle peut avoir plusieurs contraintes uniques et non NULL). La théorie des bases de données relationnelles dit que chaque table doit avoir une clé primaire. Cette règle n'est pas appliquée par PostgreSQL, mais il vaut mieux la respecter autant que possible.

## 5.3.5. Clés Étrangères

Une contrainte de clé étrangère stipule que les valeurs dans cette colonne (ou un groupe de colonnes) doit correspondre aux valeurs apparaissant dans des rangées d'une autre table. Nous disons que ceci maintient l'*intégrité référentielle* entre deux tables liées.

Disons que vous avez la table de produits que nous avons déjà utilisé plusieurs fois:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Disons aussi que vous avez une table stockant les commandes de ces produits. Nous voulons aussi nous assurer que la table des commandes ne contienne que des commandes concernant des produits qui existent réellement. Alors, nous définissons une contrainte de clé étrangère dans la table des commandes qui référence la table produit:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

Maintenant, il est impossible de créer des commandes avec une entrée `product_no` qui n'apparaît pas dans la table products.

Nous disons que, dans cette situation, la table de commandes est la table *référente* et la table products est la table *référée*. De la même façon, il y a des colonnes référentes et des colonnes référées.

On peut aussi raccourcir la commande ci−dessus en

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

parce qu'en l'absence d'une liste de colonnes, la clé primaire de la table référente est utilisée comme colonne référée.

Une clé étrangère peut aussi contraindre et référencer un groupe de colonnes. Comme d'habitude, il faut aussi l'écrire sous forme de contrainte de table. Voici un exemple de syntaxe:

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Bien sûr, le nombre et le type des colonnes contraintes doivent correspondre au nombre et au type des colonnes référées.

Vous pouvez affecter votre propre nom pour une contrainte de clé étrangère de la façon habituelle.

Une table peut contenir plus d'une contrainte de clé étrangère. Ceci peut être utilisé pour implémenter des relations n à n entre tables. Disons que vous avez des tables contenant des produits et des commandes mais vous voulez maintenant autoriser une commande qui contient peut−être beaucoup de produits (ce que la structure ci−dessus ne permet pas). On pourrait utiliser cette structure de table:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Notez aussi que la clé primaire chevauche les clés étrangères dans la dernière table.

Nous savons que les clés étrangères n'autorisent pas la création de commandes qui ne sont pas liés à un produit. Et si un produit est retiré après qu'une commande qui y réfère soit créée ? SQL vous permet aussi de le gérer. Intuitivement, nous avons plusieurs options :

- Interdire d'effacer un produit référé
- Effacer aussi les commandes
- Autre chose ?

Pour illustrer ce cas, implémentons la politique suivante sur l'exemple de relations n à n évoquée plus haut: Quand quelqu'un veut retirer un produit qui est encore référencé par un ordre (via `ordre_items`), on l'interdit. Si quelqu'un retire une commande, les éléments de l'ordre sont aussi retirés.

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Restreindre les suppressions et supprimer en cascade sont les deux options les plus communes. `RESTRICT` empêche la suppression d'une ligne référencée. `NO ACTION` signifie que si des lignes de références existent lors de la vérification de la contrainte, une erreur est levée  ceci est le comportement par défaut si vous n'avez rien spécifié. (La différence essentielle entre ces deux choix est que `NO ACTION` autorise la déférence de la vérification plus tard dans la transaction alors que `RESTRICT` ne le permet pas.) `CASCADE` spécifie que, quand une ligne référencée est supprimée, les lignes la référençant devraient aussi être automatiquement supprimées. Il existe deux autres options : `SET NULL` et `SET DEFAULT`. Celles−ci font que les colonnes de références soient initialisées à NULL ou à leur valeur par défaut, respectivement quand la ligne référencée est supprimée. Notez qu'elles ne vous excusent pas d'observer les contraintes. Par exemple, si une action spécifie `SET DEFAULT` mais que la valeur par défaut ne satisferait pas la clé étrangère, l'opération échouera.

Sur le même principe que `ON DELETE`, il y a aussi `ON UPDATE` qui est évoqué lorsqu'une colonne référencée est modifiée (mise à jour). Les actions possibles sont les mêmes.

Il y a plus d'informations sur la mise à jour et la suppression de données dans Chapitre 6.

Enfin, nous devrions dire que la clé étrangère peut référencer des colonnes qui sont une clé primaire ou forment une contrainte unique. Si la clé étrangère référence une contrainte unique, il y a des possibilités supplémentaires selon que l'on souhaite faire correspondre les valeurs NULL. Ceux−ci sont expliqués dans la documentation de référence pour *CREATE TABLE*.

## 5.4. Colonnes Systèmes

Chaque table a plusieurs *colonnes systèmes* qui sont implicitement définis par le système. De ce fait, ces noms ne peuvent être utilisés comme noms de colonnes définis par l'utilisateur. (Notez que ces restrictions sont différentes si le nom est un mot−clé ou pas ; citez un nom ne vous permettra pas d'échapper à ces restrictions.) Vous n'avez pas vraiment besoin de vous préoccuper de ces colonnes, simplement savoir qu'elles existent.

`oid`

L'identifiant objet (*object ID*) d'une rangée. Ceci est un numéro de série qui est automatiquement rajouté par PostgreSQL à toutes les rangées de tables (sauf si la table a été créée avec `WITHOUT OIDS`, auquel cas cette colonne n'est pas présente). Cette colonne est de type oid (même nom que la colonne) ; voir la Section 8.12 pour plus d'informations sur ce type.

`tableoid`

L' OID de la table contenant cette rangée. Cette colonne est particulièrement utile pour les requêtes qui sélectionnent de hiérarchies héritées, puisque sans elle, il est difficile de dire de quelle table vient

une rangée. `tableoid` peut être joint à la colonne `oid` de `pg_class` pour obtenir le nom de la table.

`xmin`

L'identité (transaction ID) de la transaction d'insertion de cette version de la rangée. (Une version de rangée est un état individuel d'une rangée; chaque mise à jour d'une rangée crée une nouvelle version de rangée pour la même rangée logique.)

`cmin`

L'identifiant de commande (à partir de zéro) au sein de la transaction d'insertion.

`xmax`

L'identité (transaction ID) de la transaction de suppression, ou zéro pour une version de rangée non effacée. Il est possible pour cette colonne d'être non NULL dans une version de rangée visible: Ceci indique normalement que la transaction de suppression n'a pas été effectuée, ou qu'une tentative de suppression a été annulée.

`cmax`

L'identifiant de commande au sein d'une transaction de suppression, ou zéro.

`ctid`

La localisation physique de la version de rangée au sein de sa table. Notez que bien que le `ctid` peut être utilisé pour trouver la version de rangée très rapidement, le `ctid` d'une rangée changera chaque fois qu'il est mis à jour ou déplacé par la commande `VACUUM FULL`. Donc, `ctid` est inutile en tant qu'identifiant de rangée à long terme. L'OID, ou encore mieux un numéro de série définie par l'utilisateur, devrait être utilisé pour identifier des rangées logiques.

Les OID sont des nombres de 32 bits et sont attribués d'un seul compteur. Dans une base de données grande ou vieille, il est possible que le compteur boucle sur lui−même. Donc il est peu pertinent de partir du principe que les OID sont uniques, sauf si vous prenez les précautions nécessaires. Si vous avez besoin d'identifier les lignes dans une table, l'utilisation d'un générateur de séquence est fortement recommandée. Néanmoins, les OID peuvent aussi être utilisés à condition que quelques précautions soient prises :

- Une contrainte unique devrait être créée sur la colonne OID de chaque table pour laquelle l'OID sera utilisée pour identifier les lignes.
- Les OID ne devraient jamais être supposés uniques entre tables ; utilisez la combinaison de `tableoid` et de l'OID de la ligne si vous avez besoin d'un identifiant sur la base complète.
- Les tables en question devraient être créées en utilisant `WITH OIDS` pour s'assurer de la compatibilité avec les versions futures de PostgreSQL. Il est planifié que `WITHOUT OIDS` deviendra l'option par défaut.

Les identifiants de transaction sont aussi des nombres de 32 bits. Dans une base de données de longue vie, il est possible pour les ID de transaction de boucler sur eux−mêmes. Ceci n'est pas un problème fatal avec des procédures de maintenance appropriées; voir le Chapitre 21 pour les détails. Il est, par contre, imprudent de dépendre de l'aspect unique des ID de transaction à long terme (plus d'un milliard de transactions).

Les identifiants de commande sont aussi des nombres de 32 bits. Ceci crée une limite dure de $2^{32}$ (4 milliards) commandes SQL au sein d'une seule transaction. En pratique, cette limite n'est pas un problème — notez que la limite est sur le nombre de commandes SQL, pas le nombre de rangées traitées.

# 5.5. Héritage

Créons deux tables. La table capitales contient les capitales d'état qui sont aussi des villes. Naturellement, la table capitales doit hériter de villes.

```
CREATE TABLE villes (
    nom             text,
    population      float,
    altitude        int     -- (in ft)
);

CREATE TABLE capitales (
    etat           char(2)
) INHERITS (villes);
```

Dans ce cas, une rangée de capitales *hérite* de tous les attributs (nom, population, et altitude) de son parent villes. Les capitales d'état ont un attribut supplémentaire state qui donne leur état. Dans PostgreSQL, une table peut hériter de zéro tables ou plus et une requête peut référencer toutes les rangées d'une table ou toutes les rangées d'une table plus celles de ses descendants.

> **Note :** La hiérarchie d'héritage est en fait un graphe acyclique dirigé.

Par exemple, la requête suivante cherche les noms de toutes les villes, y compris les capitales d'état, qui se situent à une altitude de plus de 500 pieds:

```
SELECT nom, altitude
    FROM villes
    WHERE altitude > 500;
```

qui retourne:

```
   nom     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

D'un autre côté, la requête suivante cherche toutes les villes qui ne sont pas des capitales d'état et qui sont situés à une altitude de plus de 500 pieds:

```
SELECT nom, altitude
    FROM ONLY villes
    WHERE altitude > 500;

   nom     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
```

Ici, le << ONLY >> avant villes indique que la requête ne devrait être lancée que sur villes et non les tables en dessous de villes dans la hiérarchie d'héritage. Beaucoup des commandes donc nous avons déjà discuté –– `SELECT`, `UPDATE` et `DELETE` –– gèrent cette syntaxe << ONLY >>.

> **Obsolète :** Dans les précédentes versions de PostgreSQL, le comportement par défaut était de ne pas inclure les tables enfants dans les requêtes. Il a été prouvé que cela amenait facilement

des erreurs et est aussi en violation du standard SQL:1999. Avec l'ancienne syntaxe, pour obtenir les sous−tables, vous ajoutez * au nom de la table. Par exemple

```
SELECT * from villes*;
```

Vous pouvez toujours spécifié explicitement le parcours des tables enfants en ajoutant *, ainsi qu'en spécifiant explicitement les tables enfants en écrivant << ONLY >>. Mais, depuis la version 7.1, le comportement par défaut pour un nom de table non décoré est de parcourir aussi ses tables enfants alors qu'avant ce n'était pas le comportement par défaut. Pour obtenir l'ancien comportement par défaut, initialisez l'option de configuration SQL_Inheritance à off, ainsi

```
SET SQL_Inheritance TO OFF;
```

ou ajoutez une ligne dans votre fichier postgresql.conf.

Dans certain cas, vous souhaiterez savoir dans quel table provient une rangée donnée. Il y a une colonne système appelée TABLEOID dans chaque table qui peut vous donner la table d'origine:

```
SELECT c.tableoid, c.nom, c.altitude
FROM villes c
WHERE c.altitude > 500;
```

qui renvoie :

```
 tableoid |   name    | altitude
----------+-----------+----------
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

(Si vous essayez de reproduire cet exemple, vous obtiendrez probablement des OID numériques différents.) En faisant une jointure avec pg_class, vous pourrez voir les noms de tables actuelles:

```
SELECT p.relname, v.nom, v.altitude
FROM villes v, pg_class p
WHERE v.altitude > 500 and v.tableoid = p.oid;
```

ce qui retourne:

```
 relname   |   nom     | altitude
-----------+-----------+----------
 villes    | Las Vegas |     2174
 villes    | Mariposa  |     1953
 capitales | Madison   |      845
```

Une table peut très bien hériter de plusieurs tables. Dans ce cas−là, les colonnes de la table fille correspondent à l'union des colonnes provenant des tables parentes et des colonnes définies dans la table fille.

Une limitation sérieuse de la fonctionnalité d'héritage est que les index (incluant les contraintes uniques) et les contraintes de clés étrangères s'appliquent seulement à des tables seules, pas à leurs héritiers. Ceci est vrai pour le côté de référence et le côté référencé d'une contrainte de clé étrangère. Du coup, dans les termes de

l'exemple ci–dessus :

- Si nous déclarons `villes.nom` comme `UNIQUE` ou comme une `PRIMARY KEY`, ceci n'empêchera pas la table `capitales` d'avoir des lignes avec des noms dupliqués dans `villes`. Et ces lignes dupliquées pourraient par défaut s'afficher dans les requêtes sur `villes`. En fait, par défaut, `capitales` n'aurait pas du tout de contrainte unique et, du coup, pourrait contenir plusieurs lignes avec le même nom. Vous pouvez ajouter une contrainte unique à `capitales` mais ceci n'empêcherait pas la duplication comparée à `villes`.
- De façon similaire, si nous devions spécifier que `villes.nom` `REFERENCES` une autre table, cette contrainte ne serait pas automatiquement propager à `capitales`. Dans ce cas, vous pourriez contourner ceci en ajoutant manuellement la même contrainte `REFERENCES` à `capitales`.
- Spécifier que la colonne d'une autre table `REFERENCES villes(nom)` autoriserait l'autre table à contenir les noms des villes mais pas les noms des capitales. Il n'existe pas de bons contournements pour ce cas.

Ces déficiences seront probablement corrigées dans une future version mais en attendant, un soucis considérable est nécessaire dans la décision de l'utilité de l'héritage pour votre problème.

# 5.6. Modification des tables

Quand on crée une table et qu'on se rend compte qu'on a fait une erreur ou que les besoins de l'application ont changés, on peut alors effacer la table et la recréer. Mais ceci n'est pas pratique si la table contient déjà des données ou si la table est référencée par d'autres objets de la base de données (une contrainte de clé étrangère). Par conséquent, PostgreSQL offre une série de commandes permettant de modifier une table existante. Notez que ceci est distinct au niveau du concept avec la modification des données contenues dans la table ; ici, nous sommes intéressés par la modification de la définition ou de la structure de la table.

Vous pouvez

- ajouter des colonnes,
- retirer des colonnes,
- ajouter des contraintes,
- retirer des contraintes,
- modifier les valeurs par défaut,
- modifier le type de données des colonnes,
- renommer des colonnes,
- renommer des tables.

Toutes ces actions sont réalisées en utilisant la commande *ALTER TABLE*.

## 5.6.1. Ajouter une colonne

Pour ajouter une colonne, utilisez une commande comme ceci :

```
ALTER TABLE products ADD COLUMN description text;
```

La nouvelle colonne est initialement remplie avec la valeur par défaut (NULL si vous n'avez pas spécifié de clause `DEFAULT`).

Vous pouvez aussi définir des contraintes sur la colonne au même moment en utilisant la syntaxe habituelle:

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

En fait, toutes les options applicables à la description d'une colonne dans `CREATE TABLE` peuvent être utilisées ici. Néanmoins, gardez en tête que la valeur par défaut doit satisfaire les contraintes données. Sinon, `ADD` échouera. Autrement, vous pouvez ajouter les contraintes plus tard (voir ci−dessous) après avoir rempli la nouvelle colonne correctement.

## 5.6.2. Retirer une Colonne

Pour retirer une colonne, utilisez une commande comme celle−ci :

```
ALTER TABLE products DROP COLUMN description;
```

Les données de cette colonne disparaissent. Les contraintes de table impliquant la colonne sont aussi supprimées. Néanmoins, si la colonne est référencée par une contrainte de clé étrangère dans une autre table, PostgreSQL ne supprimera pas silencieusement cette contrainte. Vous pouvez autoriser la suppression de tout ce qui dépend de la colonne en ajoutant `CASCADE` :

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

Voir Section 5.10 pour une description du mécanisme général derrière ceci.

## 5.6.3. Ajouter une Contrainte

Pour ajouter une contrainte, la syntaxe de contrainte de table est utilisée. Par exemple:

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

Pour ajouter une contrainte non NULL, qui ne peut pas être écrite sous forme d'une contrainte de table, utilisez cette syntaxe:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

La contrainte sera vérifiée immédiatement, donc les données de la table doivent remplir la contrainte avant qu'elle soit ajoutée.

## 5.6.4. Retirer une Contrainte

Pour retirer la contrainte, il faut connaître son nom. Si vous lui avez donné un nom, alors c'est facile. Sinon, le système a attribué un nom généré que vous devez découvrir. La commande `\d` *`tablename`* de psql peut être utile ici; d'autres interfaces peuvent aussi donner le moyen d'examiner les détails de table. Alors, la commande est:

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

(Si vous traitez avec un nom de contrainte généré comme `$2`, n'oubliez pas qu'il faudra l'entourer de guillemets double pour en faire un identifiant valable.)

Comme avec la suppression d'une colonne, vous avez besoin d'ajouter `CASCADE` si vous voulez supprimer une contrainte qui dépend d'autre chose. Un exemple concerne la contrainte de clé étrangère qui dépend d'une contrainte de clé unique ou primaire sur le(s) colonne(s) référencée(s).

Ça fonctionne de la même manière pour toutes les types de contrainte sauf les contraintes non NULL. Pour retirer une contrainte non NULL, utilisez

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

(Rappelez vous que les contraintes non NULL n'ont pas de noms.)

## 5.6.5. Modifier la valeur par défaut d'une colonne

Pour mettre une nouvelle valeur par défaut sur une colonne, utilisez une commande comme celle–ci:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Notez que ceci n'affecte pas les lignes existantes dans la table, cela ne modifie que la valeur par défaut pour les futures commandes `INSERT`.

Pour retirer toute valeur par défaut, utilisez

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

Ceci est équivalent à mettre la valeur par défaut à NULL. En conséquence, ce n'est pas une erreur de retirer une valeur par défaut qui n'a pas été définie car la valeur NULL est la valeur par défaut implicite.

## 5.6.6. Modifier le type de données d'une colonne

Pour convertir une colonne en un autre type de données, utilisez une commande comme ceci :

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

Elle ne réussira seulement si chaque entrée dans la colonne peut être convertie dans le nouveau type par une conversion implicite. Si une conversion plus complexe est nécessaire, vous pouvez ajouter une clause `USING` qui spécifie comment calculer les nouveaux valeurs à partir des anciennes.

PostgreSQL tentera de convertir la valeur par défaut de la colonne, si elle en a une. Mais ces conversions pourraient échouer ou pourraient produire des résultats surprenants. Il est souvent mieux de supprimer les contraintes sur une colonne avant de modifier son type, puis d'ajouter les contraintes modifiées convenablement.

### 5.6.7. Renommer une colonne

Pour renommer une colonne:

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

### 5.6.8. Renommer une Table

Pour renommer une table:

```
ALTER TABLE products RENAME TO items;
```

## 5.7. Privilèges

Quand vous créez un objet base de données, vous en devenez le propriétaire. Avec les paramètres par défaut, seul le propriétaire d'un objet peut faire quelque chose avec. Pour que d'autres utilisateurs puissent s'en servir, des *privilèges* doivent être accordés. (Néanmoins, les utilisateurs qui sont superutilisateurs ont toujours accès à n'importe quel objet.)

Il y a plusieurs privilèges différents : `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `RULE`, `REFERENCES`, `TRIGGER`, `CREATE`, `TEMPORARY`, `EXECUTE`, et `USAGE`. Les droits applicables à un objet particulier varient suivant le type d'objet (table, fonction, etc.) Pour une information complète sur les différents types de privilèges gérés par PostgreSQL, lisez la page de référence GRANT. La section et les chapitres suivants vous montreront aussi comment ces privilèges sont utilisés.

Le droit de modifier ou de détruire est le privilège du propriétaire seul.

> **Note :** Pour modifier le propriétaire d'une table, d'un index, d'une séquence ou d'une vue, utilisez la commande ALTER TABLE. Il existe des commandes `ALTER` correspondant aux autres types d'objets.

Pour accorder des privilèges, la commande `GRANT` est utilisé. Par exemple, si `joe` est un utilisateur existant et `accounts` une table existante, le privilège de mettre la table à jour peut être accordé avec

```
GRANT UPDATE ON accounts TO joe;
```

Pour accorder un privilège à un groupe, utilisez

```
GRANT SELECT ON accounts TO GROUP staff;
```

Le nom d'<< utilisateur >> spéciale `PUBLIC` peut être utilisé pour accorder un privilège à tout les utilisateurs du système. Écrire `ALL` au lieu d'un privilège spécifique accorde tous les privilèges adéquats pour ce type d'objet.

Pour révoquer un privilège, utilisez la commande approprié ci dessous; `REVOKE` :

```
REVOKE ALL ON accounts FROM PUBLIC;
```

Les privilèges spéciaux du propriétaire de l'objet (c'est–à–dire, le droit de faire des `DROP`, `GRANT`, `REVOKE`, etc.) sont toujours implicitement ceux du propriétaire et ne peuvent être ni accordés ni révoqués. Mais le propriétaire de l'objet peut choisir de révoquer ses propres privilèges ordinaires pour, par exemple, mettre une table en lecture seule pour soi–même en plus que pour les autres.

Habituellement, seul le propriétaire de l'objet (ou un superutilisateur) peut donner ou révoquer les droits sur un objet. Néanmoins, il est possible de donner un droit << avec une option de don de droits >>, qui donne à celui qui le reçoit de le donner à d'autres. Si cette option est ensuite révoquée, alors tous ceux qui ont reçu le droit de cet utilisateur (directement ou indirectement via la chaîne des dons) perdront leur droit. Pour des détails, voir les pages de références GRANT et REVOKE.

# 5.8. Schémas

Un cluster de bases de données PostgreSQL contient une ou plusieurs bases nommées. Les utilisateurs et groupes d'utilisateurs sont partagés sur le cluster tout entier mais aucune autres données n'est partagées parmi les bases. Une connexion cliente donnée sur le serveur peut accéder aux données d'une seule base, celle spécifiée dans la connexion de requête.

> **Note :** Les utilisateurs d'un cluster n'ont pas forcément le privilège d'accéder toutes les bases du cluster. Le partage des noms d'utilisateur veut dire qu'il ne peut pas y avoir plusieurs utilisateurs nommés `joe`, par exemple, dans deux bases du même cluster; mais le système peut être configuré pour autoriser `joe` à accéder qu'à certaines bases.

Une base de données contient un ou plusieurs *schémas* nommés, qui, eux, contiennent des tables. Les schémas contiennent aussi d'autres types d'objets nommés, y compris des types de données, fonctions et opérateurs. Seul le nom d'objet peut être utilisé sans conflit; par exemple, `schema1` et `mon_schema` peuvent tout les deux contenir des tables nommées `ma_table`. Contrairement aux bases de données; les schémas ne sont pas séparés de manière rigide: un utilisateur peut accéder aux objets de n'importe lequel des schémas de la base de données auxquels il se connecte s'il a les privilèges pour le faire.

Il y a plusieurs raisons pour lesquelles quelqu'un voudrait utiliser les schémas:

- Pour autoriser beaucoup d'utilisateurs d'utiliser une base de données sans se gêner les uns les autres.
- Pour organiser des objets de bases de données en groupes logiques afin de faciliter leur gestion.
- Les applications tierces peuvent être mises dans des schémas séparés pour qu'il n'y ait pas de collision avec les noms d'autres objets.

Les schémas sont comparables aux répertoires au niveau du système d'exploitation. sauf que les schémas ne peuvent pas être imbriqués.

## 5.8.1. Créer un Schéma

Pour créer un schéma, utilisez la commande `CREATE SCHEMA`. Donnez au schéma un nom de votre choix. Par exemple:

```
CREATE SCHEMA mon_schema;
```

Pour créer ou accéder aux objets dans un schéma, écrivez un *nom qualifié* qui consiste en le nom du schéma et le nom de la table séparés par un point:

```
schema.table
```

Ceci fonctionne partout où un nom de table est attendu, ceci incluant les commandes de modification de la table et les commandes d'accès aux données discutées dans les chapitres suivants. (En bref, nous parlerons uniquement des tables mais les mêmes idées s'appliquent aux autres genres d'objets nommés, comme les types et les fonctions.)

En fait, la syntaxe encore plus générale

```
database.schema.table
```

peut être utilisé aussi mais, pour le moment, ceci n'existe que pour être conforme au standard SQL. Si vous écrivez un nom de base de données, il devrait être celui de la base auquel vous êtes connecté.

Donc, pour créer une table dans le nouveau schéma, utilisez

```
CREATE TABLE mon_schema.ma_table (
 ...
);
```

Pour effacer un schéma vide (tout les objets a l'intérieur ont été effacés), utilisez

```
DROP SCHEMA mon_schema;
```

Pour effacer un schéma, y compris les objets qu'il contient, utilisez

```
DROP SCHEMA mon_schema CASCADE;
```

Lisez Section 5.10 pour une description du mécanisme général derrière tout ceci.

Cependant, vous voudriez modifier le schéma utilisé par quelqu'un d'autre (puisque c'est l'une des méthodes par lesquelles on peut restreindre l'activité de vos utilisateurs à des espaces de nom définis). La syntaxe pour ceci est:

```
CREATE SCHEMA nom_schema AUTHORIZATION nom_utilisateur;
```

Vous pouvez même omettre le nom du schéma auquel cas, le nom du schéma sera le même que le nom d'utilisateur. Voir Section 5.8.6 pour voir comment cela peut être utile.

Les noms de schéma commençant par `pg_` sont réservés pour les besoins du système et ne peuvent pas être crées par les utilisateurs.

## 5.8.2. Le Schéma Public

Dans les sections précédentes, on créait des tables sans spécifier un nom de schéma. Par défaut, ces tables (et autres objets) sont automatiquement mis dans un schéma nommé << public >>. Toute nouvelle base de données contient un tel schéma. Donc, ces instructions sont équivalents:

```
CREATE TABLE products ( ... );
```

and

```
CREATE TABLE public.products ( ... );
```

## 5.8.3. Le Chemin de Recherche de Schéma

Les noms qualifiés sont pénibles à écrire et il est, de toutes façons, préférable de ne pas coder un nom de schéma dans une application. Donc, les tables sont souvent appelées par des noms non *qualifiés* qui s'apparente souvent au nom de la table lui même. Le système détermine quel table est appelée en suivant un *chemin de recherche* qui est une liste de schémas a regarder. La première table correspondante est considérée comme la table voulue. S'il n'y a pas de correspondance, une erreur est soulevée, même si des noms de table correspondants existent dans d'autres schémas dans la base.

Le premier schéma dans le chemin de recherche est appelé le schéma courant. En plus d'être le premier schéma parcouru, il est aussi le schéma dans lequel de nouvelles tables seront créées si la commande `CREATE TABLE` ne précise pas de nom de schéma.

Pour voir le chemin de recherche courant, utilisez la commande suivante:

```
SHOW search_path;
```

Dans la configuration par défaut, ceci renvoie:

```
 search_path
--------------
 $user,public
```

Le premier élément précise qu'un schéma avec le même nom que l'utilisateur en cours doit être parcouru. Le deuxième élément renvoie au schéma public que nous avons déjà vu.

Le premier schéma existant dans le chemin de recherche est l'endroit par défaut pour la création de nouveaux objets. Ceci est la raison pour laquelle les objets sont créés dans le schéma public. Quand les objets sont liés dans tout autre contexte sans une qualification de schéma (modification de table, modification de données ou requête de commande), le chemin de recherche est traversé jusqu'à ce qu'un objet correspondant soit trouvé. Donc, dans la configuration par défaut, tout accès non qualifié ne peut que se référer au schéma public.

Pour mettre notre nouveau schéma dans le chemin, nous utilisons

```
SET search_path TO mon_schema,public;
```

(Nous ne mettons pas le `$user` ici car nous n'en avons pas besoin pour l'instant.) Et nous pouvons pas accéder à la table sans qualification de schéma:

```
DROP TABLE ma_table;
```

Aussi, puisque `mon_schema` est le premier élément dans le chemin, les nouveaux objets seront crées à l'intérieur.

On pourrait aussi écrire

```
SET search_path TO mon_schema;
```

Alors nous n'avons pas accès au schéma public sans qualification explicite. Il n'y a rien de spécial à propos du schéma public hormis le fait qu'il existe par défaut. Il peut aussi être effacé.

Voir aussi Section 9.19 qui détaille les autres façons de manipuler le chemin de recherche de schéma.

Le chemin de recherche fonctionne de la même façon pour les noms de type de données, noms de fonction, et noms d'opérateur que pour le noms de tables. Les types de données et de fonction peuvent être qualifiés de la même façon que les noms de table. Si vous avez besoin d'écrire un nom d'opérateur qualifié dans une expression, il y'a une condition spéciale: vous devez écrire

```
OPERATOR(schéma.opérateur)
```

Ceci est nécessaire afin d'éviter une ambiguïté syntaxique. Un exemple est

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

En pratique, on dépend souvent du chemin de recherche pour les opérateurs, afin de ne pas avoir à écrire quelque chose d'aussi peu présentable.

## 5.8.4. Schémas et Privilèges

Par défaut, les utilisateurs ne peuvent pas accéder aux objets dans les schémas qui ne leurs appartiennent pas. Pour autoriser cela, le propriétaire du schéma doit donner le privilège `USAGE` sur le schéma. Pour autoriser les utilisateurs à manipuler les objets d'un schéma, des privilèges supplémentaires devront peut−être être accordés, suivant l'objet.

Un utilisateur peut aussi être autorisé à créer des objets dans le schéma de quelqu'un d'autre. Pour permettre ceci, le privilège `CREATE` doit être accordé. Notez que, par défaut, tout le monde à les privilèges `CREATE` et `USAGE` sur le schéma `public`. Ceci permet à tout les utilisateurs qui sont capables de se connecter à une base de données de créer des objets dans son schéma `public`. Si vous ne souhaitez pas ce comportement, vous pouvez révoquer ce privilège:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

(Le premier << public >> est le schéma, le second << public >> veut dire << chaque utilisateur >>. Dans le premier cas, c'est un identifiant. Dans le second, c'est un mot clé, d'où la casse différente. Souvenez−vous des règles de Section 4.1.1.)

## 5.8.5. Le Catalogue de Système de Schéma

En plus des schémas `publics` crées par les utilisateurs, chaque base de données contient un schéma `pg_catalog`, qui contient les tables systèmes et tous les types de données, fonctions et opérateurs intégrés. `pg_catalog` fait toujours, de fait, partie du chemin de recherche. S'il n'est pas nommé explicitement dans le chemin, il est parcouru implicitement *avant* la recherche dans les schémas du chemin. Ceci garantit qui les noms internes seront toujours trouvables . Par contre, vous pouvez explicitement placer `pg_catalog` à la fin

si vous préférez que les noms définis par les utilisateurs surchargent les noms internes.

Dans les versions de PostgreSQL antérieures à la 7.3, les noms de table commençant par `pg_` étaient réservés. Ceci n'est plus vrai: vous pouvez créer une telle table si vous le voulez dans n'importe quel schéma non système. Par contre, il vaut mieux continuer d'éviter de tels noms pour garantir que vous n'aurez pas de conflit si une prochaine version définit une table système qui porte le même nom que votre table. (Avec le chemin de recherche par défaut, une référence non qualifiée à votre table pointera au lieu vers la table système.) Les tables systèmes continueront de suivre la convention de porter des noms commençant par `pg_` pour qu'ils n'aient pas de conflit avec des noms de table non qualifiés tant que les utilisateurs éviteront le préfixe `pg_`.

## 5.8.6. Méthodes d'utilisation

Les schémas peuvent être utilisés pour organiser vos données de plusieurs manières. Il y en a plusieurs qui sont recommandés et sont facilement supportés par la configuration par défaut:

- Si vous ne créez aucun schéma, alors tout les utilisateurs auront accès au schéma public implicitement. Ceci simule la situation dans laquelle les schémas ne sont pas disponibles. Cette situation est recommandé lorsque il n'y a qu'un seul utilisateur ou quelques utilisateurs coopérants dans une base de données. Cette configuration permet aussi une transition en douceur d'une situation ou on ne connaît pas le schéma.
- Vous pouvez créer un schéma pour chaque utilisateur avec un nom identique à celui de l'utilisateur. Souvenez vous que le chemin de recherche par défaut commence par `$user` qui correspond au nom d'utilisateur. Donc si chaque utilisateur a un schéma distinct, ils accèdent à leurs propres schémas par défaut.

  Si vous utilisez cette configuration alors vous devriez peut−être aussi révoquer l'accès au schéma public (ou l'effacer complètement) pour que les utilisateurs soient réellement limités à leur propre schéma.
- Pour installer des applications partagées (tables utilisables par tout le monde, fonctionnalités supplémentaires fournies par des applications tiers, etc), insérez les dans des schéma séparés. Rappelez vous que vous devez donner les permissions appropriées pour permettre aux utilisateurs d'y accéder. Les utilisateurs peuvent alors se référer à ces objets additionnels en qualifiant les noms avec un nom de schéma ou ils peuvent mettre les schémas supplémentaires dans leur chemin de recherche, s'ils le souhaitent.

## 5.8.7. Portabilité

Dans le standard SQL, la notion d'objets dans le même schéma appartenant à des utilisateurs différents n'existe pas. De plus, certaines implémentations ne vous permettent pas de créer des schémas qui ont un nom différent de celui de leur propriétaire. En fait, les concepts de schéma et d'utilisateur sont presque équivalents dans un système de base de données qui n'implémente que le support basique des schémas spécifiés dans le standard. A partir de ce constat, beaucoup d'utilisateurs considèrent les noms qualifiés qui correspondre réellement à *`utilisateur`*`.`*`table`*. C'est comme cela que PostgreSQL se comporte si vous créez un schéma par utilisateur pour chaque utilisateur.

De plus, il n'y a aucun concept d'un schéma `public` dans le standard SQL. Pour plus de conformité au standard, vous ne devriez pas utilisez (et sans doute effacer) le schéma `public`.

Bien sur, certains systèmes de bases de données n'implémentent pas du tout les schémas. ou donnent le support d'espace de nommage en autorisant (peut−être de façon limité) des accès sur plusieurs bases de

données alors la portabilité maximale sera obtenue en n'utilisant pas du tout les schémas.

# 5.9. D'autres Objets Base de Données

Les tables sont les objets centraux dans une structure de base de données relationnelles. Mais ce ne sont pas les seuls objets qui existent dans une base de données. Plusieurs autres types d'objets peuvent être crées afin de pour rendre l'utilisation et la gestion des données plus efficace ou pratique. Ils ne seront pas abordés dans ce chapitre mais nous vous en faisons une liste ici pour que vous soyez informés de ce qui est possible.

- Vues
- Fonctions et opérateurs
- Types de données et domaines
- Triggers et règles de réécriture

Des informations détaillées sur ces sujets apparaissent dans Partie V.

# 5.10. Gestion des Dépendances

Lorsque vous créez des structures de base complexes impliquant beaucoup de tables avec des contraintes de clés étrangères, des vues, des triggers, des fonctions, etc, vous créez implicitement un filet de dépendances entre les objets. Par exemple, une table avec une contrainte de clé étrangère dépend de la base à laquelle elle fait référence.

Pour garantir l'intégrité de la structure entière de la base, PostgreSQL vérifie que vous ne pouvez pas effacer des objets dont d'autres objets dont dépendants. Par exemple, la tentative d'effacer la table des produits que nous avons utilisé dans Section 5.3.5, avec la tables des commandes qui en dépend, donnera un message d'erreur comme celui–ci:

```
DROP TABLE products;

NOTICE:  constraint orders_product_no_fkey on table orders depends on table products
ERROR:  cannot drop table products because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

Le message d'erreur contient un indice utile: Si vous ne souhaitez pas effacer les objets dépendants individuellement, vous pouvez lancer

```
DROP TABLE products CASCADE;
```

et tout les objets seront effacés. Dans ce cas, cela n'effacera pas la table des commandes mais seulement la contrainte de clé étrangère. (Si vous voulez vérifier ce que `DROP ... CASCADE` fera, lancez `DROP` sans `CASCADE` et lisez les messages `NOTICE`.)

Toutes les commandes de suppression dans PostgreSQL supportent l'utilisation de `CASCADE`. Bien sur, la nature des dépendances varie avec la nature des objets. Vous pouvez aussi écrire `RESTRICT` au lieu de `CASCADE` pour obtenir le comportement par défaut qui est d'empêcher les suppressions d'objets sur lesquelles dépendent d'autres objets.

**Note :** D'après le standard SQL, spécifier l'un de `RESTRICT` ou `CASCADE` est requis. Aucun

système de base de donnée ne force cette règle de cette manière actuellement mais le choix du comportement par défaut, `RESTRICT` ou `CASCADE`, varie suivant le système.

**Note :** Les dépendances de contraintes de clés étrangères et de colonnes en série des versions de PostgreSQL antérieures à 7.3 ne seront *pas* maintenues ou créées pendant le processus de mise à jour. Tout autre type de dépendance sera proprement créé pendant une mise à jour à partir d'une base de données antérieure à la 7.3.

---

# Chapitre 6. Manipulation de données

Le chapitre précédent expliquait comment créer des tables et d'autres structures pour stocker vos données. Nous allons maintenant remplir ces tables avec des données. Ce chapitre montre comment insérer, mettre à jour et supprimer des données des tables. Nous allons aussi montrer des méthodes pour effectuer des changements automatiquement dans les données quand certains événements ont lieu : les déclencheurs et les règles de réécriture. Le chapitre suivant expliquera enfin comment extraire des données perdues depuis longtemps dans la base de données.

## 6.1. Insérer des données

Quand une table est créée, elle ne contient aucune donnée. La première chose à faire, c'est d'y insérer des données. Sinon, la base de données n'est pas d'une grande utilité. Les données sont insérées ligne par ligne. Bien sûr, il est possible d'insérer plus d'une ligne mais il n'est pas possible d'entrer moins qu'une ligne à la fois. Même si vous ne connaissez les valeurs que pour quelques colonnes, une ligne complète doit être créée.

Pour créer une nouvelle ligne, utilisez la commande `INSERT`. La commande a besoin du nom de la table et d'une valeur pour chaque colonne de cette table. Par exemple, utilisons produits, la table des produits de Chapitre 5 :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric
);
```

Un exemple de commande pour insérer une ligne serait :

```
INSERT INTO produits VALUES (1, 'Fromage', 9.99);
```

Les données sont listées dans l'ordre dans lequel les colonnes apparaissent dans la table, séparées par des virgules. Souvent, les données sont des littéraux (constantes) mais les expressions scalaires sont aussi acceptées.

La syntaxe précédente a le défaut qu'il faut connaître l'ordre des colonnes. Pour éviter ce problème, vous pouvez aussi lister les colonnes explicitement. Par exemple, les deux commandes suivantes ont le même effet que la précédente :

```
INSERT INTO produits (no_produit, nom, prix) VALUES (1, 'Fromage', 9.99);
INSERT INTO produits (nom, prix, no_produit) VALUES ('Fromage', 9.99, 1);
```

Beaucoup d'utilisateurs recommandent de toujours lister les noms de colonnes.

Si vous ne connaissez pas les valeurs de certaines colonnes, vous pouvez les omettre. Dans ce cas, elles seront remplies avec leur valeur par défaut. Par exemple,

```
INSERT INTO produits (no_produit, nom) VALUES (1, 'Fromage');
INSERT INTO produits VALUES (1, 'Fromage');
```

La seconde instruction est une extension de PostgreSQL. Elle remplit les colonnes de gauche à droite avec toutes les valeurs données, et les autres prennent leur valeur par défaut.

Pour plus de clarté, vous pouvez aussi explicitement demander les valeurs par défaut pour des colonnes spécifiques ou pour la ligne complète.

```
INSERT INTO produits (no_produit, nom, prix) VALUES (1, 'Fromage', DEFAULT);
INSERT INTO produits DEFAULT VALUES;
```

> **Astuce :** Pour faire des chargements en masse (insertion de grandes quantités de données), jetez un œil à la commande *COPY*. Elle n'est pas aussi souple que la commande `INSERT` mais elle est plus efficace.

## 6.2. Modifier des données

La modification de données déjà présentes dans la base est appelée mise à jour (update en anglais). Vous pouvez mettre à jour une ligne spécifique, toutes les lignes d'une table ou un sous−ensemble des lignes de la table. Chaque colonne peut être mise à jour séparément ; les autres colonnes ne sont pas modifiées.

Pour faire une mise à jour, il faut trois informations :

1. le nom de la table et de la colonne à mettre à jour ;
2. la nouvelle valeur de la colonne ;
3. les lignes à mettre à jour.

Nous avons vu dans le Chapitre 5 que le SQL ne donne pas par défaut d'identifiant unique pour les lignes. Du coup, il n'est pas nécessairement possible d'indiquer directement quelle ligne il faut mettre à jour. À la place, nous indiquons quelles conditions une ligne doit remplir pour être mise à jour. Si la table a une clé primaire (qu'elle soit déclarée ou non), nous pouvons indiquer une ligne unique en donnant une condition qui porte sur la clé primaire. Les outils graphiques d'accès aux bases de données utilisent ce principe pour vous permettre de modifier les lignes individuellement.

Par exemple, cette commande modifie tous les produits qui ont un prix de 5 et met leur prix à 10.

```
UPDATE produits SET prix = 10 WHERE prix = 5;
```

Ceci peut mettre à jour zéro, une ou plusieurs lignes. Ce n'est pas une erreur d'exécuter une commande `UPDATE` qui ne met à jour aucune ligne.

Voyons cette commande en détail. Tout d'abord, il y a le mot clé `UPDATE` suivi par le nom de la table. Comme d'habitude, le nom de la table peut être précisé par un nom de schéma, sans quoi le schéma est recherché dans le chemin. Ensuite, il y a le mot clé `SET` suivi par le nom de la colonne, un signe égal et la nouvelle valeur de la colonne. La nouvelle valeur de la colonne peut être une constante ou une expression scalaire. Par exemple, pour augmenter de 10% le prix de tous les produits, on peut exécuter :

```
UPDATE produits SET prix = prix * 1.10;
```

Comme vous le voyez, l'expression donnant la nouvelle valeur peut faire référence à la valeur actuelle dans la ligne. Nous n'avons pas encore parlé de la clause `WHERE`. Si elle est omise, cela veut dire que toutes les lignes de la table sont modifiées. Si elle est présente, seules les lignes qui remplissent la condition `WHERE` sont mises

à jour. Remarquez que le signe égal dans la clause SET est une affectation, alors que celui de la clause WHERE est une comparaison, mais cela ne crée pas d'ambiguïté. Bien sûr, la condition WHERE n'est pas nécessairement un test d'égalité. De nombreux autres opérateurs existent (voir le Chapitre 9). Mais l'expression doit s'évaluer en une expression booléenne.

Il est possible de mettre plus d'une colonne à jour dans une commande UPDATE en indiquant plusieurs colonnes dans la clause SET. Par exemple :

```
UPDATE matable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

## 6.3. Supprimer des données

Jusqu'ici, nous avons expliqué comment ajouter des données à une table et comment les modifier. Il nous reste à voir comment les enlever quand elles ne sont plus nécessaires. De la même façon que pour l'insertion, la suppression ne peut se faire que par ligne entière. Dans la section précédente, nous avons expliqué que le SQL ne propose pas de moyen d'accéder à une ligne particulière. C'est pourquoi la suppression de lignes se fait en indiquant les conditions à remplir par les lignes à supprimer. S'il y a une clé primaire dans la table, alors il est possible d'indiquer exactement la ligne à supprimer. Mais on peut aussi supprimer un groupe de lignes qui remplissent une condition, ou même toutes les lignes d'une table d'un coup.

On utilise la commande DELETE pour supprimer des lignes ; la syntaxe est très similaire à la commande UPDATE. Par exemple, pour supprimer toutes les lignes de la table produits qui ont un prix de 10, on exécute :

```
DELETE FROM produits WHERE prix = 10;
```

En indiquant simplement

```
DELETE FROM produits;
```

on supprime toutes les lignes de la table. Attention aux mauvaises manipulations !

# Chapitre 7. Requêtes

Les précédents chapitres ont expliqué comme créer des tables, comment les remplir avec des données et comment manipuler ces données. Maintenant, nous discutons enfin de la façon de récupérer ces données depuis la base de données.

## 7.1. Survol

Le processus et la commande de récupération des données sont appelés une *requête*. En SQL, la commande `SELECT` est utilisé pour spécifier des requêtes. La syntaxe générale de la commande `SELECT` est

```
SELECT liste_select FROM
expression_table
[_specification_tri]
```

Les sections suivantes décrivent le détail de la liste de sélection, l'expression des tables et la spécification du tri.

Le type de requête le plus simple est de la forme

```
SELECT * FROM table1;
```

En supposant qu'il existe une table appelée `table1`, cette commande récupérera toutes les lignes et toutes les colonnes de `table1`. (La méthode de récupération dépend de l'application cliente. Par exemple, le programme psql affichera une table, façon art ASCII, alors que les bibliothèques du client offriront des fonctions d'extraction de valeurs individuelles à partir du résultat de la requête.) `*` comme liste de sélection signifie que toutes les colonnes de l'expression de table seront récupérées. Une liste de sélection peut aussi être un sous−ensemble des colonnes disponibles ou effectuer un calcul en utilisant les colonnes. Par exemple, si `table1` dispose des colonnes nommées `a`, `b` et `c` (et peut−être d'autres), vous pouvez lancer la requête suivante :

```
SELECT a, b + c FROM table1;
```

(en supposant que `b` et `c` soient de type numérique). Voir Section 7.3 pour plus de détails.

`FROM table1` est un type très simple d'expression de tables : il lit une seule table. En général, les expressions de tables sont des constructions complexes de tables de base, de jointures et de sous−requêtes. Mais vous pouvez aussi entièrement omettre l'expression de table et utiliser la commande `SELECT` comme une calculatrice :

```
SELECT 3 * 4;
```

Ceci est plus utile si les expressions de la liste de sélection renvoient des résultats variants. Par exemple, vous pouvez appeler une fonction de cette façon :

```
SELECT random();
```

# 7.2. Expressions de table

Une *expression de table* calcule une table. L'expression de table contient une clause `FROM` qui peut être suivie des clauses `WHERE`, `GROUP BY` et `HAVING`. Les expressions de table triviales réfèrent simplement à une table sur le disque, une table de base, mais des expressions plus complexes peuvent être utilisées pour modifier ou combiner des tables de base de différentes façon.

Les clauses optionnelles `WHERE`, `GROUP BY` et `HAVING` dans l'expression de table spécifient un tube de transformations successives réalisées sur la table dérivée de la clause `FROM`. Toutes ces transformations produisent une table virtuelle fournissant les lignes à passer à la liste de sélection qui choisira les lignes à afficher de la requête.

## 7.2.1. La clause `FROM`

La clause *Clause FROM* dérive une table à partir d'une ou plusieurs tables données dans une liste de référence dont les tables sont séparées par des virgules.

```
FROM reference_table [,
reference_table [, ...]]
```

Une référence de table pourrait être un nom de table (avec en option le nom du schéma) ou une table dérivée comme une sous−requête, une table jointe ou une combinaison complexe de celles−ci. Si plus d'une référence de tables est listé dans la clause `FROM`, elle sont jointes pour former une table virtuelle intermédiaire qui pourrait être le sujet des transformations des clauses `WHERE`, `GROUP BY`, et `HAVING` et est finalement le résultat des expressions de table.

Lorsqu'une référence de table nomme une table qui est la table parent d'une table suivant la hiérarchie de l'héritage, la référence de table produit les lignes non seulement de la table mais aussi des successeurs de cette table sauf si le mot clé `ONLY` précède le nom de la table. Néanmoins, la référence produit seulement les colonnes qui apparaissent dans la table nommée... toute colonne ajoutée dans une sous−table est ignorée.

### 7.2.1.1. Tables jointes

Une table jointe est une table dérivée de deux autres tables (réelles ou dérivées) suivant les règles du type de jointure particulier. Les jointures internes (inner), externes (outer) et croisées (cross) sont disponibles.

**Types de jointures**

Jointure croisée (cross join)
`T1 CROSS JOIN T2`

Pour chaque combinaison de lignes provenant de *T1* et *T2*, la table dérivée contiendra une ligne consistant de toutes les colonnes de *T1* suivies de toutes les colonnes de *T2*. Si les tables ont respectivement N et M lignes, la table jointe en aura N * M.

`FROM T1 CROSS JOIN T2` est équivalent à `FROM T1, T2`. C'est aussi équivalent à `FROM T1 INNER JOIN T2 ON TRUE` (voir ci−dessous).

Jointures qualifiées (qualified joins)

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

Les mots INNER et OUTER sont optionnelles dans toutes les formes. INNER est la valeur par défaut ; LEFT, RIGHT et FULL impliquent une jointure externe.

La *condition de la jointure* est spécifiée dans la clause ON ou USING ou implicitement par le mot NATURAL. La condition de jointure détermine les lignes des deux tables source considérées comme << correspondante >>, comme l'explique le paragraphe ci−dessous.

La clause ON est le type le plus général de condition de jointure : il prend une expression booléenne du même genre que celui utilisée dans une clause WHERE. Une paires de lignes de *T1* et *T2* correspondent si l'expression ON est évaluée à vraie (true) pour ces deux lignes.

USING est la notation raccourcie : elle prend une liste de noms de colonnes séparées par des virgules, que les tables jointes ont en commun, et forme une condition de jointure spécifiant l'égalité de chacune de ces paires de colonnes. De plus, la sortie de JOIN USING a une colonne pour chaque paires égales des colonnes en entrée, suivies par toutes les autres colonnes de chaque table. Du coup, USING (a, b, c) est équivalent à ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c) avec l'exception que si ON est utilisé, il y aura deux colonnes a, b, puis c dans le résultat, alors qu'avec USING, il n'y en aurait eu qu'une de chaque.

Enfin, NATURAL est un format raccourci de USING : il forme une liste USING consistant exactement des noms de colonnes apparaissant à la fois dans deux tables en entrée. Comme avec USING, ces colonnes apparaissent seulement une fois dans la table de sortie.

Les types possibles de jointures qualifiées sont :

INNER JOIN
: Pour chaque ligne R1 de T1, la table jointe a une ligne pour chaque ligne de T2 satisfaisant la condition de jointure avec R1.

LEFT OUTER JOIN
: Tout d'abord, une jointure interne est réalisée. Puis, pour chaque ligne de T1 qui ne satisfont pas la condition de jointure avec toutes les lignes de T2, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T2. Du coup, la table jointe a au moins une ligne pour chaque ligne de T1 quelque soient les conditions.

RIGHT OUTER JOIN
: Tout d'abord, une jointure interne est réalisée. Puis, pour chaque ligne de T2 qui ne satisfait pas la condition de jointure avec une ligne de T1, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T1. C'est l'inverse d'une jointure gauche : la table résultante aura une ligne pour chaque ligne de T2 quelque soient les conditions.

FULL OUTER JOIN
: Tout d'abord, une jointure interne est réalisée. Puis, pour chaque ligne de T1 qui ne satisfait pas la condition de jointure avec une ligne de T2, une ligne jointe est ajouté avec des valeurs NULL dans les colonnes de T2. De plus, pour chaque ligne de T2 qui ne satisfait pas la condition de jointure avec les lignes de T1, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T1.

Les jointures de tous les types peuvent être chaînées ensemble ou imbriquées : soit les deux soit une des deux, parmi *T1* et *T2*, pourraient être des tables. Les parenthèses peuvent être utilisées autour des clauses JOIN pour contrôler l'ordre de jointure. En l'absence de parenthèses, les clauses JOIN sont imbriquées de gauche à droite.

Pour rassembler tout ceci, supposons que nous avons une table t1

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

et une table t2

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

nous obtenons les résultats suivants des différentes jointures :

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
```

```
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

La condition de jointure spécifiée avec `ON` peut aussi contenir des conditions sans relation directe avec la jointure. Ceci peut prouver son utilité pour quelques requêtes mais son utilisation doit avoir été réfléchie. Par exemple :

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

### 7.2.1.2. Alias de table et de colonne

Un nom temporaire peut être donné aux tables et aux références de tables complexe, qui sera ensuite utilisé pour référencer la table dérivée dans la suite de la requête. Cela s'appelle un *alias de table*.

Pour créer un alias de table, écrivez

```
FROM reference_table AS
alias
```

or

```
FROM reference_table alias
```

Le mot clé `AS` est du bruit. *`alias`* peut être tout identifiant.

Une application typique des alias de table est l'affectation d'identifieurs cours pour les noms de tables longs, ce qui permet de garder des clauses de jointures lisibles. Par exemple :

```
SELECT * FROM nom_de_table_tres_tres_long s JOIN un_autre_nom_trs_long a ON
s.id = a.num;
```

L'alias devient le nouveau nom de la référence de la table pour la requête courante — il n'est plus possible de référencer la table avec son nom d'origine. Du coup,

```
SELECT * FROM ma_table AS m WHERE ma_table.a > 5;
```

n'est pas valide pour une syntaxe SQL. Ce qui va réellement se passer (c'est une extension de PostgreSQL au standard) est qu'une référence de table implicite est ajoutée à la clause `FROM`, de façon à ce que la requête soit exécutée comme si elle était écrite ainsi

```
SELECT * FROM ma_table AS m, ma_table AS ma_table WHERE ma_table.a > 5;
```

ce qui résultera en une jointure croisée, ce qui n'est habituellement pas ce que vous voulez.

Les alias de table sont disponibles principalement pour aider à l'écriture de requête mais ils deviennent nécessaires pour joindre une table avec elle−même, par exemple

```
SELECT * FROM ma_table AS a CROSS JOIN ma_table AS b ...
```

De plus, un alias est requis si la référence de la table est une sous−requête (voir Section 7.2.1.3).

Les parenthèses sont utilisées pour résoudre les ambiguïtés. L'instruction suivante affectera l'alias `b` au résultat de la jointure contrairement à l'exemple précédente :

```
SELECT * FROM (ma_table AS a CROSS JOIN ma_table) AS b ...
```

Une autre forme d'alias de tables donne des noms temporaires aux colonnes de la table ainsi qu'à la table :

```
FROM reference_table [AS]
alias ( colonne1
[, colonne2 [,
...]] )
```

Si le nombre d'alias de colonnes spécifié est plus petit que le nombre de colonnes dont dispose la table réelle, les colonnes suivantes ne sont pas renommées. Cette syntaxe est particulièrement utile dans le cas de jointure de même table ou dans le cas de sous−requêtes.

Quand un alias est appliqué à la sortie d'une clause `JOIN` en utilisant n'importe laquelle de ces formes, l'alias cache le nom original à l'intérieur du `JOIN`. Par exemple,

```
SELECT a.* FROM ma_table AS a JOIN ta_table AS b ON ...
```

est du SQL valide mais

```
SELECT a.* FROM (ma_table AS a JOIN ta_table AS b ON ...) AS c
```

n'est pas valide : l'alias de table `a` n'est pas visible en dehors de l'alias `c`.

#### 7.2.1.3. Sous−requêtes

Une sous−requête spécifiant une table dérivée doit être enfermée dans des parenthèses et *doit* se voir affecté un alias de table. (Voir Section 7.2.1.2.) Par exemple :

```
FROM (SELECT * FROM table1) AS nom_alias
```

Cette exemple est équivalent à `FROM table1 AS nom_alias`. Des cas plus intéressants, qui ne peuvent pas être réduit à une jointure pleine, arrivent quand la sous−requête implique un groupement ou une agrégat.

#### 7.2.1.4. Fonctions de table

Les fonctions de table sont des fonctions produisant un ensemble de lignes composées de types de données de base (types scalaires) ou de types de données composés (lignes de table). Elles sont utilisées comme une table, une vue ou une sous−requête de la clause `FROM` d'une requête. Les colonnes renvoyées par les fonctions de table peuvent être incluses dans une clause `SELECT`, `JOIN` ou `WHERE` de la même manière qu'une colonne de table, vue ou sous−requête.

Si une fonction de table renvoie un type de données de base, la colonne de résultat est nommée comme la fonction. Si la fonction renvoie un type composite, les colonnes résultants ont le même nom que les attributs individuels du type.

Une fonction de table pourrait avoir un alias dans la clause `FROM` mais elle pourrait être laissée sans alias. Si une fonction est utilisée dans la clause `FROM` sans alias, le nom de la fonction est utilisé comme nom de table résultante.

Quelques exemples :

```
CREATE TABLE truc (trucid int, trucsousid int, trucnom text);

CREATE FUNCTION recuptruc(int) RETURNS SETOF foo AS $$
    SELECT * FROM truc WHERE trucid = $1;
$$ LANGUAGE SQL;

SELECT * FROM recuptruc(1) AS t1;

SELECT * FROM truc
    WHERE trucsousid IN (select trucsousid from recuptruc(truc.trucid) z
                            where z.trucid = truc.trucid);

CREATE VIEW vue_recuptruc AS SELECT * FROM recuptruc(1);
SELECT * FROM vue_recuptruc;
```

Dans certains cas, il est utile de définir des fonctions de table pouvant renvoyer des ensembles de colonnes différentes suivant la façon dont elles sont appelées. Pour supporter ceci, la fonction de table est déclarée comme renvoyant le pseudotype `record`. Quand une telle fonction est utilisée dans une requête, la structure de ligne attendue doit être spécifiée dans la requête elle−même, de façon à ce que le système sache comment analyser et planifier la requête. Considérez cet exemple :

```
SELECT *
    FROM dblink('dbname=mabd', 'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

La fonction `dblink` exécute une requête distante (voir `contrib/dblink`). Elle déclare renvoyer le type `record` car elle pourrait être utilisée pour tout type de requête. L'ensemble de colonnes réelles doit être spécifié dans la requête appelant de façon à ce que l'analyseur sache, par exemple, ce que comment étendre `*`.

## 7.2.2. La clause `WHERE`

La syntaxe de *Clause WHERE* est

```
WHERE condition_recherche
```

où *condition_recherche* est toute expression de valeur (voir Section 4.2) renvoyant une valeur de type `boolean`.

Après la réalisation de la clause `FROM`, chaque ligne de la table virtuelle dérivée est vérifiée avec la condition de recherche. Si le résultat de la vérification est positive (true), la ligne est conservée dans la table de sortie, sinon (c'est–à–dire si le résultat est faux ou nul, la ligne est abandonnée. La condition de recherche référence typiquement au moins quelques colonnes de la table générée dans la clause `FROM` ; ceci n'est pas requis mais, dans le cas contraire, la clause `WHERE` n'aurait aucune utilité.

> **Note :** La condition de jointure d'une jointure interne peut être écrite soit dans la clause `WHERE` soit dans la clause `JOIN`. Par exemple, ces expressions de tables sont équivalentes :
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```
>
> et
>
> ```
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
> ```
>
> ou même peut–être
>
> ```
> FROM a NATURAL JOIN b WHERE b.val > 5
> ```
>
> Laquelle vous utilisez est plutôt une affaire de style. La syntaxe `JOIN` dans la clause `FROM` n'est probablement pas aussi portable vers les autres systèmes de gestion de bases de données SQL. Pour les jointures externes, il n'y a pas d'autres choix : elles doivent être faites dans la clause `FROM` clause. Une clause `ON`/`USING` d'une jointure externe n'est *pas* équivalente à une condition `WHERE` parce qu'elle détermine l'ajout de lignes (pour les lignes qui ne correspondent pas en entrée) ainsi que pour la suppression de lignes dans le résultat final.

Voici quelques exemples de clauses `WHERE` :

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

`fdt` est la table dérivée dans la clause `FROM`. Les lignes qui ne correspondent pas à la condition de la recherche de la clause `WHERE` sont éliminées de la table `fdt`. Notez l'utilisation de sous−requêtes scalaires en tant qu'expressions de valeurs. Comme n'importe quelle autre requête, les sous−requêtes peuvent employer des expressions de tables complexes. Notez aussi comment `fdt` est référencée dans les sous−requêtes. Qualifier `c1` comme `fdt.c1` est seulement nécessaire si `c1` est aussi le nom d'une colonne dans la table d'entrée dérivée de la sous−requête. Mais qualifier le nom de colonne ajoute à la clarté même lorsque cela n'est pas nécessaire. Cet exemple montre comment le nom de colonne d'une requête externe est étendue dans les requêtes internes.

## 7.2.3. Les clauses `GROUP BY` et `HAVING`

Après avoir passé le filtre `WHERE`, la table d'entrée dérivée peut être sujette à un regroupement en utilisant la clause `GROUP BY` et à une élimination de groupe de lignes avec la clause `HAVING`.

```
SELECT liste_selection
    FROM ...
    [WHERE ...]
    GROUP BY reference_colonne_regroupement
[,
reference_colonne_regroupement]...
```

La clause *Clause GROUP BY* est utilisée pour regrouper des lignes d'une table partageant les mêmes valeurs dans toutes les colonnes précisées. L'ordre dans lequel ces colonnes sont indiquées importe peu. L'effet est de combiner chaque ensemble de lignes partageant des valeurs communes en un seul groupe de ligne représentant toutes les lignes du groupe. Ceci se fait en éliminant les redondances dans la sortie et/ou pour calculer les agrégats s'appliquant à ces groupes. Par exemple :

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

Dans la seconde requête, nous n'aurions pas pu écrire `SELECT * FROM test1 GROUP BY x` parce qu'il n'existe pas une seule valeur pour la colonne `y` pouvant être associé avec chaque autre groupe. Les colonnes de regroupement peuvent être référencées dans la liste de sélection car elles ont une valeur constante unique par groupe.

En général, si une table est groupée, les colonnes qui ne sont pas utilisées dans le regroupement ne peuvent pas être référencées sauf dans les expressions d'agrégats. Voici un exemple d'expressions d'agrégat :

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Ici, `sum` est la fonction d'agrégat qui calcule une seule valeur pour le groupe entier. Plus d'informations sur les fonctions d'agrégats disponibles sont proposées dans Section 9.15.

> **Astuce :** Le regroupement sans expressions d'agrégats calcule effectivement l'ensemble les valeurs distinctes d'une colonne. Ceci peut aussi se faire en utilisant la clause `DISTINCT` (voir Section 7.3.3).

Voici un autre exemple : il calcule les ventes totales pour chaque produit (plutôt que le total des ventes sur tous les produits).

```
SELECT produit_id, p.nom, (sum(v.unite) * p.prix) AS ventes
    FROM produits p LEFT JOIN ventes v USING (produit_id)
    GROUP BY produit_id, p.nom, p.prix;
```

Dans cet exemple, les colonnes `produit_id`, `p.nom` et `p.prix` doivent être dans la clause `GROUP BY` car elles sont référencées dans la liste de sélection de la requête. (Suivant la façon dont est conçue la table produits, le nom et le prix pourraient être totalement dépendants de l'ID du produit, donc des regroupements supplémentaires pourraient théoriquement être inutiles mais ceci n'est pas encore implémenté.) La colonne `s.unite` n'a pas besoin d'être dans la liste `GROUP BY` car elle est seulement utilisée dans l'expression de l'agrégat (`sum(...)`) représentant les ventes d'un produit. Pour chaque produit, la requête renvoie une ligne de résumé sur les ventes de ce produit.

En SQL strict, `GROUP BY` peut seulement grouper les colonnes de la table source mais PostgreSQL étend ceci en autorisant `GROUP BY` à grouper aussi les colonnes de la liste de sélection. Grouper par expressions de valeurs au lieu de simples noms de colonnes est aussi permis.

Si une table a été groupée en utilisant la clause `GROUP BY` mais que seul certains groupes sont intéressants, la clause `HAVING` peut être utilisée, plus comme une clause `WHERE`, pour éliminer les groupes d'une table groupée. Voici la syntaxe :

```
SELECT liste_selection FROM ... [WHERE
...] GROUP BY ... HAVING
expression_booléenne
```

Les expressions de la clause `HAVING` peuvent référer à la fois aux expressions groupées et aux expressions non groupées (ce qui impliquent nécessairement une fonction d'agrégat).

Exemple :

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
```

```
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

De nouveau, un exemple plus réaliste :

```
SELECT produit_id, p.nom, (sum(v.unite) * (p.prix - p.cout)) AS profit
    FROM produits p LEFT JOIN ventes v USING (produit_id)
    WHERE v.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY produit_id, p.nom, p.prix, p.cout
    HAVING sum(p.prix * s.unite) > 5000;
```

Dans l'exemple ci–dessus, la clause `WHERE` sélectionne les lignes par une colonne qui n'est pas groupée (l'expression est vraie seulement pour les ventes des quatre dernières semaines) alors que la clause `HAVING` restreint la sortie aux groupes dont le total des ventes dépasse 5000. Notez que les expressions d'agrégats n'ont pas besoin d'être identiques dans toutes les parties d'une requête.

# 7.3. Listes de sélection

Comme montré dans la section précédente, l'expression de table pour la commande `SELECT` construit une table virtuelle intermédiaire en combinant les tables, vues, en éliminant les lignes, en groupant, etc. Cette table est finalement passée à la réalisation de la *liste de sélection*. Cette liste détermine les *colonnes* de la table intermédiaire à afficher.

## 7.3.1. Éléments de la liste de sélection

La forme la plus simple de liste de sélection est `*` qui émet toutes les colonnes que l'expression de table produit. Sinon, une liste de sélection est une liste d'expressions de valeurs séparées par des virgules (comme défini dans Section 4.2). Par exemple, cela pourrait être une liste des noms de colonnes :

```
SELECT a, b, c FROM ...
```

Les noms de colonnes `a`, `b` et `c` sont soit les noms actuels des colonnes des tables référencées dans la clause `FROM` soit les alias qui leur ont été donnés (voir l'explication dans Section 7.2.1.2). L'espace de nom disponible dans la liste de sélection est le même que dans la clause `WHERE` sauf si le regroupement est utilisé, auquel cas c'est le même que dans la clause `HAVING`.

Si plus d'une table a une colonne du même nom, le nom de la table doit aussi être donné comme dans

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

En travaillant avec plusieurs tables, il est aussi utile de demander toutes les colonnes d'une table particulière :

```
SELECT tbl1.*, tbl2.a FROM ...
```

(Voir aussi Section 7.2.2.)

Si une expression de valeur arbitraire est utilisée dans la liste de sélection, il ajoute conceptuellement une nouvelle colonne virtuelle dans la table renvoyée. L'expression de valeur est évaluée une fois pour chaque ligne avec une substitution des valeurs de lignes avec les références de colonnes. Mais les expressions de la liste de sélection n'ont pas à référencer les colonnes dans l'expression de la table de la clause FROM ; elles pourrait être des expressions arithmétiques constantes, par exemple.

## 7.3.2. Labels de colonnes

Les entrées dans la liste de sélection peuvent se voir affecter des noms pour la suite de l'exécution. La << suite de l'exécution >> dans ce cas est une spécification optionnelle du tri et l'application client (c'est–à–dire les en–têtes de colonne pour l'affichage). Par exemple :

```
SELECT a AS value, b + c AS sum FROM ...
```

Si aucun nom de colonne en sortie n'est spécifié en utilisant AS, le système affecte un nom par défaut. Pour les références de colonne simple, c'est le nom de la colonne référencée. Pour les appels de fonction, il s'agit du nom de la fonction. Pour les expressions complexes, le système générera un nom générique.

> **Note :** Le nom des colonnes en sortie est différent ici de ce qui est fait dans la clause FROM (voir Section 7.2.1.2). Ce tube vous permettra en fait de renommer deux fois la même colonne mais le nom choisi dans la liste de sélection est celui qui sera passé.

## 7.3.3. DISTINCT

Après la réalisation de la liste de sélection, la table résultant pourrait être optionnellement sujet à l'élimination des lignes dupliquées. Le mot clé DISTINCT est écrit directement après SELECT pour spécifier ceci :

```
SELECT DISTINCT liste_selection ...
```

(Au lieu de DISTINCT, le mot clé ALL peut être utilisé pour spécifier le comportement par défaut, la récupération de toutes les lignes.)

Évidemment, les deux lignes sont considérées distinctes si elles diffèrent dans au moins une valeur de colonne. Les valeurs NULL sont considérées égales dans cette comparaison.

Alternativement, une expression arbitraire peut déterminer quelles lignes doivent être considérées distinctes :

```
SELECT DISTINCT ON (expression [,
expression ...])
liste_selection ...
```

Ici, *expression* est une expression de valeur arbitraire, évaluée pour toutes les lignes. Les lignes dont toutes les expressions sont égales sont considérées comme dupliquées et seule la première ligne de cet ensemble est conservée dans la sortie. Notez que la << première ligne >> d'un ensemble est non prévisible sauf si la requête est triée sur assez de colonnes pour garantir une ordre unique des colonnes arrivant dans le filtre DISTINCT. (Le traitement de DISTINCT ON parvient après le tri de ORDER BY.)

La clause DISTINCT ON ne fait pas partie du standard SQL et est quelque fois considéré comme étant un mauvais style à cause de la nature potentiellement indéterminée de ses résultats. Avec l'utilisation judicieuse de GROUP BY et de sous−requêtes dans FROM, la construction peut être évitée mais elle représente souvent l'alternative la plus agréable.

## 7.4. Combiner des requêtes

Les résultats de deux requêtes peuvent être combinés en utilisant les opérations d'ensemble : union, intersection et différence. La syntaxe est

```
requete1 UNION [ALL]
requete2
requete1 INTERSECT [ALL]
requete2
requete1 EXCEPT [ALL]
requete2
```

*requete1* et *requete2* sont les requêtes pouvant utiliser toutes les fonctionnalités discutées ici. Les opérations d'ensemble peuvent aussi être combinées et chaînées, par exemple

```
requete1 UNION requete2
UNION requete3
```

signifie réellement

```
(requete1 UNION requete2)
UNION requete3
```

UNION ajoute effectivement le résultat de *requete2* au résultat de *requete1* (bien qu'il n'y ait pas de garantie qu'il s'agit de l'ordre dans lequel les lignes sont réellement renvoyées). De plus, il élimine les lignes dupliquées du résultat, de la même façon que DISTINCT, sauf si UNION ALL est utilisée.

INTERSECT renvoie toutes les lignes qui sont à la fois dans le résultat de *requete1* et dans le résultat de *requete2*. Les lignes dupliquées sont éliminées sauf si INTERSECT ALL est utilisé.

EXCEPT renvoie toutes les lignes qui sont dans le résultat de *requete1* mais pas dans le résultat de *requete2*. (Ceci est quelque fois appelé la *différence* entre deux requêtes.) De nouveau, les lignes dupliquées sont éliminées sauf si EXCEPT ALL est utilisé.

Pour calculer l'union, l'intersection ou la différence de deux requêtes, les deux requêtes doivent être << compatibles pour une union >>, ce qui signifie qu'elles doivent renvoyer le même nombre de colonnes et que les colonnes correspondantes doivent avoir des types de données compatibles, comme décrit dans Section 10.5.

## 7.5. Tri de lignes

Après qu'une requête ait produit une table en sortie (après que la liste de sélection ait été traitée), elle peut être optionnellement triée. Si le tri n'a pas été choisi, les lignes sont renvoyées dans un ordre non spécifié. L'ordre réel dans ce cas dépendra des types de plan de parcours et de jointure et de l'ordre sur le disque mais vous ne

devez pas vous y fier. Un tri particulier en sortie peut seulement être garantie si l'étape de tri est choisie explicitement.

La clause `ORDER BY` spécifie l'ordre de tri :

```
SELECT liste_selection
    FROM expression_table
    ORDER BY colonne1 [ASC | DESC]
[, colonne2 [ASC | DESC]
...]
```

*colonne1*, etc. réfèrent les colonnes de la liste de sélection. Ceci peut soit être le nom de sortie d'une colonne (voir Section 7.3.2) ou le numéro d'une colonne. Voici quelques exemples :

```
SELECT a, b FROM table1 ORDER BY a;
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, sum(b) FROM table1 GROUP BY a ORDER BY 1;
```

Comme une extension du standard SQL, PostgreSQL autorise aussi l'ordre des expressions arbitraires :

```
SELECT a, b FROM table1 ORDER BY a + b;
```

Les références des noms de colonnes dans la clause `FROM`, absentes de la liste de sélection, sont aussi autorisées :

```
SELECT a FROM table1 ORDER BY b;
```

Mais ces extensions ne fonctionnent pas dans les requêtes impliquant `UNION`, `INTERSECT` ou `EXCEPT` et ne sont pas portables dans les autres bases de données SQL.

Chaque spécification de colonne pourrait être suivie d'un `ASC` ou `DESC` optionnelle pour initialiser la direction du tri (ascendant ou descendant). L'ordre `ASC` est la valeur par défaut. L'ordre ascendant place les plus petites valeurs en premier où << plus petit >> est défini en terme de l'opérateur <. De façon similaire, l'ordre descendant est déterminé avec l'opérateur >. [4]

Si plus d'une colonne de tri est spécifiée, les entrées suivantes sont utilisées pour trier les lignes qui sont égales sous l'ordre imposé par les colonnes de tri précédent.

## 7.6. `LIMIT` et `OFFSET`

`LIMIT` et `OFFSET` vous permet de retrouver seulement une portion des lignes générées par le reste de la requête :

```
SELECT liste_selection
    FROM expression_table
    [LIMIT { numero | ALL }]
[OFFSET numero]
```

Si un nombre limite est donné, pas plus que ce nombre de lignes sera renvoyé (mais peut–être moins si la requête récupère moins de lignes). `LIMIT ALL` est indique à ne pas spécifier la clause `LIMIT`.

`OFFSET` indique de passer ce nombre de lignes avant de renvoyer les lignes restantes. `OFFSET 0` est identique à oublier la clause `OFFSET`. Si à la fois `OFFSET` et `LIMIT` apparaissent, alors les `OFFSET` lignes sont laissés avant de commencer le renvoi des `LIMIT` lignes.

Lors de l'utilisation de `LIMIT`, il est important d'utiliser une clause `ORDER BY` contraignant les lignes résultantes dans un ordre unique. Sinon, vous obtiendrez un sous−ensemble non prévisible de lignes de la requête. Vous pourriez demander des lignes 10 à 20 mais dans quel ordre ? L'ordre est inconnu si vous ne spécifiez pas `ORDER BY`.

L'optimiseur de requêtes prend `LIMIT` en compte lors de la génération d'un plan de requêtes, de façon à ce que vous obtenez différents plans (avec différents ordres de lignes) suivant ce que vous donnez à `LIMIT` et `OFFSET`. Du coup, utiliser des valeurs `LIMIT`/`OFFSET` différentes pour sélectionner des sous−ensembles différents d'un résultat de requête *donnera des résultats inconsistants* sauf si vous forcez un ordre de résultat prévisible avec `ORDER BY`. Ceci n'est pas un bogue ; c'est une conséquence inhérente du fait que le SQL ne promette par de délivrer les résultats d'une requête dans un ordre particulier sauf si `ORDER BY` est utilisé pour contraindre l'ordre.

Les lignes passées par une clause `OFFSET` devront toujours être traitées à l'intérieur du serveur ; du coup, un `OFFSET` important peut être inefficace.

# Chapitre 8. Types de données

PostgreSQL a un large choix de types de données disponibles nativement. Les utilisateurs peuvent ajouter de nouveaux types à PostgreSQL en utilisant la commande `CREATE TYPE`.

Tableau 8−1 montre tous les types de données généraux disponibles nativement. La plupart des types de données alternatifs listés dans la colonne << Alias >> sont les noms utilisés en interne par PostgreSQL pour des raisons historiques. De plus, certains types de données internes ou obsolètes sont disponibles, mais ils ne sont pas listés ici.

**Tableau 8−1. Types de données**

| Nom | Alias | Description |
|---|---|---|
| `bigint` | `int8` | Entier signé de 8 octets |
| `bigserial` | `serial8` | Entier de 8 octets à incrémentation automatique |
| `bit [ (n) ]` | | Suite de bits de longueur fixe |
| `bit varying [ (n) ]` | `varbit` | Suite de bits de longueur variable |
| `boolean` | `bool` | Booléen (Vrai/Faux) |
| `box` | | Boîte rectangulaire dans le plan |
| `bytea` | | Donnée binaire (<< tableau d'octets >>) |
| `character varying [ (n) ]` | `varchar [ (n) ]` | Suite de caractères de longueur variable |
| `character [ (n) ]` | `char [ (n) ]` | Suite de caractères de longueur fixe |
| `cidr` | | Adresse réseau IPv4 ou IPv6 |
| `circle` | | Cercle dans le plan |
| `date` | | Date du calendrier (année, mois, jour) |
| `double precision` | `float8` | Nombre à virgule flottante de double précision |
| `inet` | | Adresse d'ordinateur IPv4 ou IPv6 |
| `integer` | `int`, `int4` | Entier signé de 4 octets |
| `interval [ (p) ]` | | Intervalle de temps |
| `line` | | Ligne infinie dans le plan |
| `lseg` | | Segment de droite dans le plan |

| | | |
|---|---|---|
| macaddr | | adresse MAC |
| money | | montant d'une devise |
| numeric [ (*p*, *s*) ] | decimal [ (*p*, *s*) ] | Nombre exact de la précision indiquée |
| path | | Chemin géométrique dans le plan |
| point | | Point géométrique dans le plan |
| polygon | | Chemin géométrique fermé dans le plan |
| real | float4 | Nombre à virgule flottante de simple précision |
| smallint | int2 | Entier signé de 2 octets |
| serial | serial4 | Entier de 4 octets à incrémentation automatique |
| text | | Chaîne de caractères de longueur variable |
| time [ (*p*) ] [ without time zone ] | | Heure du jour |
| time [ (*p*) ] with time zone | timetz | Heure du jour, avec fuseau horaire |
| timestamp [ (*p*) ] [ without time zone ] | | Date et heure |
| timestamp [ (*p*) with time zone | timestamptz | Date et heure, avec fuseau horaire |

**Compatibilité :** Les types suivants sont conformes à la norme SQL: bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (avec et sans fuseau horaire), timestamp (avec et sans fuseau horaire).

Chaque type de données a une représentation externe déterminée par ses fonctions d'entrée et de sortie. De nombreux type de données internes ont un format externe évident. Cependant, certains types sont soit spécifiques à PostgreSQL, comme les chemins géométriques, ou ont différents formats possibles, comme les types de données de date et d'heure. Certaines fonctions d'entrée et de sortie ne sont pas inversables: Le résultat de la fonction de sortie peut perdre de la précision comparé à l'entrée initiale.

# 8.1. Types numériques

Les types numériques sont constitués d'entiers de 2, 4 ou 8 octets, de nombre à virgule flottante de 4 ou 8 octets, et de décimaux à précision sélectionnable.

**Tableau 8–2. Types numériques**

| Nom | | Description | Étendue |
|---|---|---|---|

| | Taille de stockage | | |
|---|---|---|---|
| smallint | 2 octets | entier de faible étendue | –32768 à +32767 |
| integer | 4 octets | entiers les plus courants | –2147483648 à +2147483647 |
| bigint | 8 octets | grands entiers | –9223372036854775808 à 9223372036854775807 |
| decimal | variable | Précision indiquée par l'utilisateur. Valeurs exactes | pas de limite |
| numeric | variable | Précision indiquée par l'utilisateur. Valeurs exactes | pas de limite |
| real | 4 octets | Précision variable. Valeurs inexactes | précision de 6 décimales |
| double precision | 8 octets | Précision variable. Valeurs inexactes | précision de 15 décimales |
| serial | 4 octets | entier à incrémentation automatique | 1 à 2147483647 |
| bigserial | 8 octets | entier de grande taille à incrémentation automatique | 1 à 9223372036854775807 |

La syntaxe des constantes pour les types numériques est décrite dans Section 4.1.2. Les types numériques ont un ensemble complet d'opérateurs arithmétiques et de fonctions. Référez vous à Chapitre 9 pour plus d'informations. Les sections suivantes décrivent ces types en détail.

## 8.1.1. Types entiers

Les types `smallint`, `integer`, et `bigint` stockent des nombres entiers, c'est à dire sans décimale, de différentes étendues. Toute tentative d'y stocker une valeur trop grande ou trop petite produit une erreur.

Le type `integer` est le plus courant. Il offre un bon compromis entre capacité, espace utilisé et performance. Le type `smallint` n'est utilisé que si l'économie d'espace disque est le premier critère de choix. Le type `bigint` ne doit être utilisé que si le type `integer` n'offre pas une étendue suffisante, car le type `integer` est nettement plus rapide.

Le type `bigint` peut ne pas fonctionner correctement sur toutes les plates–formes, car il repose sur la capacité du compilateur à supporter les entiers de 8 octets. Sur une machine qui ne les supporte pas, `bigint` se comporte comme `integer` (mais prend bien huit octets d'espace de stockage). Ceci dit, nous ne connaissons pas de plate–forme raisonnable sur laquelle il en va ainsi.

SQL ne spécifie que les types de données `integer` (ou `int`) et `smallint`. Le type `bigint`, et les noms de types `int2`, `int4`, et `int8` sont des extensions, qui sont partagées avec d'autres systèmes de bases de données SQL.

## 8.1.2. Nombre à précision arbitraire

Le type `numeric` peut stocker des nombres avec jusqu'à 1000 chiffres significatifs et effectuer des calculs exacts. Il est spécialement recommander pour stocker les montants financiers et autres quantités pour lesquelles l'exactitude est indispensable. Néanmoins, l'arithmétique sur les valeurs `numeric` est très lent

comparé aux types entiers ou au type à virgule flottante décrits dans la section suivante.

Dans ce qui suit, on utilise les termes suivants: L'*échelle* d'un `numeric` est le nombre de chiffres décimaux de la partie fractionnaire. La *précision* d'un `numeric` est le nombre total de chiffres significatifs dans le nombre entier, c'est à dire avant et après la virgule. Donc, le nombre 23,5141 a une précision de 6 et une échelle de 4. On peut considérer que les entiers ont une échelle de 0.

La précision maximum et l'échelle maximum d'une colonne `numeric` peuvent être réglés. Pour déclarer une colonne de type numérique, il faut utiliser la syntaxe.

```
NUMERIC(précision, échelle)
```

La précision pour être strictement positive, l'échelle positive ou NULL. Alternativement,

```
NUMERIC(précision)
```

indique une échelle de 0. Préciser

```
NUMERIC
```

sans précision ni échelle crée une colonne dans laquelle on peut stocker des valeurs de n'importe quelle précision ou échelle, jusqu'à la limite de précision. Une colonne de ce type ne forcera aucune valeur entrée à une précision particulière, alors que les colonnes `numeric` avec une échelle forcent les valeurs entrées à cette échelle. (Le standard SQL demande une précision par défaut de 0, c'est à dire de forcer la transformation en entiers. Nous trouvons ça inutile. Si vous êtes soucieux de portabilité, précisez toujours la précision et l'échelle explicitement.)

Si l'échelle d'une valeur est supérieure à l'échelle d'une colonne, le système tentera d'arrondir la valeur au nombre de chiffres après la virgule spécifiée. Ensuite, si le nombre de chiffres à gauche du point décimal dépasse la précision déclarée moins l'échelle déclarée, une erreur est levée.

Les valeurs numériques sont stockées physiquement sans zéro avant ou après. Du coup, la précision déclarée et l'échelle de la colonne sont des valeurs maximums, pas des allocations fixes. (En ce sens, le type numérique est plus proche de `varchar(n)` que de `char(n)`.)

En plus des valeurs numériques ordinaires, le type `numeric` autorise la valeur spéciale `NaN`, signifiant << not−a−number >> (NdT : pas un nombre). Toute opération sur `NaN` renvoie un autre `NaN`. En écrivant cette valeur comme une constante dans une requête SQL, vous devez placer des guillemets autour. Par exemple, `UPDATE table SET x = 'NaN'`. En saisie, la chaîne `NaN` est reconnue quelque soit la casse utilisée.

Les types `decimal` et `numeric` sont équivalents. Les deux types sont dans le standard SQL.

## 8.1.3. Types à virgule flottante

Les types de données `real` et `double precision` sont des types numériques à précision variable inexacts. En pratique, ils sont généralement conformes à la norme IEEE 754 pour l'arithmétique binaire à virgule flottante (respectivement simple et double précision), dans la mesure où les processeurs, le système d'exploitation et le compilateur les supportent.

Inexact signifie que certaines valeurs ne peuvent être converties exactement dans le format interne, et sont stockées sous forme d'approximations, si bien que stocker puis réafficher ces valeurs peut faire apparaître de légers écarts. Prendre en compte ces erreurs et la façon dont elles se propagent au cours des calculs est le sujet d'une branche entière des mathématiques et de l'informatique. Nous n'en dirons pas plus que ce qui suit:

- Si vous avez besoin d'un stockage et de calculs exacts, comme pour les valeurs monétaires, utilisez plutôt le type `numeric`.
- Si vous voulez faire des calculs compliqués avec ces types pour quoi que ce soit d'important, et particulièrement si vous comptez sur certains comportements aux limites (infinis, zéro), alors vous devriez étudier le comportement de votre plate−forme avec soin.
- Tester l'égalité de deux valeurs à virgule flottante peux ne pas donner le résultat attendu.

Sur la plupart des plates−formes, le type `real` a une étendue d'au moins 1E−37 à 1E37 avec une précision d'au moins 6 chiffres décimaux. Le type `double precision` a généralement une étendue de 1E−307 à 1E+308 avec une précision d'au moins 15 chiffres. Les valeurs trop grandes ou trop petites produisent une erreur. Un arrondi peut avoir lieu si la précision d'un nombre en entrée est trop grande. Les nombres trop proches de zéro qui ne peuvent être représentés autrement que par zéro produisent une erreur (underflow).

En plus des valeurs numériques ordinaires, les types à virgule flottante ont plusieurs valeurs spéciales :

```
Infinity
-Infinity
NaN
```

Elles représentent les valeurs spéciales de l'IEEE 754 , respectivement << infinity >> (NdT : infini), << negative infinity >> (NdT : infini négatif) et << not−a−number >> (NdT : pas un nombre). (Sur une machine dont l'arithmétique à virgule flottante ne suit pas l'IEEE 754, ces valeurs ne fonctionneront probablement pas comme c'est attendu.) Lors de la saisie de ces valeurs en tant que constantes dans une commande SQL, vous devez placer des guillemets autour. Par exemple, `UPDATE table SET x = 'Infinity'`. En entrée, ces valeurs sont reconnues quelque soit la casse utilisée.

PostgreSQL autorise aussi la notation `float` du standard SQL, ainsi que `float(`*p*`)` pour indiquer des types numériques inexacts. Ici, *p* indique la précision minimale acceptable en chiffres binaires. PostgreSQL accepte `float(1)` à `float(24)`, transformés en type `real` et `float(25)` to `float(53)`, transformés en type `double precision`. Toute valeur de *p* hors de la zone de valeurs possible produit une erreur. `float` sans précision est compris comme `double precision`.

> **Note :** Avant PostgreSQL 7.4, la précision d'un `float(`*p*`)` était supposée indiquer une précision en chiffres décimaux. Nous l'avons corrigée pour respecter le standard SQL, qui indique que la précision est indiquée en chiffres binaires. L'affirmation que les `real` et les `double precision` ont exactement 24 et 53 bits dans la mantisse est correcte pour les implémentations des nombres à virgule flottante respectant le standard IEEE. Sur les plates−formes non−IEEE, c'est peut−être un peu sous−estimé, mais pour plus de simplicité, la gamme de valeurs pour *p* est utilisée sur toutes les plates−formes.

## 8.1.4. Types Série

Les types de données `serial` et `bigserial` ne sont pas de vrais types, mais plutôt un raccourci de notation pour décrire des colonnes d'identifiants uniques (similaires à la propriété `AUTO_INCREMENT` utilisée par d'autres SGBD). Dans la version actuelle, indiquer

```
CREATE TABLE nom_de_table (
    nom_de_colonne SERIAL
);
```

est équivalent à écrire :

```
CREATE SEQUENCE nom_de_table_colname_seq;
CREATE TABLE nom_de_table (
    nom_de_colonne integer DEFAULT nextval('nom_de_table_nom_de_colonne_seq') NOT NULL
);
```

Ainsi, nous avons créé une colonne d'entiers et fait en sorte que ses valeurs par défaut soient assignées par un générateur de séquence. Une contrainte `NOT NULL` est ajoutée pour s'assurer qu'une valeur NULL ne puisse pas être explicitement insérée. Dans la plupart des cas, vous voudrez aussi ajouter une contrainte `UNIQUE` ou `PRIMARY KEY` pour interdire que des doublons soient créés par accident, mais ce n'est pas automatique.

> **Note :** Avant PostgreSQL 7.3, `serial` sous entendait `UNIQUE`. Ce n'est plus automatique. Si vous souhaitez qu'une colonne serial soit unique ou soit une clé primaire, il faut le préciser, comme pour un autre type.

Pour insérer la valeur suivante de la séquence dans la colonne `serial`, il faut faire en sorte d'utiliser la valeur par défaut de la colonne. Cela peut se faire de deux façons: soit en excluant cette colonne de la liste des colonnes de la commande `INSERT`, ou en utilisant le mot clé `DEFAULT`.

Les types `serial` et `serial4` sont identiques: ils créent tous les deux des colonnes `integer`. Les types `bigserial` et `serial8` fonctionnent de la même façon, et créent des colonnes `bigint`. `bigserial` doit être utilisé si vous pensez utiliser plus de $2^{31}$ identifiants dans toute la vie de la table.

La séquence créée pour une colonne `serial` est automatiquement supprimée quand la colonne correspondante est supprimée, et ne peut l'être autrement. (Ce n'était pas le cas avant la version 7.3 de PostgreSQL. Notez que ce lien de suppression automatique de séquence ne fonctionnera pas pour une base restaurée d'une sauvegarde SQL (dump) antérieure à la version 7.3. La sauvegarde ne contient en effet pas l'information nécessaire à l'établissement du lien de dépendance.) De plus, ce lien de dépendance n'est mis que pour la colonne de type `serial` elle même. Si d'autres colonnes référencent la séquence (par exemple en appellent la fonction `nextval`), elles ne peuvent plus fonctionner si la séquence est supprimée. Utiliser le type `serial` de cette façon n'est pas recommandé. Si vous souhaitez remplir plusieurs colonnes avec le même générateur de séquence, créez la séquence indépendamment.

## 8.2. Types monétaires

> **Note :** Le type `money` est obsolète. Utilisez plutôt les types `numeric` ou `decimal`, en combinaison avec la fonction `to_char`.

Le type `money` stocke un montant d'une devise avec un nombre fixe de chiffres après la virgule. Voir Tableau 8–3. De nombreux formats sont acceptés en entrée, dont les entiers et les nombre à virgule flottante ainsi que les formats classiques de devises, comme `'$1,000.00'`. Le format de sortie est généralement ce dernier, mais dépend de la localisation.

**Tableau 8–3. Types monétaires**

| Nom | Taille de stockage | Description | Étendue |
|---|---|---|---|
| money | 4 octets | montants de devise | –21474836.48 à +21474836.47 |

# 8.3. Types caractères

**Tableau 8–4. Types caractères**

| Nom | Description |
|---|---|
| `character varying(`*n*`)`, `varchar(`*n*`)` | Longueur variable avec limite |
| `character(`*n*`)`, `char(`*n*`)` | longueur fixe, comblé avec des espaces |
| `text` | longueur variable illimitée |

Tableau 8–4 montre les types génériques disponibles dans PostgreSQL.

SQL définit deux types de caractères principaux: `character varying(`*n*`)` et `character(`*n*`)`, où *n* est un entier positif. Ces deux types peuvent stocker des chaînes de caractères de taille inférieure ou égale à *n*. Une tentative d'insérer une chaîne trop longue donnera une erreur, à moins que les caractères en trop soient tous des espaces, auquel cas la chaîne sera tronquée à la taille maximale. (Cette exception un rien étrange est imposée par la norme SQL). Si la chaîne à stocker est plus petite que la taille déclarée, elle sera complétée par des espaces pour le type `character`, et elle sera stockée telle quelle pour le type `character varying`.

Si vous transtypez (cast) explicitement une valeur en `character varying(`*n*`)` ou en `character(`*n*`)`, alors une chaîne trop longue sera tronquée à *n* caractères sans que cela génère d'erreur. (Ce comportement est aussi imposé par la norme SQL.)

> **Note :** Avant PostgreSQL 7.2, les chaînes trop longues étaient toujours tronquées sans générer d'erreur, que ce soit par une transformation explicite ou implicite.

Les notations `varchar(`*n*`)` et `char(`*n*`)` sont des alias pour `character varying(`*n*`)` et `character(`*n*`)`, respectivement. `character` sans indication de taille est équivalent à `character(1)`. Si `character varying` est utilisé sans indicateur de taille, le type accepte des chaînes de toutes tailles. Il s'agit là d'une spécificité de PostgreSQL.

De plus, PostgreSQL propose aussi le type `text`, qui permet de stocker des chaînes de n'importe quelle taille. Bien que le type `text` ne soit pas dans le standard SQL, plusieurs autres systèmes de gestion de bases de données SQL le proposent aussi.

Les valeurs de type `character` sont physiquement alignées avec des espaces pour la longueur *n* spécifiée, et sont stockées et affichées de cette façon. Néanmoins, les espaces d'alignement sont traités sémantiquement sans signification. Les espaces en fin ne sont pas utilisés lors de la comparaison de deux valeurs de type `character`, et ils seront supprimés lors de la conversion d'une valeur `character` en un des types chaîne. Notez que les espaces en fin *sont* ont sémantiquement une signification pour les valeurs de type `character varying` et `text`.

L'espace de stockage utilisé pour les données de ces types est de 4 octets en plus de la taille de la chaîne, plus le remplissage dans le cas du type `character`. Les grandes chaînes sont automatiquement compressées par

le système, si bien que l'espace effectivement utilisé peut être inférieur. Les grandes chaînes sont aussi stockées dans des tables d'arrière plan, afin de ne pas ralentir l'accès aux autres colonnes plus petites. Dans tous les cas, la taille maximale possible pour une chaîne de caractères est de l'ordre 1 Go. (La taille maximale pour *n* dans la déclaration de type est plus petite que cela. Il ne serait pas très utile de le changer parce qu'avec l'encodage des caractères sur plusieurs octets, le nombre de caractères et d'octets peuvent être très différents. Si vous voulez stocker de longues chaînes sans limite de taille spécifique, utilisez le type `text` ou le type `character varying` sans indiquer de taille, plutôt que d'indiquer une limite de taille arbitraire.)

> **Astuce :** Il n'y a pas de différence de performance entre ces trois types, à part la place disque supplémentaire pour le type qui remplit les vides avec des espaces. Bien que `character(n)` a des avantages en terme de performance dans certains autres systèmes de bases de données, il ne dispose pas de ce type d'avantages dans PostgreSQL. Dans la plupart des situations, les types `text` ou `character varying` devraient être utilisés à leur place.

Voir Section 4.1.2.1 pour avoir plus d'informations sur la syntaxe des littéraux de chaînes, et Chapitre 9 pour avoir des informations sur les opérateurs et les fonctions. L'ensemble de caractères de la base de données détermine l'ensemble de caractères utilisé pour stocker les valeurs texte ; pour plus d'informations sur le support des ensembles de caractères, référez−vous à Section 20.2.

**Exemple 8−1. Utilisation des types caractères**

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- (1)
  a   | char_length
------+-------------
 ok   |           2

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('bien      ');
INSERT INTO test2 VALUES ('trop long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('trop long'::varchar(5)); -- troncature explicite
SELECT b, char_length(b) FROM test2;
   b   | char_length
-------+-------------
 ok    |           2
 bien  |           5
 trop  |           5
```

**(1)**

La fonction `char_length` est décrite dans Section 9.4.

Il y a deux autres types caractères de taille constante dans PostgreSQL, décrits dans Tableau 8−5. Le type `name` existe *seulement* pour le stockage des identifiants dans les catalogues systèmes, et n'est pas destiné à être utilisé par les utilisateurs normaux. Sa taille est actuellement définie à 64 octets (63 utilisables plus le terminateur), mais doit être référencée en utilisant la constante `NAMEDATALEN`. La taille est définie à la compilation (et est donc ajustable pour des besoins particuliers). La taille maximale par défaut pourrait changer dans une prochaine version. Le type `"char"` (notez les guillemets) est différent de `char(1)` car il n'utilise qu'un seul octet de stockage. Il est utilisé dans les catalogues systèmes comme un type d'énumération économique.

**Tableau 8–5. Types caractères spéciaux**

| Nom | Taille de stockage | Description |
|---|---|---|
| `"char"` | 1 octet | type interne de 1 caractère |
| `name` | 64 octets | type interne pour les noms d'objets |

# 8.4. Types de données binaires

Le type de données `bytea` permet de stocker des chaînes binaires; voir Tableau 8–6.

**Tableau 8–6. Types de données binaires**

| Nom | Espace de stockage | Description |
|---|---|---|
| `bytea` | 4 octets plus la taille de la chaîne binaire à stocker | Chaîne binaire de longueur variable |

Une chaîne binaire est une séquence d'octets. Les chaînes binaires se distinguent des chaînes de caractères par deux caractéristiques: D'abord, les chaînes binaires permettent de stocker des octets de valeurs zéro, et autres caractères << non imprimables >> (habituellement définis comme des octets en dehors de l'échelle de 32 à 126). Les chaînes de caractères interdisent les octets de valeur zéro et interdisent aussi toute valeur d'octet et séquence d'octets invalide suivant le codage de l'ensemble de caractères sélectionné pour la base de données. Ensuite, les opérations sur les chaînes binaires traitent réellement les octets alors que le traitement de chaînes de caractères dépend de la configuration de la locale. En bref, les chaînes binaires sont appropriées pour le stockage de données que le développeur considère comme des << octets bruts >> alors que les chaînes de caractères sont appropriées pour stocker du texte.

Lors de la saisie de valeurs de type `bytea`, certaines valeurs d'octets *doivent* être préparées avec des caractères d'échappement(mais toutes les valeurs *peuvent* l'être) lorsqu'elles sont font partie d'une chaîne littérale dans une commande SQL. En général, pour échapper un octet, il faut le convertir en nombre octal sur 3 caractères, précédés par deux antislashs. Tableau 8–7 affiche les caractères qui doivent être échappés, et donne les séquences d'échappement possibles.

**Tableau 8–7. `bytea` Octets littéraux à échapper**

| Valeur décimale de l'octet | Description | Représentation échappée d'entrée | Exemple | Représentation de sortie |
|---|---|---|---|---|
| 0 | octet zéro | `'\\000'` | `SELECT '\\000'::bytea;` | `\000` |
| 39 | apostrophe | `'\''` ou `'\\047'` | `SELECT '\''::bytea;` | `'` |
| 92 | antislash | `'\\'` ou `'\\134'` | `SELECT '\\'::bytea;` | `\\` |
| de 0 à 31 et de 127 à 255 | octets << non affichables >> | `'\\`*xxx*`'` (valeur octal) | `SELECT '\\001'::bytea;` | `\001` |

La nécessité d'échapper les octets << non affichables >> varie en fait suivant les paramétrages de la locale. Quelques fois, vous pouvez vous en sortir sans les échapper. Notez que le résultat de chacun des exemples de

Tableau 8−7 était d'une longueur exacte d'un octet, même si la représentation en sortie de l'octet zéro et de l'antislash font plus d'un caractère.

La raison pour laquelle il faut écrire autant de antislashs, comme indiqué dans Tableau 8−7, est qu'une chaîne binaire doit passer à travers deux phases d'analyse dans le serveur PostgreSQL. Le premier antislash de chaque paire est vu comme un caractère d'échappement par l'analyseur de littéral chaîne et est donc consommé, laissant le second antislash de la paire. Le antislash restant est compris par la fonction d'entrée de PostgreSQL comme le début d'une valeur octale sur trois caractères, ou comme l'échappement d'un autre antislash. Par exemple, un littéral de chaînes passé au serveur comme `'\\001'` devient `\001` après être passé à travers l'analyseur de littéral chaîne. Le `\001` est envoyé à la fonction d'entrée de `bytea`, qui le convertit en un octet simple ayant une valeur décimale de 1. Notez que le caractère apostrophe n'est pas traité spécialement par `bytea`, et suit donc les règles normales pour les littéraux de chaîne. Voir aussi Section 4.1.2.1.)

Les octets de `Bytea` sont aussi échappés en sortie. En général, chaque octet << non−imprimable >> est converti en équivalent octal sur trois caractères et précédé d'un antislash. La plupart des caractères << imprimables >> sont représentés par leur représentation standard dans le jeu de caractère du client. Les octets de valeur décimale 92 (antislash) ont une représentation alternative spéciale. Les détails sont dans Tableau 8−8.

**Tableau 8−8. `bytea` Octets échappés en sortie**

| Valeur décimale de l'octet | Description | Représentation de sortie échappée | Exemple | Résultat de sortie |
|---|---|---|---|---|
| 92 | antislash | `\\` | `SELECT '\\134'::bytea;` | `\\` |
| 0 à 31 et 127 à 255 | octets<< non−imprimables | *xxx* (valeur octale) | `SELECT '\\001'::bytea;` | `\001` |
| 32 à 126 | octets << imprimables >> | Représentation de l'ensemble de caractères du client | `SELECT '\\176'::bytea;` | ~ |

En fonction du client que vous utilisez pour accéder à PostgreSQL, vous pouvez avoir un travail d'échappement supplémentaire à effectuer pour échapper et déséchapper les chaînes `bytea`. Par exemple, il se peut que vous deviez échapper les sauts de lignes et retours à la ligne si votre programme client les traduit automatiquement.

Le standard SQL définit un type binaire différent, appelé `BLOB` ou `BINARY LARGE OBJECT`. Le format d'entrée est différent de celui du `bytea`, mais les fonctions et opérateurs fournis sont globalement les mêmes.

# 8.5. Types date/heure

PostgreSQL supporte l'ensemble des types date et heure de SQL, montrés dans Tableau 8−9. Les opérations disponibles sur ces types de données sont décrit dans Section 9.9.

**Tableau 8−9. Types date et heure**

| Nom | Taille de stockage | Description | Valeur minimale | Valeur maximale | Résolution |
|---|---|---|---|---|---|
| `timestamp [ (p) ] [ without time zone ]` | 8 octets | date et heure | 4713 avant JC | 5874897 après JC | 1 microseconde / 14 chiffres |
| `timestamp [ (p) ] with time zone` | 8 octets | date et heure, avec fuseau horaire | 4713 avant JC | 5874897 après JC | 1 microseconde / 14 chiffres |
| `interval [ (p) ]` | 12 octets | intervals de temps | −178000000 années | 178000000 années | 1 microseconde / 14 chiffres |
| `date` | 4 octets | dates seulement | 4713 avant JC | 32767 après JC | 1 jour |
| `time [ (p) ] [ without time zone ]` | 8 octets | heures seulement | 00:00:00.00 | 23:59:59.99 | 1 microseconde / 14 chiffres |
| `time [ (p) ] with time zone` | 12 octets | heures seulement, avec fuseau horaire | 00:00:00.00+12 | 23:59:59.99−12 | 1 microseconde / 14 chiffres |

**Note :** Avant PostgreSQL 7.3, écrire seulement `timestamp` était équivalent à `timestamp with time zone`. Ceci a été changé pour une meilleure compatibilité avec le standard SQL.

`time`, `timestamp`, et `interval` acceptent une précision optionnelle *p*, qui précise le nombre de chiffres après la virgule pour les secondes. Par défaut, il n'y a pas de limite explicite à la précision. Les valeurs acceptées pour *p* vont de 0 à 6 pour les types `timestamp` et `interval`.

**Note :** Quand les valeurs `timestamp` sont stockées en tant que nombre à virgule flottante (actuellement le défaut), la limite réelle de la précision pourrait être inférieure à 6. Les valeurs `timestamp` sont stockées en tant que nombre de secondes avant ou après le 1er janvier 2000 à minuit. La précision par microseconde est obtenue pour les dates proches du 1er janvier 2000 mais la précision se dégrade pour les dates suivantes. Quand les valeurs `timestamp` sont stockées en tant qu'entier sur huit octets (une option au moment de la compilation), la précision en microseconde est disponible pour toute l'étendue des valeurs. Néanmoins, les valeurs de type timestamp codées avec des entiers sur huit octets ont une échelle de date plus limitées que celle indiquée ci−dessus : de 4713 avant Jésus−Christ à 294276 après Jésus−Christ. La même option de compilation détermine si les valeurs `time` et `interval` sont stockées en tant que nombre à virgule flottante ou en tant qu'octet sur huit chiffres. Dans le cas de la virgule flottante, les valeurs `interval` larges voient leur précision se dégrader au fur et à mesure que l'intervalle croît.

**Note :** Lorsque les valeurs de type `timestamp` sont stockées comme des nombres à virgule flottante de précision double (ce qui est le cas par défaut), la limite de précision effective peut être inférieure à 6. Les valeurs de `timestamp` sont stockées comme des secondes depuis le avant ou après minuit le 1er janvier 2001. La précision de la milliseconde est atteinte pour quelques années autour de cette date mais la précision se dégrade si on s'en éloigne. Lorsque les valeurs de `timestamp` sont stockés comme des entiers sur 8 octets, ce qui est une option à la compilation, la précision de la microseconde est disponible sur tout l'intervalle des valeurs. Néanmoins, cet intervalle est encore plus limité :: de 4713 avant Jésus Christ à 294276 après Jésus Christ.

**Note :** Avant PostgreSQL 7.3, indiquer juste `timestamp` était équivalent à `timestamp with time zone`. Ce comportement a été changé en respect du standard SQL.

Pour les types `time`, l'intervalle accepté pour *p* est de 0 à 6 lorsque les entiers sur 8 octets sont utilisés, ou de 0 à 10 lorsque le stockage se fait sous forme de nombre à virgule flottante.

Le type `time with time zone` est défini dans le standard SQL, mais sa définition lui prête des propriétés qui font douter de son utilité. Dans la plupart des cas, une combinaison de `date`, `time`, `timestamp without time zone`, et `timestamp with time zone` devrait permettre de résoudre toutes les fonctionnalités de date et heure nécessaires à une application.

Les types `abstime` et `reltime` sont des types de précision moindre, utilisés en interne. Il n'est pas recommandé de les utiliser dans de nouvelles applications. Au contraire, il est souhaitable de migrer l'existant vers un autre type approprié. Ces types internes pourraient disparaître dans une future version.

## 8.5.1. Entrée des dates et heures

La saisie de dates et heures et possible dans la plupart des formats raisonnables, dont ISO8601, compatible SQL, traditionnel POSTGRES, et d'autres. Pour certains formats, l'ordre des jours, mois et années en entrée est ambigu. Il est alors possible de préciser l'ordre attendu pour ces champs. Réglez le paramètre DateStyle à `MDY` pour choisir une interprétation mois−jour−année, à `DMY` pour jour−mois−année, à `YMD` pour année−mois−jour.

PostgreSQL est plus flexible que la norme SQL ne l'exige pour la manipulation des dates et des heures. Voir Annexe B pour connaître les règles exactes de reconnaissance des dates et heures, ainsi que les formats de champs texte comme les mois, les jours de la semaine et les fuseaux horaires.

Rappelez vous que chaque littéral date ou heure à entrer doit être mis entre apostrophes, comme les chaînes de caractères. Référez vous à Section 4.1.2.5 pour plus d'information. SQL requière la syntaxe suivante:

```
type [ (p) ] 'value'
```

où *p* dans la spécification optionnelle de précision est un entier correspondant au nombre de chiffres après la virgule dans le champ secondes. La précision peut être précisée pour les types `time`, `timestamp`, et `interval`. Les valeurs admissibles sont mentionnées plus haut. Si aucune précision n'est indiquée dans une spécification de constante, elle prend la précision de la valeur littérale.

### 8.5.1.1. Dates

Tableau 8−10 montre les formats de date possibles pour les entrées de type `date`.

**Tableau 8−10. Saisie de date**

| Exemple | Description |
|---|---|
| January 8, 1999 | sans ambiguïté quel que soit le style de date (`datestyle`) |
| 1999−01−08 | ISO−8601; 8 janvier, quel que soit le mode (format recommandé) |
| 1/8/1999 | 8 janvier en mode MDY`MDY`; 1er Août en mode `DMY` |

| | |
|---|---|
| 1/18/1999 | 18 janvier en mode MDY; rejeté dans les autres modes |
| 01/02/03 | 2 janvier 2003 en mode MDY; 1er février 2003 en mode DMY; 3 février 2003 en mode YMD |
| 1999−Jan−08 | 8 janvier dans tous les modes |
| Jan−08−1999 | 8 janvier dans tous les modes |
| 08−Jan−1999 | 8 janvier dans tous les modes |
| 99−Jan−08 | 8 janvier en mode YMD, erreur sinon |
| 08−Jan−99 | 8 janvier, sauf en mode YMD: erreur |
| Jan−08−99 | 8 janvier, sauf en mode YMD: erreur |
| 19990108 | ISO−8601; 8 janvier 1999 dans tous les modes |
| 990108 | ISO−8601; 8 janvier 1999 dans tous les modes |
| 1999.008 | Année et jour dans l'année |
| J2451187 | Jour du calendrier Julien |
| January 8, 99 BC | année 99 avant Jésus Christ |

#### 8.5.1.2. Heures

Les types heure−du−jour sont `time [ (p) ] without time zone` et `time [ (p) ] with time zone`. Écrire juste `time` est équivalent à `time without time zone`

Les valeurs d'entrée valides pour ces types sont constituées d'une heure du jour suivi d'un fuseau horaire optionnel. (voir Tableau 8−11 et Tableau 8−12.) Si un fuseau est précisé pour le type `time without time zone`, il est ignoré sans message d'erreur.

**Tableau 8−11. Saisie d'heure**

| Exemple | Description |
|---|---|
| 04:05:06.789 | ISO 8601 |
| 04:05:06 | ISO 8601 |
| 04:05 | ISO 8601 |
| 040506 | ISO 8601 |
| 04:05 AM | Identique à 04:05; AM n'affecte pas la valeur |
| 04:05 PM | Identique à 16:05; l'heure doit être <= 12 |
| 04:05:06.789−8 | ISO 8601 |
| 04:05:06−08:00 | ISO 8601 |
| 04:05−08:00 | ISO 8601 |
| 040506−08 | ISO 8601 |
| 04:05:06 PST | fuseau horaire précisé en lettres |

**Tableau 8−12. Saisie des zones de date**

| Exemple | Description |
|---|---|
| PST | Heure standard du Pacifique (Pacific Standard Time) |

| | |
|---|---|
| −8:00 | Décalage ISO−8601 pour la zone PST |
| −800 | Décalage ISO−8601 pour la zone PST |
| −8 | Décalage ISO−8601 pour la zone PST |
| zulu | Abréviation des militaires pour GMT |
| z | Version courte de zulu |

Référez−vous à Annexe B pour une liste de noms de fuseaux horaire reconnus en entrée.

### 8.5.1.3. Horodatages (time stamps)

Les valeurs d'entrée valides sont constituées par la concaténation d'une date, d'une heure, suivi d'un fuseau horaire optionnel, d'un qualificatif optionnel AD (avant Jésus Christ) ou BC (après Jésus Christ). (Autrement, AD/BC peut apparaître avant le fuseau horaire mais ce n'est pas l'ordre préféré.) Ainsi :

```
1999-01-08 04:05:06
```

et

```
1999-01-08 04:05:06 -8:00
```

sont des valeurs valides, qui suivent le standard ISO 8601. De plus, le format

```
January 8 04:05:06 1999 PST
```

très courant, est supporté.

Le standard SQL différencie les littéraux timestamp without time zone et timestamp with time zone par l'existence d'un << + >> ; ou << − >>. Du coup, suivant le standard,

```
TIMESTAMP '2004-10-19 10:23:54'
```

est du type timestamp without time zone alors que

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

est du type timestamp with time zone. PostgreSQL diffère du standard en réclamant que les littéraux timestamp with time zone soient explicitement typés :

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

Si un littéral n'est pas spécifiquement indiqué comme étant de type timestamp with time zone, PostgreSQL ignorera silencieusement toute indication de fuseau horaire dans la valeur littérale. C'est−à−dire que la valeur résultante date/time est dérivée des champs date/time dans la valeur en entrée et n'est pas ajustée pour le fuseau horaire.

Pour timestamp with time zone, la valeur stockée en interne est toujours en UTC (Temps Universel Coordonné), aussi connu sous le nom de GMT. Les valeurs d'entrée qui ont un fuseau horaire explicite sont converties en UTC en utilisant le décalage approprié. Si aucun fuseau horaire n'est précisé, alors le système considère que la date est dans le fuseau horaire indiqué par le paramètre système timezone, et la convertit en UTC en utilisant le décalage de la zone timezone.

Quand une valeur `timestamp with time zone` est affichée, elle est toujours convertie de UTC vers le fuseau horaire courant (variable `timezone`), et affichée comme une heure locale de cette zone. Pour voir l'heure dans un autre fuseau horaire, il faut soit changer la valeur de `timezone` ou utiliser la construction `AT TIME ZONE` (voir Section 9.9.3).

Les conversions entre `timestamp without time zone` et `timestamp with time zone` considèrent normalement que la valeur `timestamp without time zone` utilise le fuseau horaire `timezone`. Une zone différente peut être choisie en utilisant `AT TIME ZONE`.

#### 8.5.1.4. Intervalles

Les valeurs de type `interval` utilisent la syntaxe suivante:

```
[@] quantité unité [quantité unité...] [direction]
```

Où: *`quantité`* est un nombre (éventuellement signé); *`unité`* est `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, ou des abréviations ou des pluriels de ces unités; *`direction`* peut être `ago` ou vide. L'arobase (`@`) est du bruit optionnel. Les valeurs des différentes unités sont implicitement ajoutées en utilisant le signe approprié.

Les quantités de jours, heures, minutes et secondes peuvent être précisées sans unité explicite. Par exemple `'1 12:59:10'` est lu comme `'1 day 12 hours 59 min 10 sec'` (1 jour, 12 heures, 59 minutes, 10 secondes).

La précision optionnelle doit être entre 0 et 6, et prend la précision du littéral comme valeur par défaut.

#### 8.5.1.5. Valeurs spéciales

PostgreSQL supporte aussi plusieurs valeurs de dates spéciales, par simplicité, comme montré dans Tableau 8–13. Les valeurs `infinity` et `-infinity` ont une représentation spéciale dans le système et seront affichées de la même façon. Les autres sont simplement des facilités de notation qui seront converties en dates/heures ordinaires lorsqu'elles seront lues. (En particulier, `now` et les chaînes relatives sont converties en une valeur temps spécifique dès qu'elles sont lues.) Toutes ces valeurs sont traitées comme des constantes normales, et doivent être écrites entre apostrophes.

**Tableau 8–13. Saisie de dates/heures spéciales**

| Chaînes entrées | Types valides | Description |
|---|---|---|
| `epoch` | `date`, `timestamp` | 1970–01–01 00:00:00+00 (date système zéro d'Unix) |
| `infinity` | `timestamp` | plus tard que toutes les autres dates |
| `-infinity` | `timestamp` | plus tôt que toutes les autres dates |
| `now` | `date`, `time`, `timestamp` | heure de début de la transaction courante |
| `today` | `date`, `timestamp` | minuit aujourd'hui |
| `tomorrow` | `date`, `timestamp` | minuit demain |
| `yesterday` | `date`, `timestamp` | minuit hier |
| `allballs` | | |

| | | |
|---|---|---|
| allballs | time | 00:00:00.00 UTC |

Les fonctions suivantes, compatibles avec le standard SQL, peuvent aussi être utilisées pour obtenir l'heure courante pour le type de données correspondant : CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP. Les quatre derniers acceptent une spécification de la précision en option. (Voir Section 9.9.4.) Notez néanmoins que ce sont des fonctions SQL et qu'elles ne sont *pas* reconnues en tant que chaînes en entrée de la donnée.

## 8.5.2. Affichage des Date/Heure

Le format de sortie des types date/heure peut être choisi parmi un des quatre formats de date suivants: ISO 8601, SQL (Ingres), Traditionnel POSTGRES, et Allemand, en utilisant la commande SET datestyle. Le format par défaut est le format ISO, comme demandé par le standard SQL. Le nom du format d'affichage << SQL >> est un accident historique. Tableau 8−14 montre des exemples de chaque format d'affichage. Le format d'un type date ou time est bien sur celui de la partie date ou heure, comme montré dans les exemples.

**Tableau 8−14. Styles d'affichage de date/heure**

| Spécification de style | Description | Exemple |
|---|---|---|
| ISO | standard ISO 8601/SQL | 1997−12−17 07:37:16−08 |
| SQL | style traditionnel | 12/17/1997 07:37:16.00 PST |
| POSTGRES | style original | Wed Dec 17 07:37:16 1997 PST |
| German | style régional | 17.12.1997 07:37:16.00 PST |

Dans les styles SQL et POSTGRES, les jours apparaissent avant le mois si l'ordre des champs DMY a été précisé, sinon les mois apparaissent avant les jours. (Voir Section 8.5.1 pour savoir comment ce paramètre affecte l'interprétation des valeurs entrées.) Tableau 8−15 montre un exemple.

**Tableau 8−15. Convention d'ordre des dates**

| Réglage de >datestyle (style de date) | Ordre d'entrée | Exemple d'affichage |
|---|---|---|
| SQL, DMY | *jour*/*mois*/*année* | 17/12/1997 15:37:16.00 CET |
| SQL, MDY | *mois*/*jour*/*année* | 12/17/1997 07:37:16.00 PST |
| Postgres, DMY | *jour*/*mois*/*année* | Wed 17 Dec 07:37:16 1997 PST |

L'affichage du type interval ressemble au format de saisie, sauf que les unités comme century ou week sont converties en années et jours, et que ago est converti en un signe approprié. En mode ISO, l'affichage ressemble à:

```
[ quantité unité [ ... ] ] [ jours ] [ heures:minutes:secondes ]
```

Les styles de date/heure peuvent être sélectionnés soit en utilisant la commande SET datestyle, soit en utilisant le paramètre DateStyle du fichier de configuration postgresql.conf, soit avec la variable

d'environnement PGDATESTYLE sur le serveur ou le client. La fonction de formatage to_char (voir Section 9.8) permet aussi, de manière plus flexible, pour formater les affichages de date/heure.

## 8.5.3. Fuseaux horaires

Les fuseaux horaires et les conventions d'heures sont influencées par des décisions politiques, pas seulement par la géométrie de la terre. Les fuseaux horaires se sont un peu standardisés au cours du vingtième siècle, mais continuent d'être soumis à des changements arbitraires, particulièrement en respect des règles de changement d'heure. PostgreSQL supporte actuellement les règles de changement d'heure pour la période de 1902 à 2038 (correspondant à l'échelle intégrale du temps système Unix). Les périodes en dehors de cette échelle sont prises en tant que << temps standard >> pour le fuseau horaire sélectionné, quelque soit la partie de l'année où elles tombent.

PostgreSQL se veut compatible avec les définitions standard SQL pour un usage typique. Néanmoins, le standard SQL possède un mélange bizarre de types de date/heure et de possibilités. Deux problèmes sont évidents:

- Bien que le type date n'ait pas de fuseau horaire associé, le type heure peut en avoir un. Les fuseaux horaires, dans le monde réel, ne peuvent avoir de sens qu'associés à une date et à une heure, vu que l'écart peut varier avec l'heure d'été.
- Le fuseau horaire par défaut est précisé comme un écart numérique constant avec l'UTC. Il n'est pas possible de s'adapter à l'heure d'été ou d'hiver lorsque l'on fait des calculs arithmétiques qui passent les limites de l'heure d'été et l'heure d'hiver.

Pour ne pas avoir ces difficultés, nous recommandons d'utiliser des types de date/heure qui contiennent à la fois une date et une heure lorsque vous utilisez les fuseaux horaires. Nous recommandons de *ne pas* utiliser le type time with time zone. Ce type est néanmoins proposé par PostgreSQL pour les applications existantes et pour assurer la compatibilité avec le standard SQL. PostgreSQL utilise votre fuseau horaire pour tous les types qui ne contiennent qu'une date ou une heure.

Toutes les dates et heures, conscientes du fuseau horaire, sont stockées en interne en UTC. Ils sont convertis en heure locale dans la zone spécifiée par le paramètre de configuration timezone avant d'être affiché au client.

Le paramètre de configuration timezone peut être initialisé dans le fichier postgresql.conf ou de tous les autres moyens standards décrit dans Section 16.4. Il existe aussi quelques moyens spéciaux pour le configurer :

- Si timezone n'est spécifié ni dans postgresql.conf ni comme option en ligne de commande pour postmaster, le serveur tente d'utiliser la valeur de la variable d'environnement TZ comme fuseau horaire par défaut. Si TZ n'est pas définie ou ne fait pas partie des noms de fuseau horaire connus par PostgreSQL, le serveur tente de déterminer le fuseau horaire par défaut du système d'exploitation en vérifiant le comportement de la fonction localtime() de la bibliothèque C. Le fuseau horaire par défaut est sélectionné comme la correspondance la plus proche parmi les fuseaux horaires connus par PostgreSQL.
- La commande SQL SET TIME ZONE configure le fuseau horaire pour cette session. C'est une autre façon d'indiquer SET TIMEZONE TO avec une syntaxe plus compatible avec les spécifications SQL.

- La variable d'environnement PGTZ, si elle est mise à jour par le client, est utilisée par les applications basées sur libpq pour envoyer une commande SET TIME ZONE au serveur lors de la connexion.

Voir Annexe B pour avoir une liste des fuseaux horaires disponibles.

### 8.5.4. Types internes

PostgreSQL utilise les dates Juliennes pour tous les calculs de date/heure. Elles ont la propriété intéressante de permettre le calcul de toute date entre 4713 avant Jésus Christ et loin dans le futur, si on utilise le fait que l'année dure 365,2425 jours.

Les conventions de date antérieures au 19ème siècle offrent une lecture intéressante, mais ne sont pas assez consistantes pour être codées dans un gestionnaire de dates.

## 8.6. Type Boolean

PostgreSQL dispose du type boolean du standard SQL. Le type booléen ne peut avoir que deux états:<< true >> (vrai) et << false >> (faux). Un troisième état, << unknown >> (inconnu), est représenté par la valeur NULL de SQL.

Les valeurs littérales valides pour l'état << vrai >> sont:

```
TRUE
't'
'true'
'y'
'yes'
'1'
```

Pour l'état << faux >>, les valeurs suivantes peuvent être utilisées:

```
FALSE
'f'
'false'
'n'
'no'
'0'
```

Il est recommandé d'utiliser TRUE et FALSE (qui sont compatibles avec la norme SQL).

**Exemple 8–2. Utilisation du type `boolean`.**

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
```

```
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

Exemple 8–2 montre que les valeurs booléennes sont affichées en utilisant les lettres `t` et `f`.

> **Astuce :** Les valeurs de booléennes ne peuvent être directement converties en d'autres types. Par exemple, `CAST( boolval AS integer)` ne fonctionne pas. Il faut pour cela utiliser la construction `CASE`: `CASE WHEN boolval THEN 'value if true' ELSE 'value if false' END`. Voir Section 9.13.

le type `boolean` utilise 1 octet de stockage.

# 8.7. Types géométriques

Les types de données géométriques représentent des objets à deux dimensions. Tableau 8–16 liste les types disponibles dans PostgreSQL. Le type le plus fondamental, le point, forme la base pour tous les autres types.

**Tableau 8–16. Types géométriques**

| Nom | Taille de stockage | Représentation | Description |
|---|---|---|---|
| `point` | 16 octets | Point du plan | (x,y) |
| `line` | 32 octets | Ligne infinie (pas entièrement implémenté) | ((x1,y1),(x2,y2)) |
| `lseg` | 32 octets | Segment de droite fini | ((x1,y1),(x2,y2)) |
| `box` | 32 octets | Boite rectangulaire | ((x1,y1),(x2,y2)) |
| `path` | 16+16n octets | Chemin fermé (similaire à un polygone) | ((x1,y1),...) |
| `path` | 16+16n octets | Chemin ouvert | [(x1,y1),...] |
| `polygon` | 40+16n octets | Polygone (similaire à un chemin fermé) | ((x1,y1),...) |
| `circle` | 24 octets | Cercle | <(x,y),r> (centre et rayon) |

Un large ensemble de fonctions et d'opérateurs permettent d'effectuer différentes opérations géométriques, comme l'agrandissement, la translation, la rotation, la détermination des intersections. Elles sont expliquées dans Section 9.10.

## 8.7.1. Points

Les points sont les blocs fondamentaux pour construire les types géométriques. Les valeurs de type `point` sont spécifiées en utilisant la syntaxe suivante:

```
( x , y )
  x , y
```

où *x* et *y* sont les coordonnées respectives sous forme de nombre à virgule flottante.

## 8.7.2. Segments de droites

Les segments de droites (`lseg`) sont représentés sous forme paires de points, en utilisant la syntaxe suivante:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

où `(x1,y1)` et `(x2,y2)` sont les points extrémités du segment.

## 8.7.3. Boxes

Les boites (rectangles) sont représentées par paires de points qui sont les coins opposés de la boite, en utilisant la syntaxe suivante:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

où `(x1,y1)` et `(x2,y2)` sont les coins opposés du rectangle.

Les rectangles sont affichés en utilisant la première syntaxe. Les coins sont réordonnées lors de la saisie. Le coin en haut à gauche, puis le coin en bas à droite. Les autres coins peuvent être saisis, mais les coins en bas à gauche et en haut à droite sont déterminés à partir de l'entrée et stockés.

## 8.7.4. Chemins

Les chemins ( type `path` ) sont représentés par des listes de points connectés. Les chemins peuvent être *ouverts*, si le premier et le dernier point ne sont pas considérés comme connectés, ou *fermés*, si le premier et le dernier point sont considérés connectés.

Les valeurs de type `path` sont entrées avec la syntaxe suivante.

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

où les points sont les extrémités des segments de droites qui forment le chemin. Les crochets ( `[]` ) indiquent un chemin ouvert, alors que les parenthèses ( `()` ) indiquent un chemin fermé.

Les chemins sont affichés avec la première syntaxe.

### 8.7.5. Polygones

Les polygones ( type `polygon`) sont représentés par des ensembles de points (les vertex du polygone). Ils devraient probablement être considérés comme des chemins fermés, mais ils sont stockés différemment et ont leurs propres routines de manipulation.

Les valeurs de type `polygon` sont saisies avec la syntaxe suivante:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

où les points sont les extrémités des segments de droites qui forment les limites du polygone.

Les polygones sont affichés en utilisant la première syntaxe.

### 8.7.6. Cercles

Les cercles (type `circle`) sont représentés par le point central et le rayon. Les valeurs de type `circle` sont saisies avec la syntaxe suivante:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

où `(x,y)` est le centre et *r* est le rayon du cercle.

Les cercles sont affichés en utilisant la première syntaxe.

## 8.8. Types d'adresses réseau

PostgreSQL offre des types de données pour stocker des adresses IPv4, IPv6 et MAC, décrites dans Tableau 8−17 Il est préférable d'utiliser ces types plutôt que des types texte standards pour stocker des adresses réseau car ils proposent un contrôle de syntaxe lors de la saisie et plusieurs opérations et fonctions spécialisées (voir Section 9.11).

**Tableau 8−17. Types d'adresses réseau**

| Nom | Taille de stockage | Description |
|---|---|---|
| `cidr` | 12 or 24 octets | réseaux IPv4 et IPv6 |
| `inet` | 12 or 24 octets | hôtes et réseaux IPv4 et IPv6 |
| `macaddr` | 6 bytes | adresses MAC |

Lors du tri d'un type `inet` ou `cidr`, Les adresses IPv4 apparaîtront toujours avant les adresses IPv6, Y compris les adresses IPv4 encapsulées, comme ::10.2.3.4 or ::ffff::10.4.3.2.

### 8.8.1. `inet`

Le type `inet` contient une adresse d'hôte IPv4 ou IPv6, et optionnellement l'identité de son sous réseau, le tout dans un seul champ. L'identité du sous réseau est représentée en indiquant combien de bits de l'adresse hôte constituent l'adresse réseau (le << masque réseau >>). Si le masque réseau est 32 et l'adresse de type IPV4, alors la valeur n'indique pas un sous réseau, juste un hôte. En IPv6, la longueur de l'adresse est de 128 bits, si bien que 128 bits définissent une adresse réseau unique. Notez que si vous ne voulez utiliser que des adresses de réseau, il est préférable d'utiliser le type `cidr` plutôt que le type `inet`.

Le format de saisie pour ce type est *`address/y`* où *`address`* est une adresse IPv4 ou IPv6 et *`y`* est le nombre de bits du masque réseau. Si *`y`* est omis, alors le masque vaut 32 pour IPv4 et 128 pour IPv6, et la valeur représente un hôte unique. A l'affichage, la portion */y* est supprimée si le masque réseau indique un hôte unique.

### 8.8.2. `cidr`

Le type `cidr` contient une spécification de réseau IPv4 ou IPv6. L'entrée et la sortie suivent les conventions Classless Internet Domain Routing. Le format pour indiquer un réseau est *`address/y`* où *`address`* est le réseau représenté sous forme d'une adresse IPv4 ou IPv6 et *`y`* est le nombre de bits du masque réseau. Si *`y`* est omis, il calculé en utilisant les règles de l'ancien système de classes d'adresses, à ceci près qu'il sera au moins assez grand pour inclure tous les octets saisis. C'est une erreur de spécifier une adresse réseau avec des bits à droite du masque spécifié.

Tableau 8−18 donne des exemples.

**Tableau 8−18. `cidr` Exemples de saisie de types**

| Saisie de `cidr` | Affichage de `cidr` | `abbrev(cidr)` |
|---|---|---|
| 192.168.100.128/25 | 192.168.100.128/25 | 192.168.100.128/25 |
| 192.168/24 | 192.168.0.0/24 | 192.168.0/24 |
| 192.168/25 | 192.168.0.0/25 | 192.168.0.0/25 |
| 192.168.1 | 192.168.1.0/24 | 192.168.1/24 |
| 192.168 | 192.168.0.0/24 | 192.168.0/24 |
| 128.1 | 128.1.0.0/16 | 128.1/16 |
| 128 | 128.0.0.0/16 | 128.0/16 |
| 128.1.2 | 128.1.2.0/24 | 128.1.2/24 |
| 10.1.2 | 10.1.2.0/24 | 10.1.2/24 |
| 10.1 | 10.1.0.0/16 | 10.1/16 |
| 10 | 10.0.0.0/8 | 10/8 |
| 10.1.2.3/32 | 10.1.2.3/32 | 10.1.2.3/32 |
| 2001:4f8:3:ba::/64 | 2001:4f8:3:ba::/64 | 2001:4f8:3:ba::/64 |
| 2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128 | 2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128 | 2001:4f8:3:ba:2e0:81ff:fe22:d1f1 |

| | | |
|---|---|---|
| ::ffff:1.2.3.0/120 | ::ffff:1.2.3.0/120 | ::ffff:1.2.3/120 |
| ::ffff:1.2.3.0/128 | ::ffff:1.2.3.0/128 | ::ffff:1.2.3.0/128 |

### 8.8.3. Comparaison de `inet` et `cidr`

La différence principale entre les types de données `inet` et `cidr` est que `inet` accepte des valeurs avec des bits non nuls à la droite du masque de réseau, alors que `inet` ne l'accepte pas.

> **Astuce :** Si n'aimez pas le format d'affichage des valeurs `inet` et `cidr`, essayez les fonctions `host`, `text`, et `abbrev`.

### 8.8.4. `macaddr`

Le type `macaddr` stocke des adresses MAC, c'est à dire des adresses de cartes réseau Ethernet (mais les adresses MAC sont aussi utilisées dans d'autres cas). Les entrées sont acceptées dans de nombreux formats, dont:

```
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08-00-2b-01-02-03'
'08:00:2b:01:02:03'
```

Qui indiquent tous la même adresse. Les majuscules et les minuscules sont acceptées pour les chiffres de `a` à `f`. L'affichage se fait toujours selon le dernier format.

Le répertoire `contrib/mac` de la distribution des sources de PostgreSQL contient des outils qui peuvent être utilisés pour trouver les noms des fabricants de matériel liés à des adresses MAC.

## 8.9. Types champs de bits

Les champs de bits sont des chaînes de 0 et de 1. Ils peuvent être utilisés pour stocker ou visualiser des masques de bits. Il y a deux types champs de bits SQL: `bit(`*n*`)` et `bit varying(`*n*`)`, où *n* est un entier positif.

Les données de type `bit` doivent avoir une longueur de exactement *n* bits. Essayer d'y affecter une chaîne de bits plus longue ou plus courte déclenche une erreur. Les données de type `bit varying` ont une longueur variable, mais ne peuvent dépasser une taille de *n* bits. Les chaînes plus longues sont rejetées. Écrire `bit` sans longueur est équivalent à `bit(1)`, alors que `bit varying` sans longueur indique une taille illimitée.

> **Note :** Lors d'un transtypage explicite (cast) d'une chaîne de bits vers un champ de type `bit(`*n*`)`, la chaîne obtenue sera complétée avec des zéros ou bien tronquée, pour obtenir une taille d'exactement *n* bits, sans que cela produise une erreur. De la même façon, si une chaîne de bits est explicitement transtypée vers un champ de type `bit varying(`*n*`)`, elle sera tronquée si elle fait plus de *n* bits.

**Note :** Avant PostgreSQL 7.2, les données de type `bit` étaient toujours tronquées (ou complétées avec des zéros) silencieusement, que le transtypage soit explicite ou non. Ce comportement a été modifié pour se conformer au standard SQL.

Voir Section 4.1.2.3 pour plus d'information sur la syntaxe des constantes de champs de bits. Les opérateurs logiques bit à bit et les manipulations de chaînes de bits sont décrits dans Section 9.6.

**Exemple 8–3. Utilisation des types de champs de bits**

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
ERROR:  bit string length 2 does not match type bit(3)
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;
  a  |  b
-----+-----
 101 | 00
 100 | 101
```

# 8.10. Tableaux

PostgreSQL autorise de définir des colonnes d'une table comme des tableaux multidimensionnels à longueur variable. Des tableaux de n'importe quel type, même défini par l'utilisateur, peuvent être créés. (Néanmoins, les tableaux de type composite ou de domaines ne sont pas encore supportés.)

## 8.10.1. Déclaration des types de tableaux

Pour illustrer l'utilisation des types de tableaux, créons cette table :

```
CREATE TABLE sal_emp (
    nom              text,
    paye_par_semaine integer[],
    planning         text[][]
);
```

Comme indiqué ci–dessus, un type de donnée tableau est nommé en ajoutant des crochets (`[]`) au type de donnée des éléments du tableau. La commande ci–dessus créera une table nommée `sal_emp` avec une colonne de type `text` (`nom`), un tableau à une dimension de type `integer` (`paye_par_semaine`), représentant le salaire d'un employé par semaine et un tableau à deux dimensions de type `text` (`planning`), représentant le planning hebdomadaire de l'employé.

La syntaxe pour `CREATE TABLE` permet de spécifier la taille exacte des tableaux, par exemple :

```
CREATE TABLE tictactoe (
    carres   integer[3][3]
);
```

Néanmoins, l'implémentation actuelle n'oblige pas au respect des limites en taille du tableau –– le comportement est identique à celui des tableaux dont la longueur n'a pas été spécifiée.

En fait, l'implémentation actuelle n'oblige pas non plus à déclarer le nombre de dimensions. Les tableaux d'un type d'élément particulier sont tous considérés comme étant du même type, sans vérification de la taille ou du nombre de dimensions. Donc, déclarer le nombre de dimensions ou la taille dans `CREATE TABLE` a uniquement un but de documentation, cela n'affecte pas le comportement lors de l'exécution.

Une syntaxe alternative, conforme au standard SQL:1999, pourrait être utilisée pour les tableaux à une dimension. `paye_par_semaine` pourrait avoir été définie ainsi :

```
    paye_par_semaine  integer ARRAY[4],
```

Cette syntaxe nécessite une constante de type entier pour indiquer la taille du tableau. Néanmoins, comme indiqué précédemment, PostgreSQL n'impose aucune restriction sur la taille.

## 8.10.2. Saisie de valeurs de type tableau

Pour écrire une valeur de type tableau comme une constante littérale, encadrez les valeurs des éléments par des accolades et séparez−les par des virgules. (Si vous connaissez le C, ce n'est pas différent de la syntaxe C pour initialiser les structures.) Vous pouvez mettre des guillemets doubles autour des valeurs des éléments, et devez le faire si elles contiennent des virgules ou des accolades. (Plus de détails ci−dessous.) Le format général d'une constante de type tableau est donc le suivant :

```
'{ val1 delim val2 delim ... }'
```

où *`delim`* est le caractère de délimitation pour ce type, tel qu'il est enregistré dans son entrée `pg_type`. Parmi les types de données standards fournis par la distribution PostgreSQL, le type `box` utilise un point−virgule (`;`) mais tous les autres utilisent une virgule (`,`). Chaque *`val`* est soit une constante du type des éléments du tableau ou un sous−tableau. Voici un exemple d'une constante tableau

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Cette constante a deux dimensions, un tableau 3 par 3 consistant en trois sous−tableaux d'entiers.

(Ces types de constantes de tableau sont en fait un cas particulier des constantes de type générique abordées dans la Section 4.1.2.5. La constante est traitée initialement comme une chaîne et passée à la routine de conversion d'entrées de tableau. Une spécification explicite du type pourrait être nécessaire.)

Maintenant, nous pouvons montrer quelques instructions `INSERT`.

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"rendez-vous", "repas"}, {}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"rencontre", "repas"}, {"rencontre"}}');
    ERROR:  multidimensional arrays must have array expressions with matching dimensions
```

Notez que les tableaux à plusieurs dimensions doivent avoir des limites correspondantes pour chaque dimension. Une différence provoque une erreur à l'exécution.

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"rencontre", "repas"}, {"entrainement", "présentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"petit-déjeuner", "consulting"}, {"rencontre", "repas"}}');
```

Une limitation de l'implémentation actuelle des tableaux existe, les éléments individuels d'un tableau ne peuvent avoir la valeur SQL NULL. Le tableau entier peut être NULL mais vous ne pouvez pas avoir des éléments NULL dans un tableau avec d'autres éléments non NULL. (Ceci est susceptible de changer dans le futur.)

Le résultat des deux insertions précédentes ressemble à ceci :

```
SELECT * FROM sal_emp;
 nom   |      paye_par_semaine     |      planning
-------+---------------------------+--------------------
Bill  | {10000,10000,10000,10000} | {{rencontre,repas},{entrainement,présentation}}
Carol | {20000,25000,25000,25000} | {{petit-déjeuner,consulting},{rencontre,repas}}
(2 rows)
```

La syntaxe du constructeur `ARRAY` peut aussi être utilisée :

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['rendez-vous', 'repas'], ['entrainement','présentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['petit-déjeuner', 'consulting'], ['rencontre', 'repas']]);
```

Notez que les éléments du tableau sont des constantes SQL ordinaires ou des expressions ; par exemple, les chaînes de caractères littérales sont encadrées par des guillemets simples au lieu de guillemets doubles comme cela serait le cas dans un tableau littéral. La syntaxe du constructeur `ARRAY` est discutée plus en profondeur dans la Section 4.2.10.

## 8.10.3. Accès aux tableaux

Maintenant, nous pouvons lancer quelques requêtes sur la table. Tout d'abord, montrons comment accéder à un seul élément du tableau à la fois. Cette requête retrouve le nom des employés dont la paye a changé la deuxième semaine :

```
SELECT nom FROM sal_emp WHERE paye_par_semaine[1] <> paye_par_semaine[2];

 nom
-------
 Carol
(1 row)
```

Les nombres du tableau sont écrit entre crochets. Par défaut, PostgreSQL utilise la convention des nombres commençant à 1 pour les tableaux, c'est–à–dire un tableau à *n* éléments commence avec `array[1]` et finit

avec `array[n]`.

Cette requête récupère la paye de la troisième semaine pour tous les employés :

```
SELECT paye_par_semaine[3] FROM sal_emp;

 paye_par_semaine
------------------
          10000
          25000
(2 rows)
```

Nous pouvons aussi accéder à des parties rectangulaires arbitraires ou à des sous−tableaux. Une partie d'un tableau est notée par l'écriture *extrémité basse*`:`*extrémité haute* quelle que soit la dimension des tableaux. Par exemple, cette requête retrouve le premier élément du planning de Bill pour les deux premiers jours de la semaine :

```
SELECT planning[1:2][1:1] FROM sal_emp WHERE nom = 'Bill';

      planning
--------------------
 {{rendez-vous},{entrainement}}
(1 row)
```

Nous aurions aussi pu écrire

```
SELECT planning[1:2][1] FROM sal_emp WHERE nom = 'Bill';
```

en ayant le même résultat. Une opération d'indiçage de rangée est toujours prise pour représenter une tranche de rangée si un indice quelconque est écrit sous la forme *inférieur*`:`*supérieur*. Une limite basse de 1 est supposée pour toute tranche dont seule une valeur est spécifiée. Voici un autre exemple :

```
SELECT planning[1:2][2] FROM sal_emp WHERE nom = 'Bill';

         planning
---------------------------
 {{rendez-vous,repas},{entrainement,présentation}}
(1 row)
```

Récupérer en dehors des limites actuelles d'un tableau amène une valeur SQL NULL, pas une erreur. Par exemple, si `planning` a les dimensions `[1:3][1:2]`, alors référencer `planning[3][3]` a un résultat NULL. De la même façon, une référence sur un tableau avec le mauvais nombre d'indices amène une valeur NULL plutôt qu'une erreur. Récupérer une partie d'un tableau complètement en dehors des limites actuelles renvoie un tableau NULL ; mais si la partie demandée est partiellement intégrée aux limites du tableau, alors il est silencieusement réduit à la région d'intersection.

Les dimensions actuelles de toute valeur d'un tableau sont disponibles avec la fonction `array_dims` :

```
SELECT array_dims(planning) FROM sal_emp WHERE nom = 'Carol';

 array_dims
------------
 [1:2][1:1]
(1 row)
```

`array_dims` donne un résultat de type `text`, ce qui est pratique à lire mais peut–être moins simple à interpréter pour les programmes. Les dimensions sont aussi récupérables avec `array_upper` et `array_lower`, qui renvoient respectivement la limite haute et basse d'un tableau spécifié.

```
SELECT array_upper(planning, 1) FROM sal_emp WHERE nom = 'Carol';

 array_upper
-------------
           2
(1 row)
```

## 8.10.4. Modification de tableaux

La valeur d'un tableau peut être complètement remplacée :

```
UPDATE sal_emp SET paye_par_semaine = '{25000,25000,27000,27000}'
    WHERE nom = 'Carol';
```

ou en utilisant la syntaxe de l'expression `ARRAY` :

```
UPDATE sal_emp SET paye_par_semaine = ARRAY[25000,25000,27000,27000]
    WHERE nom = 'Carol';
```

On peut aussi mettre à jour un seul élément d'un tableau :

```
UPDATE sal_emp SET paye_par_semaine[4] = 15000
    WHERE nom = 'Bill';
```

ou faire une mise à jour par tranche :

```
UPDATE sal_emp SET paye_par_semaine[1:2] = '{27000,27000}'
    WHERE nom = 'Carol';
```

Une valeur de tableau enregistrée peut être agrandie pour affecter un élément adjacent à ceux déjà présents ou en affectant à une partie adjacente une partie des données déjà présentes. Par exemple, si le tableau `mon_tableau` a pour le moment quatre éléments, il en aura cinq après une mise à jour qui a affecté `mon_tableau[5]`. Actuellement, l'agrandissement de cette façon est seulement autorisé pour les tableaux à une dimension, et non pas pour les tableaux multidimensionnels.

L'affectation de parties d'un tableau permet la création de tableaux dont l'indice de départ n'est pas 1. Par exemple, vous pourriez affecter `mon_tableau[-2:7]` pour créer un tableau avec les valeurs d'indices allant de –2 à 7.

Les valeurs de nouveaux tableaux peuvent aussi être construites en utilisant l'opérateur de concaténation, `||`.

```
SELECT ARRAY[1,2] || ARRAY[3,4];
   ?column?
---------------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
```

```
(1 row)
```

L'opérateur de concaténation autorise un élément à être placé au début ou à la fin d'un tableau à une dimension. Il accepte aussi deux tableaux à *N* dimensions, ou un tableau à *N* dimensions et un à *N+1* dimensions.

Lorsqu'un élément seul est placé au début d'un tableau à une dimension, le résultat est un tableau disposant d'une limite inférieure égale à la limite inférieure de l'opérande du côté droit moins un. Lorsqu'un élément est placé à la fin d'un tableau à une dimension, le résultat est un tableau contenant la limite inférieure de l'opérande gauche. Par exemple :

```
SELECT array_dims(1 || ARRAY[2,3]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

Lorsque deux tableaux ayant un même nombre de dimensions sont concaténés, le résultat conserve la limite inférieure de l'opérande gauche. Le résultat est un tableau comprenant chaque élément de l'opérande gauche suivi de chaque élément de l'opérande droit. Par exemple :

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

Lorsqu'un tableau à *N* dimensions est placé au début ou à la fin d'un tableau à *N+1* dimensions, le résultat est analogue au cas ci−dessus. Chaque sous−tableau de dimension *N* est en quelque sorte un élément de la dimension externe d'un tableau à *N+1* dimensions. Par exemple :

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [0:2][1:2]
(1 row)
```

Un tableau peut aussi être construit en utilisant les fonctions `array_prepend`, `array_append` ou `array_cat`. Les deux premières supportent seulement les tableaux à une dimension alors que `array_cat` supporte les tableaux multidimensionnels. Notez que l'opérateur de concaténation vu ci−dessus est préféré à l'utilisation directe de ces fonctions. En fait, les fonctions sont utilisées principalement pour l'implémentation de l'opérateur de concaténation. Néanmoins, elles pourraient être directement utiles dans la création d'agrégats définis par l'utilisateur. Quelques exemples :

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
   array_cat
---------------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

## 8.10.5. Recherche dans des tableaux

Pour rechercher une valeur dans un tableau, vous devez vérifier chaque valeur dans le tableau. Ceci peut se faire à la main si vous connaissez la taille du tableau. Par exemple :

```
SELECT * FROM sal_emp WHERE paye_par_semaine[1] = 10000 OR
                            paye_par_semaine[2] = 10000 OR
                            paye_par_semaine[3] = 10000 OR
                            paye_par_semaine[4] = 10000;
```

Néanmoins, ceci devient rapidement fastidieux pour les gros tableaux et n'est pas très utile si la taille du tableau n'est pas connue. Une autre méthode est décrite dans Section 9.17. La requête ci−dessus est remplaçable par :

```
SELECT * FROM sal_emp WHERE 10000 = ANY (paye_par_semaine);
```

De plus, vous pouvez trouvez les lignes où le tableau n'a que des valeurs égales à 10000 avec :

```
SELECT * FROM sal_emp WHERE 10000 = ALL (paye_par_semaine);
```

> **Astuce :** Les tableaux ne sont pas toujours initialisés ; rechercher des éléments spécifiques d'un tableau pourrait être un signe d'une mauvaise conception de la base de données. Utilisez plutôt une table séparée avec une ligne pour chaque élément faisant parti du tableau. Cela sera plus simple pour une recherche et fonctionnera mieux dans le cas d'un grand nombre d'éléments.

## 8.10.6. Syntaxe d'entrée et de sortie des tableaux

La représentation externe du type texte d'une valeur d'un tableau consiste en des éléments interprétés suivant les règles de conversion d'entrées/sorties pour le type de l'élément du tableau, plus des décorations indiquant la structure du tableau. L'affichage consiste en des accolades (`{` et `}`) autour des valeurs du tableau et des caractères de délimitation entre éléments adjacents. Le caractère délimiteur est habituellement une virgule (`,`) mais peut être autre chose : il est déterminé par le paramètre `typdelim` du type de l'élément tableau (parmi les types de données standards supportés par l'implémentation de PostgreSQL, le type `box` utilise un point−virgule (`;`) mais tous les autres utilisent la virgule). Dans un tableau multidimensionnel, chaque dimension (row, plane, cube, etc.) utilise son propre niveau d'accolades et les délimiteurs doivent être utilisés entre des entités adjacentes au sein d'accolades de même niveau.

La routine de sortie du tableau placera des guillemets doubles autour des valeurs des éléments si elles sont des chaînes vides ou contiennent des accolades, des caractères délimiteurs, des guillemets doubles, des antislash ou des espaces. Les guillemets doubles et les antislash intégrés aux valeurs des éléments seront échappés avec un antislash. Pour les types de données numériques, on peut supposer sans risque que les doubles guillemets n'apparaîtront jamais, mais pour les types de données texte, vous devez vous préparer à gérer la présence et l'absence de guillemets. (Ceci est un changement du comportement à partir de la version pré−7.2 de PostgreSQL.)

Par défaut, la valeur de la limite basse d'un tableau est initialisée à 1. Si une des dimensions du tableau a une limite basse différente de 1, un affichage supplémentaire indiquant les dimensions réelles du tableau précède l'affichage de la structure du tableau. Cet affichage consiste en des crochets (`[]`) autour de chaque limite basse et haute d'une dimension avec un délimiteur deux−points (`:`) entre chaque. L'affichage des dimensions du tableau est suivie par un signe d'égalité (=). Par exemple :

```
SELECT 1 || ARRAY[2,3] AS array;

    array
---------------
[0:2]={1,2,3}
(1 row)

SELECT ARRAY[1,2] || ARRAY[[3,4]] AS array;

          array
-------------------------
[0:1][1:2]={{1,2},{3,4}}
(1 row)
```

Cette syntaxe peut aussi être utilisée pour spécifier des indices de tableau différents des indices par défaut. Par exemple :

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

Comme indiqué précédemment, lors de l'écriture d'une valeur de tableau, vous pourriez écrire des guillemets doubles autour de chaque élément individuel de tableau. Vous *devez* le faire si leur absence autour d'un élément induit en erreur l'analyseur de la valeur du tableau. Par exemple, les éléments contenant des crochets,

virgules (ou un caractère délimiteur), guillemets doubles, antislashs ou espace (en début comme en fin) doivent avoir des guillemets doubles. Pour placer un guillemet double ou un antislash dans une valeur d'élément d'un tableau, faites−le précéder d'un antislash. Autrement, vous pouvez échapper tous les caractères de données qui sont utilisés dans la syntaxe du tableau.

Vous pouvez ajouter des espaces avant un crochet gauche ou après un crochet droit. Vous pouvez aussi ajouter des espaces avant tout élément individuel. Dans tous les cas, les espaces seront ignorés. Par contre, les espaces à l'intérieur des éléments entre guillemets doubles ou entourés par des caractères autres que des espaces ne sont pas ignorés.

> **Note :** Rappelez−vous que ce que vous écrivez comme commande SQL sera tout d'abord interprété en tant que chaîne littérale puis en tant que tableau. Ceci double le nombre d'antislash dont vous aurez besoin. Par exemple, pour insérer une valeur de tableau de type `text` contenant un antislash et un guillemet double, vous aurez besoin d'écrire
>
> ```
> INSERT ... VALUES ('{"\\\\","\\""}');
> ```
>
> Le processeur de la chaîne littérale supprime un niveau d'antislash, donc ce qui arrive à l'analyseur de tableau ressemble à `{"\\","\""}`. À la place, les chaînes remplissant l'entrée du type de données `text` deviennent respectivement `\` et `"`. (Si nous travaillions avec un type de données dont la routine d'entrée traitait aussi les antislash de manière spéciale, `bytea` par exemple, nous pourrions avoir besoin d'au plus huit antislash dans la commande pour en obtenir un dans l'élément stocké.) Les guillemets dollar (voir Section 4.1.2.2) pourraient être utilisés pour éviter le besoin des doubles antislashs.
>
> **Astuce :** La syntaxe du constructeur `ARRAY` (voir Section 4.2.10) est souvent plus facile à utiliser que la syntaxe du tableau littéral lors de l'écriture de valeurs du tableau en commandes SQL. Avec `ARRAY`, les valeurs de l'élément individuel sont écrites de la même façon qu'elles auraient été écrites si elles n'avaient pas fait partie d'un tableau.

# 8.11. Types composites

Un *type composite* décrit la structure d'une ligne ou d'un enregistrement ; il est en essence une simple liste de noms de champs et de leur types de données. PostgreSQL autorise l'utilisation de valeurs de types composite identiques de plusieurs façons à l'utilisation des types simples. Par exemple, une colonne d'une table peut être déclarée comme étant de type composite.

## 8.11.1. Déclaration de types composite

Voici deux exemples simples de définition de types composite :

```
CREATE TYPE complexe AS (
    r       double precision,
    i       double precision
);

CREATE TYPE element_inventaire AS (
    nom             text,
    id_fournisseur  integer,
    prix            numeric
);
```

La syntaxe est comparable à `CREATE TABLE` sauf que seuls les noms de champs et leur types peuvent être spécifiés ; aucune contrainte (telle que `NOT NULL`) ne peut être inclus actuellement. Notez que le mot clé `AS` est essentiel ; sans lui, le système penserait à un autre genre de commande `CREATE TYPE` et vous obtiendriez d'étranges erreurs de syntaxe.

Après avoir défini les types, nous pouvons les utiliser pour créer des tables :

```
CREATE TABLE disponible (
    element   element_inventaire,
    nombre    integer
);

INSERT INTO disponible VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

ou des fonctions :

```
CREATE FUNCTION prix_extension(element_inventaire, integer) RETURNS numeric
AS 'SELECT $1.prix * $2' LANGUAGE SQL;

SELECT prix_extension(element, 10) FROM disponible;
```

Quand vous créez une table, un type composite est automatiquement créé, avec le même nom que la table, pour représenter le type de ligne de la table. Par exemple, si nous avions dit

```
CREATE TABLE element_inventaire (
    nom             text,
    id_fournisseur  integer REFERENCES fournisseur,
    prix            numeric CHECK (prix > 0)
);
```

alors le même type composite `element_inventaire` montré ci–dessus aurait été créé et pourrait être utilisé comme ci–dessus. Néanmoins, notez une restriction importante de l'implémentation actuelle : comme aucune contrainte n'est associée avec un type composite, les contraintes indiquées dans la définition de la table *ne sont pas appliquées* aux valeurs du type composite en dehors de la table. (Un contournement partiel est d'utiliser les types de domaine comme membres de types composites.)

## 8.11.2. Entrée d'une valeur composite

Pour écrire une valeur composite comme une constante littérale, englobez les valeurs du champ dans des parenthèses et séparez–les par des virgules. Vous pouvez placer des guillemets doubles autour de chaque valeur de champ et vous devez le faire si elle contient des virgules ou des parenthèses. (Plus de détails ci–dessous). Donc, le format général d'une constante composite est le suivant :

```
'( val1 , val2 , ... )'
```

Voici un exemple

```
'("fuzzy dice",42,1.99)'
```

qui serait une valeur valide du type `element_inventaire` défini ci–dessus. Pour rendre un champ NULL, n'écrivez aucun caractère dans sa position dans la liste. Par exemple, cette constante spécifie un troisième champ NULL :

```
'("fuzzy dice",42,)'
```

Si vous voulez un champ vide au lieu d'une valeur NULL, saisissez deux guillemets :

```
'("",42,)'
```

Ici, le premier champ est une chaîne vide non NULL alors que le troisième est NULL.

(Ces constantes sont réellement seulement un cas spécial de constantes génériques de type discutées dans Section 4.1.2.5. La constante est initialement traitée comme une chaîne et passée à la routine de conversion de l'entrée de type composite. Une spécification explicite de type pourrait être nécessaire.)

La syntaxe d'expression `ROW` pourrait aussi être utilisée pour construire des valeurs composites. Dans la plupart des cas, ceci est considérablement plus simple à utiliser que la syntaxe de chaîne littérale car vous n'avez pas à vous inquiéter des multiples couches de guillemets. Nous avons déjà utilisé cette méthode ci–dessus :

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

Le mot clé ROW est optionnel si vous avez plus d'un champ dans l'expression, donc ceci peut être simplifié avec

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

La syntaxe de l'expression `ROW` est discutée avec plus de détails dans Section 4.2.11.

## 8.11.3. Accéder aux types composite

Pour accéder à un champ d'une colonne composite, vous pouvez écrire un point et le nom du champ, un peu comme la sélection d'un champ à partir d'un nom de table. En fait, c'est tellement similaire que vous pouvez souvent utiliser des parenthèses pour éviter une confusion de l'analyseur. Par exemple, vous pouvez essayer de sélectionner des sous–champs à partir de notre exemple de table, `disponible`, avec quelque chose comme :

```
SELECT element.nom FROM disponible WHERE element.prix > 9.99;
```

Ceci ne fonctionnera pas car le nom `element` est pris pour le nom d'une table, et non pas d'un champ, suivant les règles de la syntaxe SQL. Vous devez l'écrire ainsi :

```
SELECT (element).nom FROM disponible WHERE (element).prix > 9.99;
```

ou si vous avez aussi besoin d'utiliser le nom de la table (par exemple dans une requête multi–table), de cette façon :

```
SELECT (disponible.element).nom FROM disponible WHERE (disponible.element).prix > 9.99;
```

Maintenant, l'objet entre parenthèses est correctement interprété comme une référence à la colonne `element`, puis le sous–champ peut être sélectionné à partir de lui.

Des problèmes syntaxiques similaires s'appliquent quand vous sélectionnez un champ à partir d'une valeur composite. En fait, pour sélectionner un seul champ à partir du résultat d'une fonction renvoyant une valeur

composite, vous aurez besoin d'écrire quelque chose comme

```
SELECT (ma_fonction(...)).champ FROM ...
```

Sans les parenthèses supplémentaires, ceci provoquera une erreur.

## 8.11.4. Modifier les types composite

Voici quelques exemples de la bonne syntaxe pour insérer et mettre à jour des colonnes composites. Tout d'abord pour insérer ou modifier une colonne entière :

```
INSERT INTO matab (col_complexe) VALUES((1.1,2.2));

UPDATE matab SET col_complexe = ROW(1.1,2.2) WHERE ...;
```

Le premier exemple omet `ROW`, le deuxième l'utilise ; nous pouvons le faire des deux façons.

Nous pouvons mettre à jour un sous−champ individuel d'une colonne composite :

```
UPDATE matab SET col_complexe.r = (col_complexe).r + 1 WHERE ...;
```

Notez ici que nous n'avons pas besoin de (et, en fait, ne pouvons pas) placer des parenthèses autour des noms de colonnes apparaissant juste après `SET`, mais nous avons besoin de parenthèses lors de la référence à la même colonne dans l'expression à droite du signe d'égalité.

Et nous pouvons aussi spécifier des sous−champs comme cibles de la commande `INSERT` :

```
INSERT INTO matab (col_complexe.r, col_complexe.i) VALUES(1.1, 2.2);
```

Si tous les sous−champs d'une colonne ne sont pas spécifiés, ils sont remplis avec une valeur NULL.

## 8.11.5. Syntaxe en entrée et sortie d'un type composite

La représentation texte externe d'une valeur composite consiste en des éléments qui sont interprétés suivant les règles de conversion d'entrées/sorties pour les types de champs individuels, plus des décorations indiquant la structure composite. Cette décoration consiste en des parenthèses ( `(` et `)` ) autour de la valeur entière ainsi que des virgules ( `,` ) entre les éléments adjacents. Des espace blancs en dehors des parenthèses sont ignorés mais à l'intérieur des parenthèses, ils sont considérés comme faisant partie de la valeur du champ et pourrait ou non être significatif suivant les règles de conversion de l'entrée pour le type de données du champ. Par exemple, dans

```
'(  42)'
```

l'espace blanc sera ignoré si le type du champ est un entier, mais pas s'il s'agit d'un champ de type texte.

Comme indiqué précédemment, lors de l'écriture d'une valeur composite, vous pouvez saisir des guillemets doubles autour de chaque valeur de champ individuel. Vous *devez* le faire si la valeur du champ pourrait sinon gêner l'analyseur de la valeur du champ composite. En particulier, les champs contenant des parenthèses, des virgules, des guillemets doubles ou des antislashs doivent être entre guillemets doubles. Pour placer un guillemet double ou un antislash dans la valeur d'un champ composite entre guillemets, faites−le précéder d'un antislash. (De plus, une paire de guillemets doubles à l'intérieur d'une valeur de champ à guillemets doubles

est pris pour représenter un caractère guillemet double, en analogie aux règles des guillemets simples dans les chaînes SQL littérales.) Autrement, vous pouvez utiliser l'échappement par antislash pour protéger tous les caractères de données qui auraient été pris pour une syntaxe composite.

Une valeur de champ composite vide (aucun caractère entre les virgules ou parenthèses) représente une valeur NULL. Pour écrire une valeur qui est une chaîne vide plutôt qu'une valeur NULL, écrivez `""`.

La routine de sortie composite placera des guillemets doubles autour des valeurs de champs s'ils sont des chaînes vides ou s'ils contiennent des parenthèses, virgules, guillemets doubles, antislash ou espaces blancs. (Faire ainsi pour les espaces blancs n'est pas essentiel mais aide à la lecture.) Les guillemets doubles et antislashs dans les valeurs des champs seront doublés.

> **Note :** Rappelez−vous que ce que vous allez saisir dans une commande SQL sera tout d'abord interprété comme une chaîne littérale, puis comme un composite. Ceci double le nombre d'antislash dont vous avez besoin. Par exemple, pour insérer un champ `text` contenant un guillemet double et un antislash dans une valeur composite, vous devez écrire
>
> ```
> INSERT ... VALUES ('("\\"\\\\")');
> ```
>
> Le processeur des chaînes littérales supprime un niveau d'antislash de façon à ce qui arrive à l'analyseur de valeurs composites ressemble à `("\"\\")`. À son tour, la chaîne remplie par la routine d'entrée du type de données `text` devient `"\`. (Si nous étions en train de travailler avec un type de données dont la routine d'entrée traite aussi les antislashs spécialement, `bytea` par exemple, nous pourrions avoir besoin d'au plus huit antislashs dans la commande pour obtenir un antislash dans le champ composite stocké.) Le guillemet dollar (voir Section 4.1.2.2) pourrait être utilisé pour éviter le besoin des antislashs doublés.
>
> **Astuce :** La syntaxe du constructeur `ROW` est habituellement plus simple à utiliser que la syntaxe du littérale composite lors de l'écriture de valeurs composites dans des commandes SQL. Dans `ROW`, les valeurs individuelles d'un champ sont écrits de la même façon qu'ils l'auraient été en étant pas membres du composite.

# 8.12. Types identifiants d'objets

Les identifiants d'objets (OID) sont utilisés en interne par PostgreSQL comme clés primaires de différentes tables système. De plus, une colonne système OID est ajoutée aux tables créées par les utilisateurs sauf si `WITHOUT OIDS` est indiqué à la création de la table ou quand la variable de configuration default_with_oids est configurée à false. Le type `oid` représente un identificateur d'objet Il a aussi différents types alias : `regproc`, `regprocedure`, `regoper`, `regoperator`, `regclass`, et `regtype`. Tableau 8−19 donne un aperçu.

Le type `oid` est actuellement un entier de quatre octets. Du coup, il n'est pas suffisamment grand garantir l'unicité dans une grosse base de données, où même dans une très grosse table. Du coup, l'utilisation d'une colonne OID comme clé primaire d'une table créée par un utilisateur est déconseillée. Les OID surtout destinés à être des références vers les tables système.

> **Note :** Les OID sont inclus par défaut dans les tables créées par un utilisateur dans PostgreSQL 8.0.5. Néanmoins, ce comportement est amené à changer dans une future version de PostgreSQL. Éventuellement, les tables créées par un utilisateur n'incluront pas de colonne

système OID sauf si `WITH OIDS` est spécifié lors de la création de la table ou lorsque la variable de configuration `default_with_oids` est initialisée à true. Si votre application requiert la présence d'une colonne système OID dans une table, elle devrait indiquer `WITH OIDS` lors de la création de la table pour s'assurer de sa compatibilité avec les prochaines versions de PostgreSQL.

Le type `oid` lui même dispose de peu d'opérations à part la comparaison. Néanmoins, il peut être transtypé en entier (integer) et manipulé en utilisant les opérateurs habituels des entiers. (Attention aux possibles confusions entre les entiers signés et non signés si vous le faites.)

Les types alias de l'OID n'ont pas d'opérations à eux sauf pour les routines spécialisées en entrée et en sortie. Ces routines sont capables d'accepter et d'afficher des noms symboliques pour les objets systèmes, plutôt que la valeur numérique brute que le type `oid` utiliserait. Les types alias permettent de simplifier la recherche des valeurs OID pour les objets. Par exemple, pour examiner les lignes `pg_attribute` en relation avec une table `matable`, vous pourriez écrire

```
SELECT * FROM pg_attribute WHERE attrelid = 'matable'::regclass;
```

plutôt que

```
SELECT * FROM pg_attribute
WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'matable');
```

Bien que cela ne semble pas si difficile, c'est toujours trop simplifié. Une sous−sélection bien plus compliqué serait nécessaire pour sélectionner le bon OID s'il existait plusieurs tables nommées `matable` dans différents schémas. Le convertisseur d'entrées `regclass` gère la recherche de la table suivant le paramétrage du chemin des schémas et il fera donc la << bonne recherche >> automatiquement. De façon similaire, convertir une OID d'une table en `regclass` est facile pour l'affichage symbolique d'un OID numérique.

**Tableau 8−19. Types identifiants d'objet**

| Nom | Référence | Description | Exemple |
|---|---|---|---|
| `oid` | tous | identifiant d'objet numérique | `564182` |
| `regproc` | `pg_proc` | nom de fonction | `sum` |
| `regprocedure` | `pg_proc` | fonction avec les types d'arguments | `sum(int4)` |
| `regoper` | `pg_operator` | nom d'opérateur | `+` |
| `regoperator` | `pg_operator` | opérateur avec types d'arguments | `*(integer,integer)` ou `-(NONE,integer)` |
| `regclass` | `pg_class` | nom de relation | `pg_type` |
| `regtype` | `pg_type` | nom de type de données | `integer` |

Tous les types alias d'OID acceptent des noms avec un préfixe de schéma, et affichent des noms préfixés par un schémas si l'objet ne peut être trouvé dans le chemin de recherche courant sans préfixe de schéma. Les types alias `regproc` et `regoper` n'acceptent que des noms uniques en entrée (sans surcharge), si bien qu'ils sont d'un usage limité. Dans la plupart des cas, `regprocedure` ou `regoperator` sont plus appropriés. Pour `regoperator`, les opérateurs unaires sont identifiés en écrivant `NONE` pour les opérandes non utilisés.

Un autre type identifiant utilisé par le système est `xid`, ou identifiant (abrégé xact) de transaction. C'est le type de données des colonnes systèmes `xmin` et `xmax`. Les identifiants de transactions sont des quantités sur

32 bits.

Un troisième type identifiant utilisé par le système est `cid`, ou identifiant de commande. C'est le type de données des colonnes systèmes `cmin` et `cmax`. Les identifiants de commandes sont aussi des quantités sur 32 bits.

Un dernier type identifiant utilisé par le système est `tid`, ou identifiant de tuple (identifiant de ligne). C'est le type de données des colonnes système `ctid`. Un tuple est une paire (numéro de bloc, index de tuple dans le bloc) qui identifie l'emplacement physique de la ligne dans sa table.

(Les colonnes systèmes sont expliquées plus en détail dans Section 5.4.)

# 8.13. Pseudo−Types

Le système de types de PostgreSQL contient un certain nombre de types à usage spécial qui sont collectivement appelés des *pseudo−types*. Un pseudo−type ne peut être utilisé pour une colonne de table, mais il peut être utilisé pour déclarer un argument de fonction ou un type de résultat. Chacun des pseudo−types disponibles est utile dans des situations où le comportement d'une fonction ne correspond pas simplement à accepter ou retourner des valeurs d'un type de données SQL spécifique. Tableau 8−20 liste les pseudo−types existants.

**Tableau 8−20. Pseudo−Types**

| Nom | Description |
|---|---|
| `any` | Indique qu'une fonction accepte n'importe quel type de données, quel qu'il soit. |
| `anyarray` | Indique qu'une fonction accepte tout type tableau (voir Section 31.2.6). |
| `anyelement` | Indique qu'une fonction accepte tout type de données. (voir Section 31.2.6). |
| `cstring` | Indique qu'une fonction accepte ou retourne une chaîne de caractères du langage C (terminée par un NULL). |
| `internal` | Indique qu'une fonction accepte ou retourne un type de données interne du serveur de bases de données. |
| `language_handler` | Une fonction d'appel de langage procédural est déclarée retourner un `language_handler`. |
| `record` | Identifie une fonction qui retourne un type de ligne non spécifié. |
| `trigger` | Une fonction trigger est déclarée comme retournant un type `trigger`. |
| `void` | Indique qu'une fonction ne retourne pas de valeur. |
| `opaque` | Un type de données obsolète qui servait précédemment à tous les usages cités ci−dessus. |

Les fonctions codées en C (incluses dans le produit ou chargées dynamiquement) peuvent déclarer, accepter ou retourner chacun de ces pseudo−types. Il est de la responsabilité de l'auteur de la fonction de s'assurer que la fonction se comporte normalement lorsqu'un pseudo−type est utilisé comme type d'argument.

Les fonctions codées en langage procédural peuvent utiliser les pseudo−types si leur langage le permet. A ce jour, tous les langages interdisent l'usage d'un pseudo−type comme argument, et n'acceptent que `void` et `record` comme type retourné (plus `trigger` lorsque la fonction est utilisée comme trigger). Certains supportent aussi les fonctions polymorphes en utilisant les types `anyarray` et `anyelement`.

Le pseudo−type `internal` sert à déclarer des fonctions qui ne sont appelées que par la base de données en interne, et non pas directement par une requête SQL. Si une fonction a au moins un argument de type `internal`, alors elle ne peut être appelée depuis SQL. Pour préserver la sécurité de cette restriction, il est important de suivre cette règle de codage: ne créez pas de fonction qui retourne un `internal` si elle n'a pas au moins un argument de type `internal`.

# Chapitre 9. Fonctions et opérateurs

PostgreSQL fournit un grand nombre de fonctions et d'opérateurs pour les types de données intégrés. Les utilisateurs peuvent aussi définir leurs propres fonctions et opérateurs comme décrit dans Partie V. Les commandes `\df` et `\do` de psql sont utilisées pour afficher respectivement la liste de toutes les fonctions et de tous les opérateurs.

Si la portabilité vous concerne, prenez note que la plupart des fonctions et opérateurs décrits dans ce chapitre, à l'exception des opérateurs arithmétiques et logiques les plus triviaux et quelques fonctions spécifiquement indiquées, ne font pas partie du standard SQL. Quelques−unes des fonctionnalités étendues sont présentes dans d'autres systèmes de gestion de bases de données SQL et dans la plupart des cas, ces fonctionnalités sont compatibles et cohérentes à de nombreuses implémentations. Ce chapitre n'est pas non plus exhaustif ; des fonctions supplémentaires apparaissent dans les sections adéquates du manuel.

## 9.1. Opérateurs logiques

Voici la liste des opérateurs logiques habituels :

```
AND
OR
NOT
```

SQL utilise une logique booléenne à trois valeurs et où la valeur NULL représente << inconnu >>. Observez les tables de vérité suivantes :

| *a* | *b* | *a* AND *b* | *a* OR *b* |
|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE |
| TRUE | FALSE | FALSE | TRUE |
| TRUE | NULL | NULL | TRUE |
| FALSE | FALSE | FALSE | FALSE |
| FALSE | NULL | FALSE | NULL |
| NULL | NULL | NULL | NULL |

| *a* | NOT *a* |
|---|---|
| TRUE | FALSE |
| FALSE | TRUE |
| NULL | NULL |

Les opérateurs `AND` et `OR` sont commutatifs, c'est−à−dire que l'échange des opérandes gauche et droit n'affecte pas le résultat. Mais, voir Section 4.2.12 pour plus d'informations sur l'ordre d'évaluation des sous−expressions.

# 9.2. Opérateurs de comparaison

Les opérateurs de comparaison habituels sont disponibles, comme l'indique Tableau 9−1.

**Tableau 9−1. Opérateurs de comparaison**

| Opérateur | Description |
|---|---|
| < | inférieur à |
| > | supérieur à |
| <= | inférieur ou égal à |
| >= | supérieur ou égal à |
| = | égal à |
| <> ou != | différent de |

**Note :** L'opérateur != est converti en <> au moment de l'analyse. Il n'est pas possible d'implémenter les opérateurs != et <> pour faire d'autres choses.

Les opérateurs de comparaison sont disponibles pour tous les types de données où cela a un sens. Tous les opérateurs de comparaison sont des opérateurs binaires renvoyant des valeurs du type `boolean`; des expressions comme `1 < 2 < 3` ne sont pas valides (car il n'existe pas d'opérateur < pour comparer une valeur booléenne avec `3`).

En plus des opérateurs de comparaison, la construction spéciale `BETWEEN` est disponible.

```
a BETWEEN x AND
y
```

est équivalent à

```
a >= x AND
a <= y
```

De même,

```
a NOT BETWEEN x AND
y
```

est équivalent à

```
a < x OR
a > y
```

Il n'y a pas de différence entre les deux formes respectives si ce n'est les cycles CPU requis pour ré−écrire en interne la première sous la forme de la seconde.

Pour vérifier si une valeur est NULL ou non, utilisez les constructions

```
expression IS NULL
```

```
expression IS NOT NULL
```

ou la construction équivalente, non standard,

```
expression ISNULL
expression NOTNULL
```

N'écrivez *pas* `expression = NULL` parce que `NULL` n'est pas << égal à >> `NULL`. (La valeur NULL représente une valeur inconnue, et il est impossible de dire si deux valeurs inconnues sont égales.) Ce comportement est conforme au standard SQL.

> **Astuce :** Quelques applications pourraient s'attendre à ce que `expression = NULL` renvoie true si `expression` s'évalue comme la valeur NULL. Il est chaudement recommandé que ces applications soient modifiées pour se conformer au standard SQL. Néanmoins, si cela ne peut pas être fait, le paramètre de configuration transform_NULL_equals est disponible. S'il est activé, PostgreSQL convertira les clauses `x = NULL` en `x IS NULL`. C'était le comportement par défaut dans les versions 6.5 à 7.1 de PostgreSQL.

Un opérateur standard de comparaison renvoie NULL (signifiant << inconnu >>) si une des entrées est NULL. Une autre façon de comparer est d'utiliser la syntaxe `IS DISTINCT FROM`:

```
expression IS DISTINCT FROM expression
```

Pour des entrées différentes de NULL, le résultat est le même que celui obtenu par l'opérateur <>. Cependant, si les deux entrées sont NULL, alors cela retourne faux et si une des deux entrées est NULL, alors cela retourne vrai. Dans ce cas–là, NULL n'est plus considéré comme comme un état de l'expression mais comme la valeur de l'expression.

Les valeurs booléennes peuvent aussi être testées en utilisant les constructions

```
expression IS TRUE
expression IS NOT TRUE
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

Elles renverront toujours true ou false, jamais une valeur NULL, même si l'opérande est NULL. Une entrée NULL est traitée comme la valeur logique << inconnue >>. Notez que `IS UNKNOWN` et `IS NOT UNKNOWN` sont réellement identiques à `IS NULL` et `IS NOT NULL`, respectivement, sauf que l'expression en entrée doit être de type booléen.

## 9.3. Fonctions et opérateurs mathématiques

Des opérateurs mathématiques sont fournis pour beaucoup de types PostgreSQL. Pour les types sans conventions mathématiques communes pour toutes les permutations possibles (c'est–à–dire les types dates/time), nous décrivons le comportement actuel dans les sections ci–dessous.

Tableau 9–2 affiche les opérateurs mathématiques disponibles.

**Tableau 9–2. Opérateurs mathématiques**

| Opérateur | Description | Exemple | Résultat |
|---|---|---|---|
| + | addition | 2 + 3 | 5 |
| – | soustraction | 2 – 3 | -1 |
| * | multiplication | 2 * 3 | 6 |
| / | division (la division entière tronque les résultats) | 4 / 2 | 2 |
| % | modulo (reste) | 5 % 4 | 1 |
| ^ | exponentiel | 2.0 ^ 3.0 | 8 |
| \|/ | racine carrée | \|/ 25.0 | 5 |
| \|\|/ | racine cubique | \|\|/ 27.0 | 3 |
| ! | factoriel | 5 ! | 120 |
| !! | factoriel (opérateur préfixe) | !! 5 | 120 |
| @ | valeur absolue | @ -5.0 | 5 |
| & | AND bit par bit | 91 & 15 | 11 |
| \| | OR bit par bit | 32 \| 3 | 35 |
| # | XOR bit par bit | 17 # 5 | 20 |
| ~ | NOT bit par bit | ~1 | -2 |
| << | décalage gauche | 1 << 4 | 16 |
| >> | décalage droit | 8 >> 2 | 2 |

Les opérateurs bits à bits fonctionnent seulement sur les types de données intregal alors que les autres sont disponibles pour tous les types de données numériques. Les opérateurs bits bits sont aussi disponibles pour les types de chaînes de bits `bit` et `bit varying` comme le montre Tableau 9–10.

Tableau 9–3 affiche les fonctions mathématiques disponibles. Dans la table, `dp` signifie `double precision`. Beaucoup de ces fonctions sont fournies dans de nombreuses formes composées de types d'argument différents. Sauf lorsque c'est indiqué, toute forme donnée d'une fonction renvoie le même type de données que son argument. Les fonctions utilisant des données de type `double precision` sont pour la plupart implémentées avec la bibliothèque C du système hôte ; la précision et le comportement dans les cas particuliers peuvent varier suivant le système hôte.

**Tableau 9–3. Fonctions mathématiques**

| Fonction | Type renvoyé | Description | Exemple | Résultat |
|---|---|---|---|---|
| abs(*x*) | (identique à *x*) | valeur absolue | abs(-17.4) | 17.4 |
| cbrt(dp) | dp | racine cubique | cbrt(27.0) | 3 |
| ceil(dp ou numeric) | (identique à l'argument) | plus petit entier supérieur à l'argument | ceil(-42.8) | -42 |
| ceiling(dp ou numeric) | (identique à l'entrée) | plus petit entier supérieur à l'argument (alias pour ceil) | ceiling(-95.3) | -95 |

| | | | | |
|---|---|---|---|---|
| degrees(dp) | dp | radians vers degrés | degrees(0.5) | 28.6478897565412 |
| exp(dp ou numeric) | (identique à l'argument) | exponentiel | exp(1.0) | 2.71828182845905 |
| floor(dp ou numeric) | (identique à l'argument) | plus grand entier inférieur à l'argument | floor(-42.8) | -43 |
| ln(dp ou numeric) | (identique à l'argument) | logarithme | ln(2.0) | 0.693147180559945 |
| log(dp ou numeric) | (identique à l'argument) | logarithme base 10 | log(100.0) | 2 |
| log(b numeric, x numeric) | numeric | logarithm en base b | log(2.0, 64.0) | 6.0000000000 |
| mod(y, x) | (identiques aux types des arguments) | reste de y/x | mod(9,4) | 1 |
| pi() | dp | constante << π >> | pi() | 3.14159265358979 |
| power(a dp, b dp) | dp | a élevé à la puissance b | power(9.0, 3.0) | 729 |
| power(a numeric, b numeric) | numeric | a élevé à la puissance b | power(9.0, 3.0) | 729 |
| radians(dp) | dp | degrés vers radians | radians(45.0) | 0.785398163397448 |
| random() | dp | valeur au hasard entre 0.0 et 1.0 | random() | |
| round(dp ou numeric) | (identique à l'argument) | arrondi à l'entier le plus proche | round(42.4) | 42 |
| round(v numeric, s integer) | numeric | arrondi pour s décimales | round(42.4382, 2) | 42.44 |
| setseed(dp) | integer | initialise la recherche pour les appels à random() | setseed(0.54823) | 1177314959 |
| sign(dp ou numeric) | (identique à l'argument) | signe de l'argument (−1, 0, +1) | sign(-8.4) | -1 |
| sqrt(dp ou numeric) | (identique à l'argument) | racine carré | sqrt(2.0) | 1.4142135623731 |
| trunc(dp ou numeric) | (identique à l'argument) | tronque vers zéro | trunc(42.8) | 42 |
| trunc(v numeric, s integer) | numeric | tronque sur s décimales | trunc(42.4382, 2) | 42.43 |
| width_bucket(op | integer | renvoie le jeton | width_bucket(5.35, | 3 |

| | | | | |
|---|---|---|---|---|
| numeric, b1 numeric, b2 numeric, count integer) | | auquel l'opérande était affecté dans un histogramme à équidistance du nombre de jetons, une limite supérieure de b1 et une limite inférieure de b2 | 0.024, 10.06, 5) | |

Enfin, Tableau 9–4 affiche les fonctions trigonométriques disponibles. Toutes les fonctions trigonométriques prennent des arguments et renvoient des valeurs de type double precision.

**Tableau 9–4. Fonctions trigonométriques**

| Fonction | Description |
|---|---|
| acos(*x*) | cosinus inversé |
| asin(*x*) | sinus inverse |
| atan(*x*) | tangente inverse |
| atan2(*x*, *y*) | tangent inverse de *x*/ *y* |
| cos(*x*) | cosinus |
| cot(*x*) | cotangente |
| sin(*x*) | sinus |
| tan(*x*) | tangente |

# 9.4. Fonctions et opérateurs de chaînes

Cette section décrit les fonctions et opérateurs pour examiner et manipuler des valeurs de type chaîne de caractères. Dans ce contexte, les chaînes incluent les valeurs des types character, character varying et text. Sauf cas contraire précisé, toutes les fonctions listées ci–dessous fonctionnent sur tous ces types mais faites attention des effets potentiels du remplissage automatique lors de l'utilisation du type character. Généralement, les fonctions décrites ici fonctionnent aussi sur les données d'autres types en convertissant au préalable la donnée en une représentation de type chaîne. Quelques fonctions existent aussi nativement pour le type chaînes bit à bit.

SQL définit quelques fonctions de type chaîne avec une syntaxe particulière où certains mots clés sont utilisés à la place de virgule pour séparer les arguments. Des détails sont disponibles dans Tableau 9–5. Ces fonctions sont aussi implémentées en utilisant la syntaxe rationnelle pour l'appel de fonctions. (Voir Tableau 9–6.)

**Tableau 9–5. Fonctions et opérateurs SQL pour le type chaîne**

| Fonction | Type renvoyé | Description | Exemple | Résultat |
|---|---|---|---|---|
| string \|\| string | text | Concaténation de chaîne | 'Post' \|\| 'greSQL' | PostgreSQL |

| | | | | |
|---|---|---|---|---|
| `bit_length (string)` | `integer` | Nombre de bits dans une chaîne | `bit_length('jose')` | `32` |
| `char_length (string)` ou `character_length (string)` | `integer` | Nombre de caractères dans une chaîne | `char_length('jose')` | `4` |
| `convert(string using nom_conversion)` | `text` | Modifie le codage en utilisant le nom de conversion cité. Les conversions peuvent être définies par `CREATE CONVERSION`. De même, il existe quelques noms de conversion pré−définis. Voir Tableau 9−7 pour les noms de conversion disponibles. | `convert('PostgreSQL' using iso_8859_1_to_utf_8)` | `'PostgreSQL'` in Unicode (UTF−8) encoding |
| `lower (string)` | `text` | Convertit une chaîne en minuscule | `lower('TOM')` | `tom` |
| `octet_length(string)` | `integer` | Nombre d'octets dans une chaîne | `octet_length('jose')` | `4` |
| `overlay(string placing string from integer [for integer])` | `text` | Remplace la sous−chaîne | `overlay('Txxxxas' placing 'hom' from 2 for 4)` | `Thomas` |
| `position(substring in string)` | `integer` | Emplacement des sous−chaînes spécifiées | `position('om' in 'Thomas')` | `3` |
| `substring(string [from integer] [for integer])` | `text` | Extrait une sous−chaîne | `substring('Thomas' from 2 for 3)` | `hom` |
| `substring(string from` *`modele`*`)` | `text` | Extrait une sous−chaîne correspondant à l'expression rationnelle POSIX | `substring('Thomas' from '...$')` | `mas` |
| `substring(string from` *`pattern`* `for` *`escape`*`)` | `text` | Extrait une sous−chaîne correspondant à l'expression rationnelle SQL | `substring('Thomas' from '%#"o_a#"_' for '#')` | `oma` |
| `trim([leading \| trailing \| both]` | `text` | Supprime la plus grande chaîne | `trim(both 'x' from 'xTomxx')` | `Tom` |

| | | | | |
|---|---|---|---|---|
| [caractères] from chaîne) | | contenant seulement les caractères (un espace par défaut) à partir du début, de la fin ou des deux extrémités (respectivement start, end, both) de la chaîne. | | |
| upper (string) | text | Convertit une chaîne en majuscule | upper('tom') | TOM |

D'autres fonctions de manipulation de chaînes sont disponibles et listées dans Tableau 9–6. Certaines d'entre elles sont utilisées en interne pour implémenter les fonctions de chaîne répondant au standard SQL listées dans Tableau 9–5.

**Tableau 9–6. Autres fonctions de chaîne**

| Fonction | Type renvoyé | Description | Exemple | Résultat |
|---|---|---|---|---|
| ascii(text) | integer | Code ASCII du premier caractère de l'argument | ascii('x') | 120 |
| btrim(chaîne text [, caractères text]) | text | Supprime la chaîne la plus longue consistant seulement de caractères compris dans caractères (un espace par défaut) entre le début et la fin de chaîne. | btrim('xyxtrimyyx', 'xy') | trim |
| chr(integer) | text | Caractère correspondant au code ASCII donné | chr(65) | A |
| convert(chaîne text, [codage_source name,] codage_destination name) | text | Convertit une chaîne dans le codage codage_destination. Le codage initial est spécifié par codage_source. Si codage_source est omis, le codage de la base de données est pris en compte. | convert( 'texte_en_unicode', 'UNICODE', 'LATIN1') | texte_en_unico représenté dans le co ISO 8859–1 |
| decode(chaîne text, type text) | bytea | Décode les données binaires à partir de chaîne codées auparavant avec encode. Le type de paramètre est le même que encode. | decode('MTIzAAE=', 'base64') | 123\000\001 |
| encode(données bytea, type text) | text | Code les données binaires en une représentation en | encode( '123\\000\\001', | MTIzAAE= |

| | | | | |
|---|---|---|---|---|
| | | ASCII uniquement. Les types supportés sont : base64, hex, escape. | 'base64') | |
| initcap (text) | text | Convertit la première lettre de chaque mot en majuscule et le reste en minuscule. Les mots sont des séquences de caractères alphanumériques séparés par des caractères non alphanumériques. | initcap('bonjour thomas') | Bonjour Thomas |
| length (chaîne text) | integer | Nombre de caractères dans chaîne | length('jose') | 4 |
| lpad(chaîne text, longueur integer [, remplissage text]) | text | Remplit chaîne pour avoir une chaîne de longueur longueur en ajoutant les caractères remplissage (un espace par défaut). Si chaîne a un taille supérieure à longueur, alors elle est tronquée (sur la droite). | lpad('hi', 5, 'xy') | xyxhi |
| ltrim(string chaîne [, caractères text]) | text | Supprime la chaîne la plus longue contenant seulement les caractères de chaîne (un espace par défaut) à partir du début de la chaîne. | ltrim('zzzytrim', 'xyz') | trim |
| md5(chaîne text) | text | Calcule la clé MD5 de chaîne, renvoyant le résultat en hexadécimal. | md5('abc') | 900150983cd24f<br>d6963f7d28e17f |
| pg_client_encoding() | name | Nom du codage client actuel | pg_client_encoding() | SQL_ASCII |
| quote_ident(chaîne text) | text | Renvoie la chaîne entre guillemets à utiliser comme identifiant dans une chaîne d'instructions SQL. Les guillemets sont seulement ajoutés si nécessaire (c'est–à–dire si la chaîne contient des caractères non identifieurs). Les guillemets faisant partie de la chaîne sont doublés. | quote_ident('Foo bar') | "Foo bar" |
| quote_literal(string text) | text | Renvoie la chaîne correctement entre | quote_literal( 'O\'Reilly') | 'O''Reilly' |

| | | | | |
|---|---|---|---|---|
| | | guillemets pour être utilisée comme une chaîne littérale dans une chaîne d'instructions SQL. Les guillemets compris dans la chaîne et les antislash sont correctement doublés. | | |
| repeat(chaîne text, nombre integer) | text | Repère le texte chaîne nombre fois | repeat('Pg', 4) | PgPgPgPg |
| replace(chaîne text, àpartirde text, vers text) | text | Remplace dans chaîne toutes les occurrences de la sous−chaîne àpartirde avec la sous−chaîne vers. | replace( 'abcdefabcdef', 'cd', 'XX') | abXXefabXXef |
| rpad(chaîne text, longueur integer [, remplissage text]) | text | Remplit chaîne sur une longueur de longueur caractères en ajoutant les caractères remplissage (un espace par défaut). Si la chaîne a une taille supérieure à longueur, elle est tronquée. | rpad('hi', 5, 'xy') | hixyx |
| rtrim(chaîne text [, caractères text]) | text | Supprime la chaîne la plus longue contenant uniquement les caractères provenant de caractères (un espace par défaut) depuis la fin de chaîne. | rtrim('trimxxxx', 'x') | trim |
| split_part(chaîne text, délimiteur text, champ integer) | text | Divise chaîne par rapport au délimiteur et renvoie le champ donné (en comptant à partir de 1) | split_part( 'abc~@~def~@~ghi', '~@~', 2) | def |
| strpos(chaîne, sous−chaîne) | int | Emplacement de la sous−chaîne spécifiée (identique à position(sous−chaîne in sous−chaîne), mais notez l'ordre inverse des arguments) | strpos('high', 'ig') | 2 |
| substr(chaîne, from [, nombre]) | text | Extrait la sous−chaîne (identique à substring(chaîne from àpartirde for nombre)) | substr('alphabet', 3, 2) | ph |
| to_ascii(text [, codage]) | text | Convertit le texte en ASCII à partir de | to_ascii('Karel') | Karel |

| | | | | |
|---|---|---|---|---|
| | | n'importe quelle autre forme de codage [a] | | |
| to_hex(nombre integer ou bigint) | text | Convertit nombre dans sa représentation hexadécimale équivalente | to_hex(2147483647) | 7fffffff |
| translate(chaîne text, àpartirde text, vers text) | text | Tout caractère dans chaîne qui correspond à un caractère dans l'ensemble àpartirde est remplacé par le caractère correspondant dans l'ensemble vers. | translate('12345', '14', 'ax') | a23x5 |

Remarques :
a. La fonction to_ascii supporte la conversion de LATIN1, LATIN2, LATIN9 et WIN1250 uniquement.

**Tableau 9–7. Conversions intégrés**

| Nom de la conversion [a] | Codage source | Codage destination |
|---|---|---|
| ascii_to_mic | SQL_ASCII | MULE_INTERNAL |
| ascii_to_utf_8 | SQL_ASCII | UNICODE |
| big5_to_euc_tw | BIG5 | EUC_TW |
| big5_to_mic | BIG5 | MULE_INTERNAL |
| big5_to_utf_8 | BIG5 | UNICODE |
| euc_cn_to_mic | EUC_CN | MULE_INTERNAL |
| euc_cn_to_utf_8 | EUC_CN | UNICODE |
| euc_jp_to_mic | EUC_JP | MULE_INTERNAL |
| euc_jp_to_sjis | EUC_JP | SJIS |
| euc_jp_to_utf_8 | EUC_JP | UNICODE |
| euc_kr_to_mic | EUC_KR | MULE_INTERNAL |
| euc_kr_to_utf_8 | EUC_KR | UNICODE |
| euc_tw_to_big5 | EUC_TW | BIG5 |
| euc_tw_to_mic | EUC_TW | MULE_INTERNAL |
| euc_tw_to_utf_8 | EUC_TW | UNICODE |
| gb18030_to_utf_8 | GB18030 | UNICODE |
| gbk_to_utf_8 | GBK | UNICODE |
| iso_8859_10_to_utf_8 | LATIN6 | UNICODE |
| iso_8859_13_to_utf_8 | LATIN7 | UNICODE |
| iso_8859_14_to_utf_8 | LATIN8 | UNICODE |
| iso_8859_15_to_utf_8 | LATIN9 | UNICODE |
| iso_8859_16_to_utf_8 | LATIN10 | UNICODE |
| iso_8859_1_to_mic | LATIN1 | MULE_INTERNAL |
| iso_8859_1_to_utf_8 | LATIN1 | UNICODE |
| iso_8859_2_to_mic | LATIN2 | MULE_INTERNAL |

| | | |
|---|---|---|
| iso_8859_2_to_utf_8 | LATIN2 | UNICODE |
| iso_8859_2_to_windows_1250 | LATIN2 | WIN1250 |
| iso_8859_3_to_mic | LATIN3 | MULE_INTERNAL |
| iso_8859_3_to_utf_8 | LATIN3 | UNICODE |
| iso_8859_4_to_mic | LATIN4 | MULE_INTERNAL |
| iso_8859_4_to_utf_8 | LATIN4 | UNICODE |
| iso_8859_5_to_koi8_r | ISO_8859_5 | KOI8 |
| iso_8859_5_to_mic | ISO_8859_5 | MULE_INTERNAL |
| iso_8859_5_to_utf_8 | ISO_8859_5 | UNICODE |
| iso_8859_5_to_windows_1251 | ISO_8859_5 | WIN |
| iso_8859_5_to_windows_866 | ISO_8859_5 | ALT |
| iso_8859_6_to_utf_8 | ISO_8859_6 | UNICODE |
| iso_8859_7_to_utf_8 | ISO_8859_7 | UNICODE |
| iso_8859_8_to_utf_8 | ISO_8859_8 | UNICODE |
| iso_8859_9_to_utf_8 | LATIN5 | UNICODE |
| johab_to_utf_8 | JOHAB | UNICODE |
| koi8_r_to_iso_8859_5 | KOI8 | ISO_8859_5 |
| koi8_r_to_mic | KOI8 | MULE_INTERNAL |
| koi8_r_to_utf_8 | KOI8 | UNICODE |
| koi8_r_to_windows_1251 | KOI8 | WIN |
| koi8_r_to_windows_866 | KOI8 | ALT |
| mic_to_ascii | MULE_INTERNAL | SQL_ASCII |
| mic_to_big5 | MULE_INTERNAL | BIG5 |
| mic_to_euc_cn | MULE_INTERNAL | EUC_CN |
| mic_to_euc_jp | MULE_INTERNAL | EUC_JP |
| mic_to_euc_kr | MULE_INTERNAL | EUC_KR |
| mic_to_euc_tw | MULE_INTERNAL | EUC_TW |
| mic_to_iso_8859_1 | MULE_INTERNAL | LATIN1 |
| mic_to_iso_8859_2 | MULE_INTERNAL | LATIN2 |
| mic_to_iso_8859_3 | MULE_INTERNAL | LATIN3 |
| mic_to_iso_8859_4 | MULE_INTERNAL | LATIN4 |
| mic_to_iso_8859_5 | MULE_INTERNAL | ISO_8859_5 |
| mic_to_koi8_r | MULE_INTERNAL | KOI8 |
| mic_to_sjis | MULE_INTERNAL | SJIS |
| mic_to_windows_1250 | MULE_INTERNAL | WIN1250 |
| mic_to_windows_1251 | MULE_INTERNAL | WIN |
| mic_to_windows_866 | MULE_INTERNAL | ALT |
| sjis_to_euc_jp | SJIS | EUC_JP |
| sjis_to_mic | SJIS | MULE_INTERNAL |
| sjis_to_utf_8 | SJIS | UNICODE |
| tcvn_to_utf_8 | TCVN | UNICODE |
| uhc_to_utf_8 | UHC | UNICODE |

| | | |
|---|---|---|
| utf_8_to_ascii | UNICODE | SQL_ASCII |
| utf_8_to_big5 | UNICODE | BIG5 |
| utf_8_to_euc_cn | UNICODE | EUC_CN |
| utf_8_to_euc_jp | UNICODE | EUC_JP |
| utf_8_to_euc_kr | UNICODE | EUC_KR |
| utf_8_to_euc_tw | UNICODE | EUC_TW |
| utf_8_to_gb18030 | UNICODE | GB18030 |
| utf_8_to_gbk | UNICODE | GBK |
| utf_8_to_iso_8859_1 | UNICODE | LATIN1 |
| utf_8_to_iso_8859_10 | UNICODE | LATIN6 |
| utf_8_to_iso_8859_13 | UNICODE | LATIN7 |
| utf_8_to_iso_8859_14 | UNICODE | LATIN8 |
| utf_8_to_iso_8859_15 | UNICODE | LATIN9 |
| utf_8_to_iso_8859_16 | UNICODE | LATIN10 |
| utf_8_to_iso_8859_2 | UNICODE | LATIN2 |
| utf_8_to_iso_8859_3 | UNICODE | LATIN3 |
| utf_8_to_iso_8859_4 | UNICODE | LATIN4 |
| utf_8_to_iso_8859_5 | UNICODE | ISO_8859_5 |
| utf_8_to_iso_8859_6 | UNICODE | ISO_8859_6 |
| utf_8_to_iso_8859_7 | UNICODE | ISO_8859_7 |
| utf_8_to_iso_8859_8 | UNICODE | ISO_8859_8 |
| utf_8_to_iso_8859_9 | UNICODE | LATIN5 |
| utf_8_to_johab | UNICODE | JOHAB |
| utf_8_to_koi8_r | UNICODE | KOI8 |
| utf_8_to_sjis | UNICODE | SJIS |
| utf_8_to_tcvn | UNICODE | TCVN |
| utf_8_to_uhc | UNICODE | UHC |
| utf_8_to_windows_1250 | UNICODE | WIN1250 |
| utf_8_to_windows_1251 | UNICODE | WIN |
| utf_8_to_windows_1256 | UNICODE | WIN1256 |
| utf_8_to_windows_866 | UNICODE | ALT |
| utf_8_to_windows_874 | UNICODE | WIN874 |
| windows_1250_to_iso_8859_2 | WIN1250 | LATIN2 |
| windows_1250_to_mic | WIN1250 | MULE_INTERNAL |
| windows_1250_to_utf_8 | WIN1250 | UNICODE |
| windows_1251_to_iso_8859_5 | WIN | ISO_8859_5 |
| windows_1251_to_koi8_r | WIN | KOI8 |
| windows_1251_to_mic | WIN | MULE_INTERNAL |
| windows_1251_to_utf_8 | WIN | UNICODE |
| windows_1251_to_windows_866 | WIN | ALT |
| windows_1256_to_utf_8 | WIN1256 | UNICODE |
| windows_866_to_iso_8859_5 | ALT | ISO_8859_5 |

| | | |
|---|---|---|
| windows_866_to_koi8_r | ALT | KOI8 |
| windows_866_to_mic | ALT | MULE_INTERNAL |
| windows_866_to_utf_8 | ALT | UNICODE |
| windows_866_to_windows_1251 | ALT | WIN |
| windows_874_to_utf_8 | WIN874 | UNICODE |

Remarques :
a. Les noms des conversions suivent un schéma de nommage standard : le nom officiel de la source de codage avec tous les caractères non alpha−numériques remplacés par des tirets bas suivis d'un _to_ suivi par le nom de destination du codage après avoir suivi le même traitement que le nom de la source. Du coup, les noms pourraient dévier de noms de codage personnalisés.

# 9.5. Fonctions et opérateurs de chaînes binaires

Cette section décrit les fonctions et opérateurs pour examiner et manipuler des valeurs de type bytea.

SQL définit quelques fonctions de chaînes avec une syntaxe spéciale où certains mots clés, plutôt que des virgules, sont utilisés pour séparés les mots clés. Les détails font partie de Tableau 9−8. Quelques fonctions sont aussi implémentées en utilisant la syntaxe habituelle pour l'appel de fonction. (Voir Tableau 9−9.)

**Tableau 9−8. Fonctions et opérateurs SQL pour les chaînes binaires**

| Fonction | Type renvoyé | Description | Exemple | Résultat |
|---|---|---|---|---|
| chaîne \|\| chaîne | bytea | Concaténation de chaîne | '\\\\Post'::bytea \|\| '\\047gres\\000'::bytea | \\Post'gres\000 |
| octet_length(chaîne) | integer | Nombre d'octets dans une chaîne binaire | octet_length( 'jo\\000se'::bytea) | 5 |
| position(sous-chaîne in chaîne) | integer | Emplacement de la sous−chaîne indiquée | position('\\000om'::bytea in 'Th\\000omas'::bytea) | 3 |
| substring(chaîne [from integer] [for integer]) | bytea | Extrait la sous−chaîne | substring('Th\\000omas'::bytea from 2 for 3) | h\000o |
| trim([both] octets from chaînes) | bytea | Supprime la plus longue chaîne contenant seulement les octets dans octets au début et à la | trim('\\000'::bytea from '\\000Tom\\000'::bytea) | Tom |

| | | | | |
|---|---|---|---|---|
| | | fin de chaîne | | |
| get_byte(chaîne, décalage) | integer | Extrait un octet de la chaîne. | get_byte('Th\\000omas'::bytea, 4) | 109 |
| set_byte(chaîne, décalage, nouvelle_valeur) | bytea | Enregistre un octet dans la chaîne. | set_byte('Th\\000omas'::bytea, 4, 64) | Th\000o@as |
| get_bit(chaîne, décalage) | integer | Extrait un bit de la chaîne. | get_bit('Th\\000omas'::bytea, 45) | 1 |
| set_bit(chaîne, décalage, nouvelle_valeur) | bytea | Enregistre un bit dans la chaîne. | set_bit('Th\\000omas'::bytea, 45, 0) | Th\000omAs |

Les fonctions de manipulations supplémentaires de chaînes binaires sont disponibles et sont listées dans Tableau 9–9. Certaines sont utilisées en interne pour implémenter les fonctions de chaînes suivant le standard SQL listées dans Tableau 9–8.

**Tableau 9–9. Autres fonctions sur les chaînes binaires**

| Fonction | Type retourné | Description | Exemple | Résultat |
|---|---|---|---|---|
| btrim(chaîne bytea, octets bytea) | bytea | Supprime la plus longue chaîne consistant seulement des octets de octets à partir du début et de la fin de chaîne. | btrim('\\000trim\\000'::bytea, '\\000'::bytea) | trim |
| length (chaîne) | integer | Longueur de la chaîne binaire | length('jo\\000se'::bytea) | 5 |
| decode(chaîne text, type text) | bytea | Décode la chaîne binaire à partir de chaîne auparavant codé avec encode. Le type de paramètre est identique à celui d'encode. | decode('123\\000456', 'escape') | 123\000456 |
| encode(chaîne bytea, type text) | text | Code la chaîne binaire en une représentation en ASCII uniquement. Les types supportés sont : base64, hex, escape. | encode('123\\000456'::bytea, 'escape') | 123\000456 |

# 9.6. Fonctions et opérateurs pour les chaînes de bits

Cette section décrit les fonctions et opérateurs pour examiner et manipuler les chaînes de bits, qui sont des valeurs du type `bit` et `bit varying`. En dehors des opérateurs de comparaison habituels, les opérateurs montrés dans Tableau 9−10 peuvent être utilisés. Les opérandes de chaînes de bits `&`, `|` et `#` doivent être de même longueur. Lors d'un décalage de bits, la longueur originale de la chaîne est préservée comme le montre les exemples.

**Tableau 9−10. Opérateurs sur les chaînes de bits**

| Opérateur | Description | Exemple | Résultat |
|---|---|---|---|
| `\|\|` | concaténation | `B'10001' \|\| B'011'` | `10001011` |
| `&` | AND bit à bit | `B'10001' & B'01101'` | `00001` |
| `\|` | OR bit à bit | `B'10001' \| B'01101'` | `11101` |
| `#` | XOR bit à bit | `B'10001' # B'01101'` | `11100` |
| `~` | NOT bit à bit | `~ B'10001'` | `01110` |
| `<<` | décalage gauche bit à bit | `B'10001' << 3` | `01000` |
| `>>` | décalage droit bit à bit | `B'10001' >> 2` | `00100` |

Les fonctions SQL suivantes fonctionnent sur les chaînes de bits ainsi que sur les chaînes de caractères : `length`, `bit_length`, `octet_length`, `position`, `substring`.

De plus, il est possible de convertir des valeurs intégrales en ou à partir du type `bit`. Quelque exemples :

```
44::bit(10)                    0000101100
44::bit(3)                     100
cast(-44 as bit(12))           111111010100
'1110'::bit(4)::integer        14
```

Notez que la conversion de << bit >> signifie la conversion de `bit(1)` et, du coup, il délivrera seulement le bit de poids faible de l'entier.

> **Note :** Avant PostgreSQL 8.0, convertir une entier en `bit(n)` aurait copié les `n` bits les plus à gauche de l'entier alors que, maintenant, il copie les `n` bits les plus à droite. De plus, convertir un entier en une chaîne de bits d'une largeur plus grande que l'entier lui−même changera le signe côté gauche.

# 9.7. Correspondance de modèles

Il existe trois différents approches aux correspondances de modèles fournies par PostgreSQL: l'opérateur SQL traditionnel `LIKE`, le plus récent `SIMILAR TO` (ajouté dans SQL:1999) et les expressions rationnelles du type POSIX. De plus, une fonction de correspondance de modèles, `substring`, est disponible. Elle utilise soit le style `SIMILAR TO` soit le style POSIX des expressions rationnelles.

> **Astuce :** Si vous avez besoin des correspondances de modèles qui vont au delà de ça, considérez l'écriture d'une fonction en Perl ou Tcl.

### 9.7.1. LIKE

```
chaîne LIKE modèle
[ESCAPE caractère d'échappement]
chaîne NOT LIKE modèle
[ESCAPE caractère d'échappement]
```

Chaque *modèle* définit un ensemble de chaîne. L'expression LIKE renvoie true si la *chaîne* est contenu dans l'ensemble de chaînes représenté par le *modèle*. (Comme attendu, l'expression NOT LIKE renvoie false si LIKE renvoie true et vice versa. Une expression équivalente est NOT (*chaîne* LIKE *modèle*).)

Si le *modèle* ne contient pas de signe de pourcentage ou de tirets bas, alors le modèle représente seulement la chaîne elle−même ; dans ce cas, LIKE agit exactement comme l'opérateur d'égalité. Un tiret bas (_) dans *modèle* correspond à un seul caractère ; un signe de pourcentage (%) correspond à toutes les chaînes de zéro à plus de caractères.

Quelques exemples :

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

Le modèle LIKE correspond toujours à la chaîne entière. Pour faire correspondre une séquence à l'intérieur d'une chaîne, le modèle doit donc commencer et finir avec un signe de pourcentage.

Pour faire correspondre un vrai tiret bas ou un vrai signe de pourcentage sans qu'ils remplacent d'autres caractères, le caractère correspondant dans *modèle* doit être précédé du caractère d'échappement. Par défaut, il s'agit de l'antislash mais un autre caractère peut être sélectionné en utilisant la clause ESCAPE. Pour correspondre au caractère d'échappement lui−même, écrivez deux fois ce caractère.

Notez que l'antislash a déjà une signification particulière dans les chaînes littérales, donc écrite une constante du modèle avec un antislash signifie écrire quatre antislash dans l'instruction. Vous pouvez éviter ceci en sélectionnant un autre caractère d'échappement avec ESCAPE ; ensuite un antislash n'est plus spécial dans un LIKE. (Mais il est toujours spécial dans l'analyseur de chaînes littérales, donc vous aurez besoin des deux.)

Il est aussi possible de ne sélectionner aucun caractère d'échappement en écrivant ESCAPE ''. Ceci désactive complètement le mécanisme d'échappement, ce qui rend impossible la désactivation de la signification particulière du tiret bas et du signe de pourcentage dans le modèle.

Le mot clé ILIKE est utilisé à la place de LIKE pour faire des correspondances sans tenir compte de la casse mais en tenant compte de la locale active. Ceci ne fait pas partie du standard SQL mais est une extension PostgreSQL.

L'opérateur ~~ est équivalent à LIKE alors que ~~* correspond à ILIKE. Il existe aussi les opérateurs !~~ et !~~* représentant respectivement NOT LIKE et NOT ILIKE. Tous ces opérateurs sont spécifiques à PostgreSQL.

## 9.7.2. Expressions rationnelles SIMILAR TO

```
chaîne SIMILAR TO modèle
[ESCAPE caractère d'échappement]
chaîne NOT SIMILAR TO
modèle [ESCAPE
caractère d'échappement]
```

L'opérateur SIMILAR TO renvoie true ou false suivant le fait que son modèle corresponde ou non à la chaîne donnée. C'est identique à LIKE sauf qu'il interprète le modèle en utilisant la définition SQL d'une expression rationnelle. Les expressions rationnelles SQL99 sont un curieux mélange de la notation LIKE et de la notation habituelle des expressions rationnelles.

Comme LIKE, l'opérateur SIMILAR TO réussit uniquement si son modèle correspond à la chaîne entière ; ceci ne ressemble pas aux pratiques habituelles des expressions rationnelles où le modèle peut se situer n'importe où dans la chaîne. Ainsi comme LIKE, SIMILAR TO utilise _ et % comme caractères joker dénotant respectivement un caractère seul et toute chaîne (ils sont comparables à . and .* dans les expressions compatibles POSIX).

En plus de ces fonctionnalités empruntées à LIKE, SIMILAR TO supporte trois méta−caractères de correspondance de modèle empruntés aux expressions rationnelles de POSIX :

- | dénote le contraire (une des deux alternatives).
- * dénote la répétition des précédents éléments, entre zéro et un nombre illimité de fois.
- + dénote la répétition des précédents éléments, entre une et un nombre illimité de fois.
- Parenthèses () peuvent être utilisées pour grouper des éléments en un seul élément logique.
- Une expression entre crochets [...] spécifie une classe de caractères, comme dans les expressions rationnelles POSIX.

Notez que les répétitions (? and {...}) ne sont pas permises bien qu'elles existent en POSIX. De même, le point (.) n'est pas un méta−caractère.

Comme avec LIKE, un antislash désactive la signification spéciale de tous les méta−caractères ; un autre caractère d'échappement peut être spécifié avec ESCAPE.

Quelques exemples :

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

La fonction substring avec trois paramètres, substring(chaîne from *modèle* for *caractère d'échappement*), fournit une extraction qu'une sous−chaîne correspondant à un modèle d'expression rationnelle SQL. Comme avec SIMILAR TO, le modèle spécifié doit correspondre à l'entière chaîne de données, sinon la fonction échouera et renverra NULL. Pour indiquer la partie du modèle qui devrait être renvoyée en cas de succès, le modèle doit contenir deux occurrences du caractères d'échappement suivi d'un guillemet double ("). Le texte correspondant à la portion du modèle entre ces deux marqueurs est renvoyé.

Quelques exemples :

```
substring('foobar' from '%#"o_b#"%' for '#')
oob
substring('foobar' from '#"o_b#"%' for '#')
NULL
```

## 9.7.3. Expressions rationnelles POSIX

Tableau 9−11 liste les opérateurs disponibles pour la correspondance de modèles en utilisant les expressions rationnelles POSIX.

**Tableau 9−11. Opérateurs de correspondance des expressions rationnelles**

| Opérateur | Description | Exemple |
|---|---|---|
| ~ | Correspond à l'expression rationnelle, en tenant compte de la casse | `'thomas' ~ '.*thomas.*'` |
| ~* | Correspond à l'expression rationnelle, sans tenir compte de la casse | `'thomas' ~* '.*Thomas.*'` |
| !~ | Ne correspond pas à l'expression rationnelle, en tenant compte de la casse | `'thomas' !~ '.*Thomas.*'` |
| !~* | Ne correspond pas à l'expression rationnelle, sans tenir compte de la casse | `'thomas' !~* '.*vadim.*'` |

Les expressions rationnelles POSIX fournissent un moyen plus puissant pour la correspondance de modèles par rapport aux opérateurs `LIKE` et `SIMILAR TO`. Beaucoup d'outils Unix comme `egrep`, `sed` ou `awk` utilise un langage de correspondance de modèles similaire à celui décrit ici.

Une expression rationnelle est une séquence de caractères qui est une définition abrégée d'un ensemble de chaînes (un *ensemble rationnel*). Une chaîne est dite correspondante à une expression rationnelle si elle est membre de l'ensemble rationnel décrit par l'expression rationnelle. Comme avec `LIKE`, les caractères modèles correspondent exactement aux caractères de chaîne sauf s'ils représentent des caractères spéciaux dans le langage des expressions rationnelles mais les expressions rationnelles utilisant des caractères spéciaux différents de ceux de `LIKE`. Contrairement aux modèles pour `LIKE`, une expression rationnelle est autorisée à correspondre quelque part dans la chaîne, sauf si l'expression rationnelle est explicitement ancrée au début ou à la fin de la chaîne.

Quelques exemples :

```
'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^(b|c)' false
```

La fonction `substring` avec deux paramètres, `substring(chaîne from modèle)`, fournit une extraction d'une sous−chaîne correspondant à un modèle d'expression rationnelle POSIX. Il renvoie NULL s'il n'y a pas de correspondance et la portion de texte correspondant au modèle dans le cas contraire. Mais si le modèle contient des parenthèses, la portion de texte ayant correspondu à la première sous−expression entre parenthèses (la première dont la parenthèse gauche apparaît) est renvoyée. Vous pouvez placer ces parenthèses entre l'expression complète si vous voulez utiliser des parenthèses à l'intérieur sans déclencher cette exception. Si vous avez besoin des parenthèses dans le modèle avant la sous−expression que vous voulez extraire, voir les parenthèses sans capture décrites plus bas.

Quelques exemples :

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

Les expressions rationnelles de PostgreSQL sont implémentées en utilisant un paquetage écrit par Henry Spencer. La plupart des descriptions d'expressions rationnelles ci−dessous sont copiés directement de sa page man.

### 9.7.3.1. Détails des expressions rationnelles

Les expressions rationnelles (REs), ainsi qu'elles sont définies dans POSIX 1003.2, viennent en deux formats : les ER *étendues* ou ERE (en gros celles de `egrep`) et les ER *basiques* ou ERB (principalement celles d'`ed`). PostgreSQL supporte les deux formes et implémente aussi quelques extensions ne faisant pas partie du standard POSIX mais devenant malgré tout de plus en plus populaire à cause de leur disponibilité dans les langages de programmation tels que Perl et Tcl. Les ER utilisant ces extensions non POSIX sont appelées des ER *avancées* ou des ARE dans cette documentation. Les ARE sont un surensemble exact des ERE alors que les ERB ont des incompatibilités de notation (sauf parler du fait qu'elles sont bien plus limitées). Nous décrivons tout d'abord les formats ARE et ERE, notant les fonctionnalités qui s'appliquent seulement aux ARE et puis nous décrivons la différence avec les ERB.

> **Note :** La forme des expressions rationnelles acceptées par PostgreSQL peut être choisie en initialisant le paramètres à l'exécution regex_flavor (décrit dans Section 16.4). La configuration habituelle est `advanced` (NdT : pour avancées) mais il est possible de sélectionner `extended` (pour étendues) avec un maximum de compatibilité avec les versions antérieures à la 7.4 de PostgreSQL.

Une expression rationnelle est définie par une ou plusieurs branches *branches* séparées par des caractères `|`. Elle établit une correspondance avec tout ce qui correspond à une des branches.

Une branche contient des *atomes quantifiés* ou des *contraintes*, concaténés. Elle établit une correspondance pour le premier suivie d'une correspondance pour le second, etc ; une branche vide établit une correspondance avec une chaîne vide.

Un atome quantifié est un *atome* suivi le cas échéant d'un seul *quantificateur*. Sans quantificateur, il établit une correspondance avec l'atome. Avec un quantificateur, il peut établir autant de correspondances avec l'atome que possible. Un *atome* peut être toute possibilité montrée dans Tableau 9−12. Les quantifieurs possibles et leurs significations sont disponibles dans Tableau 9−13.

Une *contrainte* correspond à une chaîne vide mais correspond seulement si des conditions spécifiques sont rencontrées. Une contrainte peut être utilisée sauf qu'elle ne peut pas être suivie d'un quantificateur. Les contraintes simples sont affichées dans Tableau 9−14 ; quelques contraintes supplémentaires sont décrites après.

**Tableau 9−12. Atomes d'expressions rationnelles**

| Atome | Description |
|---|---|
| `(`*re*`)` | (où *re* est toute expression rationnelle) établie une correspondance avec *re*, la correspondance étant conservée pour un rapport possible |

| | |
|---|---|
| `(?:re)` | comme ci−dessus mais la correspondance n'est pas conservée (un ensemble de parenthèses << sans capture >>) (seulement ARE) |
| `.` | correpond à tout caractère seul |
| `[caractères]` | une *expression entre crochets*, correspondant à n'importe lequel des `caractères` (voir Section 9.7.3.2 pour plus de détails) |
| `\k` | (où `k` est tout sauf un caractère alpha−numérique) établit une correspondance avec un caractère ordinaire, c'est−à−dire `\\` correspond à un caractère antislash |
| `\c` | où `c` est un caractère alphanumérique (probablement suivi d'autres caractères) est un *échappement*, voir Section 9.7.3.3 (ARE seulement ; pour les ERE et ERB, ceci correspond à `c`) |
| `{` | lorsqu'il est suivi d'un caractère autre qu'un chiffre, établit une correspondance avec l'accolade ouvrante `{` ; si elle est suivie d'un chiffre, c'est le début d'une `limite` (voir ci−dessous) |
| `x` | où `x` est un simple caractère sans signification et correspondant donc à ce caractère |

Une ER pourrait ne pas terminer avec `\`.

> **Note :** Rappelez−vous que l'antislash (`\`) a déjà une signification particulière dans les chaînes littérales PostgreSQL. Pour écrire un modèle constant contenant un antislash, vous devez écrire deux antislashs dans l'instruction.

**Tableau 9−13. Quantifiant d'expressions rationnelles**

| Quantifiant | Correspondance |
|---|---|
| `*` | une séquence de 0 ou plus correspondance de l'atome |
| `+` | une séquence de 1 ou plus correspondance de l'atome |
| `?` | une séquence 0 ou 1 correspondance de l'atome |
| `{m}` | une séquence d'exactement `m` correspondance de l'atome |
| `{m,}` | une séquence de `m` ou plus correspondance de l'atome |
| `{m,n}` | une séquence de `m` à `n` (inclus) correspondance de l'atome ; `m` ne doit pas être supérieur à `n` |
| `*?` | version non gourmande de `*` |
| `+?` | version non gourmande de `+` |
| `??` | version non gourmande de `?` |
| `{m}?` | version non gourmande de `{m}` |
| `{m,}?` | version non gourmande de `{m,}` |
| `{m,n}?` | version non gourmande de `{m,n}` |

Les formes utilisant `{...}` sont connus comme des *limite*s. Les nombres `m` et `n` à l'intérieur d'une limite sont des entiers décimaux non signés avec des droits allant de 0 à 255 inclus.

Les quantifiants *non gourmands* (disponibles uniquement avec les ERA) correspondent aux même possibilités que leur équivalent normal (*gourmand*) mais préfèrent le plus petit nombre plutôt que le plus grand nombre de correspondance. Voir Section 9.7.3.5 pour plus de détails.

> **Note :** Un quantifiant ne peut pas immédiatement suivre un autre quantifiant. Un quantifiant ne peut pas commencer une expression ou une sous−expression ou suivre `^` ou `|`.

**Tableau 9−14. Contraintes des expressions rationnelles**

| Contrainte | Description |
|---|---|
| ^ | correspond au début d'une chaîne |
| $ | correspond à la fin d'une chaîne |
| (?=*re*) | *positive lookahead* correspond à tout point où une sous−chaîne correspondant à *re* commence (uniquement pour les ERA) |
| (?!*re*) | *negative lookahead* correspond à tout point où aucune sous−chaîne correspondant à *re* commence (uniquement pour les ERA) |

Les contraintes << lookahead >> ne doivent pas contenir de *références arrières* (voir Section 9.7.3.3), et toutes les parenthèses contenues sont considérées comme non capturantes.

### 9.7.3.2. Expressions avec crochets

Une *expression entre crochets* est une liste de caractères contenus dans [ ]. Cela correspond normalement à tout caractère de la liste (mais voir plus bas). Si la liste commence avec ^, cela correspond à tout caractère *non* compris dans la liste. Si deux caractères de la liste sont séparés par un tiret (−), c'est un raccourci pour tous les caractères compris entre ces deux, c'est−à−dire qu'en ASCII, [0−9] correspond à tout chiffre. Il est illégal d'avoir deux séquences partageant la même fin, par exemple a−c−e. Ces séquences dépendent grandement de la façon dont elles sont créées, donc les programmes portables devraient éviter leur utilisation.

Pour inclure un littéral ] dans la liste, faites en sorte qu'il soit le premier caractère (suivant un possible ^). Pour inclure un littéral −, faites en sorte qu'il soit le premier ou le dernier caractère, ou qu'il soit dans le deuxième point final d'une séquence. Pour utiliser un littéral − comme premier point final d'une séquence, englobez−le dans [. et .] pour en faire un élément de cohésion (voir ci−dessous). Avec l'exception de ces caractères, quelques combinaisons utilisant [ (voir les paragraphes suivants) et les échappements (uniquement pour les ERA), tous les autres caractères spéciaux perdent leur signification spéciale à l'intérieur d'une expression entre crochets. En particulier, \ n'est pas spécial lorsqu'il suit les règles des ERE ou des ERB bien qu'il soit spécial (en tant qu'introduction d'un échappement) dans les ERA.

À l'intérieur d'une expression entre crochets, un élément liant (un caractère, une séquence de caractères multiples qui se suivent comme s'il n'y avait qu'un seul caractère, ou le nom d'une séquence liée) englobé dans [. et .] correspond à une séquence de caractères de cet élément liant. La séquence est un simple élément de la liste de l'expression entre crochets. Une expression entre crochets contenant un élément liant avec plusieurs caractères peut donc correspondre à plus d'un caractère, c'est−à−dire que si la séquence liante inclut un élément liant ch, alors la ER [[.ch.]]*c établit une correspondance avec les cinq premiers caractères de chchcc.

> **Note :** PostgreSQL n'a pas d'éléments multi−caractères qui se suivent. Cette information décrit les comportements futurs possibles.

À l'intérieur d'une expression entre crochets, un élément liant englobé dans [= et =] est une classe d'équivalence, correspondant aux séquences de caractères de tous les éléments liant équivalent à celui−là, lui−même étant compris. (S'il n'existe pas d'éléments liants correspondants, le traitement est comme si les délimiteurs englobant étaient [. et .].) Par exemple, si o et ^ sont les membres d'une classe d'équivalence, alors [[=o=]], [[=^=]] et [o^] sont tous synonymes. Une classe d'équivalence ne peut pas être le point final d'une séquence.

À l'intérieur d'une expression entre crochets, le nom d'une classe de caractères englobé dans [: et :]

correspond à la liste de tous les caractères appartenant à cette classe. Les noms de classes de caractères standards sont `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`. Ils correspondent aux classes de caractère définies dans ctype. Une locale pourrait en fournir d'autres. Une classe de caractères pourrait ne pas être utilisée comme point final d'une séquence.

Il existe deux cas spéciaux d'expressions entre crochets : les expressions entre crochets `[[:<:]]` et `[[:>:]]` sont contraintes, correspondant à des chaînes vides respectivement au début et à la fin d'un mot. Un mot est défini comme une séquence de caractères qui n'est ni précédée ni suivie de caractères. Un caractère de mot est un caractère `alnum` (comme défini par ctype) ou un tiret bas. C'est une extension, compatible avec mais non spécifiée dans POSIX 1003.2, et devrait être utilisé avec précaution dans les logiciels conçus pour être portables sur d'autres systèmes. Les échappements de contraintes décrites ci−dessous sont généralement préférables (elles ne sont pas plus standard mais elles sont certainement plus simple à saisir).

### 9.7.3.3. Échappement d'expressions rationnelles

Les *échappements* sont des séquences spéciales commençant avec \ et suivies d'un caractère alphanumérique. Les échappements viennent en plusieurs variétés : entrée de caractère, raccourci de classe et références. Un \ suivi d'un caractère alphanumérique mais ne constituant pas un échappement valide est illégal dans une ERA. Pour les ERE, il n'y pas d'échappement : en dehors d'une expression entre crochets, un \ suivi d'un caractère alphanumérique signifie simplement ce caractère (ordinaire) et, à l'intérieur d'une expression entre crochets, \ est un caractère ordinaire. (ce dernier est la vraie incompatibilité entre les ERE et les ERA.)

Les *échappements d'entrée de caractère* existent pour faciliter la spécification de caractères non affichables ou indésirables pour toute autre raison dans les ER. Ils sont disponibles dans Tableau 9−15.

Les *échappements de raccourci de classe* fournissent des raccourcis pour des classes de caractères communément utilisées. Ils sont disponibles dans Tableau 9−16.

Un *échappement à contrainte* est une contrainte, correspondant à la chaîne vide si les conditions spécifiques sont rencontrées, écrite avec un échappement. Ils sont disponibles dans Tableau 9−17.

Une *référence* (*n*) correspond à la même chaîne que la sous−expression précédente entre parenthèses spécifiée par le nombre *n* (voir Tableau 9−18). Par exemple, `([bc])\1` correspond à `bb` ou `cc` mais ni `bc` ni `cb`. La sous−expression doit complètement précédée la référence dans la ER. Les sous−expressions sont numérotées dans l'ordre des parenthèses ouvrantes. Les parenthèses non capturantes ne définissent pas de sous−expressions.

> **Note :** Gardez à l'esprit qu'un symbole d'échappement \ devra être doublé si vous souhaitez le saisir comme une chaîne SQL constante. Par exemple :
>
> `'123' ~ '^\\d{3}'` *true*

**Tableau 9−15. Échappements d'entrée de caractère des expressions rationnelles**

| Échappement | Description |
|---|---|
| \a | caractère alerte (sonnerie), comme en C |
| \b | effacement, comme en C |
| \B | synonyme de \ pour aider à la réduction des doubles antislashs |
| \c*X* | |

| | |
|---|---|
| | (où *X* est un caractère quelconque) le caractère dont les 5 bits de poids faible sont les mêmes que ceux de *X* et dont tous les autres bits sont à zéro |
| \e | le caractère dont le nom de séquence liante est ESC ou, en dehors de ça, le caractère de valeur octale 033 |
| \f | retour chariot, comme en C |
| \n | retour à la ligne, comme en C |
| \r | retour à la ligne, comme en C |
| \t | tabulation horizontale, comme en C |
| \u*wxyz* | (où *wxyz* est exactement quatre chiffres hexadécimaux) le caractère Unicode U+*wxyz* dans l'ordre des octets locaux |
| \U*stuvwxyz* | (où *stuvwxyz* représentent exactement huit chiffres héxadécimaux) réservé pour une extension Unicode vers le 32 bits, un peu hypothétique |
| \v | tabulation verticale, comme en C |
| \x*hhh* | (où *hhh* est toute séquence de chiffres héxadécimaux) le caractère dont la valeur hexadécimale est 0x*hhh* (un simple caractère, peu importe le nombre de chiffres hexadécimaux utilisés) |
| \0 | le caractère dont la valeur est 0 |
| *xy* | (où *xy* représente exactement deux chiffres octaux et qui n'est pas une *référence*) le caractère dont la valeur octale est 0*xy* |
| *xyz* | (où *xyz* représente exactement trois chiffres octaux et qui n'est pas une *référence*) le caractère dont la valeur octale est 0*xyz* |

Les chiffres hexadécimaux sont 0–9, a–f et A–F. Les chiffres octaux sont 0–7.

Les échappements d'entrée de caractère sont toujours pris comme des caractères ordinaires. Par exemple, \135 est ] en ASCII mais \135 ne termine pas une expression entre crochets.

**Tableau 9–16. Échappement de raccourcis de classe des expressions rationnelles**

| Échappement | Description |
|---|---|
| \d | [[:digit:]] |
| \s | [[:space:]] |
| \w | [[:alnum:]_] (notez que le tiret bas est inclus) |
| \D | [^[:digit:]] |
| \S | [^[:space:]] |
| \W | [^[:alnum:]_] (notez que le tiret bas est inclus) |

À l'intérieur d'expressions entre crochets, \d, \s, et \w perdent leur crochet externe, et \D, \S et \W sont illégales. (Donc, par exemple, [a-c\d] est équivalent à [a-c[:digit:]]. De plus, [a-c\D], qui est équivalent à [a-c^[:digit:]], est illégal.)

**Tableau 9–17. Échappements de contraintes des expressions rationnelles**

| Échappement | Description |
|---|---|
| \A | correspond seulement au début de la chaîne (voir Section 9.7.3.5 pour comprendre comment ceci diffère de ^) |

| | |
|---|---|
| \m | correspond seulement au début d'un mot |
| \M | correspond seulement à la fin d'un mot |
| \y | correspond seulement au début ou à la fin d'un mot |
| \Y | correspond seulement à un point qui n'est ni le début ni la fin d'un mot |
| \Z | correspond seulement à la fin d'une chaîne (voir Section 9.7.3.5 pour comprendre comment ceci diffère de $) |

Un mot est défini suivant la spécification de [[:<:]] et [[:>:]] ci−dessus. Les contraintes d'échappements sont illégales à l'intérieur d'expressions entre crochets.

**Tableau 9−18. Références dans les expressions rationnelles**

| Échappement | Description |
|---|---|
| *m* | (où *m* est un chiffre différent de zéro) une référence de la *m*ème sous−expression |
| *mnn* | (où *m* est un chiffre différent de zéro et *nn* quelques chiffres supplémentaires et la valeur décimale *mnn* n'est pas plus grande que le nombre de parenthèses fermantes capturantes vues jusqu'à maintenant) une référence de la *mnn*ème sous−expression |

**Note :** Il existe une ambiguïté historique inhérente entre les échappements d'entrée de caractères en octal et les référencées, ambiguïté résolue par des heuristiques, comme montré ci−dessus. Un zéro en début indique toujours un échappement octal. Un seul caractère différent de zéro et suivi d'un autre caractère est toujours pris comme une référence. Une séquence à plusieurs chiffres ne commençant pas avec un zéro est pris comme une référence si une sous−expression convenable (c'est−à−dire que le nombre fait partie des numéros de référence). Dans le cas contraire, il est pris pour un nombre octal.

### 9.7.3.4. Métasyntaxe des expressions rationnelles

En plus de la syntaxe principale décrite ci−dessus, il existe quelques formes spéciales et autres possibilités syntaxiques disponibles.

Normalement, la recette de ER en cours d'utilisation est déterminé par regex_flavor. Néanmoins, cela peut être surchargé par un préfixe *directeur*. Si un ER commence avec ***:, le reste de l'ER est considéré comme une ERA, quelque soit regex_flavor. Si un ER commence avec ***=, le reste de l'ER est considéré comme une chaîne littérale, tous les caractères étant donc considérés ordinaires.

Une ERA pourrait commencer avec des *options intégrées* : une séquence (?*xyz*) (où *xyz* correspond à un ou plusieurs caractères alphabétiques) spécifie les options affectant le reste de l'ER. Ces options surchargent toutes options précédemment déterminées (incluant le type de l'ER et la sensibilité à la casse). Les lettres d'options disponibles sont indiquées dans Tableau 9−19.

**Tableau 9−19. Lettres d'option intégré à une ERA**

| Option | Description |
|---|---|
| b | le reste de l'ER est une ERB |
| c | activation de la sensibilité à la casse (surcharge l'opérateur type) |
| e | le reste de l'ER est une ERE |
| i | désactivation de la sensibilité à la casse (voir Section 9.7.3.5) (surcharge l'opérateur type) |

| | |
|---|---|
| m | synonyme historique pour n |
| n | activation de la sensibilité aux nouvelles lignes (voir Section 9.7.3.5) |
| p | activation partielle de la sensibilité aux nouvelles lignes (voir Section 9.7.3.5) |
| q | le reste de l'ER est une chaîne littérale (<< entre guillemets >>), composé uniquement de caractères ordinaires |
| s | désactivation de la sensibilité aux nouvelles lignes (par défaut) |
| t | syntaxe légère (par défaut ; voir ci−dessous) |
| w | désactivation de la sensibilité aux nouvelles lignes (<< étrange >>) correspondantes (voir Section 9.7.3.5) |
| x | syntaxe étendue (voir ci−dessous) |

Les options intégrées prennent effet au ) terminant la séquence. Elles pourraient seulement apparaître au début d'une ERA (après le directeur ***: s'il y en a un).

En plus de la syntaxe habituelle d'une ER (*légère*), dans laquelle tous les caractères ont une signification, il existe une syntaxe *étendue*, disponible en signifiant l'option intégrée x. Avec la syntaxe étendue, les caractères d'espace blanc d'une ER sont ignorés comme le sont tous les caractères entre un # et la nouvelle ligne suivante (ou la fin de l'ER). Ceci permet le commentaire d'une ER complexe. Il existe trois exceptions à cette règle de base :

- un caractère espace blanc # suivi d'un \ est retenu
- un caractère espace blanc # à l'intérieur d'une expression entre crochets est retenu
- un espace blanc et des commentaires ne peuvent pas apparaître à l'intérieur de symboles multi−caractères, tels que (?:

Dans ce cas, les caractères d'espace blanc sont l'espace, la tabulation, le retour chariot et tout caractère appartenant à la classe de caractère *space*.

Enfin, dans une ARE, à l'extérieur des expressions entre crochets, la séquence (?#*ttt*) (où *ttt* est tout texte ne contenant pas )) est un commentaire totalement ignoré. De nouveau, cela n'est pas permis entre les caractères des symboles multi−caractères comme (?:. De tels commentaires sont plus un artéfact historique qu'une fonctionnalité utile et leur utilisation est obsolète ; utilisez plutôt la syntaxe étendue.

*Aucune* de ces extensions métasyntaxe n'est disponible si un directeur initial ***= a spécifié que la saisie de l'utilisateur doit être traité comme une chaîne littérale plutôt que comme une ER.

### 9.7.3.5. Règles de correspondance des expressions rationnelles

Dans l'hypothèse qu'une ER pourrait correspondre à plus d'une sous−chaîne d'une chaîne donnée, l'ER correspond à la première. De même, dans l'hypothèse que l'ER pourrait correspondre à plus d'une sous−chaîne commençant au même endroit, soit la correspondance la plus longue possible soit la correspondance la plus courte possible sera prise suivant si l'ER est *affamée* ou *non−affamée*.

Le fait qu'une ER est affamée ou non est déterminé par les règles suivantes :

- La plupart des atomes, et toutes les contraintes, n'ont pas d'attribut de faim (parce qu'ils ne correspondent pas à des quantités variables de texte de toute façon).
- Ajouter des parenthèses autour d'une ER ne change pas sa faim.
- Un atome quantifié avec un quantificateur à répétition fixe ({*m*} ou {*m*}?) a la même faim (peut−être aucune) que l'atome lui−même.

- Un atome quantifié avec d'autres quantifieurs standards (incluant `{m,n}` avec *m* égal à *n*) est affamé (préfère la plus grande correspondance).
- Un atome quantifié avec un quantificateur non affamé (incluant `{m,n}?` avec *m* égal à *n*) n'est pas affamé (préfère la plus courte correspondance).
- Une branche — c'est–à–dire une ER qui n'a pas d'opérateur `|` de haut niveau — est aussi affamé que le premier atome quantifié dans lui qui a un attribut de faim.
- Une ER consistant en deux branches, ou plus, connectées par l'opérateur `|` est toujours affamée.

Les règles ci–dessus associent les attributs de faim pas seulement avec les atomes quantifiés individuels, mais aussi avec les branches et les ER entiers contenant des atomes quantifiés. Cela signifie que la correspondance est faite d'une telle façon que la branche, ou l'ER complète, correspond à la sous–chaîne la plus longue ou la plus courte possible *comme un tout*. Une fois la longueur de la longueur entière déterminée, la partie en lui qui correspond à toute sous–expression particulière est déterminée sur la base de l'attribut de faim de cette sous–expression, avec les sous–expressions commençant plus tôt dans l'ER ayant priorité sur celles commençant après.

Un exemple de ce que cela signifie :

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Result: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Result: 1
```

Dans le premier cas, l'ER est affamé en un tout parce que `Y*` est affamé. Elle peut correspondre au début de `Y` et cela correspond à la chaîne la plus longue commençant à partir de là, donc `Y123`. La sortie est la partie entre parenthèses, soit `123`. Dans le second cas, l'ER en un tout n'est pas affamé car `Y*?` ne l'est pas non plus. Cela peut correspondre au début de `Y` et cela correspond à la chaîne la plus courte commençant là, donc `Y1`. La sous–expression `[0-9]{1,3}` est affamée mais cela ne peut pas changer la décision sur la longueur totale de la correspondance ; donc, cela le force à correspondre à seulement `1`.

En bref, quand une ER contient à la fois des sous–expressions affamées et non affamées, la longueur de la correspondance totale est soit aussi longue que possible soit aussi courte que possible, suivant l'attribut affecté à l'ER complète. Les attributs affectés aux sous–expressions affectent seulement quelles parties de cette correspondance ils sont autorisés à << manger >>.

Les quantifieurs `{1,1}` et `{1,1}?` peuvent être utilisés pour forcer la préférence la plus longue ou la plus courte, respectivement, sur une sous–expression ou sur l'ER complète.

Les longueurs de correspondance sont mesurés en caractères, et non pas en éléments liants. Une chaîne vide est considérée comme plus grande que pas de correspondance du tout. Par exemple : `bb*` correspond aux trois caractères du milieu de `abbbc` ; `(week|wee)(night|knights)` correspond aux dix caractères de `weeknights` ; lorsque `(.*).*` correspond à `abc`, la sous–expression entre parenthèses correspond aux trois caractères ; et quand `(a*)*` correspond à `bc`, à la fois l'ER et la sous–expression entre parenthèses correspondent à une chaîne vide.

Si la correspondance ne tient pas compte de la casse, l'effet revient à ce que toutes les distinctions de casse aient disparu de l'alphabet. Quand un caractère alphabétique, existant en plusieurs cas, apparaît comme un caractère ordinaire en dehors d'une expression entre crochets, il est effectivement transformé en une expression entre crochets contenant les deux cas, c'est–à–dire que `x` devient `[xX]`. Quand il apparaît dans une expression entre crochets, toutes les transformations de casse sont ajoutées à l'expression entre crochets, c'est–à–dire que `[x]` devient `[xX]` et que `[^x]` devient `[^xX]`.

Si la sensibilité aux retours chariots est spécifiée, `.` et les expressions entre crochets utilisant ne correspondront jamais au caractère de retour à la ligne (de façon à ce que les correspondances ne croisent jamais les retours chariots sauf si l'ER arrange ceci explicitement), et `^` et `$` correspondront à la chaîne vide respectivement après et avant un retour chariot, en plus de correspondre respectivement au début et à la fin de la chaîne. Mais les échappements ERA `\A` and `\Z` continuent de correspondre *seulement* au début et à la fin de la chaîne.

Si la sensibilité partielle aux retours chariot est spécifiée, ceci affecte `.` et les expressions entre crochets comme avec la sensibilité aux retours chariot mais sans `^` et `$`.

Si la sensibilité partielle inverse aux retours chariot est spécifiée, ceci affecte `^` et `$` comme avec la sensibilité aux retours chariot mais sans `.` et les sous−expressions. Ceci n'est pas très utile mais est fournie pour des raisons de symétrie.

### 9.7.3.6. Limites et compatibilité

Aucune limite particulière n'est imposée sur la longueur des ER dans cette implémentation. Néanmoins, les programmes souhaitant être particulièrement portables ne devraient pas employer d'ER de plus de 256 octets car l'implémentation POSIX peut refuser d'accepter de telles ER.

La seule fonctionnalité des ERA, incompatible avec les ERE POSIX, est que `\` ne perd pas sa signification spéciale à l'intérieur des expressions entre crochets. Toutes les autres fonctionnalités ERA utilisent une syntaxe qui est illégale ou a des effets indéfinis ou non spécifiés dans les ERE POSIX ; la syntaxe `***` des directeurs est en dehors de la syntaxe POSIX pour les ERB et les ERE.

Un grand nombre des extensions ERA sont empruntées de Perl mais certaines ont été modifiées pour les nettoyer et quelques extensions Perl ne sont pas présentes. Les incompatibilités incluent `\b`, `\B`, le manque de traitement spécial pour le retour à la ligne en fin de chaîne, l'ajout d'expressions entre crochets aux expressions affectées par les correspondance avec retour à la ligne, les restrictions sur les parenthèses et les références dans les contraintes, et la correspondance des chaînes suivant leur taille (au lieu de la première rencontrée).

Deux incompatibilités importantes existent entre les syntaxes ERA et ERE reconnues par les pré−versions 7.4 de PostgreSQL:

- Dans les ERA, `\` suivi d'un caractère alphanumérique est soit un échappement soit une erreur alors que dans les versions précédentes, c'était simplement un autre moyen d'écrire un caractère alphanumérique. Ceci ne devrait pas poser trop de problèmes car il n'existe aucune raison pour écrire une telle séquence dans les versions précédentes.
- Dans les ERA, `\` reste un caractère spécial y compris à l'intérieur de `[]`, donc un `\` littéral à l'intérieur d'une expression entre crochets doit être écrit `\\`.

Alors que ces différences ne devraient pas poser problèmes pour la plupart des applications, vous pouvez les éviter si nécessaire en initialisant `regex_flavor` à `extended`.

### 9.7.3.7. Expressions rationnelles basiques

Les ERB diffèrent des ERE sur plusieurs aspects. `|`, `+` et `?` sont des caractères ordinaires et il n'existe pas d'équivalent pour leur fonctionnalité. Les délimiteurs sont `\{` et `\}`, avec `{` et `}` étant eux−même des caractères ordinaires. Les parenthèses pour les sous−expressions imbriquées sont `\(` et `\)`, avec `(` et `)` étant

eux−mêmes des caractères ordinaires. ^ est un caractère ordinaire sauf au début d'une ER ou au début d'une sous−expression entre parenthèses, $ est un caractère ordinaire sauf à la fin d'une ER ou à la fin d'une sous−expression entre parenthèses et * est un caractère ordinaire s'il apparaît au début d'une ER ou au début d'une sous−expression entre parenthèses (après un possible ^). Enfin, les références à un seul chiffre sont disponibles, et \< et \> sont des synonymes pour respectivement [[:<:]] et [[:>:]] ; aucun autre échappement n'est disponible.

# 9.8. Fonctions de formatage des types de données

Les fonctions de formatage de PostgreSQL fournissent un ensemble d'outils puissants pour convertir différent types de données (date/heure, entier, nombre à virgule flottante, numérique) en des chaînes formatées et pour convertir des chaînes formatées en des types de données spécifiques. Tableau 9−20 les listes. Ces fonctions suivent toutes une convention d'appels commune : le premier argument est la valeur à formater et le second argument est un modèle définissant le format de sortie ou d'entrée.

**Tableau 9−20. Fonctions de formatage**

| Fonction | Type en retour | Description | Exemple |
|---|---|---|---|
| to_char(timestamp, text) | text | convertit un champ date/heure (timestamp) en une chaîne | to_char(current_timestamp, 'HH12:MI:SS') |
| to_char(interval, text) | text | convertit un champ de type interval en une chaîne | to_char(interval '15h 2m 12s', 'HH24:MI:SS') |
| to_char(int, text) | text | convertit un champ de type integer en une chaîne | to_char(125, '999') |
| to_char(double precision, text) | text | convertit un champ de type real/double precision en une chaîne | to_char(125.8::real, '999D9') |
| to_char(numeric, text) | text | convertit un champ de type numeric en une chaîne | to_char(-125.8, '999D99S') |
| to_date(text, text) | date | convertit une chaîne en date | to_date('05 Dec 2000', 'DD Mon YYYY') |
| to_timestamp(text, text) | timestamp with time zone | convertit une chaîne string en un champ de type timestamp | to_timestamp('05 Dec 2000', 'DD Mon YYYY') |
| to_number(text, text) | numeric | convertit une chaîne en champ de type numeric | to_number('12,454.8-', '99G999D9S') |

Attention : to_char(interval, text) est obsolète et ne devrait plus être utilisé dans du code nouvellement écrit. Elle sera supprimée dans la prochaine version.

Dans une chaîne modèle en sortie (for `to_char`), certains modèles sont reconnus et remplacés avec les données proprement formatées à partir de la valeur à formater. Tout texte qui n'est pas un modèle est copié sans modification. De même, sur une chaîne modèle en entrée (pour tout sauf `to_char`), les modèles identifient les parties de la chaîne en entrée à récupérer et les valeurs à trouver.

Tableau 9–21 affiche les modèles disponibles pour formater les valeurs de types date et heure.

**Tableau 9–21. Modèles pour le formatage de champs de type date/heure**

| Modèle | Description |
|---|---|
| `HH` | heure du jour (01–12) |
| `HH12` | heure du jour (01–12) |
| `HH24` | heure du jour (00–23) |
| `MI` | minute (00–59) |
| `SS` | seconde (00–59) |
| `MS` | milliseconde (000–999) |
| `US` | microseconde (000000–999999) |
| `SSSS` | secondes passées depuis minuit (0–86399) |
| `AM` ou `A.M.` ou `PM` ou `P.M.` | indicateur du méridien (en majuscule) |
| `am` ou `a.m.` ou `pm` ou `p.m.` | indicateur du méridien (en minuscule) |
| `Y,YYY` | année (4 chiffres et plus) avec virgule |
| `YYYY` | year (quatre chiffres et plus) |
| `YYY` | les trois derniers chiffres de l'année |
| `YY` | les deux derniers chiffres de l'année |
| `Y` | le dernier chiffre de l'année |
| `IYYY` | année ISO (4 chiffres ou plus) |
| `IYY` | trois derniers chiffres de l'année ISO |
| `IY` | deux derniers chiffres de l'année ISO |
| `I` | dernier chiffre de l'année ISO |
| `BC` ou `B.C.` ou `AD` ou `A.D.` | indicateur de l'ère (majuscule) |
| `bc` ou `b.c.` ou `ad` ou `a.d.` | indicateur de l'ère (minuscule) |
| `MONTH` | nom complet du mois en majuscule (espaces ajoutés pour arriver à neuf caractères) |
| `Month` | nom complet du mois (espaces ajoutés pour arriver à neuf caractères) |
| `month` | nom complet du mois en minuscule (espaces ajoutés pour arriver à neuf caractères) |
| `MON` | abréviation du nom du mois en majuscule (trois caractères) |
| `Mon` | abréviation du nom du mois (trois caractères) |
| `mon` | abréviation du nom du mois en minuscule (trois caractères) |
| `MM` | numéro du mois (01–12) |
| `DAY` | nom complet du jour en majuscule (espaces ajoutés pour arriver à neuf caractères) |

| | |
|---|---|
| Day | nom complet du jour (espaces ajoutés pour arriver à neuf caractères) |
| day | nom complet du jour en minuscule (espaces ajoutés pour arriver à neuf caractères) |
| DY | abréviation du nom du jour en majuscule (3 caractères) |
| Dy | abréviation du nom du jour (3 caractères) |
| dy | abréviation du nom du jour en minuscule (3 caractères) |
| DDD | jour de l'année (001−366) |
| DD | jour du mois (01−31) |
| D | jour de la semaine (de 1 à 7, dimanche étant le 1) |
| W | numéro de semaine du mois (de 1 à 5) (la première semaine commence le premier jour du mois.) |
| WW | numéro de semaine dans l'année (de 1 à 53) (la première semaine commence le premier jour de l'année.) |
| IW | numéro de semaine ISO de l'année (le premier jeudi de la nouvelle année correspond à la semaine 1.) |
| CC | siècle (2 chiffres) |
| J | Julian Day (nombre de jours depuis le 1er janvier 4712 avant JC) |
| Q | trimestre |
| RM | mois en nombre romain (I−XII ; I étant janvier) (en majuscule) |
| rm | mois en nombre romain (i−xii; i étant janvier) (en minuscule) |
| TZ | nom du fuseau horaire (en majuscule) |
| tz | nom du fuseau horaire (en minuscule) |

Certains modificateurs peuvent être appliqués à tout modèle pour changer leur comportement. Par exemple, FMMonth est le modèle Month avec le modificateur FM. Tableau 9−22 affiche les modificateurs de modèles pour le formatage des dates/heures.

**Tableau 9−22. Modificateurs de modèles pour le formatage des dates/heures**

| Modificateur | Description | Exemple |
|---|---|---|
| préfixe FM | mode remplissage (supprime les espaces et les zéros) | FMMonth |
| suffixe TH | suffixe du nombre ordinal en majuscule | DDTH |
| suffixe th | suffixe du nombre ordinal en minuscule | DDth |
| préfixe FX | option global de format fixé (voir les notes d'utilisation) | FX Month DD Day |
| suffixe SP | mode épeler (pas encore implémenté) | DDSP |

Notes d'utilisation pour le formatage date/heure :

- FM supprime les zéro au début et les espaces en fin qui, autrement aurait été ajouté pour que la sortie du modèle soit de taille fixe.
- to_timestamp et to_date passent les espaces blancs multiples de la chaîne en entrée si l'option FX n'est pas utilisée. FX doit être spécifié et être le premier élément du modèle. Par exemple, to_timestamp('2000 JUN', 'YYYY MON') est correct mais to_timestamp('2000 JUN', 'FXYYYY MON') renvoie une erreur car to_timestamp n'attend qu'un seul espace.
- Le texte ordinaire est autorisé dans les modèles to_char et sera remis en sortie de façon littérale. Vous pouvez placez une sous−chaîne entre des guillemets doubles pour forcer son interprétation en tant que texte littéral même s'il contient des mots clés de modèles. Par exemple, dans '"Hello

`Year "YYYY"`, les caractères `YYYY` seront remplacés par l'année mais le seul `Y` du mot `Year` ne le sera pas.

- Si vous voulez avoir un guillemet double dans la sortie, vous devez le précéder d'un antislash, par exemple `'\\"YYYY Month\\"'`. (Deux antislashs sont nécessaire parce qu'un seul antislash a déjà une signification spéciale dans une chaîne.)
- La conversion `YYYY` d'une chaîne vers un champ de type `timestamp` ou `date` a une restriction dans le cas où vous utilisez une année avec plus de quatre chiffres. Vous devez utiliser des caractères non numériques ou un modèle après `YYYY`, sinon l'année sera toujours interprétée sur quatre chiffres. Par exemple, pour l'année 20000) : `to_date('200001131', 'YYYYMMDD')` sera interprété comme une année à quatre chiffres ; utilisez à la place un séparateur non décimal après l'année comme `to_date('20000-1131', 'YYYY-MMDD')` ou `to_date('20000Nov31', 'YYYYMonDD')`.
- Dans les conversions des chaînes de caractères vers `timestamp` ou `date`, le champ `CC` est ignoré s'il y a un champ `YYY`, `YYYY` ou `Y,YYY`. Si `CC` est utilisé avec `YY` ou `Y`, alors l'année est calculée comme `(CC-1)*100+YY`.
- Les valeurs en millisecondes (`MS`) et microsecondes (`US`) dans une conversion d'une chaîne vers un champ de type `timestamp` sont utilisées comme faisant partie de la fraction décimale des secondes. Par exemple, `to_timestamp('12:3', 'SS:MS')` n'est pas trois millisecondes mais 300 car la conversion le compte comme 12 + 0,3 secondes. Ceci signifie que le format `SS:MS`, les valeurs d'entrées `12:3`, `12:30` et `12:300` spécifient le même nombre de millisecondes. Pour obtenir trois millisecondes, vous devez utiliser `12:003` car la conversion compte avec 12 + 0,003 = 12,003 secondes.

  Voici un exemple encore plus complexe : `to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US')` représente 15 heures, 12 minutes et 2 secondes + 20 millisecondes + 1230 microsecondes = 2,021230 secondes.
- la numérotation du jour de la semaine de `to_char` (voir le modèle de formatage de 'D') est différente de celle de la fonction `extract`.

Tableau 9–23 affiche les modèles disponibles pour le formatage des valeurs numériques.

**Tableau 9–23. Modèles pour le formatage de valeurs numériques**

| Modèle | Description |
|---|---|
| `9` | valeur avec le nombre spécifié de chiffres |
| `0` | valeur avec des zéros de remplissage |
| `.` (point) | point décimal |
| `,` (virgule) | séparateur de groupe (milliers) |
| `PR` | valeur négative entre crochets |
| `S` | signe accroché au nombre (utilise la locale) |
| `L` | symbole monétaire (utilise la locale) |
| `D` | point décimale (utilise la locale) |
| `G` | séparateur de groupe (utilise la locale) |
| `MI` | signe moins dans la position spécifiée (si le nombre est inférieur à 0) |
| `PL` | signe plus dans la position spécifiée (si le nombre est supérieur à 0) |

| | |
|---|---|
| SG | signe plus/moins dans la position spécifiée |
| RN | numéro romain (entrée entre 1 et 3999) |
| TH ou th | suffixe du nombre ordinal |
| V | décalage d'un nombre spécifié de chiffres (voir les notes) |
| EEEE | notation scientifique (pas encore implémentée) |

Notes d'utilisation pour le formatage des nombres :

- Un signe formaté en utilisant SG, PL ou MI n'est pas ancré au nombre ; par exemple, to_char(-12, 'S9999') produit '  -12' mais to_char(-12, 'MI9999') produit '-  12'. L'implémentation Oracle n'autorise pas l'utilisation de MI devant 9, mais requiert plutôt que 9 précède MI.
- 9 résulté en une valeur avec le même nombre de chiffre. Si un chiffre n'est pas disponible, il est remplacé par un espace.
- TH ne convertit pas les valeurs inférieures à zéro et ne convertit pas les nombres à fraction.
- PL, SG, et TH sont des extensions PostgreSQL.
- V multiplie effectivement les valeurs en entrée par 10^*n*, où *n* est le nombre de chiffres suivant V. to_char ne supporte pas l'utilisation de V combiné avec un point décimal. (donc 99.9V99 n'est pas autorisé.)

Tableau 9–24 affiche quelques exemples de l'utilisation de la fonction to_char.

**Tableau 9–24. Exemples avec `to_char`**

| Expression | Résultat |
|---|---|
| to_char(current_timestamp, 'Day, DD  HH12:MI:SS') | 'Tuesday  , 06  05:39:18' |
| to_char(current_timestamp, 'FMDay, FMDD  HH12:MI:SS') | 'Tuesday, 6  05:39:18' |
| to_char(-0.1, '99.99') | '  -.10' |
| to_char(-0.1, 'FM9.99') | '-.1' |
| to_char(0.1, '0.9') | ' 0.1' |
| to_char(12, '9990999.9') | '    0012.0' |
| to_char(12, 'FM9990999.9') | '0012.' |
| to_char(485, '999') | ' 485' |
| to_char(-485, '999') | '-485' |
| to_char(485, '9 9 9') | ' 4 8 5' |
| to_char(1485, '9,999') | ' 1,485' |
| to_char(1485, '9G999') | ' 1 485' |
| to_char(148.5, '999.999') | ' 148.500' |
| to_char(148.5, 'FM999.999') | '148.5' |
| to_char(148.5, 'FM999.990') | '148.500' |
| to_char(148.5, '999D999') | ' 148,500' |
| to_char(3148.5, '9G999D999') | ' 3 148,500' |
| to_char(-485, '999S') | '485-' |

| | |
|---|---|
| to_char(-485, '999MI') | '485-' |
| to_char(485, '999MI') | '485 ' |
| to_char(485, 'FM999MI') | '485' |
| to_char(485, 'PL999') | '+485' |
| to_char(485, 'SG999') | '+485' |
| to_char(-485, 'SG999') | '-485' |
| to_char(-485, '9SG99') | '4-85' |
| to_char(-485, '999PR') | '<485>' |
| to_char(485, 'L999') | 'DM 485 |
| to_char(485, 'RN') | '        CDLXXXV' |
| to_char(485, 'FMRN') | 'CDLXXXV' |
| to_char(5.2, 'FMRN') | 'V' |
| to_char(482, '999th') | ' 482nd' |
| to_char(485, '"Good number:"999') | 'Good number: 485' |
| to_char(485.8, '"Pre:"999" Post:" .999') | 'Pre: 485 Post: .800' |
| to_char(12, '99V999') | ' 12000' |
| to_char(12.4, '99V999') | ' 12400' |
| to_char(12.45, '99V9') | ' 125' |

# 9.9. Fonctions et opérateurs pour date/heure

Tableau 9–26 affiche les fonctions disponibles pour le traitement des valeurs date et heure avec des détails apparaissant dans les sous–sections suivantes. Tableau 9–25 illustre les comportements des opérateurs d'arithmétique basique (+, *, etc.). Pour les fonctions de formatage, référez–vous à Section 9.8. Vous devriez être familier avec les informations de base sur les types de données date/heure d'après Section 8.5.

Toutes les fonctions et opérateurs décrit ci–dessous acceptant une entrée de type `time` ou `timestamp` viennent en deux variantes : une prenant `time with time zone` ou `timestamp with time zone` et une autre prenant `time without time zone` ou `timestamp without time zone`. Pour faire bref, ces variantes ne sont pas affichées séparément. De plus, les opérateurs + et * viennent en paires commutatives (par exemple, à la fois date + integer et integer + date) ; nous en montrons seulement une pour chacune des paires.

**Tableau 9–25. Opérateurs date/heure**

| Opérateur | Exemple | Résultat |
|---|---|---|
| + | date '2001-09-28' + integer '7' | date '2001-10-05' |
| + | date '2001-09-28' + interval '1 hour' | timestamp '2001-09-28 01:00' |
| + | date '2001-09-28' + time '03:00' | timestamp '2001-09-28 03:00' |
| + | interval '1 day' + interval '1 hour' | interval '1 day 01:00' |
| + | timestamp '2001-09-28 01:00' + interval '23 hours' | timestamp '2001-09-29 00:00' |

| | | |
|---|---|---|
| + | time '01:00' + interval '3 hours' | time '04:00' |
| - | - interval '23 hours' | interval '-23:00' |
| - | date '2001-10-01' - date '2001-09-28' | integer '3' |
| - | date '2001-10-01' - integer '7' | date '2001-09-24' |
| - | date '2001-09-28' - interval '1 hour' | timestamp '2001-09-27 23:00' |
| - | time '05:00' - time '03:00' | interval '02:00' |
| - | time '05:00' - interval '2 hours' | time '03:00' |
| - | timestamp '2001-09-28 23:00' - interval '23 hours' | timestamp '2001-09-28 00:00' |
| - | interval '1 day' - interval '1 hour' | interval '23:00' |
| - | timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00' | interval '1 day 15:00' |
| * | interval '1 hour' * double precision '3.5' | interval '03:30' |
| / | interval '1 hour' / double precision '1.5' | interval '00:40' |

**Tableau 9-26. Fonctions date/heure**

| Fonction | Code de retour | Description | Exemple | Résultat |
|---|---|---|---|---|
| age(timestamp, timestamp) | interval | Soustrait les arguments, produisant un résultat << symbolique >> qui utilise les années et les mois | age(timestamp '2001-04-10', timestamp '1957-06-13') | 43 years 9 mons 27 days |
| age(timestamp) | interval | Soustrait à partir de la date courante (current_date) | age(timestamp '1957-06-13') | 43 years 8 mons 3 days |
| current_date | date | Date d'aujourd'hui ; voir Section 9.9.4 | | |
| current_time | time with time zone | Heure du jour ; voir Section 9.9.4 | | |
| current_timestamp | timestamp with time zone | Date et heure du jour ; voir Section 9.9.4 | | |
| date_part(text, timestamp) | double precision | Obtenir un sous-champ (équivalent à extract) ; voir Section 9.9.1 | date_part('hour', timestamp '2001-02-16 20:38:40') | 20 |
| date_part(text, interval) | double precision | Obtenir un sous-champ (équivalent à extract) ; voir Section 9.9.1 | date_part('month', interval '2 years 3 months') | 3 |
| date_trunc(text, timestamp) | timestamp | Tronquer jusqu'à la précision spécifiée ; voir aussi Section 9.9.2 | date_trunc('hour', timestamp '2001-02-16 | 2001-02-16 20:00:00 |

| | | | | |
|---|---|---|---|---|
| | | | 20:38:40') | |
| extract(field from timestamp) | double precision | Obtenir un sous-champ ; voir Section 9.9.1 | extract(hour from timestamp '2001-02-16 20:38:40') | 20 |
| extract(field from interval) | double precision | Obtenir un sous-champ ; voir Section 9.9.1 | extract(month from interval '2 years 3 months') | 3 |
| isfinite(timestamp) | boolean | Test d'un type time stamp fini (différent de l'infini) | isfinite(timestamp '2001-02-16 21:28:30') | true |
| isfinite(interval) | boolean | Test d'un intervalle fini | isfinite(interval '4 hours') | true |
| localtime | time | Heure du jour ; voir Section 9.9.4 | | |
| localtimestamp | timestamp | Date et heure ; voir Section 9.9.4 | | |
| now() | timestamp with time zone | Date et heure courante (équivalent à current_timestamp) ; voir Section 9.9.4 | | |
| timeofday() | text | Date et heure courante ; voir Section 9.9.4 | | |

En plus de ces fonctions, l'opérateur SQL OVERLAPS est supporté :

```
( début1, fin1 ) OVERLAPS
( début2, fin2 )
( début1, longueur1 )
OVERLAPS ( début2,
longueur2
)
```

Cette expression renvoie vrai (true) lorsque les deux périodes de temps (définies par leur point final) se surchargent, et faux dans le cas contraire. Les points finaux peuvent être spécifiés comme des pairs de dates, d'heures ou de timestamps ; ou comme une date, une heure ou un timestamp suivi d'un intervalle.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Résultat :true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Résultat :false
```

## 9.9.1. EXTRACT, date_part

```
EXTRACT (champ FROM
source)
```

La fonction extract récupère des sous-champs de valeurs date/heure, tels que l'année ou l'heure. *source* est une expression de valeur de type timestamp, time ou interval. (Les expressions de type date seront converties en timestamp et peuvent aussi être utilisées.) *champ* est un identifiant ou une chaîne qui

sélectionne le champ à extraire de la valeur source. La fonction `extract` renvoie des valeurs de type `double precision`. Ce qui suit est une liste de noms de champs valides :

`century`
: Le siècle

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Résultat : 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 21
```

Le premier siècle commence le 1er janvier de l'an 1 (0001−01−01 00:00:00 AD) bien qu'ils ne le savaient pas à cette époque. Cette définition s'applique à tous les pays ayant un calendrier Grégorien. Le siècle 0 n'existe pas. Vous allez de −1 à 1. Si vous n'êtes pas d'accord, adressez votre plainte à : Le Papa, Cathédrale Saint−Pierre de Rome, Vatican.

Les versions de PostgreSQL antérieures à la 8.0 ne suivaient pas la numérotation conventionnelle des siècles mais renvoyaient uniquement le champ année divisée par 100.

`day`
: Le champ jour (du mois) : de 1 à 31

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat: 16
```

`decade`
: Le champ année divisée par 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 200
```

`dow`
: Le jour de la semaine (de 0 à 6 ; dimanche étant le 0) (uniquement pour les valeurs de type `timestamp`)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 5
```

Notez que la numérotation du jour de la semaine est différent de celui de la fonction `to_char`.

`doy`
: Le jour de l'année (de 1 à 365/366) (uniquement pour les valeurs `timestamp`)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 47
```

`epoch`
: Pour les valeurs de type `date` et `timestamp`, le nombre de secondes depuis le 1er janvier 1970 (exactement depuis le 1970−01−01 00:00:00−00 (peut être négatif) ; pour les valeurs de type `interval`, le nombre total de secondes dans l'intervalle

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-08');
Résultat :982384720

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Résultat :442800
```

Voici comment convertir une valeur epoch en une date/heure :

```
SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720 * INTERVAL '1 second';
```

`hour`

Le champ heure (0 – 23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 20
```

`microseconds`

Le champ secondes, incluant la partie décimale, multiplié par 1 000 000. Notez que ceci inclut les secondes complètes.

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Résultat :28500000
```

`millennium`

Le millénaire

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 3
```

Les années 1900 sont dans le second millénaire. Le troisième millénaire commence le 1er janvier 2001.

Les versions de PostgreSQL précédant la 8.0 ne suivaient pas les conventions de numérotation des millénaires, mais renvoyaient seulement le champ année divisé par 1000.

`milliseconds`

Le champ secondes, incluant la partie décimale, multiplié par 1000. Notez que ceci inclut les secondes complètes.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Résultat :28500
```

`minute`

Le champ minutes (0 – 59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 38
```

`month`

Pour les valeurs de type `timestamp`, le numéro du mois dans l'année (de 1 à 12) ; pour les valeurs de type `interval`, le nombre de mois, modulo 12 (0 – 11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Résultat : 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Résultat : 1
```

`quarter`

Le trimestre (1 – 4) dont le jour fait partie (uniquement pour les valeurs de type `timestamp`)

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 1
```

`second`

Le champs secondes, incluant la partie décimale (0 – 59[5])

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
```

*Résultat :* 40

```
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
```
*Résultat :*28.5

`timezone`
: Le décalage du fuseau horaire depuis UTC, mesuré en secondes. Les valeurs positives correspondent aux fuseaux horaires à l'est d'UTC, les valeurs négatives à l'ouest d'UTC.

`timezone_hour`
: Le composant heure du décalage de la zone horaire

`timezone_minute`
: Le composant minute du décalage de la zone horaire

`week`
: Le numéro de la semaine dans l'année, auquel appartient le jour. Par définition (ISO 8601), la première semaine d'une année contient le 4 janvier de cette année. (La semaine avec l' ISO−8601 commence un lundi.) Autrement dit, le premier jeudi d'une année se trouve dans la première semaine de cette année. (uniquement pour les valeurs de type `timestamp`)

  À cause de ceci, il est possible pour les dates de début janvier de faire partie de la 52è ou 53è semaine de l'année précédente. Par exemple, `2005-01-01` fait partie de la 53è semaine de 2004 et `2006-01-01` fait partie de la 52è semaine de l'année 2005.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
```
*Résultat :* 7

`year`
: Le champ année. Gardez en tête qu'il n'y a pas de `0 AD`, donc soustraire `BC` années de `AD` années devra se faire avec attention.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
```
*Résultat :*2001

La fonction `extract` a pour but principal l'exécution de calcul. Pour le formatage des valeurs date/heure en affichage, voir Section 9.8.

La fonction `date_part` est modelé sur l'équivalent traditionnel Ingres de la fonction `extract` du standard SQL :

```
date_part('champ', source)
```

Notez, ici, que le paramètre *champ* doit être une valeur de type chaîne et non pas un nom. Les noms de champ valide pour `date_part` sont les mêmes que pour `extract`.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Résultat : 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Résultat : 4
```

## 9.9.2. `date_trunc`

La fonction `date_trunc` est conceptuellement similaire à la fonction `trunc` pour les nombres.

```
date_trunc('champ',
source)
```

*source* une expression de valeur de type `timestamp` ou `interval`. (Les valeurs de type `date` et `time` sont converties automatiquement en respectivement `timestamp` ou `interval`.) *champ* indique la précision pour tronquer la valeur en entrée. La valeur de retour est de type `timestamp` ou `interval` avec tous les champs qui sont moins signifiants que l'ensemble sélectionné de zéro (ou pour la date et le mois).

Les valeurs valides pour *champ* sont :

```
microseconds
milliseconds
second
minute
hour
day
week
month
year
decade
century
millennium
```

Exemples :

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Résultat : 2001-02-16
20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Résultat : 2001-01-01
00:00:00
```

## 9.9.3. AT TIME ZONE

La construction `AT TIME ZONE` permet les conversions du type << time stamps >> vers les différents fuseaux horaires. Tableau 9–27 affiche ses variantes.

**Tableau 9–27. Variantes AT TIME ZONE**

| Expression | Type de retour | Description |
|---|---|---|
| `timestamp without time zone AT TIME ZONE` *zone* | `timestamp with time zone` | Convertit l'heure locale avec une zone horaire donnée vers l'UTC |
| `timestamp with time zone AT TIME ZONE` *zone* | `timestamp without time zone` | Convertit la zone horaire UTC vers l'heure avec la zone horaire donnée |
| `time with time zone AT TIME ZONE` *zone* | `time with time zone` | Convertit l'heure locale pour les différentes zones horaires |

Dans ces expressions, le fuseau horaire désiré *zone* peut être spécifié soit comme une chaîne texte (par exemple, `'PST'`) ou comme un intervalle (c'est–à–dire `INTERVAL '-08:00'`). Dans le cas textuel, les

noms de fuseaux sont ceux affichés dans Tableau B−4. (Il pourrait être utile de supporter les noms plus généraux affichés dans Tableau B−6 mais ceci n'est pas encore implémenté.)

Exemples (supposant que la zone horaire locale est `PST8PDT`) :

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
Résultat : 2001-02-16
19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
Résultat : 2001-02-16
18:38:40
```

Le premier exemple prend un << zone−less time stamp >> et l'interprète comme heure MST (UTC−7) pour produire un << time stamp >> UTC, qui effectue une rotation vers PST (UTC−8) pour l'affichage. Le deuxième exemple prend un type << time stamp >> spécifié en EST (UTC−5) et le convertit en l'heure locale en MST (UTC−7).

La fonction `timezone(`*`zone`*`, `*`timestamp`*`)` est équivalente pour la structure conforme au SQL *`timestamp`* `AT TIME ZONE` *`zone`*.

## 9.9.4. Date/Heure courante

Les fonctions suivantes sont disponibles pour obtenir la date courante et/ou l'heure :

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME ( precision )
CURRENT_TIMESTAMP ( precision )
LOCALTIME
LOCALTIMESTAMP
LOCALTIME ( precision )
LOCALTIMESTAMP ( precision )
```

`CURRENT_TIME` et `CURRENT_TIMESTAMP` délivrent les valeurs avec indication du fuseau horaire ; `LOCALTIME` et `LOCALTIMESTAMP` délivrent les valeurs avec indication du fuseau horaire.

`CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, et `LOCALTIMESTAMP` peuvent se voir attribuer de façon optionnelle le paramètre de précision, qui cause l'arrondissement du résultat pour le nombre de chiffres de fraction dans le champ des secondes. Sans le paramètre de précision, le résultat est donné avec la précision complète.

> **Note :** Avant PostgreSQL 7.2, les paramètres de précision n'existaient pas et le résultat était toujours donné en secondes entières.

Quelques exemples :

```
SELECT CURRENT_TIME;
Résultat :14:39:53.662522-05

SELECT CURRENT_DATE;
Résultat :2001-12-23
```

```
SELECT CURRENT_TIMESTAMP;
Résultat : 2001-12-23
14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
Résultat : 2001-12-23
14:39:53.66-05

SELECT LOCALTIMESTAMP;
Résultat : 2001-12-23
14:39:53.662522
```

La fonction `now()` est l'équivalent traditionnel PostgreSQL de `CURRENT_TIMESTAMP`.

Il existe aussi la fonction `timeofday()` qui, pour des raisons historiques, renvoie une chaîne de type `text` plutôt qu'une valeur de type `timestamp` :

```
SELECT timeofday();
Résultat : Sat Feb 17
19:07:32.000126 2001 EST
```

Il est important de savoir que `CURRENT_TIMESTAMP` et les fonctions relatives renvoient l'heure de début de la transaction courante ; leurs valeurs ne changent pas lors de la transaction. Ceci est considérée comme une fonctionnalité : le but est de permettre à une transaction seule d'avoir une notion consistante de l'heure << courante >>, donc les modifications multiples à l'intérieur de la même transaction partagent la même heure. `timeofday()` renvoie l'heure de l'horloge et change lors des transactions.

> **Note :** D'autres systèmes de bases de données pourraient mettre à jour ces valeurs plus fréquemment.

Tous les types de données date/heure acceptent aussi la valeur littérale spéciale `now` pour spécifier la date et l'heure actuelle. Du coup, les trois suivants renvoient aussi le même résultat :

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';
```

> **Astuce :** Vous ne voulez pas utiliser la troisième forme lors de la spécification de la clause `DEFAULT` pour la création d'une table. Le système convertira `now` vers une valeur de type `timestamp` dès que la constante est analysée, de façon à ce que la valeur par défaut soit nécessaire, l'heure de la création de la table serait utilisée ! Les deux premières formes ne seront pas évaluées jusqu'à ce que la valeur par défaut soit utilisée car ce sont des appels de fonctions. Donc, ils donneront le comportement désiré d'avoir la valeur par défaut au moment de l'insertion de la ligne.

# 9.10. Fonctions et opérateurs géométriques

Les types géométriques `point`, `box`, `lseg`, `line`, `path`, `polygon` et `circle` ont un large ensemble de support natif des fonctions et opérateurs natifs, affichés dans Tableau 9–28, Tableau 9–29 et Tableau 9–30.

**Tableau 9–28. Opérateurs géométriques**

| Opérateur | Description | Exemple |
|---|---|---|
| + | Translation | box '((0,0),(1,1))' + point '(2.0,0)' |
| - | Translation | box '((0,0),(1,1))' - point '(2.0,0)' |
| * | Échelle/rotation | box '((0,0),(1,1))' * point '(2.0,0)' |
| / | Échelle/rotation | box '((0,0),(2,2))' / point '(2.0,0)' |
| # | Point ou boîte d'intersection | '((1,-1),(-1,1))' # '((1,1),(-1,-1))' |
| # | Nombre de points dans le chemin ou le polygone | # '((1,0),(0,1),(-1,0))' |
| @-@ | Longueur ou circonférence | @-@ path '((0,0),(1,0))' |
| @@ | Centre | @@ circle '((0,0),10)' |
| ## | Point le plus proche entre le premier et le second opérande | point '(0,0)' ## lseg '((2,0),(0,2))' |
| <-> | Distance entre | circle '((0,0),1)' <-> circle '((5,0),1)' |
| && | Surcharge ? | box '((0,0),(1,1))' && box '((0,0),(2,2))' |
| &< | Ne s'étend pas à droite de ? | box '((0,0),(1,1))' &< box '((0,0),(2,2))' |
| &> | Ne s'étend pas à gauche de ? | box '((0,0),(3,3))' &> box '((0,0),(2,2))' |
| << | À gauche de ? | circle '((0,0),1)' << circle '((5,0),1)' |
| >> | À droite de ? | circle '((5,0),1)' >> circle '((0,0),1)' |
| <^ | En dessous de ? | circle '((0,0),1)' <^ circle '((0,5),1)' |
| >^ | Au dessus de ? | circle '((0,5),1)' >^ circle '((0,0),1)' |
| ?# | Intersection ? | lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))' |
| ?- | Horizontal ? | ?- lseg '((-1,0),(1,0))' |
| ?- | Sont-ils alignés horizontalement ? | point '(1,0)' ?- point '(0,0)' |
| ?\| | Vertical ? | ?\| lseg '((-1,0),(1,0))' |
| ?\| | Sont-ils verticalement alignés ? | point '(0,1)' ?\| point '(0,0)' |
| ?-\| | Perpendiculaire ? | lseg '((0,0),(0,1))' ?-\| lseg '((0,0),(1,0))' |
| ?\|\| | Parallèle ? | lseg '((-1,0),(1,0))' ?\|\| lseg '((-1,2),(1,2))' |

| | | |
|---|---|---|
| ~ | Contient ? | `circle '((0,0),2)' ~ point '(1,1)'` |
| @ | Contenu ou contenant ? | `point '(1,1)' @ circle '((0,0),2)'` |
| ~= | Identique à ? | `polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'` |

**Tableau 9–29. Fonctions géométriques**

| Fonction | Type de retour | Description | Exemple |
|---|---|---|---|
| `area (`*object*`)` | `double precision` | aire | `area(box '((0,0),(1,1))')` |
| `box_intersect(box, box)` | `box` | boîte d'intersection | `box_intersect(box '((0,0),(1,1))',box '((0.5,0.5),(2,2))')` |
| `center (`*object*`)` | `point` | centre | `center(box '((0,0),(1,2))')` |
| `diameter(circle)` | `double precision` | diamètre d'un cercle | `diameter(circle '((0,0),2.0)')` |
| `height(box)` | `double precision` | taille verticale d'une boîte | `height(box '((0,0),(1,1))')` |
| `isclosed(path)` | `boolean` | un chemin fermé ? | `isclosed(path '((0,0),(1,1),(2,0))')` |
| `isopen(path)` | `boolean` | un chemin ouvert ? | `isopen(path '[(0,0),(1,1),(2,0)]')` |
| `length(`*object*`)</ entry>` | `double precision` | longueur | `length(path '((-1,0),(1,0))')` |
| `npoints(path)` | `integer` | nombre de points | `npoints(path '[(0,0),(1,1),(2,0)]')` |
| `npoints(polygon)` | `integer` | nombre de points | `npoints(polygon '((1,1),(0,0))')` |
| `pclose(path)` | `path` | convertit un chemin en chemin fermé | `pclose(path '[(0,0),(1,1),(2,0)]')` |
| `popen(path)` | `path` | convertit un chemin en chemin ouvert | `popen(path '((0,0),(1,1),(2,0))')` |
| `radius(circle)` | `double precision` | radius d'un cercle | `radius(circle '((0,0),2.0)')` |
| `width(box)` | `double precision` | taille horizontale d'une boîte | `width(box '((0,0),(1,1))')` |

**Tableau 9–30. Fonctions de conversion d'un type géométrique**

| Fonction | Type de retour | Description | Exemple |
|---|---|---|---|
| `box(circle)` | `box` | cercle vers boîte | `box(circle '((0,0),2.0)')` |
| `box(point, point)` | `box` | points vers boîte | |

| | | | |
|---|---|---|---|
| | | | `box(point '(0,0)', point '(1,1)')` |
| `box(polygon)` | `box` | polygone vers boîte | `box(polygon '((0,0),(1,1),(2,0))')` |
| `circle(box)` | `circle` | boîte vers cercle | `circle(box '((0,0),(1,1))')` |
| `circle(point, double precision)` | `circle` | centre et radius vers cercle | `circle(point '(0,0)', 2.0)` |
| `lseg(box)` | `lseg` | diagonale d'une boîte vers un segment de ligne | `lseg(box '((-1,0),(1,0))')` |
| `lseg(point, point)` | `lseg` | points vers un segment de ligne | `lseg(point '(-1,0)', point '(1,0)')` |
| `path(polygon)` | `point` | polygone vers chemin | `path(polygon '((0,0),(1,1),(2,0))')` |
| `point(double precision, double precision)` | `point` | point de construction | `point(23.4, -44.5)` |
| `point(box)` | `point` | centre de la boîte | `point(box '((-1,0),(1,0))')` |
| `point(circle)` | `point` | centre d'un cercle | `point(circle '((0,0),2.0)')` |
| `point(lseg)` | `point` | centre de lseg | `point(lseg '((-1,0),(1,0))')` |
| `point(lseg, lseg)` | `point` | intersection | `point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')` |
| `point(polygon)` | `point` | centre d'un polygone | `point(polygon '((0,0),(1,1),(2,0))')` |
| `polygon(box)` | `polygon` | boîte vers polygone à quatre points | `polygon(box '((0,0),(1,1))')` |
| `polygon(circle)` | `polygon` | cercle vers polygone à 12 points | `polygon(circle '((0,0),2.0)')` |
| `polygon(`*`npts`*`, circle)` | `polygon` | cercle vers polygone *`npts`*–point | `polygon(12, circle '((0,0),2.0)')` |
| `polygon(path)` | `polygon` | chemin vers polygone | `polygon(path '((0,0),(1,1),(2,0))')` |

Il est possible d'accéder aux deux numéros composant d'un `point` comme si c'était un tableau avec les index 0 et 1. Par exemple, si `t.p` est une colonne de type `point`, alors `SELECT p[0] FROM t` récupère les coordonnées X et `UPDATE t SET p[1] = ...` modifie les coordonnées Y. De la même façon, une valeur de type `box` ou `lseg` pourrait être traitée comme un tableau de deux valeurs de type `point`.

La fonction `area` fonctionne pour les types `box`, `circle` et `path`. La fonction `area` fonctionne seulement pour le type de données `path` si les points dans le `path` ne se coupent pas. Par exemple, le `path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH` ne fonctionne pas. Néanmoins, le `path` suivant, et visuellement identique, `'((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH` fonctionnera. Si le concept d'intersection contre sans intersection du `path` est confus pour vous, dessinez les deux `path` ci–dessus côte–à–côte sur une partie d'un papier graphe.

# 9.11. Fonctions et opérateurs pour le type des adresses réseau

Tableau 9−31 affiche les opérateurs disponibles pour les types `cidr` et `inet`. Les opérateurs de tests <<, <<=, >> et >>= d'inclusion du sous−réseau. Ils considèrent seulement les parties réseau des deux adresses, ignorant toute la partie hôte et déterminent si une partie réseau est identique à un sous−réseau ou à un autre.

**Tableau 9−31. Opérateurs `cidr` et `inet`**

| Opérateur | Description | Exemple |
|---|---|---|
| < | est plus petit que | `inet '192.168.1.5' < inet '192.168.1.6'` |
| <= | est plus petit que ou égal à | `inet '192.168.1.5' <= inet '192.168.1.5'` |
| = | est égal à | `inet '192.168.1.5' = inet '192.168.1.5'` |
| >= | est plus grand ou égal à | `inet '192.168.1.5' >= inet '192.168.1.5'` |
| > | est plus grand que | `inet '192.168.1.5' > inet '192.168.1.4'` |
| <> | n'est pas égal à | `inet '192.168.1.5' <> inet '192.168.1.4'` |
| << | est contenu dans | `inet '192.168.1.5' << inet '192.168.1/24'` |
| <<= | est contenu dans ou égal à | `inet '192.168.1/24' <<= inet '192.168.1/24'` |
| >> | contient | `inet'192.168.1/24' >> inet '192.168.1.5'` |
| >>= | contient ou est égal à | `inet '192.168.1/24' >>= inet '192.168.1/24'` |

Tableau 9−32 affiche les fonctions disponibles, utilisables avec les types `cidr` et `inet`. Les fonctions `host`, `text` et `abbrev` ont principalement pour but d'offrir des formatages d'affichage alternatifs. Vous pouvez convertir une valeur de type texte vers un type `inet` en utilisant la syntaxe de conversion normale : `inet(`*`expression`*`)` ou *`colname`*`::inet`.

**Tableau 9−32. Fonctions `cidr` et `inet`**

| Fonction | Type de retour | Description | Exemple | Résultat |
|---|---|---|---|---|
| `broadcast(inet)` | `inet` | adresse de broadcast pour le réseau | `broadcast('192.168.1.5/24')` | `192.168.1.255/24` |
| `host(inet)` | `text` | extraction de l'adresse IP comme du texte | `host('192.168.1.5/24')` | `192.168.1.5` |
| `masklen(inet)` | `integer` | extraction de la longueur du masque réseau | `masklen('192.168.1.5/24')` | `24` |
| `set_masklen(inet, integer)` | `inet` | initialise la longueur du | `set_masklen('192.168.1.5/24', 16)` | `192.168.1.5/16` |

| | | | | |
|---|---|---|---|---|
| | | masque réseau pour une valeur de type `inet` | | |
| `netmask(inet)` | `inet` | construction du masque réseau | `netmask('192.168.1.5/24')` | `255.255.255.0` |
| `hostmask(inet)` | `inet` | construction du masque de l'hôte pour le réseau | `hostmask('192.168.23.20/30')` | `0.0.0.3` |
| `network(inet)` | `cidr` | extraction de la partie réseau de l'adresse | `network('192.168.1.5/24')` | `192.168.1.0/24` |
| `text(inet)` | `text` | extraction de l'adresse IP et de la longueur du masque réseau comme texte | `text(inet '192.168.1.5')` | `192.168.1.5/32` |
| `abbrev(inet)` | `text` | format d'affichage raccourci comme texte | `abbrev(cidr '10.1.0.0/16')` | `10.1/16` |
| `family(inet)` | `integer` | extrait la famille de l'adresse ; `4` pour IPv4, `6` pour IPv6 | `family('::1')` | `6` |

Tableau 9–33 affiche les fonctions disponibles, à utiliser avec le type `macaddr`. La fonction `trunc(macaddr)` renvoie une adresse MAC avec les trois derniers octets initialisés à zéro. Ceci peut être utilisé pour associer le préfixe restant avec un manufacturier. Le répertoire `contrib/mac` dans la distribution source contient quelques outils pour créer et maintenir une table d'association.

**Tableau 9–33. Fonctions `macaddr`**

| Fonction | Type de retour | Description | Exemple | Résultat |
|---|---|---|---|---|
| `trunc(macaddr)` | `macaddr` | initialiser les trois octets à zéro | `trunc(macaddr '12:34:56:78:90:ab')` | `12:34:56:00:00:00` |

Le type `macaddr` supporte aussi les opérateurs relationnels standards (>, <=, etc.) dans un ordre lexicographique.

# 9.12. Fonctions de manipulation de séquence

Cette section décrit les fonctions de PostgreSQL pour opérer sur les *objets de séquence*. Les objets de séquence (aussi appelés des générateurs de séquence ou simplement des séquences) sont des tables spéciales à ligne seule créées avec la commande `CREATE SEQUENCE`. Un objet séquence est habituellement utilisé pour générer des identifiants uniques pour les lignes d'une table. Les fonctions de séquence, listées dans Tableau 9–34, fournissent des méthodes simples et sains avec des utilisateurs multiples pour obtenir des valeurs de séquences successives à partir de l'objet séquence.

**Tableau 9–34. Fonctions séquence**

| Fonction | Type de retour | Description |
|---|---|---|
| `nextval(text)` | `bigint` | Avance la séquence et renvoie la nouvelle valeur |
| `currval(text)` | `bigint` | Valeur de retour obtenu le plus récemment avec `nextval` |
| `setval(text, bigint)` | `bigint` | Initialise la valeur courante de la séquence |
| `setval(text, bigint, boolean)` | `bigint` | Initialise la valeur courante de la séquence et le drapeau `is_called` |

Pour des raisons largement historiques, la séquence à utiliser sur l'appel de la fonction séquence est spécifiée par un argument de type texte. Pour achever quelques compatibilité avec la gestion de noms SQL ordinaires, les fonctions séquence convertissent leur arguments en minuscule sauf si la chaîne est entre des guillemets doubles. Donc,

```
nextval('foo')      opère sur la séquence
foo
nextval('FOO')      opère sur la séquence
foo
nextval('"Foo"')    opère sur la séquence
Foo
```

Le nom de la séquence peut être qualifié avec le schéma si nécessaire :

```
nextval('mon_schema.foo')     opère sur
mon_schema.foo
nextval('"mon_schema".foo')   identique à
ci-dessus
nextval('foo')              cherche dans le chemin de recherche
pour foo
```

Bien sûr, l'argument texte peut être le résultat d'une expression, pas seulement un simple littéral, qui est utile occasionellement.

Les fonctions séquence disponibles sont :

`nextval`
: Avance l'objet séquence à sa prochaine valeur et renvoie cette valeur. Ceci se fait de façon atomique : même si les sessions multiples exécutent `nextval` de façon concurrente, chacun va recevoir

proprement une valeur de séquence distincte.

`currval`

Renvoie la valeur la plus récemment obtenue par `nextval` pour cette séquence dans la session actuelle. (Une erreur est rapportée si `nextval` n'a jamais été appelé pour cette séquence dans cette session.) Notez que, parce qu'il renvoie une valeur locale à la session, il donne une réponse prévisible que les autres sessions aient exécutées ou non la fonction `nextval` depuis que la session en cours l'a fait.

`setval`

Réinitialise la valeur du compteur de l'objet séquence. La forme avec deux paramètres initialise le champ `last_value` de la séquence à la valeur spécifiée et initialise le champ `is_called` à `true`, signifiant que le prochain `nextval` avancera la séquence avant de renvoyer une valeur. Dans la forme à trois paramètres, `is_called` pourrait être initialisé soit à `true` soit à `false`. S'il est initialisé à `false`, le prochain `nextval` retournera exactement la valeur spécifiée et l'avancement de la séquence commence avec le `nextval` suivant. Par exemple,

```
SELECT setval('foo', 42);           Next nextval
renverra 43
SELECT setval('foo', 42, true);     Identique à
ci-dessus
SELECT setval('foo', 42, false);    Next nextval
renverra 42
```

Le résultat renvoyé par `setval` est juste la valeur du second argument.

**Important :** Pour éviter le blocage des transactions concurrentes qui obtiennent des nombres de la même séquence, une opération `nextval` n'est jamais annulée ; c'est−à−dire qu'une fois que la valeur a été récupérée, elle est considérée utilisée, même si la transaction qui annule le `nextval` après. Ceci signifie que les transactions annulées pourraient laisser des << trous >> inutilisés dans la séquence des valeurs assignées. Les opérations `setval` ne sont jamais annulées.

Si un objet séquence a été créé avec les paramètres par défaut, les appels à `nextval` sur celui−ci renverront les valeurs successives à partir de 1. D'autres comportements peuvent être obtenus en utilisant des paramètres spéciaux de la commande CREATE SEQUENCE ; voir la page de référence de la commande pour plus d'informations.

# 9.13. Expressions conditionnelles

Cette section décrit les expressions conditionnelles respectant le standard SQL disponibles avec PostgreSQL.

**Astuce :** Si vos besoins dépassent les possibilités des expressions conditionnelles, vous devez considérer l'écriture d'une procédure stockée dans un langage de programmation plus expressif.

## 9.13.1. CASE

L'expression SQL `CASE` est une expression conditionnelle générique, similaire aux instructions if/else des autres langages :

```
CASE WHEN condition THEN
```

```
résultat
     [WHEN ...]
     [ELSE résultat]
END
```

Les clauses CASE peuvent être utilisées partout où une expression est valide. *condition* est une expression qui renvoie un résultat boolean. Si le résultat est vrai, alors la valeur de l'expression CASE est le *résultat* qui suit la condition. Si le résultat est faux, toute clause WHEN suivante est recherchée de la même façon. Si la *condition* WHEN est vraie, alors la valeur de l'expression est le *résultat* de la clause ELSE. Si la clause ELSE est omise et qu'aucune condition ne correspond, alors le résultat est nul.

Un exemple :

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'un'
            WHEN a=2 THEN 'deux'
            ELSE 'autres'
       END
    FROM test;

 a | case
---+-------
 1 | un
 2 | deux
 3 | autres
```

Les types de données de toutes les expressions *résultat* doivent être convertibles dans un même type de sortie. Voir Section 10.5 pour plus de détails.

La << simple >> expression CASE suivante est une variante spécialisée de la forme générale ci−dessus :

```
CASE expression
    WHEN valeur THEN
    résultat
    [WHEN ...]
    [ELSE résultat]
END
```

L'*expression* est calculée et comparée à toutes les spécifications de *valeur* des clauses WHEN jusqu'à en trouver une. Si aucune ne correspond, le *résultat* dans la clause ELSE (ou une valeur NULL) est renvoyée. Ceci est similaire à l'instruction switch en C.

L'exemple ci−dessus peut être réécrit en utilisant la syntaxe CASE simple :

```
SELECT a,
       CASE a WHEN 1 THEN 'un'
              WHEN 2 THEN 'deux'
              ELSE 'autres'
```

```
       END
    FROM test;

 a | case
---+-------
 1 | un
 2 | deux
 3 | autres
```

Une expression CASE n'évalue pas les sous−expressions qui ne sont pas nécessaires pour déterminer le résultat. Par exemple, voici une façon possible d'éviter une division par zéro :

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

### 9.13.2. COALESCE

```
COALESCE(valeur [,
...])
```

La fonction COALESCE renvoie le premier de ces arguments qui n'est pas nul. Une valeur NULL est renvoyée seulement si tous les arguments sont nuls. Ceci est souvent utile pour substituer une valeur par défaut pour les valeurs NULL lorsque la donnée est récupérée pour affichage. Par exemple :

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Comme une expression CASE, COALESCE n'évaluera pas les arguments non nécessaires à la détermination du résultat ; c'est−à−dire que tous les arguments à la droite du premier argument non nul ne sont pas évalués.

### 9.13.3. NULLIF

```
NULLIF(valeur1,
valeur2)
```

La fonction NULLIF renvoie une valeur NULL si et seulement si *valeur1* et *valeur2* sont égales. Sinon, il renvoie *valeur1*. Ceci est réalisé pour disposer de l'opération inverse de l'exemple de COALESCE donné ci−dessus :

```
SELECT NULLIF(value, '(none)') ...
```

## 9.14. Fonctions et opérateurs sur les tableaux

Tableau 9−35 affiche les opérateurs disponibles pour les types array.

**Tableau 9−35. Opérateurs array**

| Opérateur | Description | Exemple | Résultat |
|---|---|---|---|
| = | égal à | ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3] | t |

| | | | |
|---|---|---|---|
| <> | différent de | ARRAY[1,2,3] <> ARRAY[1,2,4] | t |
| < | inférieur à | ARRAY[1,2,3] < ARRAY[1,2,4] | t |
| > | supérieur à | ARRAY[1,4,3] > ARRAY[1,2,4] | t |
| <= | inférieur ou égal à | ARRAY[1,2,3] <= ARRAY[1,2,3] | t |
| >= | supérieur ou égal à | ARRAY[1,4,3] >= ARRAY[1,4,3] | t |
| \|\| | concaténation de tableaux | ARRAY[1,2,3] \|\| ARRAY[4,5,6] | {1,2,3,4,5,6} |
| \|\| | concaténation de tableaux | ARRAY[1,2,3] \|\| ARRAY[[4,5,6],[7,8,9]] | {{1,2,3},{4,5,6},{7,8,9}} |
| \|\| | concaténation d'un élément avec un tableau | 3 \|\| ARRAY[4,5,6] | {3,4,5,6} |
| \|\| | concaténation d'un tableau avec un élément | ARRAY[4,5,6] \|\| 7 | {4,5,6,7} |

Voir Section 8.10 pour plus de détails sur le comportement des opérateurs.

Tableau 9–36 affiche les fonctions disponibles à l'utilisation avec des types tableaux. Voir Section 8.10 pour plus de discussion et d'exemples d'utilisation de ces fonctions.

**Tableau 9–36. Fonctions sur `array`**

| Fonction | Type de retour | Description | Exemple | Résultat |
|---|---|---|---|---|
| array_cat (anyarray, anyarray) | anyarray | concatène deux tableaux | array_cat(ARRAY[1,2,3], ARRAY[4,5]) | {1,2,3,4,5} |
| array_append (anyarray, anyelement) | anyarray | ajoute un élément à la fin d'un tableau | array_append(ARRAY[1,2], 3) | {1,2,3} |
| array_prepend (anyelement, anyarray) | anyarray | ajoute un élément au début d'un tableau | array_prepend(1, ARRAY[2,3]) | {1,2,3} |
| array_dims (anyarray) | text | renvoie une représentation textuelle des dimensions d'un tableau | array_dims(array[[1,2,3], [4,5,6]]) | [1:2][1:3] |
| array_lower (anyarray, integer) | integer | renvoie la dimension inférieure du | array_lower(array_prepend(0, ARRAY[1,2,3]), 1) | 0 |

| | | | | |
|---|---|---|---|---|
| | | tableau donné | | |
| array_upper (anyarray, integer) | integer | renvoie la dimension supérieure du tableau donné | array_upper(ARRAY[1,2,3,4], 1) | 4 |
| array_to_string (anyarray, text) | text | concatène des éléments de tableau en utilisant le délimiteur fourni | array_to_string(array[1, 2, 3], '~^~') | 1~^~2~^~3 |
| string_to_array (text, text) | text[] | divise une chaîne en tableau d'éléments utilisant le délimiteur fourni | string_to_array( 'xx~^~yy~^~zz', '~^~') | {xx,yy,zz} |

# 9.15. Fonctions d'agrégat

Les *fonctions d'agrégat* calculent une seule valeur résultant d'un ensemble de valeurs en entrée. Tableau 9–37 affiche les fonctions d'agrégat intégrées. Les considérations spéciales sur la syntaxe des fonctions d'agrégat sont expliquées dans Section 4.2.7. Consultez Section 2.7 pour un supplément d'informations introductives.

**Tableau 9–37. Fonctions d'agrégat**

| Fonction | Type d'argument | Type de retour | Description |
|---|---|---|---|
| avg(*expression*) | smallint, integer, bigint, real, double precision, numeric ou interval | numeric pour tout argument de type entier, double precision pour tout argument en virgule flottante, sinon identique au type de données de l'argument | la moyenne (au sens arithmétique) de toutes les valeurs en entrée |
| bit_and(*expression*) | smallint, integer, bigint ou bit | identique au type de données de l'argument | le AND bit à bit de toutes les valeurs non NULL en entrée ou NULL s'il n'y en a pas |
| bit_or(*expression*) | smallint, integer, bigint ou bit | identique au type de données de l'argument | le OR bit à bit de toutes les valeurs non NULL en entrée ou NULL s'il n'y en a pas |
| bool_and(*expression*) | bool | bool | |

| | | | |
|---|---|---|---|
| | | | true si toutes les valeurs en entrée sont true, sinon false |
| bool_or(*expression*) | bool | bool | true si au moins une valeur en entrée est true, sinon false |
| count(*) | | bigint | nombre de valeurs en entrée |
| count(*expression*) | tout type | bigint | nombre de valeurs en entrée pour lesquelles l'*expression* n'est pas NULL |
| every(*expression*) | bool | bool | équivalent à bool_and |
| max(*expression*) | tout type numeric, string ou date/time | identique au type en argument | valeur maximale de l'*expression* pour toutes les valeurs en entrée |
| min(*expression*) | tout type numeric, string ou date/time | identique au type en argument | valeur minimale de l'*expression* pour toutes les valeurs en entrée |
| stddev(*expression*) | smallint, integer, bigint, real, double precision ou numeric | double precision pour les arguments en virgule flottante, sinon numeric | deviation standard des valeurs en entrée |
| sum(*expression*) | smallint, integer, bigint, real, double precision, numeric ou interval | bigint pour les arguments de type smallint ou integer, numeric pour les arguments de type bigint, double precision pour les arguments en virgule flottante, sinon identique au type de données de l'argument | somme de l'*expression* pour toutes les valeurs en entrée |
| variance(*expression*) | smallint, integer, bigint, real, double precision ou numeric | double precision pour les arguments en virgule flottante, sinon numeric | simple variance des valeurs en entrée (carré de la déviation) |

Il devrait être noté qu'en dehors de count, ces fonctions renvoient une valeur NULL si aucune ligne n'est sélectionnée. En particulier, une somme (sum) sur aucune ligne renvoie NULL et non pas zéro. La fonction coalesce pourrait être utilisée pour substituer des zéros aux valeurs NULL quand cela est nécessaire.

> **Note :** Les agrégats booléens bool_and et bool_or correspondent aux agrégats standards du SQL every et any ou some. Comme pour any et some, il semble qu'il y a une ambiguïté dans la syntaxe standard :

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

Ici, ANY peut être considéré à la fois comme une sous−requête ou comme un agrégat si l'expression de sélection ne renvoie qu'une ligne. Du coup, le nom standard ne peur être donné à ces agrégats.

**Note :** Les utilisateurs habitués à travailler avec d'autres systèmes de gestion de bases de données SQL pourraient être surpris par les caractéristiques de performances de certains fonctions d'agrégat avec PostgreSQL lorsque l'agrégat est appliqué à la table entière (autrement dit, pas de clause WHERE). En particulier, une requête identique à

```
SELECT min(col) FROM matable;
```

sera exécuté par PostgreSQL en utilisant un parcours séquentiel de la table entière. D'autres systèmes de bases de données pourraient optimiser de telles requêtes en utilisant un index sur la colonne si celui−ci est disponible. De même, les fonctions d'agrégat max() et count() requièrent toujours un parcours séquentiel si elles s'appliquent à la table entière avec PostgreSQL.

PostgreSQL ne peut pas implémenter facilement cette optimisation parce qu'elle autorise aussi les requêtes d'agrégat définies par l'utilisateur. Comme min(), max() et count() sont définies en utilisant l'API générique des fonctions d'agrégat, rien n'est prévu pour les cas spéciaux lors de l'exécution de ces fonctions sous certaines circonstances.

Heureusement, il existe un contournement simple de min() et max(). la requête montrée ci−dessous est équivalent à la requête ci−dessus, si ce n'est qu'elle prend avantage de l'index B−tree s'il existe sur cette colonne.

```
SELECT col FROM matable ORDER BY col ASC LIMIT 1;
```

Une requête similaire (obtenue en substituant DESC avec ASC dans la requête ci−dessus) peut être utilisé à la place de max()).

Malheureusement, il n'existe pas de requête triviale similaire pouvant être utilisée pour améliorer les performances de count() si cela s'applique à la table entière.

# 9.16. Expressions de sous−expressions

Cette section décrit les expressions de sous−requêtes compatibles SQL et disponibles avec PostgreSQL. Toutes les formes d'expressions documentées dans cette section renvoient des résultats booléens (true/false).

## 9.16.1. EXISTS

```
EXISTS ( sous-requête )
```

L'argument d'EXISTS est une instruction SELECT arbitraire ou une *sous−requête*. La sous−requête est évaluée pour déterminer si elle renvoie des lignes. Si elle en renvoie au moins une, le résultat d'EXISTS est vrai (<< true >>) ; si elle n'en renvoie aucune, le résultat d'EXISTS est faux (<< false >>).

La sous−requête peut faire référence à des variables de la requête englobante qui agiront comme des constantes lors d'une évaluation de la sous−requête.

La sous−requête sera exécutée suffisamment pour déterminer si une ligne est renvoyée, donc pas obligatoirement complètement. Il est déconseillé d'écrire une sous−requête qui a des effets de bord (tel que l'appel de fonctions de séquence) ; que l'effet de bord se fasse ou non serait difficile à prédire.

Comme le résultat dépend seulement du fait que des lignes sont renvoyées et, donc, que le contenu des lignes importe peu, la liste de sortie de la sous−requête est normalement inintéressant. Une convention de codage commune est l'écriture de tous les tests `EXISTS` dans la forme `EXISTS(SELECT 1 WHERE ...)`. Néanmoins, il y a des exceptions à cette règle, exceptions comme des sous−requêtes utilisant `INTERSECT`.

Ce simple exemple ressemble à une jointure interne sur `col2` mais il produit au plus une ligne en sortie pour chaque ligne de `tab1` même s'il y a plusieurs correspondances parmi les lignes de `tab2` :

```
SELECT col1 FROM tab1
    WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

## 9.16.2. `IN`

```
expression IN
(sous-requête)
```

Le côté droit est une sous−expression entre parenthèses qui retournera qu'une seule ligne. L'expression gauche est évaluée et comparée à chaque ligne du résultat de la sous−requête. Le résultat de `IN` est vrai (<< true >>) si une ligne équivalente de la sous−requête est trouvée. Le résultat est faux (<< false >>) si aucune ligne correspondante n'est trouvée (ceci incluant le cas spécial où la sous−requête ne renvoie aucune ligne).

Notez que si l'expression gauche est NULL ou s'il n'existe pas de correspondance avec les valeurs du côté droit et qu'au moins une ligne du côté droit est NULL, le résultat de la construction `IN` sera nul, et non pas faux. Ceci est en accord avec les règles normales du SQL pour les combinaisons booléennes des valeurs NULL.

Comme avec `EXISTS`, il n'est pas conseillé d'assumer que la sous−requête sera évaluée complètement.

```
constructeur_ligne IN (sous-requête)
```

Le côté gauche de la forme `IN` est un constructeur de ligne comme décrit dans Section 4.2.11. Le côté droit de la forme `IN` est une sous−requête entre parenthèses qui doit renvoyer exactement autant de colonnes qu'il n'y a d'expressions dans le côté gauche. Les expressions côté gauche sont évaluées et comparées ligne par ligne au résultat de la sous−requête. Le résultat de `IN` est vrai (<< true >>) si une ligne équivalente de la sous−requête est trouvée. Le résultat est faux (<< false >>) si aucune ligne correspondante n'est trouvée (ceci incluant le cas spécial où la sous−requête ne renvoie aucune ligne).

Comme d'habitude, les valeurs NULL dans les lignes sont combinées suivant les règles habituelles des expressions booléennes SQL. Deux lignes sont considérées égales si leurs membres correspondants sont non nuls et égaux ; les lignes diffèrent si le contenu de leurs membres sont non nuls et différents ; sinon le résultat de la comparaison de la ligne est inconnu, donc nul. Si tous les lignes résultantes sont soit différentes soit NULL, avec au moins une NULL, alors le résultat de `IN` est nul.

### 9.16.3. NOT IN

```
constructeur_ligne NOT IN (sous-requête)
```

Le côté gauche de cette forme de NOT IN est un constructeur de lignes, comme décrit dans Section 4.2.11. Le côté droit est une sous−requête entre parenthèses qui doit renvoyer une colonne exactement. L'expression gauche est évaluée et comparée à chaque ligne du résultat de la sous−requête. Le résultat de NOT IN est vrai (<< true >>) si seules des lignes différentes de la sous−requête sont trouvées (ceci incluant le cas spécial où la sous−requête ne renvoie aucune ligne). Le résultat est faux (<< false >>) si une ligne égale est trouvée.

Notez que si l'expression gauche renvoie un résultat nul ou s'il n'existe pas de valeurs du côté droit égales et qu'au moins une ligne du côté droit renvoie nul, le résultat de la construction NOT IN sera nul, et non pas vrai. Ceci est en accord avec les règles standards du SQL pour les combinaisons booléennes de valeurs NULL.

Comme avec EXISTS, il n'est pas conseillé d'assumer que la sous−requête sera évaluée complètement.

```
constructeur_ligne operator ANY
(sous-requête)
constructeur_ligne operator SOME
(sous-requête)
```

Le côté droit de la forme ANY est un constructeur de lignes, comme décrit dans Section 4.2.11. Le côté droit de cette forme de NOT IN est une sous−requête entre parenthèses qui doit renvoyer exactement autant de colonnes qu'il n'y a d'expressions dans la ligne gauche. Les expressions du côté gauche sont évaluées et comparées ligne par ligne à chaque ligne du résultat de la sous−requête. Le résultat de NOT IN est vrai (<< true >>) si seules des lignes différentes de la sous−requête sont trouvées (ceci inclut le cas spécial où la sous−requête ne renvoie aucune ligne). Le résultat est faux (<< false >>) si aucune ligne égale n'est trouvée.

Comme d'habitude, les valeurs NULL de la ligne sont combinées suivant les règles standards du SQL sur les expressions booléennes. Deux lignes sont considérées égales si tous leurs membres correspondant sont non nuls et égaux ; les lignes sont différentes si un des membres correspondants est non nul et différent ; sinon le résultat de cette comparaison de ligne est inconnu (nul). Si tous les résultats ligne sont soit différents soit nuls, avec au moins un nul, alors le résultat de NOT IN est nul.

---

### 9.16.4. ANY/SOME

```
expression opérateur ANY
(sous-requête)
expression opérateur SOME
(sous-requête)
```

Le côté droit est une sous−requête entre parenthèse qui ne doit retourner qu'une seule colonne. L'expression du côté gauche est évaluée et comparée à chaque ligne de la sous−requête en utilisant l'*opérateur* indiqué et rendant un résultat booléen. Le résultat de ANY est vrai (<< true >>) si un résultat vrai est obtenu. Le résultat est faux (<< false >>) si aucun résultat vrai n'est trouvée (ceci incluant le cas spécial où la requête ne renvoie aucune ligne).

SOME est un synonyme de ANY. IN est équivalent à = ANY.

Notez que sans succès et avec au moins une ligne NULL du côté droit pour le résultat de l'opérateur, le résultat de la construction ANY sera nul et non pas faux. Ceci est en accord avec les règles standards SQL pour

les combinaisons booléenne de valeurs NULL.

Comme avec EXISTS, il n'est pas conseillé d'assumer que la sous−requête sera évaluée complètement.

```
(expression [,
expression ...]) opérateur
ANY (sous-requête)
(expression [,
expression ...]) opérateur
SOME (sous-requête)
```

Le côté droit de cette forme ANY est une sous−requête entre parenthèses, qui doit renvoyer exactement autant de colonnes qu'il n'y a d'expressions dans la liste gauche. Les expressions du côté gauche sont évaluées et comparées ligne par ligne à chaque ligne du résultat de la sous−requête en utilisant l'*opérateur* donné. Actuellement, seuls les opérateurs = et <> sont permis dans les constructions de ligne ANY. Le résultat de ANY est vrai (<< true >>) si une ligne est trouvée. Le résultat est faux (<< false >>) si aucune ligne n'est trouvée (ceci incluant le cas spécial où la sous−requête ne renverrait aucune ligne).

Comme d'habitude, les valeurs NULL dans les lignes sont combinées avec les règles standards des expressions booléennes en SQL. Deux lignes sont considérées égales si tous les membres correspondants sont non nuls et égaux ; les lignes sont différentes si un des membres est non nul et différent ; sinon le résultat de cette comparaison de lignes est inconnu (donc nul). S'il y a au moins une ligne NULL, alors le résultat de ANY ne peut pas être faux (false) ; il sera vrai (true) ou nul.

## 9.16.5. ALL

```
expression opérateur ALL
(sous-requête)
```

Le côté droit est une sous−requête entre parenthèses qui ne doit renvoyer qu'une seule colonne. L'expression gauche est évaluée et comparée à chaque ligne du résultat de la sous−requête en utilisant l'*opérateur* qui doit renvoyer un résultat booléen. Le résultat de ALL est vrai (<< true >>) si toutes les lignes renvoient true (ceci incluant le cas spécial où la sous−requête ne renvoie aucune ligne). Le résultat est faux (<< false >>) si un résultat faux est découvert.

NOT IN est équivalent à <> ALL.

Notez que s'il n'y a aucun échec mais qu'au moins une ligne du côté droit renvoie une valeur NULL sur le résultat de l'opérateur, le résultat de la construction ALL sera nul et non pas vrai. Ceci est en accord avec les règles standards du SQL pour les combinaisons booléennes de valeurs NULL.

Comme avec EXISTS, il est déconseillé d'assumer que la sous−requête sera évaluée complètement.

```
constructeur_ligne opérateur ALL (sous-requête)
```

Le côté gauche de cette forme de ALL est un constructeur de lignes comme décrit dans Section 4.2.11. Le côté droit de cette forme ALL est une sous−requête entre parenthèses qui doit renvoyer exactement le même nombre de colonnes qu'il y a d'expressions dans la ligne gauche. Les expressions du côté gauche sont évaluées et comparées ligne par ligne à chaque ligne du résultat de la sous−requête en utilisant l'*opérateur* donné. Actuellement, seuls les opérateurs = et <> sont autorisées dans les requêtes de ligne ALL. Le résultat de ALL est << true >> si toutes les lignes de la sous−requête sont respectivement égales ou différentes (ceci incluant le cas spécial où la sous−requête ne renvoie aucune ligne). Le résultat est faux (<< false >>) si une ligne se

trouve être respectivement différente ou égale.

Comme d'habitude, les valeurs NULL des lignes sont combinées avec les règles standards des expressions booléennes en SQL. Deux lignes sont considérées égales si tous les membres correspondants sont non nuls et égaux ; les lignes sont différentes si un membre est non nul ou différent ; sinon le résultat de la comparaison de cette ligne est inconnue (nul). S'il y a au moins une ligne résultat NULL, alors le résultat de `ALL` ne peut pas être vrai (true) ; il sera faux (false) ou nul.

### 9.16.6. Comparaison de lignes complètes

```
constructeur_ligne opérateur (sous-requête)
```

Le côté gauche est un constructeur de lignes, comme décrit dans Section 4.2.11. Le côté droit est une sous−requête entre parenthèses qui doit renvoyer exactement autant de colonnes qu'il n'y a d'expressions su le côté gauche. De plus, la sous−requête ne peut pas renvoyer plus d'une ligne. (S'il ne renvoie aucune ligne, le résultat est pris comme s'il était nul.) Le côté gauche est évalué et comparé avec la ligne du résultat de la sous−requête. Actuellement, seuls les opérateurs = et <> sont autorisés dans les comparaisons de lignes. Le résultat est vrai (<< true >>) si les deux lignes sont respectivement égales ou différentes.

Comme d'habitude, les valeurs NULL des lignes sont combinées avec les règles standards des expressions booléennes SQL. Deux lignes sont considérées égales si tous les membres correspondants sont non nuls et égaux ; les lignes sont différentes si un membre correspondant est non nul et différent ; sinon le résultat de la comparaison de la ligne est inconnu (nul).

## 9.17. Comparaisons de lignes et de tableaux

Cette section décrit plusieurs constructions spécialisées pour les comparaisons multiples entre des groupes de valeurs. Ces formes sont syntaxiquement en relation avec les formes de sous−requêtes de la section précédente mais n'impliquent pas de sous−requêtes. Ces formes impliquant des sous−expressions de tableaux sont des extensions de PostgreSQL ; le reste est compatible avec SQL. Toutes les formes d'expression documentées dans cette section renvoient des résultats booléens (true/false).

### 9.17.1. `IN`

```
expression IN
(valeur[, ...])
```

Le côté droit est une liste entre parenthèses d'expressions scalaires. Le résultat est vrai (<< true >>) si le côté gauche de l'expression est égal à une des expressions du côté droit. C'est une notation raccourci de

```
expression = value1
OR
expression = value2
OR
...
```

Notez que si l'expression du côté gauche renvoie nul ou s'il n'y a pas de valeurs du côté droit égales et qu'au moins une des expressions du côté droit renvoie la valeur NULL, le résultat de la construction `IN` sera nul et non pas faux. Ceci est en accord avec les règles standards SQL pour les combinaisons booléennes de valeurs

NULL.

## 9.17.2. NOT IN

```
expression NOT IN
(valeur[, ...])
```

Le côté droit est une liste entre parenthèses d'expressions scalaires. Le résultat est vrai (<< true >>) si l'expression du côté gauche est différent de toutes les expressions du côté droit. Ceci est une notation raccourci pour

```
expression <> value1
AND
expression <> value2
AND
...
```

Notez que si l'expression du côté gauche renvoie une valeur NULL ou s'il existe des valeurs différentes du côté droit et qu'au moins une expression du côté droit renvoie la valeur NULL, le résultat de la construction `NOT IN` sera nul et non pas vrai. Ceci est en accord avec les règles standards du SQL pour les combinaisons booléennes des valeurs NULL.

> **Astuce :** `x NOT IN y` est équivalent à `NOT (x IN y)` dans tout les cas. Néanmoins, les valeurs NULL ont plus de chances de survenir pour le novice avec l'utilisation de `NOT IN` qu'en utilisant `IN`. Il est préférable d'exprimer votre condition de façon positive si possible.

## 9.17.3. ANY/SOME (array)

```
expression opérateur ANY
(expression tableau)
expression opérateur SOME
(expression tableau)
```

Le côté droit est une expression entre parenthèses qui doit renvoyer une valeur de type array. L'expression du côté gauche est évaluée et comparée à chaque élément du tableau en utilisant l'*opérateur* donné et qui doit renvoyer un résultat booléen. Le résultat de `ANY` est vrai (<< true >>) si un résultat vrai est obtenu. Le résultat est faux (<< false >>) si aucun résultat vrai n'est trouvé (ceci incluant le cas spécial où le tableau a zéro élément).

`SOME` est un synonyme pour `ANY`.

## 9.17.4. ALL (array)

```
expression opérateur ALL
(expression tableau)
```

Le côté droit est une expression entre parenthèses qui doit renvoyer une valeur de type tableau. L'expression du côté gauche est évaluée et comparée à chaque élément du tableau en utilisant l'*opérateur* donné qui doit renvoyer un résultat booléen. Le résultat de `ALL` est vrai (<< true >>) si toutes les comparaisons renvoient vrai (ceci incl ut le cas spécial où le tableau a zéro élément). Le résultat est faux (<< false >> si un résultat faux est trouvé.

### 9.17.5. Comparaison sur des lignes complètes

```
constructeur_lignes IS DISTINCT FROM
constructeur_lignes
```

Chaque côté est un constructeur de lignes comme décrit dans Section 4.2.11. Les deux valeurs de lignes doivent avoir le même nombre de lignes. Chaque côté est évaluée et ils sont comparés ligne par ligne. Actuellement, seuls les opérateurs = et <> sont autorisés dans les comparaisons de ligne. Le résultat est vrai (<< true >>) si les deux lignes sont respectivement égales ou différentes.

Comme d'habitude, les valeurs NULL des lignes sont combinées avec les règles standards des expressions booléennes SQL. Les deux lignes sont considérées égales si leur membres correspondants sont non nul et égaux ; les lignes sont différentes si un des membres correspondants est non nul et différent ; sinon le résultat de la comparaison de ligne est inconnu (nul).

```
constructeur_lignes IS DISTINCT FROM constructeur_lignes
```

Cette construction est similaire à une comparaison de lignes <> mais cela ne ramène pas de NULL pour des entrées NULL. À la place, une valeur NULL est considérée différente (distincte de) toute valeur non NULL et deux valeurs NULL sont considérées égales (non distinctes). Du coup, le résultat sera toujours soit true soit false, jamais NULL.

```
constructeur_lignes IS NULL
    constructeur_lignes IS NOT NULL
```

Ces constructions testent si la valeur d'une ligne est NULL ou non. Une valeur de ligne est considérée non NULL si au moins un champ n'est pas NULL.

## 9.18. Fonctions renvoyant des ensembles

Cette section décrit des fonctions qui peuvent renvoyer plus d'une ligne. Actuellement, les seules fonctions dans cette classe sont les séries générant des fonctions, comme détaillé dans le Tableau 9−38.

**Tableau 9−38. Séries générant des fonctions**

| Fonction | Argument Type | Type de retour | Description |
|---|---|---|---|
| `generate_series (début, fin)` | `int` ou `bigint` | `setof int` ou `setof bigint` (même type qu'en argument) | Génère une série de valeurs, commençant à `début` pour finir à `fin` avec un incrément de un. |
| `generate_series (début, fin, étape)` | `int` ou `bigint` | `setof int` ou `setof bigint` (même type qu'en argument) | Génère une série de valeurs, commençant à `début` pour finir à `fin` avec un incrément de `étape`. |

Quand `étape` est positif, aucune ligne n'est renvoyée si `début` est plus grand que `fin`. Au contraire, quand `étape` est négatif, aucune ligne n'est renvoyée si `début` est plus petit que `fin`. De même, aucune ligne n'est renvoyé pour les entrées `NULL`. Si `étape` vaut zéro, c'est considéré comme une erreur. Quelques exemples

suivent :

```
select * from generate_series(2,4);
generate_series
-----------------
2
3
4
(3 rows)

select * from generate_series(5,1,-2);
generate_series
-----------------
5
3
1
(3 rows)

select * from generate_series(4,3);
generate_series
-----------------
(0 rows)

select current_date + s.a as dates from generate_series(0,14,7) as s(a);
dates
------------
2004-02-05
2004-02-12
2004-02-19
(3 rows)
```

# 9.19. Fonctions d'information sur le système

Tableau 9–39 affiche plusieurs fonctions qui extraient des informations de la session et du système.

**Tableau 9–39. Fonctions d'information sur la session**

| Nom | Type de retour | Description |
|---|---|---|
| `current_database()` | `nom` | nom de la base de données en cours |
| `current_schema()` | `nom` | nom du schéma en cours |
| `current_schemas(boolean)` | `nom[]` | nom des schémas dans le chemin de recherche des schémas, incluant en option les schémas implicites |
| `current_user` | `nom` | nom d'utilisateur du contexte d'exécution en cours |
| `inet_client_addr()` | `inet` | adresse de la connexion distante |
| `inet_client_port()` | `int4` | port de la connexion distante |
| `inet_server_addr()` | `inet` | adresse de la connexion locale |
| `inet_server_port()` | `int4` | port de la connexion locale |
| `session_user` | `name` | nom de l'utilisateur de la session |
| `user` | `name` | équivalent à `current_user` |
| `version()` | `text` | informations sur la version de PostgreSQL |

`session_user` est habituellement l'utilisateur utilisé pour la connexion à la base de données ; mais les superutilisateurs peuvent modifier ce paramétrage avec SET SESSION AUTHORIZATION. `current_user` est l'identifiant de l'utilisateur, applicable pour les vérifications de droits. Normalement, il est identique à l'utilisateur de la session mais il change pendant l'exécution des fonctions avec l'attribut `SECURITY DEFINER`. Dans le parlé Unix, l'utilisateur de la session est le << real user >> (NdT : l'utilisateur réel) et l'utilisateur en cours est l'<< effective user >> (NdT : l'utilisateur effectif) .

> **Note :** `current_user`, `session_user` et `user` ont un statut syntaxique spécial dans SQL : ils doivent être appelés sans parenthèses à la fin.

`current_schema` renvoie le nom du premier schéma sur le chemin de recherche (ou une valeur NULL si ce dernier est vide). C'est le schéma qui sera utilisé pour toute table ou autre objet nommé créé sans spécifier de schéma cible. `current_schemas(boolean)` renvoie un tableau de noms avec tous les schémas du chemin de recherche. L'option booléenne détermine si les schémas système inclus implicitement comme `pg_catalog` sont inclus dans le chemin de recherche renvoyé.

> **Note :** Le chemin de recherche est modifiable à l'exécution. La commande est :
>
> ```
> SET search_path TO schema [, schema, ...]
> ```

`inet_client_addr` renvoie l'adresse IP du client en cours et `inet_client_port` renvoie le numéro du port. `inet_server_addr` renvoie l'adresse IP sur laquelle le serveur a accepté la connexion en cours et `inet_server_port` renvoie le numéro du port. Toutes ces fonctions renvoient NULL si la connexion en cours s'est établie via un socket de domaine Unix.

`version()` renvoie une chaîne décrivant la version du serveur PostgreSQL.

Tableau 9–40 liste les fonctions qui permettent aux utilisateurs de demander les droits d'accès. Voir Section 5.7 pour plus d'informations sur les droits.

**Tableau 9–40. Fonctions de demandes de droits d'accès**

| Nom | Type de retour | Description |
|---|---|---|
| `has_table_privilege (utilisateur, table, droit)` | `boolean` | l'utilisateur a–t'il des droits sur la table |
| `has_table_privilege (table, droit)` | `boolean` | l'utilisateur courant a–t'il des droits sur la table |
| `has_database_privilege (utilisateur, base, droit)` | `boolean` | l'utilisateur a–t'il des droits sur la base de données |
| `has_database_privilege (base, droit)` | `boolean` | l'utilisateur courant a–t'il des droits sur la base de données |
| `has_function_privilege (utilisateur, fonction, droit)` | `boolean` | l'utilisateur a–t'il des droits sur la fonction |

| | | |
|---|---|---|
| has_function_privilege (fonction, droit) | boolean | l'utilisateur courant a–t'il des droits sur la fonction |
| has_language_privilege (utilisateur, langage, droit) | boolean | l'utilisateur a–t'il des droits sur le langage |
| has_language_privilege (langage, droit) | boolean | l'utilisateur en cours a–t'il des droits sur le langage |
| has_schema_privilege (utilisateur, schéma, droit) | boolean | l'utilisateur a–t'il des droits sur le schéma |
| has_schema_privilege (schéma, droit) | boolean | l'utilisateur en cours a–t'il des droits sur le langage |
| has_tablespace_privilege( utilisateur, espacelogique, droit) | boolean | l'utilisateur a–t'il des droits sur l'espace logique |
| has_tablespace_privilege (espacelogique, droit) | boolean | l'utilisateur en cours a–t'il des droits sur l'espace logique |

has_table_privilege vérifie si l'utilisateur peut accéder à une table d'une façon particulière. L'utilisateur peut être spécifié par son nom ou par son ID (pg_user.usesysid). Si l'argument est omis, current_user est supposé. La table peut être spécifiée par nom ou par OID. (Du coup, il n'y a que six variantes de has_table_privilege pouvant être distingué par leur nombre et le types de leurs arguments.) En spécifiant un nom, il est possible de le qualifier par celui du schéma si nécessaire. Le type de droit d'accès désiré est spécifié par une chaîne de texte qui doit être évalué par une des valeurs SELECT, INSERT, UPDATE, DELETE, RULE, REFERENCES ou TRIGGER. (Néanmoins, la casse de la chaîne n'est pas significative.) Voici un exemple :

```
SELECT has_table_privilege('mon_schema.ma_table', 'select');
```

has_database_privilege vérifie si l'utilisateur peut accéder à une base de données d'une façon particulière. Les possibilités pour ses arguments sont analogues à has_table_privilege. Le type de droit d'accès désiré doit être évalué à CREATE, TEMPORARY ou TEMP (ce qui est équivalent à TEMPORARY).

has_function_privilege vérifie si un utilisateur peut accéder à une fonction d'une façon particulière. Les possibilités pour ses arguments sont analogues à has_table_privilege. En spécifiant une fonction par une chaîne texte plutôt que par son OID, l'entrée autorisée est identique au type de données regprocedure (voir Section 8.12). Le type de droit d'accès désiré doit s'évaluer à EXECUTE. Voici un exemple :

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

has_language_privilege vérifie si un utilisateur peut accéder à un langage de procédures d'une façon particulière. Les possibilités pour ses arguments sont analogues à has_table_privilege. Le type de droit d'accès désiré doit s'évaluer à USAGE.

has_schema_privilege vérifie si un utilisateur peut accéder à un schéma d'une façon particulière. Les possibilités pour ses arguments sont analogues à has_table_privilege. Le type de droit d'accès désiré

doit s'évaluer à CREATE ou USAGE.

has_tablespace_privilege vérifie si un utilisateur peut accéder à un espace logique d'une façon particulière. Les possibilités pour ses arguments sont analogues à has_table_privilege. Le type de droit d'accès désiré doit s'évaluer à CREATE.

Pour tester si un utilisateur détient une option grant sur le droit, ajoutez WITH GRANT OPTION au mot clé du droit ; par exemple 'UPDATE WITH GRANT OPTION'.

Tableau 9−41 affiche les fonctions qui déterminent si un certain objet est *visible* dans le chemin de recherche en cours. Une table est dite visible si son schéma contenant est dans le chemin de recherche et qu'aucune table du même nom apparaît avant dans le chemin de recherche. Ceci est équivalent à au fait que la table peut être référencée par nom sans qualification explicite de schéma. Par exemple, pour lister les noms de toutes les tables visibles :

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

**Tableau 9−41. Fonctions de requêtes sur la visibilité du schéma**

| Nom | Type de retour | Description |
|---|---|---|
| pg_table_is_visible(table_oid) | boolean | la table est−elle visible dans le chemin de recherche |
| pg_type_is_visible(type_oid) | boolean | le type (ou domaine) est−il visible dans le chemin de recherche |
| pg_function_is_visible (function_oid) | boolean | la fonction est−elle visible dans le chemin de recherche |
| pg_operator_is_visible (operator_oid) | boolean | l'opérateur est−il visible dans le chemin de recherche |
| pg_opclass_is_visible (opclass_oid) | boolean | la classe d'opérateur est−elle visible dans le chemin de recherche |
| pg_conversion_is_visible (conversion_oid) | boolean | la conversion est−elle visible dans le chemin de recherche |

pg_table_is_visible réalise la vérification pour les tables (ou vues, ou tout autre type d'entrée dans pg_class). pg_type_is_visible, pg_function_is_visible, pg_operator_is_visible, pg_opclass_is_visible et pg_conversion_is_visible réalisent le même type de vérification de visibilité pour les types (et domaines), les fonctions, les opérators, les classes d'opérateur et les conversions, respectivement. Pour les fonctions et opérateurs, un objet dans le chemin de recherche est visible s'il n'y a aucun objet du même nom *et de même type de données pour les arguments* précédemment dans le chemin. Pour les classes d'opérateur, les méthodes d'accès par le nom et par l'index associé sont considérées.

Toutes ces fonctions nécessitent que les OID des objets identifient l'objet à vérifier. Si vous voulez tester un objet par son nom, il est préférable d'utiliser les types d'alias d'OID (regclass, regtype, regprocedure ou regoperator), par exemple

```
SELECT pg_type_is_visible('mon_schema.widget'::regtype);
```

Notez qu'il n'y aurait aucun sens à tester un nom non qualifié de cette façon — si le nom peut être reconnu, il doit être visible.

Tableau 9–42 liste les fonctions qui extraient des informations à partir des catalogues système.

**Tableau 9–42. Fonctions d'information sur le catalogue système**

| Nom | Type de retour | Description |
|---|---|---|
| `format_type (type_oid, typemod)` | `text` | obtient le nom SQL d'un type de données |
| `pg_get_viewdef(view_name)` | `text` | récupère la commande `CREATE VIEW` de la vue (*obsolète*) |
| `pg_get_viewdef (view_name, pretty_bool)` | `text` | récupère la commande `CREATE VIEW` pour la vue (*obsolète*) |
| `pg_get_viewdef(view_oid)` | `text` | récupère la commande `CREATE VIEW` pour la vue |
| `pg_get_viewdef (view_oid, pretty_bool)` | `text` | récupère la commande `CREATE VIEW` pour la vue |
| `pg_get_ruledef(rule_oid)` | `text` | grécupère la commande `CREATE RULE` pour la règle |
| `pg_get_ruledef (rule_oid, pretty_bool)` | `text` | récupère la commande `CREATE RULE` pour la règle |
| `pg_get_indexdef (index_oid)` | `text` | récupère la commande `CREATE INDEX` pour l'index |
| `pg_get_indexdef (index_oid, column_no, pretty_bool)` | `text` | récupère la commande `CREATE INDEX` pour l'index, ou une définition d'une seule colonne de l'index quand `column_no` est différent de zéro |
| `pg_get_triggerdef(trigger_oid)` | `text` | récupère la commande `CREATE [ CONSTRAINT ] TRIGGER` pour le déclencheur |
| `pg_get_constraintdef (constraint_oid)` | `text` | récupère la définition d'une contrainte |
| `pg_get_constraintdef(constraint_oid, pretty_bool)` | `text` | récupère la définition d'une contrainte |
| `pg_get_expr(expr_text, relation_oid)` | `text` | décompile la forme interne d'une expression, en supposant que toute Var compris en lui fait référence à la relation indiquée dans le deuxième paramètre |
| `pg_get_expr(expr_text, relation_oid, pretty_bool)` | `text` | décompile la forme interne d'une expression, en supposant que toute Var compris en lui fait référence à la relation indiquée dans le deuxième paramètre |
| `pg_get_userbyid(userid)` | `name` | récupère le nom de l'utilisateur suivant l'ID donné |
| `pg_get_serial_sequence (table_name, column_name)` | `text` | récupère le nom de la séquence qu'une colonne `serial` ou `bigserial` utilise |
| | | |

| | | |
|---|---|---|
| pg_tablespace_databases (tablespace_oid) | setof oid | récupère un ensemble d'OID de la base de données ayant des objets dans l'espace logique |

format_type renvoie le nom SQL d'un type de données qui est identifié par son OID de type et peut−être par un modificateur de type. Passez NULL au modificateur de type si aucun modificateur spécifique n'est connu.

pg_get_viewdef, pg_get_ruledef, pg_get_indexdef, pg_get_triggerdef et pg_get_constraintdef reconstruisent la commande de création pour, respectivement, une vue, une règle, un index, un déclencheur ou une contrainte. (Notez que ceci est une reconstruction décompilée, pas le texte original de la commande.) pg_get_expr décompile la forme interne d'une expression individuelle, comme la valeur par défaut d'une colonne. Cela pourrait être utilisé lors de l'examen du contenu des catalogues systèmes. La plupart de ces fonctions viennent en deux variantes, une qui peut << afficher joliment >> le résultat (en option). Ce format est plus lisible mais le format par défaut a plus de chances d'être interprété de la même façon par les versions futures de PostgreSQL ; évitez d'utiliser ce format pour réaliser des sauvegardes. Passer false pour le paramètre de joli affichage renvoie le même résultat que la variante qui n'a pas ce paramètre.

pg_get_userbyid extrait le nom d'un utilisateur à partir de son identifiant. pg_get_serial_sequence récupère le nom de la séquence associée avec une colonne de type serial ou bigserial. Le nom est convenablement formaté pour être passé aux fonctions sur les séquences (voir Section 9.12). NULL est renvoyé si la colonne n'a pas de séquence attachée.

pg_tablespace_databases autorise l'examen de l'usage d'un espace logique. Elle renverra un ensemble d'OID des bases de données contenant des objets dans l'espace logique. Si cette fonction renvoie des lignes, l'espace logique n'est pas vide et ne peux pas être supprimé. Pour afficher les objets spécifiques peuplant l'espace logique, vous aurez besoin de vous connecter aux bases de données identifiées par pg_tablespace_databases et de demander leurs catalogues pg_class.

Les fonctions affichées dans Tableau 9−43 extraient des commentaires stockés précédemment avec la commande COMMENT. Une valeur NULL est renvoyée si aucun commentaire ne correspond aux paramètres spécifiés.

**Tableau 9−43. Fonctions d'informations sur les commentaires**

| Nom | Type de retour | Description |
|---|---|---|
| obj_description (object_oid, catalog_name) | text | récupère un commentaire à partir d'un objet de la base de données |
| obj_description(object_oid) | text | récupère un commentaire à partir d'un objet de la base de données (*obsolète*) |
| col_description (table_oid, column_number) | text | récupère un commentaire sur une colonne d'une table |

La forme à deux paramètres de obj_description renvoie le commentaire d'un objet de la base de données, spécifié par son OID et le nom du catalogue système le contenant. Par exemple, obj_description(123456,'pg_class') récupérerait le commentaire pour une table d'OID 123456. La forme à un paramètre de obj_description requiert seulement l'OID de l'objet. Elle est maintenant obsolète car il n'existe aucune garantie que les OID soient uniques au travers des différents catalogues système ; du coup, un mauvais commentaire pourrait être renvoyé.

`col_description` renvoie un commentaire pour une colonne de la table, spécifié par l'OID de la table et le numéro de la colonne. `obj_description` ne peut pas être utilisé pour les colonnes de table car les colonnes ne disposent d'OID.

# 9.20. Fonctions d'administration système

Tableau 9–44 affiche les fonctions disponibles pour connaître ou modifier les paramètres de configuration en exécution.

**Tableau 9–44. Fonctions de paramétrage de configuration**

| Nom | Type de retour | Description |
|---|---|---|
| `current_setting (nom_paramètre)` | `text` | valeur actuelle du paramètre |
| `set_config (nom_paramètre, nouvelle_valeur, est_locale)` | `text` | configure le paramètre et renvoie la nouvelle valeur |

La fonction `current_setting` renvoie la valeur courante du paramètre `nom_paramètre`. Elle correspond à la commande SQL `SHOW`. Un exemple :

```
SELECT current_setting('datestyle');

current_setting
-----------------
ISO, MDY
(1 row)
```

`set_config` configure le paramètre `nom_paramètre` avec `nouvelle_valeur`. Si `est_locale` vaut `true`, la nouvelle valeur s'appliquera seulement à la transaction en cours. Si vous voulez que la nouvelle valeur s'applique à la session en cours, utilisez `false` à la place. La fonction correspond à la commande SQL `SET`. Un exemple :

```
SELECT set_config('log_statement_stats', 'off', false);

set_config
------------
off
(1 row)
```

La fonction montrée dans Tableau 9–45 envoie des signaux de contrôle aux autres processus serveur. L'utilisation de cette fonction est restreinte aux superutilisateurs.

**Tableau 9–45. Fonctions d'envoi de signal aux serveurs**

| Nom | Type de retour | Description |
|---|---|---|
| `pg_cancel_backend (pid)` | `int` | Annule la requête en cours du serveur |

Cette fonction renvoie 1 en cas de succès, 0 en cas d'échec. L'identifiant du processus (`pid`) d'un serveur actif backend est disponible à partir de la colonne `procpid` dans la vue `pg_stat_activity` ou en listant les processus `postgres` sur le serveur avec ps.

Les fonctions montrées dans Tableau 9−46 aident à l'exécution de sauvegardes à chaud. L'utilisation de ces fonctions est restreinte aux superutilisateurs.

**Tableau 9−46. Fonctions de contrôle de la sauvegarde**

| Nom | Type de retour | Description |
|---|---|---|
| `pg_start_backup (texte_label)` | `text` | Début de la sauvegarde à chaud |
| `pg_stop_backup()` | `text` | Fin de la sauvegarde à chaud |

`pg_start_backup` accepte un seul paramètre qui est un label défini arbitrairement par l'utilisateur pour la sauvegarde. (Typiquement, cela sera le nom sous lequel le fichier de sauvegarde sera stocké.) La fonction écrit un fichier label dans le répertoire de données du groupe, puis renvoie le décalage du WAL de début de sauvegarde au format texte. (L'utilisateur n'a pas besoin de faire attention à la valeur du résultat mais il est fourni au cas où il pourrait être utile.)

`pg_stop_backup` supprime le fichier label créé par `pg_start_backup` et crée, à la place, un fichier historique dans l'aire des archives WAL. Ce fichier inclut le label donné à `pg_start_backup`, les décalages de début et de fin des WAL pour la sauvegarde ainsi que les heures de début et de fin de la sauvegarde. La valeur en retour est le décalage du WAL de fin (qui a de nouveau peu d'intérêt).

Pour des détails sur le bon usage de ces fonctions, voir Section 22.3.

# Chapitre 10. Conversion de types

Le mélange de différents types de données dans la même expression peut être requis, intentionnellement ou pas, par les instructions SQL. PostgreSQL possède des fonctionnalités étendues pour évaluer les expressions de type mixte.

Dans la plupart des cas, un utilisateur n'aura pas besoin de comprendre les détails du mécanisme de conversion des types. Cependant, les conversions implicites faites par PostgreSQL peuvent affecter le résultat d'une requête. Quand cela est nécessaire, ces résultats peuvent être atteints directement en utilisant la conversion *explicite* de types.

Ce chapitre introduit les mécanismes et les conventions sur les conversions de types dans PostgreSQL. Référez−vous aux sections appropriées du Chapitre 8 et du Chapitre 9 pour plus d'informations sur les types de données spécifiques, les fonctions et les opérateurs autorisés.

## 10.1. Vue d'ensemble

SQL est un langage fortement typé. C'est−à−dire que chaque élément de données est associé à un type de données qui détermine son comportement et son utilisation permise. PostgreSQL a un système de types extensible qui est beaucoup plus général et flexible que les autres implémentations de SQL. Par conséquent, la plupart des comportements de conversion de types dans PostgreSQL est régie par des règles générales plutôt que par une heuristique *ad hoc*. Cela permet aux expressions de types mixtes d'être significatives même avec des types définis par l'utilisateur.

L'analyseur de PostgreSQL divise les éléments lexicaux en seulement cinq catégories fondamentales : les entiers, les nombres non entiers, les chaînes de caractères, les identifieurs et les mots−clé. Les constantes de la plupart des types non−numériques sont d'abord classifiées comme chaînes de caractères. La définition du langage SQL permet de spécifier le nom des types avec une chaîne et ce mécanisme peut être utilisé dans PostgreSQL pour lancer l'analyseur sur le bon chemin. Par exemple, la requête

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

a deux constantes littérales, de type `text` et `point`. Si un type n'est pas spécifié pour une chaîne littérale, alors le type `inconnu` est assigné initialement pour être résolu dans les étapes ultérieures comme décrit plus bas.

Il y a quatre constructions SQL fondamentales qui exigent des règles distinctes de conversion de types dans l'analyseur de PostgreSQL :

Les appels de fonctions
: Une grande partie du système de types de PostgreSQL est construit autour d'un riche ensemble de fonctions. Les fonctions peuvent avoir un ou plusieurs arguments. Puisque que PostgreSQL permet la surcharge des fonctions, le nom seul de la fonction n'identifie pas de manière unique la fonction à appeler ; l'analyseur doit sélectionner la bonne fonction par rapport aux types des arguments fournis.

Les opérateurs
: PostgreSQL autorise les expressions avec des opérateurs de préfixe et de suffixe unaires (un argument) aussi bien que binaires (deux arguments). Comme les fonctions, les opérateurs peuvent être surchargés, et donc le même problème existe pour sélectionner le bon opérateur.

Le stockage des valeurs
: Les clauses SQL `INSERT` et `UPDATE` placent le résultat des expressions dans une table. Les expressions dans une clause doivent être en accord avec le type des colonnes cibles et peuvent être converties vers celles−ci.

Les constructions `UNION`, `CASE` et `ARRAY`
: Comme toutes les requêtes issues d'une clause `SELECT` utilisant une union doivent apparaître dans un ensemble unique de colonnes, les types de résultats de chaque clause `SELECT` doivent être assortis et convertis en un ensemble uniforme. De façon similaire, les expressions de résultats d'une construction `CASE` doivent être converties vers un type commun de façon à ce que l'ensemble de l'expression `CASE` ait un type de sortie connu. Cela est la même chose pour les constructions avec `ARRAY`.

Les catalogues systèmes stockent les informations au sujet de la validité des conversions entre les types de données et comment exécuter ces conversions. Les conversions sont appelées *casts* en anglais. Des conversions de types supplémentaires peuvent être ajoutées par l'utilisateur avec la commande `CREATE CAST`. (Cela est habituellement fait en conjonction avec la définition de nouveaux types de données. L'ensemble des conversions entre les types prédéfinis a été soigneusement choisi et le mieux est de ne pas le modifier.)

Concernant les types SQL standards, une heuristique additionnelle est fournie dans l'analyseur pour permettre de meilleures estimations du comportement approprié. Il y a plusieurs *catégories de types* basiques définies : `boolean`, `numeric`, `string`, `bitstring`, `datetime`, `timespan`, `geometric`, `network` et utilisateurs. Chaque catégorie, à l'exception des types définis par l'utilisateur, a un ou plusieurs *types pré−définis* qui sont préférentiellement choisis quand il y a une ambiguïté. Dans la catégorie des types utilisateurs, chaque type est son propre type préféré. Les expressions ambiguës (celles avec de multiples solutions d'analyse candidates) peuvent souvent être résolues quand il y a de nombreux types pré−définis possibles mais elles soulèveront une erreur quand il existe des choix multiples pour des types utilisateurs.

Toutes les règles de conversions de types sont écrites en gardant à l'esprit plusieurs principes :

- Les conversions implicites ne doivent jamais avoir de résultats surprenants ou imprévisibles.
- Les types utilisateurs dont l'analyseur n'a pas à priori connaissance, devraient être << plus hauts >> dans la hiérarchie des types. Dans les expressions de types mixtes, les types natifs doivent toujours être convertis en un type utilisateur (seulement si la conversion est nécessaire bien sûr).
- Les types utilisateurs ne sont pas reliés. Actuellement PostgreSQL n'a pas d'informations disponibles sur les relations entre les types, autres que celles fournies par les heuristiques codées en dur pour les types natifs et les relations implicites basées sur les fonctions et les conversions disponibles.
- Il n'y aura pas de surcharge depuis l'analyseur ou l'exécuteur si une requête n'a pas besoin d'une conversion de types implicite. C'est−à−dire que si une requête est bien formulée et si les types sont déjà assortis, alors la requête devra s'exécuter sans perte de temps supplémentaire et sans introduire à l'intérieur de celle−ci des appels à des conversions implicites non nécessaires.

  De plus, si une requête nécessite habituellement une conversion implicite pour une fonction et si l'utilisateur définit une nouvelle fonction avec les types des arguments corrects, l'analyseur devrait utiliser cette nouvelle fonction et ne fera plus des conversions implicites en utilisant l'ancienne fonction.

# 10.2. Opérateurs

L'opérateur spécifique à employer dans un appel d'opérateurs est déterminé par la procédure ci−dessous. Notez que cette procédure est indirectement affectée par l'ordre d'insertion des opérateurs. Regardez la Section 4.1.6 pour plus de détails.

**Résolution de types pour les opérateurs**

1. Sélectionner les opérateurs à examiner depuis le catalogue système `pg_operator`. Si un nom non−qualifié d'opérateur était utilisé (le cas habituel), les opérateurs examinés sont ceux avec un nom et un nombre d'arguments corrects et qui sont visibles dans le chemin de recherche courant (regardez la Section 5.8.3). Si un nom qualifié d'opérateur a été donné, seuls les opérateurs dans le schéma spécifique sont examinés.

    a. Si un chemin de recherche trouve de nombreux opérateurs avec des types d'arguments identiques, seul celui apparaissant le plus tôt dans le chemin sera examiné. Mais les opérateurs avec des types d'arguments différents sont examinés sur une base d'égalité indépendamment de leur position dans le chemin de recherche.
2. Vérifier que l'opérateur accepte le type exact des arguments en entrée. Si un opérateur existe (il peut en avoir uniquement un qui corresponde exactement dans l'ensemble des opérateurs considérés), utiliser cet opérateur.

    a. Si un argument lors d'une invocation d'opérateur binaire est de type `inconnu`, alors considérer pour ce contrôle que c'est le même type que l'autre argument. Les autres cas impliquant le type `inconnu` ne trouveront jamais une correspondance à ce niveau.
3. Regarder pour la meilleure correspondance.

    a. Se débarrasser des opérateurs candidats pour lesquels les types en entrée ne correspondent pas et qui ne peuvent pas être convertis (en utilisant une conversion implicite) pour correspondre. Le type `inconnu` est supposé être convertible vers n'importe quoi. Si un candidat reste, l'utiliser, sinon aller à la prochaine étape.
    b. Parcourir tous les candidats et garder ceux avec la correspondance la plus exacte par rapport aux types en entrée. (Les domaines sont considérés de la même façon que leur type de base pour cette étape.) Garder tous les candidats si aucun n'a de correspondances exactes. Si seulement un candidat reste, l'utiliser ; sinon aller à la prochaine étape.
    c. Parcourir tous les candidats et garder ceux qui acceptent les types préférés (de la catégorie des types de données en entrée) aux positions où la conversion de types aurait été requise. Garder tous les candidats si aucun n'accepte les types préférés. Si seulement un candidat reste, l'utiliser ; sinon aller à la prochaine étape.
    d. Si des arguments en entrée sont `inconnus`, vérifier la catégorie des types acceptés à la position de ces arguments par les candidats restants. À chaque position, sélectionner la catégorie `chaîne de caractères` si un des candidats accepte cette catégorie. (Cette préférence envers les chaînes de caractères est appropriée car le terme type−inconnu ressemble à une chaîne de caractères.) Dans le cas contraire, si tous les candidats restants acceptent la même catégorie de types, sélectionner cette catégorie. Dans le cas contraire échouer car le choix correct ne peut pas être déduit sans plus d'indices. Se débarrasser maintenant des candidats qui n'acceptent pas la catégorie sélectionnée. De plus, si des candidats acceptent un type préféré comme argument donné, se débarrasser des candidats qui acceptent, pour cet argument, les types qui ne sont pas préférés.

e. Si seulement un candidat reste, l'utiliser. Si aucun candidat ou si plus d'un candidat reste, alors échouer.

Quelques exemples suivent.

**Exemple 10–1. Résolution de types pour l'opérateur exponentiel**

Il n'y a qu'un seul opérateur exponentiel défini dans le catalogue et il prend deux arguments de type `double precision` (double précision). L'analyseur assigne initialement le type `integer` (entier) aux deux arguments de cette expression :

```
SELECT 2 ^ 3 AS "exp";

 exp
-----
   8
(1 row)
```

L'analyseur fait donc une conversion de types sur les deux opérandes et la requête est équivalente à

```
SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS "exp";
```

**Exemple 10–2. Résolution de types pour les opérateurs de concaténation de chaînes**

La syntaxe d'une chaîne de caractères est utilisée pour travailler avec les types chaînes mais aussi avec les types d'extensions complexes. Les chaînes de caractères avec un type non spécifié sont comparées avec les opérateurs candidats probables.

Un exemple avec un argument non spécifié :

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

Dans ce cas, l'analyseur cherche à voir s'il existe un opérateur prenant `text` pour ses deux arguments. Comme il y en a un, il suppose que le second argument devrait être interprété comme un type `text`.

Voici une concaténation sur des types non spécifiés :

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

Dans ce cas, il n'y a aucune allusion initiale sur quel type utiliser puisqu'aucun type n'est spécifié dans la requête. Donc l'analyseur regarde pour tous les opérateurs candidats et trouve qu'il existe des candidats acceptant en entrée la catégorie chaîne de caractères (string) et la catégorie morceaux de chaînes (bit–string).

Puisque la catégorie chaînes de caractères est préférée quand elle est disponible, cette catégorie est sélectionnée. Le type préféré pour la catégorie chaînes étant `texte` (text), ce type est utilisé comme le type spécifique pour résoudre les types inconnus.

**Exemple 10−3. Résolution de types pour les opérateurs de valeur absolue et de négation**

Le catalogue d'opérateurs de PostgreSQL a plusieurs entrées pour l'opérateur de préfixe `@`. Ces entrées implémentent toutes des opérations de valeur absolue pour des types de données numériques variées. Une de ces entrées est pour le type `float8` (réel) qui est le type préféré dans la catégorie des numériques. Par conséquent, PostgreSQL utilisera cette entrée quand il sera en face d'un argument non numérique :

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

Ici le système a effectué une conversion implicite du type `text` (texte) au type `float8` (réel) avant d'appliquer l'opérateur choisi. Nous pouvons vérifier que `float8` et pas un autre type a été utilisé :

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

D'un autre côté, l'opérateur préfixe ~ (négation bit par bit) est défini seulement pour les types entiers et non pas pour `float8` (réel). Ainsi, si nous essayons un cas similaire avec ~, nous obtenons :

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You may need to add explicit
type casts.
```

Ceci se produit parce que le système ne peut pas décider quel opérateur doit être préféré parmi les différents opérateurs ~ possibles. Nous pouvons l'aider avec une conversion explicite :

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

# 10.3. Fonctions

La fonction spécifique à utiliser dans une invocation de fonctions est déterminée selon les étapes suivantes.

**Résolution de types pour les fonctions**

1. Sélectionner les fonctions à examiner depuis le catalogue système `pg_proc`. Si un nom non−qualifié de fonction était utilisé, les fonctions examinées sont celles avec un nom et un nombre d'arguments

corrects et qui sont visibles dans le chemin de recherche courant (regardez la Section 5.8.3). Si un nom qualifié de fonctions a été donné, seules les fonctions dans le schéma spécifique sont examinées.

   a. Si un chemin de recherche trouve de nombreuses fonctions avec des types d'arguments identiques, seule celle apparaissant le plus tôt dans le chemin sera examinée. Mais les fonctions avec des types d'arguments différents sont examinées sur une base d'égalité indépendamment de leur position dans le chemin de recherche.

2. Vérifier que la fonction accepte le type exact des arguments en entrée. Si une fonction existe (il peut en avoir uniquement une qui corresponde exactement dans l'ensemble des fonctions considérées), utiliser cette fonction. (Les cas impliquant le type `inconnu` ne trouveront jamais de correspondance à cette étape.)
3. Si aucune correspondance n'est trouvée, regarder si l'appel à la fonction apparaît être une requête triviale de conversion de types. Cela arrive si l'appel à la fonction a juste un argument et si le nom de la fonction est le même que le nom (interne) de certains types de données. De plus, l'argument de la fonction doit être soit un type inconnu soit un type qui a une compatibilité binaire avec le type de données nommés. Quand ces conditions sont réunies, l'argument de la fonction est converti vers le type de données nommé sans aucun appel effectif à la fonction.
4. Regarder pour la meilleure correspondance.

   a. Se débarrasser des fonctions candidates pour lesquelles les types en entrée ne correspondent pas et qui ne peuvent pas être convertis (en utilisant une conversion implicite) pour correspondre. Le type `inconnu` est supposé être convertible vers n'importe quoi. Si seulement un candidat reste, utiliser le, sinon aller à la prochaine étape.
   b. Parcourir tous les candidats et garder ceux avec la correspondance la plus exacte par rapport aux types en entrée. (Les domaines sont considérés de la même façon que leur type de base pour cette étape.) Garder tous les candidats si aucun n'a de correspondances exactes. Si seulement un candidat reste, utiliser le ; sinon aller à la prochaine étape.
   c. Parcourir tous les candidats et garder ceux qui acceptent les types préférés (de la catégorie des types de données en entrée) aux positions où la conversion de types aurait été requise. Garder tous les candidats si aucun n'accepte les types préférés. Si seulement un candidat reste, utiliser le ; sinon aller à la prochaine étape.
   d. Si des arguments en entrée sont `inconnu`, vérifier les catégories de types acceptées à la position de ces arguments par les candidats restants. À chaque position, sélectionner la catégorie `chaîne de caractères` si un des candidats accepte cette catégorie. (Cette préférence envers les chaînes de caractères est appropriée depuis que le terme type−inconnu ressemble à une chaîne de caractères.) Dans le cas contraire, si tous les candidats restants acceptent la même catégorie de types, sélectionner cette catégorie. Dans le cas contraire échouer car le choix correct ne peut pas être déduit sans plus d'indices. Se débarrasser maintenant des candidats qui n'acceptent pas la catégorie sélectionnée. De plus, si des candidats acceptent un type préféré comme argument donné, se débarrasser des candidats qui acceptent, pour cet argument, les types qui ne sont pas préférés.
   e. Si seulement un candidat reste, utiliser le. Si aucun candidat ou si plus d'un candidat reste, alors échouer.

Notez que les règles de << correspondance optimale >> sont identiques pour la résolution de types concernant les opérateurs et les fonctions. Quelques exemples suivent.

**Exemple 10−4. Résolution de types pour les arguments de la fonction arrondie**

Il n'existe qu'une seule fonction round avec deux arguments. (Le premier est un numérique, le second est un entier. Ainsi, la requête suivante convertie automatiquement le type du premier argument de entier vers numérique.

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

La requête est en fait transformée par l'analyseur en

```
SELECT round(CAST (4 AS numeric), 4);
```

Puisque le type numérique est initialement assigné aux constantes numériques avec un point décimal, la requête suivante ne requièrera pas une conversion de types et pourra par conséquent être un peu plus efficace :

```
SELECT round(4.0, 4);
```

**Exemple 10–5. Résolution de types pour les fonctions retournant un segment de chaîne**

Il existe plusieurs fonctions substr, une d'entre elles prend les types texte et entier. Si cette fonction est appelée avec une constante de chaînes d'un type inconnu, le système choisi la fonction candidate qui accepte un argument issu de la catégorie préférée chaînes (c'est–à–dire de type texte).

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

Si la chaîne de caractères est déclarée comme étant du type varchar (chaîne de caractères de longueur variable), ce qui peut être le cas si elle vient d'une table, alors l'analyseur essayera de la convertir en texte :

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

Ceci est transformé par l'analyseur pour efficacement devenir

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **Note :** L'analyseur apprend depuis le catalogue pg_cast que les types texte et varchar ont une compatibilité binaire, ce qui veut dire que l'un peut être passé à une fonction qui accepte l'autre sans avoir à faire aucune conversion physique. Par conséquent, aucun appel de conversion explicite de types n'est réellement inséré dans ce cas.

Et si la fonction est appelée avec un argument de type entier, l'analyseur essayera de le convertir en texte :

```
SELECT substr(1234, 3);

 substr
--------
     34
(1 row)
```

Ceci est en fait exécuté de la façon suivante :

```
SELECT substr(CAST (1234 AS text), 3);
```

Cette transformation automatique peut réussir parce qu'il y a une invocation de conversion implicite du type `entier` vers le type `texte`.

# 10.4. Stockage de valeurs

Les valeurs qui doivent être insérées dans une table sont converties vers le type de données de la colonne de destination selon les règles suivantes.

**Conversion de types pour le stockage de valeurs**

1. Vérifier qu'il y ait une correspondance exacte avec la cible.
2. Dans le cas contraire, essayer de convertir l'expression vers le type cible. Cela réussira s'il y a une conversion (cast) enregistrée entre ces deux types. Si une expression est de type inconnu, le contenu de la chaîne littérale sera fourni à l'entrée de la routine de conversion pour le type cible.
3. Vérifier s'il y a une conversion de taille pour le type cible. Une conversion de taille est une conversion d'un type vers lui−même. Si elle est trouvée dans le catalogue `pg_cast`, appliquez−la à l'expression avant de la stocker dans la colonne de destination. La fonction d'implémentation pour une telle conversion prend toujours un paramètre supplémentaire de type `integer`, qui reçoit la longueur déclarée de la colonne de destination (en fait, il s'agit de la valeur `atttypmod` ; l'interprétation de `atttypmod` varie pour les différents types de données). La fonction de conversion est responsable de l'application de toute sémantique dépendante de la longueur comme la vérification de la taille ou une troncature.

**Exemple 10−6. Conversion de types pour le stockage de `caractères`**

Pour une colonne cible déclarée comme `character(20)`, la déclaration suivante assure que la valeur stockée a la taille correcte :

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, length(v) FROM vv;

          v           | length
----------------------+--------
 abcdef               |     20
(1 row)
```

Ce qui s'est réellement passé ici, c'est que les deux types inconnus sont résolus en `texte` par défaut, permettant à l'opérateur `||` de les résoudre comme une concaténation de `texte`. Ensuite, le résultat `texte` de l'opérateur est converti en `bpchar` ( << blank−padded char >>, le nom interne du type de données `character` (caractère)) pour correspondre au type de la colonne cible. (Comme les types `texte` et

bpchar ont une compatibilité binaire, cette conversion n'insère aucun appel réel à une fonction.) Enfin, la fonction de taille bpchar(bpchar, integer) est trouvée dans le catalogue système et appliquée au résultat de l'opérateur et à la longueur de la colonne stockée. Cette fonction de type spécifique effectue le contrôle de la longueur requise et ajoute des espaces pour combler la chaîne.

## 10.5. Constructions UNION, CASE et ARRAY

Les constructions SQL avec des UNION doivent potentiellement faire correspondre des types différents pour avoir un ensemble unique dans le résultat. L'algorithme de résolution est appliqué séparément à chaque colonne de sortie d'une requête d'union. Les constructions INTERSECT et EXCEPT résolvent des types différents de la même manière qu'UNION. Les constructions CASE et ARRAY utilisent le même algorithme pour faire correspondre les expressions qui les composent et sélectionner un type de résultat.

**Résolution des types pour UNION, CASE et ARRAY**

1. Si toutes les entrées sont du type inconnu, résoudre comme étant du type texte (le type préféré de la catégorie chaîne). Dans le cas contraire, ignorer les entrées inconnues pendant le choix du type du résultat.
2. Si toutes les entrées non−inconnues ne sont pas toutes de la même catégorie, échouer.
3. Choisir la première entrée non−inconnue qui soit un type préféré dans sa catégorie ou autoriser toutes les entrées non−inconnues à être implicitement converties vers elle.
4. Convertir toutes les entrées vers le type sélectionné.

Quelques exemples suivent.

**Exemple 10−7. Résolution de types avec des types sous−spécifiés dans une union**

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

Ici, la chaîne de type inconnu 'b' sera convertie vers le type texte.

**Exemple 10−8. Résolution de types dans une union simple**

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

Le littéral 1.2 est du type numérique (numeric en anglais) et la valeur 1, de type entier, peut être convertie implicitement vers un type numérique, donc ce type est utilisé.

**Exemple 10–9. Résolution de types dans une union transposée**

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

Dans cet exemple, le type `real` (réel) ne peut pas être implicitement converti en `integer` (entier) mais un `integer` peut être implicitement converti en `real` ; le résultat de l'union est résolu comme étant un `real`.

# Chapitre 11. Index

Les index sont une méthode courante pour augmenter les performances d'une base de données. Un index permet au serveur de bases de données de retrouver une ligne spécifique bien plus rapidement que sans index. Mais les index ajoutent aussi une surcharge au système de base de données dans son ensemble, si bien qu'ils doivent être utilisés avec discernement.

## 11.1. Introduction

Supposons que nous ayons une table comme celle–ci:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

et que l'application utilise beaucoup de requêtes de la forme

```
SELECT content FROM test1 WHERE id = constant;
```

Sans préparation supplémentaire, le système devrait lire la table `test1` en entier, ligne par ligne, pour trouver toutes les lignes qui correspondent. S'il y a beaucoup de lignes dans `test1`, et que seulement quelques lignes correspondent à la requête (peut–être même zéro ou une seule), alors, clairement, la méthode n'est pas efficace. Mais si on a indiqué au système de maintenir un index sur la colonne `id`, alors il peut utiliser une manière beaucoup plus efficace pour trouver les lignes recherchées. Par exemple, il pourrait n'avoir à parcourir que quelques niveaux d'un arbre de recherche.

Une approche similaire est utilisée dans la plupart des livres autres que ceux de fiction: les termes et concepts qui sont fréquemment recherchés par les lecteurs sont listés par ordre alphabétique à la fin du livre. Le lecteur qui recherche un mot particulier peut facilement parcourir l'index, puis aller directement à la page ou aux pages indiquée(s). De la même façon que l'auteur doit anticiper les sujets que les lecteurs risquent de rechercher, il est de la responsabilité du programmeur de prévoir quels index seraient avantageux.

La commande suivante permet de créer un index sur la colonne `id` dont nous parlons:

```
CREATE INDEX test1_id_index ON test1 (id);
```

Le nom `test1_id_index` peut être choisi librement, mais il est conseillé de choisir un nom qui rappelle le but de l'index.

Pour supprimer l'index, utilisez la commande `DROP INDEX`. Les index peuvent être ajoutés et enlevés des tables à tout moment.

Une fois un index créé, aucune intervention supplémentaire n'est nécessaire: Le système met à jour l'index lorsque la table est modifiée, et utilise l'index dans les requêtes lorsqu'il pense que c'est plus efficace qu'une lecture complète de la table. Il faut néanmoins lancer la commande `ANALYZE` régulièrement pour permettre à l'optimiseur de requêtes de prendre les bonnes décisions. Voyez Chapitre 13 pour comprendre quand et pourquoi l'optimiseur décide d'utiliser ou de ne *pas* utiliser un index.

Les index peuvent aussi bénéficier aux commandes `UPDATE` et `DELETE` qui ont des conditions de recherche. Les index peuvent de plus être utilisés dans les jointures. Ainsi, un index défini sur une colonne qui fait partie d'une condition de jointure peut accélérer significativement les requêtes avec jointures.

Lorsqu'un index est créé, le système doit le maintenir synchronisé avec la table. Cela rend plus lourdes les opérations de manipulation de données. C'est pourquoi les index qui ne sont pas essentiels ou qui ne sont pas utilisés du tout doivent être supprimés. Notez qu'une requête ou une commande de manipulation de données ne peut utiliser qu'un index par table au maximum.

## 11.2. Types d'index

PostgreSQL propose plusieurs types d'index: B−tree, R−tree, Hash et GiST. Chaque type d'index utilise un algorithme différent qui convient à un type particulier de requêtes. Par défaut, la commande `CREATE INDEX` créera un index B−tree, ce qui convient dans la plupart des situations. Les index B−tree savent traiter les égalités et les recherches sur des tranches de valeurs sur les données qui peuvent être triées. En particulier, l'optimiseur de requêtes de PostgreSQL essaie d'utiliser un index B−tree lorsque une colonne indexée est utilisée dans une comparaison qui utilise un de ces opérateurs:

```
<
<=
=
>=
>
```

Les constructions équivalentes à des combinaisons de ces opérateurs, comme `BETWEEN` et `IN`, peuvent aussi être implémentées avec une recherche par index B−tree. (Mais notez que `IS NULL` n'est pas équivalent à = et n'est pas indexable.)

L'optimiseur peut aussi utiliser un index B−tree pour des requêtes qui utilisent les opérateurs de recherche de motif `LIKE`, `ILIKE`, `~`, et `~*`, *si* le motif est au début de la chaîne à rechercher. Par exemple: `col LIKE 'foo%'` ou `col ~ '^foo'`, mais pas `col LIKE '%bar'`. Néanmoins, si votre serveur n'utilise pas la localisation C, il vous faudra créer l'index avec une classe d'opérateur spéciale pour supporter l'indexage à correspondance de modèles. Voir Section 11.6 ci−dessous.

Les index R−tree sont adaptés aux requêtes sur des données spatiales. Pour créer un index R−tree, utilisez une commande de la forme:

```
CREATE INDEX name ON table USING RTREE (column);
```

L'optimiseur de requêtes de PostgreSQL envisagera d'utiliser un index R−tree lorsqu'une colonne indexée fait partie d'une comparaison utilisant l'un de ces opérateurs:

```
<<
&<
&>
>>
@
~=
```

&&

(Voir Section 9.10 pour connaître la signification de ces opérateurs.)

Les index hachés ne supportent que les simples comparaisons d'égalité. L'optimiseur de requêtes envisagera d'utiliser un index haché lorsqu'une colonne indexée fait partie d'une comparaison utilisant l'opérateur =. La commande suivante est utilisée pour créer un index haché:

```
CREATE INDEX name ON table USING HASH (column);
```

> **Note :** Les tests ont montré que les index hachés de PostgreSQL ne sont pas plus efficaces que les index B−tree, et que la taille de l'index et le temps de création d'un index hashé sont bien moins bons. Pour ces raisons, l'utilisation des index hachés est actuellement découragée.

Les index GiST ne sont pas un seul genre d'index mais plutôt une infrastructure à l'intérieur de laquelle plusieurs stratégies d'indexage peuvent être implémentées. EN accord, les opérateurs particuliers avec lesquels un index GiST peut être utilisé varient suivant sur la stratégie d'indexage (la *classe d'opérateur*). Pour plus d'informations, voir Chapitre 48.

La méthode d'index B−tree est une implémentation des B−trees à haute concurrence de Lehman−Yao. La méthode d'index R−tree implémente les R−tree standards en utilisant l'algorithme de découpage quadratique de Guttman. La méthode d'index par hachage est une implémentation de l'algorithme de hachage linéaire de Litwin. Nous ne mentionnons les algorithmes utilisés que pour indiquer que toutes ces méthodes d'indexation sont complètement dynamiques et n'ont pas besoin d'une optimisation périodique (au contraire, par exemple, des méthodes de hachage statique).

## 11.3. Les index multicolonnes

Un index peut porter sur plus d'une colonne. Par exemple, si vous avez une table de cette forme:

```
CREATE TABLE test2 (
  majeur int,
  mineur int,
  nom varchar
);
```

(par exemple, si vous gardez votre répertoire `/dev` dans une base de données...) et que vous faites fréquemment des requêtes comme:

```
SELECT nom FROM test2 WHERE majeur = constante AND mineur = constante;
```

alors il est sans doute souhaitable de définir un index sur les colonnes `majeur` et `mineur` ensemble, par exemple avec:,

```
CREATE INDEX test2_mm_idx ON test2 (majeur, mineur);
```

Actuellement, seuls les B−trees et les index GiST supportent les index multicolonnes. Jusqu'à 32 colonnes peuvent être indexées. Cette limite peut être modifiée à la compilation de PostgreSQL. Voyez le fichier `pg_config_manual.h`.

L'optimiseur de requêtes peut utiliser un index multicolonnes pour les requêtes qui utilisent la colonnes la plus à gauche de la définition de l'index, plus un nombre quelconque des colonnes listées à sa droite, sans trou. Par exemple, un index sur `(a, b, c)` peut être utilisé dans des requêtes utilisant `a`, `b`, et `c`, ou dans des requêtes utilisant à la fois `a` et `b`, ou dans des requêtes n'utilisant que `a`, mais pas dans une requête utilisant une autre combinaison. (Dans une requête utilisant `a` et `c`, l'optimiseur pourrait choisir d'utiliser l'index pour `a`, en traitant `c` comme une colonne ordinaire non indexée.) Bien sur, chaque colonne doit être utilisée avec les opérateurs appropriés pour le type d'index. Les clauses qui comprennent un autre opérateur ne seront pas prises en compte.

Les index multicolonnes ne peuvent être utilisés que si les clauses des colonnes indexées sont jointes avec `AND`. Par exemple,

```
SELECT nom FROM test2 WHERE majeur = constant OR mineur = constant;
```

ne peut utiliser l'index `test2_mm_idx` (défini précédemment) sur les deux colonnes. (Il peut néanmoins l'utiliser pour faire une recherche sur la colonne `major`.)

Les index multicolonnes doivent être utilisés avec parcimonie. La plupart du temps, un index sur une seule colonne est suffisant et économise du temps et de l'espace disque. Les index avec plus de trois colonnes sont rarement utiles, sauf cas très particuliers.

## 11.4. Index Uniques

Les index peuvent aussi être utilisés pour garantir l'unicité des valeurs d'une colonne, ou l'unicité des valeurs combinées de plusieurs colonnes.

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

À ce jour, seuls les index B−trees peuvent être déclarés uniques.

Lorsqu'un index est déclaré unique, des lignes différentes d'une table ne pourront avoir une valeur égale. Les valeurs Nulles ne sont pas considérées comme égales. Un index unique multicolonnes ne rejettera que les cas où toutes les colonnes indexées sont égales dans deux lignes.

PostgreSQL crée automatiquement un index unique quand une contrainte unique ou une clé primaire sont définies sur une table. L'index porte sur les colonnes qui composent la clé primaire ou la contrainte d'unicité (il s'agit d'un index multicolonnes, si c'est approprié). Cet index EST le mécanisme qui vérifie la contrainte.

> **Note :** La méthode la plus appropriée pour ajouter une contrainte à une table est `ALTER TABLE ... ADD CONSTRAINT`. L'utilisation des index pour vérifier les contraintes uniques doit être considérée comme un détail d'implémentation qui ne doit pas être utilisé directement. Il faut par contre savoir qu'il n'est pas nécessaire de créer manuellement un index sur les colonnes uniques. Cela dupliquerait l'index créé automatiquement.

## 11.5. Index sur des expressions

Une colonne d'index ne correspond pas nécessairement exactement à une colonne de la table associée, mais peut être une fonction ou une expression scalaire calculée à partir d'une ou plusieurs colonnes de la table.

Cette fonctionnalité est utile pour obtenir un accès rapide aux tables basé sur les résultat des calculs.

Par exemple, une façon classique de faire des comparaisons indépendantes de la casse est d'utiliser la fonction `lower`:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

Si un index a été défini sur le résultat de `lower(col1)`, cette requête peut l'utiliser. Cet index est créé avec la commande:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Si nous avions déclaré cet index `UNIQUE`, il empêcherait la création de lignes dont la valeur de la colonne `col1` ne diffère que par la casse. Ainsi, les index sur les expressions peuvent être utilisés pour vérifier des contraintes qui ne peuvent êtres définies avec une simple contrainte.

Un autre exemple, si vous faites souvent des requêtes comme celle–ci:

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

alors il peut être utile de créer un index comme celui–ci:

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

La syntaxe de la commande `CREATE INDEX` nécessite normalement de mettre des parenthèses autour de l'expression indexés, comme dans l'exemple précédent. Les parenthèses peuvent être omises quand l'expression est juste un appel de fonction, comme dans le premier exemple.

Les expressions d'index sont relativement coûteuses à calculer, car l'expression doit être recalculée à chaque insertion ou mise à jour de chaque ligne. C'est pourquoi les index basés sur des expressions ne doivent être utilisés que quand les requêtes qui les exécutent sont très fréquentes.

## 11.6. Classes d'Opérateurs

Une définition d'index peut indiquer une *classe d'opérateurs* pour chaque colonne de l'index.

```
CREATE INDEX name ON table (column opclass [, ...]);
```

La classe d'opérateurs identifie les opérateurs que l'index doit utiliser sur cette colonne. Par exemple, un index B–tree sur une colonne de type `int4` utiliserait la classe `int4_ops`; Cette classe d'opérateurs comprend des fonctions de comparaison pour les valeurs de type `int4`. En pratique, la classe d'opérateurs par défaut pour le type de données de la colonne est généralement suffisant. Les classes d'opérateurs sont utiles pour certains types de données, pour lesquels il pourrait y avoir plus d'un comportement utile de l'index. Par exemple, nous pourrions vouloir trier une donnée de type nombre complexe soit par sa valeur absolue, soit par sa partie entière. Nous pourrions le faire en définissant deux classes d'opérateurs pour ce type de données et en sélectionnant la bonne classe en créant l'index.

Il y a quelques classes d'opérateurs en plus des classes par défaut:

- Les classes d'opérateurs `text_pattern_ops`, `varchar_pattern_ops`, `bpchar_pattern_ops`, et `name_pattern_ops` supportent les index B−tree sur les types `text`, `varchar`, `char`, et `name`, respectivement. La différence avec les classes d'opérateurs ordinaires est que les valeurs sont comparées strictement caractère par caractère plutôt que suivant les règles de tri spécifiques à la localisation. Cela rend ces index utilisables pour des requêtes qui utilisent des recherches sur des motifs (`LIKE` ou des expressions régulières POSIX) si le serveur n'utilise pas la localisation standard << C >>. Par exemple, on pourrait indexer une colonne `varchar` comme ceci:

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  Si vous utilisez la localisation C, vous pouvez à la place créer un index avec la classe d'opérateurs par défaut, qui sera utilisable pour les recherches de motifs. Notez aussi qu'il faut créer un index avec la classe d'opérateurs par défaut si vous voulez que les requêtes qui utilisent une comparaison ordinaire utilisent un index. De telles requêtes ne peuvent pas utiliser les classes d'opérateurs *xxx*`_pattern_ops`. Il est possible de créer plusieurs index sur la même colonne avec différentes classes d'opérateurs.

Les requêtes suivantes montrent toutes les classes d'opérateurs prédéfinies:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcamid = am.oid
    ORDER BY index_method, opclass_name;
```

Elle peut être étendue pour montrer tous les opérateurs inclus dans chaque classe:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opr.oprname AS opclass_operator
    FROM pg_am am, pg_opclass opc, pg_amop amop, pg_operator opr
    WHERE opc.opcamid = am.oid AND
          amop.amopclaid = opc.oid AND
          amop.amopopr = opr.oid
    ORDER BY index_method, opclass_name, opclass_operator;
```

## 11.7. Index partiels

Un *index partiel* est un index construit sur un sous−ensemble d'une table; le sous−ensemble est défini par une expression conditionnelle (appelée le *prédicat* de l'index partiel). L'index ne contient des entrées que pour les lignes de la table qui satisfont au prédicat.

L'usage principal des index partiels est d'éviter d'indexer les valeurs trop courantes. Comme une requête qui fait des recherches sur une valeur trop courante (qui correspond à plus de quelques pour−cent des lignes) n'utilisera pas cet index de toute façon, il ne sert à rien de garder ces lignes dans l'index. Cela réduit la taille de l'index, ce qui accélère les requêtes qui l'utilisent. Cela accélère aussi beaucoup d'opérations de mise à jour de la table car l'index n'a pas besoin d'être mis à jour à chaque fois. Exemple 11−1 montre une application possible de cette idée.

**Exemple 11−1. Mettre en place un index partiel pour exclure les valeurs courantes**

Supposons que vous enregistrez un journal d'accès à un serveur web dans une base de données. La plupart des accès proviennent de classes d'adresses IP internes à votre organisation, mais certaines viennent d'ailleurs (disons des employés connectés par modem). Si vos recherches sur des adresses IP concernent essentiellement les accès extérieures, vous n'avez probablement pas besoin d'indexer les classes d'adresses IP qui correspondent au sous−réseau de votre organisation.

Supposons que la table soit comme ceci:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

Pour créer un index partiel qui corresponde à notre exemple, il faut utiliser une commande comme celle−ci:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND client_ip < inet '192.168.100.255');
```

Une requête typique qui peut utiliser cet index est:

```
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

Une requête qui ne peut pas l'utiliser est:

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

Observez que cet type d'index partiel nécessite que les valeurs courantes soient prédéterminées. Si la distribution des valeurs est inhérente (du fait de la nature de l'application) et statique (ne changeant pas dans le temps), ce n'est pas trop difficile, mais si les valeurs courantes sont simplement dues au hasard, cela peut demander beaucoup de travail de maintenance.

Il est aussi possible d'exclure des valeurs de l'index qui ne correspondent pas aux requêtes courantes; ceci est montré dans Exemple 11−2. Cette méthode donne les mêmes avantages que la précédente, mais empêche les valeurs << sans intérêt >> d'être accédées. Évidemment, mettre en place des index partiels pour ce genre de scénario nécessite beaucoup de soin et d'expérimentation.

**Exemple 11−2. Mettre en place un index partiel pour exclure les valeurs inintéressantes**

Si vous avez une table qui contient des commandes facturées et des commandes non facturées, que les commandes non facturées ne prennent qu'une petite fraction de l'espace dans la table, et que ces commandes non facturées sont les plus accédées, alors vous pouvez améliorer les performances en créant un index limité aux lignes non facturées. La commande pour créer l'index ressemblerait à ceci:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

La requête suivante utilisera probablement cet index:

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

Néanmoins, l'index peut aussi être utilisé dans des requêtes qui n'utilisent pas `order_nr`, comme:

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

Ceci n'est pas aussi efficace qu'un index partiel sur la colonne `amount`, car le système doit lire l'index en entier. Néanmoins, s'il y a assez peu de commandes non facturées, l'utilisation de cet index partiel pour trouver les commandes non facturées peut être efficace.

Notez que cette requête ne peut pas utiliser cet index:

```
SELECT * FROM orders WHERE order_nr = 3501;
```

La commande 3501 peut faire partie des commandes facturées ou bien des commandes non facturées.

Exemple 11−2 illustre aussi le fait que la colonne indexée et la colonne utilisée dans le prédicat ne sont pas nécessairement les mêmes. PostgreSQL supporte tous les prédicats sur les index partiels, tant que ceux−ci ne portent que sur des champs de la table indexée. Néanmoins, il faut se rappeler le prédicat doit correspondre aux conditions utilisées dans les requêtes qui sont supposées profiter de l'index. Pour être précis, un index partiel ne peut être utilisé pour une requête que si le système peut reconnaître que la clause `WHERE` de la requête implique mathématiquement le prédicat de l'index. PostgreSQL n'a pas de méthode sophistiquée de démonstration de théorème pour reconnaître que des expressions apparemment différentes sont mathématiquement équivalentes. (Non seulement une telle méthode générale de démonstration serait extrêmement complexe à créer, mais en plus elle serait probablement trop lente pour être d'une quelconque utilité.) Le système peut reconnaître des implications d'inégalités simples, par exemple << x < 1 >> implique << x < 2 >>; sinon, la condition du prédicat doit correspondre exactement à une partie de la clause `WHERE` de la requête, sans quoi l'index ne sera pas considéré comme utilisable.

Le troisième usage possible des index partiels ne nécessite pas que l'index soit utilisé dans des requêtes. L'idée ici est de créer un index unique sur un sous−ensemble de la table, comme dans Exemple 11−3. Ceci permet de mettre en place une unicité parmi le sous−ensemble des lignes de la table qui satisfont au prédicat, sans contraindre les lignes qui n'y satisfont pas.

**Exemple 11−3. Mettre en place un index unique partiel**

Supposons que nous ayons une table qui décrive des résultats de tests. Nous voulons nous assurer qu'il n'y a qu'une seule entrée << succès >> (success) pour chaque combinaison de sujet (subject) et de résultat (target), mais il peut y avoir un nombre quelconque d'entrées << echec >>. Voici une façon de le faire.

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

C'est une méthode très efficace pour le faire quand il y a peu de tests réussis et beaucoup de tests en échec.

Enfin, un index partiel peut aussi être utilisé pour passer outre aux choix de plan d'exécution de requête du système. Il peut arriver avec certains jeux de données particuliers que le système utilise un index alors qu'il ne devrait vraiment pas le faire. Dans ce cas, on peut mettre en place l'index de telle faon qu'il ne soit pas utilisé pour la requête qui pose problème. Normalement, PostgreSQL fait des choix d'usage d'index raisonnables. Par

exemple, il les évite pour rechercher les valeurs communes, si bien que l'exemple précédent n'économise que la taille de l'index, il n'est en fait pas nécessaire pour éviter l'usage de l'index. En fait, les choix de plan d'exécution grossièrement incorrects doivent être traités comme des bogues, et être transmis à l'équipe de développement.

Gardez à l'esprit que mettre en place un index partiel indique que vous connaissez vos données au moins aussi bien que l'analyseur de requêtes, et en particulier que vous savez quand un index peut être profitable. Une telle connaissance nécessite de l'expérience et une bonne compréhension du fonctionnement des index de PostgreSQL. Dans la plupart des cas, les index partiels ne représentent pas un gros gain par rapport aux index classiques.

Vous trouverez plus d'informations sur les index partiels en lisant *The case for partial indexes*, *Partial indexing in POSTGRES: research project*, et *Generalized Partial Indexes*.

## 11.8. Examiner l'usage des index

Bien que les index de PostgreSQL n'aient pas besoin de maintenance ni d'optimisation, il est important de s'assurer que les index sont effectivement utilisés sur un système en production. On vérifie l'utilisation d'un index pour une requête particulière avec la commande *EXPLAIN*. Son utilisation dans notre cas est expliquée dans Section 13.1. Il est aussi possible de rassembler des statistiques globales sur l'utilisation des index sur un serveur en cours de fonctionnement, comme décrit dans Section 23.2.

Il est difficile de donner une procédure générale pour déterminer quels index doivent être créés. Plusieurs cas typiques ont été cités dans les exemples précédents. Une bonne dose d'expérimentation sera nécessaire dans de nombreux cas. Le reste de cette section donne quelques pistes.

- La première chose à faire est de lancer *ANALYZE*. Cette commande collecte les informations sur la distribution des valeurs dans la table. Cette information est nécessaire pour essayer de deviner le nombre lignes retournées par une requête. L'optimiseur de requêtes en a besoin pour donner des coûts réalistes aux différents plans de requêtes possibles. En l'absence de statistiques réelles, le système utilise quelques valeurs par défaut, qui ont toutes les chances d'être inadaptées. Examiner l'utilisation des index par une application sans avoir lancé `ANALYZE` préalablement est du coup une cause perdue.
- Utilisez des données réelles pour l'expérimentation. Utiliser des données de test pour mettre en place des index vous permettra de trouver les index dont vous avez besoin pour vos données de test, mais c'est tout.

  Il est particulièrement néfaste d'utiliser des jeux de données très réduits. Alors qu'une requête sélectionnant 1000 lignes parmi 100000 pourrait utiliser un index, il est peu probable qu'une requête sélectionnant 1 ligne dans une table de 100 lignes le fasse, parce que les 100 lignes tiennent probablement dans une seule page sur le disque, et qu'il n'y a aucun plan d'exécution qui puisse aller plus vite que la lecture d'une seule page.

  Soyez aussi vigilant en créant des données de test, ce qui est souvent inévitable quand l'application n'est pas encore en production. Les valeurs qui sont très similaires, complètement aléatoire, ou insérées déjà triées peuvent modifier la distribution des données et fausser les statistiques.
- Quand les index ne sont pas utilisés, il peut être utile pour les tests de forcer leur utilisation. Certains paramètres d'exécution du serveur peuvent interdire certains types de plans (décrits dans Section 16.4). Par exemple, en interdisant les lectures séquentielles de tables `enable_seqscan`) et les jointures à boucles imbriquées (`enable_nestloop`), qui sont les deux plans les plus basiques, on

forcera le système à utiliser un plan différent. Si le système continue néanmoins à choisir une lecture séquentielle ou une jointure à boucles imbriquées, alors il y a probablement un problème plus fondamental qui empêche l'utilisation de l'index, par exemple que la condition ne correspond pas à l'index. (Les sections précédentes expliquent quelles sortes de requêtes peuvent utiliser quelles sortes d'index.)

- Si l'index est effectivement utilisé en forçant son utilisation, alors il y a deux possibilités: Soit le système a raison et l'utilisation de l'index est effectivement inappropriée, soit les coûts estimés des plans de requêtes ne reflètent pas la réalité. Il faut alors comparer la durée de la requête avec et sans index. La commande `EXPLAIN ANALYZE` peut être utile pour cela.
- S'il apparaît que les estimations de coûts sont fausses, il y a de nouveau deux possibilités. Le coût total est calculé à partir du coût par ligne de chaque nœud du plan, multiplié par l'estimation de sélectivité du nœud de plan. Le coût des nœuds de plan peut être optimisé avec des paramètres d'exécution (décrits dans Section 16.4). Une estimation de sélectivité inadaptée est due à des statistiques insuffisantes. Il est peut être possible de les améliorer en optimisant les paramètres de collecte de statistiques. Voir *ALTER TABLE*).

  Si vous n'arrivez pas à ajuster les coûts pour qu'ils représentent mieux la réalité, alors vous devrez forcer l'utilisation de l'index explicitement. Vous pouvez aussi, si vous le voulez, contacter les développeurs de PostgreSQL afin qu'ils examinent le problème.

---

# Chapitre 12. Contrôle d'accès simultané

Ce chapitre décrit le comportement du système de bases de données PostgreSQL lorsque deux sessions, ou plus, essaient d'accéder aux mêmes données au même moment. Le but dans cette situation est de permettre un accès efficace pour toutes les sessions tout en maintenant une intégrité stricte des données. Chaque développeur d'applications utilisant des bases de données devrait être familier avec les thèmes couverts dans ce chapitre.

## 12.1. Introduction

Contrairement aux systèmes de bases de données traditionnelles qui utilisent des verrous pour le contrôle de concurrence, PostgreSQL maintient la consistance des données à l'aide d'un modèle multi−versions (Multiversion Concurrency Control, MVCC). Ceci signifie que, lors d'une requête à la base de données, chaque transaction voit une image des données (une *version de la base de données*) telle qu'elles étaient quelque temps auparavant, quelque soit l'état actuel des données sous−jacentes. Ceci protège la transaction de données incohérentes, causées par les mises à jour effectuées par une (autre) transaction concurrente sur les mêmes lignes de données, fournissant ainsi une *isolation des transactions* pour chaque session de base de données.

Le principal avantage de l'utilisation du modèle MVCC pour le contrôle de concurrence, contrairement au verrouillage, est que dans les verrous acquis par MVCC pour récupérer (en lecture) des données aucun conflit n'intervient avec les verrous acquis pour écrire des données. Du coup, lire ne bloque jamais l'écriture et écrire ne bloque jamais la lecture.

Les capacités de verrouillage de tables ou de lignes sont aussi disponibles dans PostgreSQL pour les applications ne pouvant pas s'adapter facilement au comportement de MVCC. Néanmoins, un bon usage de MVCC fournira généralement de meilleures performances que les verrous.

## 12.2. Isolation des transactions

Le standard SQL définit quatre niveaux d'isolation de transaction pour empêcher trois phénomènes de se produire lors de transactions concurrentes. Ces phénomènes indésirables sont :

lecture sale
: Une transaction lit des données écrites par une transaction concurrente non validée.

lecture non reproductible
: Une transaction relit des données qu'elle a lu précédemment et trouve que les données ont été modifiées par une autre transaction (validée depuis la lecture initiale).

lecture fantôme
: Une transaction ré−exécute une requête renvoyant un ensemble de lignes satisfaisant une condition de recherche et trouve que l'ensemble des lignes satisfaisant la condition a changé du fait d'une autre transaction récemment validée.

Les quatre niveaux d'isolation de transaction et les comportements correspondants sont décrits dans le Tableau 12−1.

**Tableau 12−1. Niveaux d'isolation des transactions SQL**

| Niveau d'isolation | Lecture sale | Lecture non reproductible | Lecture fantôme |
|---|---|---|---|
| Uncommited Read (en français, << Lecture de données non validées >>) | Possible | Possible | Possible |
| Commited Read (en français, << Lecture de données validées >>) | Impossible | Possible | Possible |
| Repeatable Read (en français, << Lecture répétée >>) | Impossible | Impossible | Possible |
| Serializable (en français, << Sérialisable >>) | Impossible | Impossible | Impossible |

Dans PostgreSQL, vous pouvez demander un des quatre niveaux standards d'isolation de transaction. Mais, en interne, il existe seulement deux niveaux distincts d'isolation, qui correspondent aux niveaux Read Committed et Serializable. Lorsque vous sélectionnez le niveau Read Uncommitted, vous obtenez réellement Read Committed, et quand vous sélectionnez Repeatable Read, vous obtenez réellement Serializable, donc le niveau d'isolation actuel pourrait être plus strict que ce que vous sélectionnez. Ceci est permis par le standard SQL. Les quatre niveaux d'isolation définissent seulement quel phénomène ne doit pas survenir, ils ne définissent pas ce qui doit arriver. La raison pour laquelle PostgreSQL fournit seulement deux niveaux d'isolation est qu'il s'agit de la seule façon raisonnable de faire correspondre les niveaux d'isolation standards avec l'architecture de contrôle des accès simultanés multiversion. Le comportement des niveaux standards d'isolation est détaillé dans les sous−sections suivantes.

Pour initialiser le niveau d'isolation d'une transaction, utilisez la commande SET TRANSACTION.

## 12.2.1. Niveau d'isolation Read committed (lecture des seules données validées)

*Read Commited* est le niveau d'isolation par défaut de PostgreSQL. Lorsqu'une transaction fonctionne avec ce niveau d'isolation, une requête `SELECT` voit uniquement les données validées avant que la requête soit lancée ; elle ne voit jamais, lors de l'exécution de la requête, ni les données non validées ni les modifications validées par les transactions concurrentes (néanmoins, le `SELECT` voit les effets des précédentes mises à jour dans sa propre transaction, et ce même si elles ne sont pas validées). Dans les faits, une requête `SELECT` voit une image de la base de données à l'instant où la requête a été lancée. Notez que des commandes `SELECT` successives peuvent voir des données différentes, y compris si elles font partie de la même transaction, si d'autres transactions valident des modifications de données pendant l'exécution du premier `SELECT`.

Les commandes `UPDATE`, `DELETE` et `SELECT FOR UPDATE` se comportent de la même façon que `SELECT` en ce qui concerne la recherche des lignes cibles : elles ne trouveront que les lignes cibles qui ont été validées avant le début de la commande. Néanmoins, une telle ligne cible pourrait avoir déjà été mise à jour (ou supprimée ou marquée pour mise à jour) par une autre transaction concurrente au moment où elle est découverte. Dans ce cas, le processus de mise à jour attendra que la première transaction soit validée ou annulée (si elle est toujours en cours). Si la première mise à jour est annulée, alors ses effets sont niés et le deuxième processus peut exécuter la mise à jour des lignes originellement trouvées. Si la première mise à jour est validée, la deuxième mise à jour ignorera la ligne si la première mise à jour l'a supprimée, sinon elle essaiera d'appliquer son opération à la version mise à jour de la ligne. La condition de recherche de la commande (la clause `WHERE`) est ré−évaluée pour savoir si la version mise à jour de la ligne correspond toujours à la condition de recherche. Dans ce cas, la deuxième mise à jour continue son opération, en commençant par la version mise à jour de la ligne.

À cause de la règle ci−dessus, une commande de mise à jour a la possibilité de voir une image non cohérente : elle peut voir les effets de commandes de mises à jour concurrentes affectant les mêmes lignes que celles qu'elle essaie de mettre à jour mais elle ne voit pas les effets de ces commandes sur les autres lignes de la base de données. Ce comportement rend le mode de lecture validée non convenable pour les commandes qui impliquent des conditions de recherche complexes. Néanmoins, il est correct pour les cas simples. Par exemple, considérons la mise à jour de balances de banque avec des transactions comme

```
BEGIN;
UPDATE comptes SET balance = balance + 100.00 WHERE no_compte = 12345;
UPDATE comptes SET balance = balance - 100.00 WHERE no_compte = 7534;
COMMIT;
```

Si deux transactions comme celle−ci essaient de modifier en même temps la balance du compte 12345, nous voulons clairement que la deuxième transaction commence à partir de la version mise à jour de la ligne du compte. Comme chaque commande n'affecte qu'une ligne prédéterminée, la laisser voir la version mise à jour de la ligne ne crée pas de soucis de cohérence.

Comme dans le mode de lecture validée, chaque nouvelle commande commence avec une nouvelle image incluant toutes les transactions validées jusque là, les commandes suivantes dans la transaction verront les effets de la transaction concurrente validée dans tous les cas. La problématique ici est de savoir si à l'intérieur d'une *même* commande, nous avons ou non une vision totalement cohérente de la base de données.

L'isolation partielle de la transaction fournie par le mode de lecture validée est adéquate pour de nombreuses applications et ce mode est rapide et simple à utiliser. Néanmoins, pour les applications réalisant des requêtes et mises à jour complexes, il pourrait être nécessaire de garantir une vue plus cohérente de la base de données que ce que fournit le mode Read Commited.

## 12.2.2. Niveau d'isolation sérialisable

Le niveau *sérialisable* est le niveau d'isolation de transaction le plus strict. Ce niveau émule l'exécution de la transaction en série, comme si les transactions avaient été exécutées l'une après l'autre, en série plutôt que parallèlement. Néanmoins, les applications utilisant ce niveau doivent être préparées à tenter de nouveau les transactions suite aux échecs de la sérialisation.

Quand une transaction est dans le niveau sérialisable, une requête `SELECT` voit seulement les données validées avant le début de la transaction ; elle ne voit jamais les données non validées et les modifications validées lors de l'exécution de la transaction par des transactions concurrentes (néanmoins, le `SELECT` voit bien les effets des mises à jour précédentes exécutées à l'intérieur de sa propre transaction même si elles ne sont pas encore validées). C'est différent du niveau Read Commited dans la mesure où `SELECT` voit une image du début de la transaction et non pas du début de la requête en cours à l'intérieur de la transaction. Du coup, les commandes `SELECT` successives à l'intérieur d'une même transaction voit toujours les mêmes données.

Les commandes `UPDATE`, `DELETE` et `SELECT FOR UPDATE` se comportent de la même façon que `SELECT` en ce qui concerne la recherche de lignes cibles : elles trouveront seulement les lignes cibles qui ont été validées avant le début de la transaction. Néanmoins, une telle ligne cible pourrait avoir été mise à jour (ou supprimée ou marquée pour mise à jour) par une autre transaction concurrente au moment où elle est utilisée. Dans ce cas, la transaction sérialisable attendra que la première transaction de mise à jour soit validée ou annulée (si celle−ci est toujours en cours). Si la première mise à jour est annulée, les effets sont inversés et la transaction sérialisable peut continuer avec la mise à jour de la ligne trouvée à l'origine. Mais si le processus

de mise à jour est validé (et que la ligne est mise à jour ou supprimée, pas simplement sélectionnée en vue d'une mise à jour), alors la transaction sérialisable sera annulée avec le message

```
ERROR:  could not serialize access due to concurrent update
```

parce qu'une transaction sérialisable ne peut pas modifier les lignes changées par d'autres transactions après que la transaction sérialisable ait commencé.

Quand l'application reçoit ce message d'erreurs, elle devrait annuler la transaction actuelle et ré−essayer la transaction complète. La seconde fois, la transaction voit les modifications déjà validées comme faisant partie de sa vue initiale de la base de données, donc il n'y a pas de conflit logique en utilisant la nouvelle version de la ligne comme point de départ pour la mise à jour de la nouvelle transaction.

Notez que seules les transactions de modifications ont besoin d'être tentées de nouveau ; les transactions en lecture seule n'auront jamais de conflits de sérialisation.

Le mode sérialisable fournit une garantie rigoureuse que chaque transaction accède à une vue totalement cohérente de la base de données. Néanmoins, l'application doit être prête à tenter de nouvelles transactions lorsque des mises à jour concurrentes rendent impossibles de soutenir l'illusion d'une exécution en série. Comme le coût de re−lancement de transactions complexes pourrait être significatif, ce mode est seulement recommandé lors de mise à jour contenant une logique suffisamment complexe pour donner de mauvaises réponses dans le mode de lecture validée. Plus communément, le mode sérialisable est nécessaire quand une transaction exécute plusieurs commandes successives qui doivent avoir des vues identiques de la base de données.

---

#### 12.2.2.1. Isolation sérialisable contre vraie sérialisation

La signification intuitive (et la définition mathématique) de l'exécution << sérialisable >> est que toute paire de transactions concurrentes validées avec succès apparaîtra comme ayant été exécutée en série, l'une après l'autre — bien que celle survenant en premier n'est pas prévisible. Il est important de réaliser qu'interdire les comportements indésirables listés dans le Tableau 12−1 n'est pas suffisant pour garantir une vraie exécution en série et, en fait, le mode sérialisable de PostgreSQL *ne garantie pas une exécution en série dans ce sens*. Comme exemple, considérez une table `ma_table`, contenant initialement

```
 classe | valeur
--------+--------
      1 |     10
      1 |     20
      2 |    100
      2 |    200
```

Supposons que la transaction sérialisable A traite

```
SELECT SUM(valeur) FROM ma_table WHERE classe = 1;
```

puis insère le résultat (30) comme `valeur` dans une nouvelle ligne avec `classe` = 2. En concurrence, la transaction serialisable B traite

```
SELECT SUM(valeur) FROM ma_table WHERE classe = 2;
```

et obtient le résultat 300, qu'il insère dans une nouvelle ligne avec `classe` = 1. Donc, les deux transactions valident. Aucun des comportements indiqués comme non désirables n'est survenu, pourtant nous avons un résultat qui n'aurait pû arriver dans tout ordre sériel. Si A a été exécuté avant B, B aurait trouvé la somme 330, et non pas 300. De façon similaire, l'autre ordre aurait eu comme résultat une somme différente pour le calcul par A.

Pour garantir une vraie sérialisation mathématique, il est nécessaire que le système de bases de données force le *verrouillage du prédicat*, ce qui signifie qu'une transaction ne peut pas insérer ou modifier une ligne qui aurait correspondue à la condition `WHERE` d'une requête dans une autre transaction concurrente. Par exemple, une fois que la transaction A a exécuté la requête `SELECT ... WHERE classe = 1`, un système à verrouillage de prédicat aurait interdit l'insertion de toute ligne de classe 1 par la transaction B jusqu'à la validation de la transaction A. [6] Un tel système de verrous est complexe à implémenter et extrêmement coûteux à l'exécution car toutes les sessions doivent être conscientes des détails de chaque requête exécutée par chaque transaction concurrente. Et cette grande dépense est pratiquement complètement perdue car, en pratique, la plupart des applications ne posent pas ce genre de problèmes (l'exemple ci−dessus est assez petit et a peu de chance de représenter de vrais logiciels). Du coup, PostgreSQL n'implémente pas le verrouillage de prédicat et, autant que nous le sachions, aucun DBMS en production ne le gère.

Dans ces cas où la possibilité d'une exécution non sérialisable est un vrai hasard, les problèmes peuvent être prévenus par l'utilisation appropriée d'un verrou explicite. Les sections suivantes comprennent plus de discussion sur ce sujet.

## 12.3. Verrouillage explicite

PostgreSQL fournit de nombreux modes de verrous pour contrôler les accès concurrents aux données des tables. Ces modes peuvent être utilisés pour contrôler le verrouillage par l'application dans des situations où MVCC n'a pas le comportement désiré. De plus, la plupart des commandes PostgreSQL acquièrent automatiquement des verrous avec les modes appropriés pour s'assurer que les tables référencées ne sont pas supprimées ou modifiées de façon incompatible lorsque la commande s'exécute (par exemple, `ALTER TABLE` ne peut pas être exécuté en même temps que d'autres opérations sur la même table).

Pour examiner une liste des verrous actuels dans un serveur de base de données, utilisez la vue système `pg_locks` (Section 41.33). Pour plus d'informations sur la surveillance du statut du sous−système de gestion des verrous, référez−vous au Chapitre 23.

### 12.3.1. Verrous de niveau table

La liste ci−dessous affiche les modes de verrous disponibles et les contextes dans lesquels ils sont automatiquement utilisés par PostgreSQL. Vous pouvez aussi acquérir explicitement n'importe lequel de ces verrous avec la commande LOCK. Rappelez−vous que tous ces modes de verrous sont des verrous au niveau table, même si le nom contient le mot << row >> (NdT : ligne) ; les noms des modes de verrous sont historiques. Dans une certaine mesure, les noms reflètent l'utilisation typique de chaque mode de verrou — mais la sémantique est identique. La seule vraie différence entre un mode verrou et un autre est l'ensemble des modes verrous avec lesquels ils rentrent en conflit. Deux transactions ne peuvent pas conserver des verrous de modes en conflit sur la même table au même moment (néanmoins, une transaction n'entre jamais en conflit avec elle−même. Par exemple, elle pourrait acquérir un verrou `ACCESS EXCLUSIVE` et acquérir plus tard un verrou `ACCESS SHARE` sur la même table). Des modes de verrou sans conflit pourraient être détenus en même temps par plusieurs transactions. Notez, en particulier, que certains modes de

verrous sont en conflit avec eux−même (par exemple, un verrou ACCESS EXCLUSIVE ne peut pas être détenu par plus d'une transaction à la fois) alors que d'autres n'entrent pas en conflit avec eux−même (par exemple, un verrou ACCESS SHARE peut être détenu par plusieurs transactions). Une fois acquis, un verrou est conservé jusqu'à la fin de la transaction.

**Modes de verrous au niveau table**

ACCESS SHARE
: En conflit avec le mode verrou ACCESS EXCLUSIVE.

  Les commandes SELECT et ANALYZE acquièrent un verrou sur ce mode avec les tables référencées. En général, tout requête lisant seulement une table et ne la modifiant pas obtiendra ce mode de verrou.

ROW SHARE
: En conflit avec les modes de verrous EXCLUSIVE et ACCESS EXCLUSIVE.

  La commande SELECT FOR UPDATE acquiert un verrou de ce mode avec la table cible (en plus des verrous ACCESS SHARE des autres tables référencées mais pas sélectionnées FOR UPDATE).

ROW EXCLUSIVE
: En conflit avec les modes de verrous SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, et ACCESS EXCLUSIVE.

  Les commandes UPDATE, DELETE et INSERT acquièrent ce mode de verrou sur la table cible (en plus des verrous ACCESS SHARE sur toutes les autres tables référencées). En général, ce mode de verrouillage sera acquis par toute commande modifiant des données de la table.

SHARE UPDATE EXCLUSIVE
: En conflit avec les modes de verrous SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE et ACCESS EXCLUSIVE. Ce mode protège une table contre les modifications concurrentes de schéma et l'exécution de VACUUM.

  Acquis par VACUUM (sans FULL).

SHARE
: En conflit avec les modes de verrous ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE et ACCESS EXCLUSIVE. Ce mode protège une table contre les modifications de données concurrentes.

  Acquis par CREATE INDEX.

SHARE ROW EXCLUSIVE
: En conflit avec les modes de verrous ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE et ACCESS EXCLUSIVE.

  Ce mode de verrouillage n'est automatiquement acquis par aucune commande PostgreSQL.

EXCLUSIVE
: En conflit avec les modes de verrous ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE et ACCESS EXCLUSIVE. Ce mode autorise uniquement les verrous ACCESS SHARE concurrents, c'est−à−dire que seules les lectures à partir de la table peuvent être effectué en parallèle avec une transaction contenant ce mode de verrouillage.

  Ce mode de verrouillage n'est automatiquement acquis par aucune commande PostgreSQL.

ACCESS EXCLUSIVE

Entre en conflit avec tous les modes (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE et ACCESS EXCLUSIVE). Ce mode garantie que le détenteur est la seule transaction à accéder à la table de quelque façon que ce soit.

Acquis par les commandes ALTER TABLE, DROP TABLE, REINDEX, CLUSTER et VACUUM FULL. C'est aussi le mode de verrou par défaut des instructions LOCK TABLE qui ne spécifient pas explicitement de mode de verrouillage.

**Astuce :** Seul un verrou ACCESS EXCLUSIVE bloque une instruction SELECT (sans FOR UPDATE).

## 12.3.2. Verrous au niveau ligne

En plus des verrous au niveau table, il existe des verrous au niveau ligne. Un verrou sur une ligne spécifique est automatiquement acquis lorsque la ligne est mise à jour (ou supprimée ou marquée pour mise à jour). Le verrou est détenu jusqu'à la fin de la transaction, que ce soit une validation ou une annulation. Les verrous au niveau ligne n'affectent pas les requêtes sur les données ; ils bloquent seulement les *modifieurs d'une même ligne*. Pour acquérir un verrou au niveau ligne sans modifier réellement la ligne, sélectionnez la ligne avec SELECT FOR UPDATE. Notez qu'une fois un verrou au niveau ligne acquis, la transaction pourrait mettre à jour la ligne plusieurs fois sans peur des conflits.

PostgreSQL ne garde en mémoire aucune information sur les lignes modifiées, il n'y a donc aucune limite sur le nombre de lignes verrouillées à un moment donné. Néanmoins, verrouiller une ligne peut causer une écriture disque ; ainsi, par exemple, SELECT FOR UPDATE modifiera les lignes sélectionnées pour les marquer et cela résultera en des écritures disques.

En plus des verrous tables et lignes, les verrous partagés/exclusifs sur les pages sont utilisés pour contrôler la lecture et l'écriture des pages de table dans l'ensemble de tampons partagées. Ces verrous sont immédiatement relâchés une fois la ligne récupérée ou mise à jour. Les développeurs d'application ne sont normalement pas concernés par les verrous au niveau page mais nous les mentionnons dans un souci d'exhaustivité.

## 12.3.3. Verrous morts (blocage)

L'utilisation de verrous explicites accroît le risque de *verrous morts* lorsque deux transactions (voire plus) détiennent chacune un verrou que l'autre convoite. Par exemple, si la transaction 1 a acquis un verrou exclusif sur la table A puis essaie d'acquérir un verrou exclusif sur la table B alors que la transaction 2 possède déjà un verrou exclusif sur la table B et souhaite maintenant un verrou exclusif sur la table A, alors aucun des deux ne peut continuer. PostgreSQL détecte automatiquement ces situations de blocage et les résout en annulant une des transactions impliquées, permettant ainsi à l'autre (aux autres) de se terminer (quelle est exactement la transaction annulée est difficile à prévoir mais vous ne devriez pas vous en préoccuper).

Notez que les verrous morts peuvent aussi se produire en résultat à des verrous de niveau ligne (et du coup, ils peuvent se produire même si le verrouillage explicite n'est pas utilisé). Considérons le cas où il existe deux transactions concurrentes modifiant une table. La première transaction exécute :

```
UPDATE comptes SET balance = balance + 100.00 WHERE no_compte = 11111;
```

Elle acquiert un verrou au niveau ligne sur la ligne spécifiée par le numéro de compte (no_compte). Ensuite, la deuxième transaction exécute :

```
UPDATE comptes SET balance = balance + 100.00 WHERE no_compte = 22222;
UPDATE comptes SET balance = balance - 100.00 WHERE no_compte = 11111;
```

La première instruction `UPDATE` acquiert avec succès un verrou au niveau ligne sur la ligne spécifiée, donc elle réussit à mettre à jour la ligne. Néanmoins, la deuxième instruction `UPDATE` trouve que la ligne qu'elle essaie de mettre à jour a déjà été verrouillée, alors elle attend la fin de la transaction ayant acquis le verrou. Maintenant, la première transaction exécute :

```
UPDATE comptes SET balance = balance - 100.00 WHERE no_compte = 22222;
```

La première transaction essaie d'acquérir un verrou au niveau ligne sur la ligne spécifiée mais ne le peut pas : la deuxième transaction détient déjà un verrou. Donc, elle attend la fin de la transaction deux. Du coup, la première transaction est bloquée par la deuxième et la deuxième est bloquée par la première : une condition de blocage, un verrou mort. PostgreSQL détectera cette situation et annulera une des transactions.

La meilleure défense contre les verrous morts est généralement de les éviter en s'assurant que toutes les applications utilisant une base de données acquièrent des verrous sur des objets multiples dans un ordre cohérent. Dans l'exemple ci−dessus, si les deux transactions avaient mis à jour les lignes dans le même ordre, aucun blocage n'aurait eu lieu. Vous devriez vous assurer que le premier verrou acquis sur un objet dans une transaction est dans le plus haut mode qui sera nécessaire pour cet objet. S'il n'est pas possible de vérifier ceci à l'avance, alors les blocages devront être gérés à l'exécution en ré−essayant les transactions annulées à cause de blocage.

Tant qu'aucune situation de blocage n'est détectée, une transaction cherchant soit un verrou de niveau table soit un verrou de niveau ligne attendra indéfiniment que les verrous en conflit soient relâchés. Ceci signifie que maintenir des transactions ouvertes sur une longue période de temps (par exemple en attendant une saisie de l'utilisateur) est parfois une mauvaise idée.

## 12.4. Vérification de cohérence des données au niveau de l'application

Comme les lecteurs ne verrouillent pas les données avec PostgreSQL, quelque soit le niveau d'isolation, les données lues par une transaction peuvent être surchargées par une autre transaction concurrente. En d'autres termes, si une ligne est renvoyée par `SELECT`, cela ne signifie pas que la ligne est toujours courante au moment où elle est renvoyée (c'est−à−dire, quelque temps après que la requête courante n'ait commencé). La ligne pourrait avoir été modifiée ou supprimée par une transaction validée après le lancement de cette requête. Même si la ligne est toujours valide << maintenant >>, elle pourrait être modifiée ou supprimée avant que la transaction courante ne soit validée ou annulée.

Une autre façon de penser à ceci est que chaque transaction voit une image du contenu de la base de données et les transactions concurrentes en cours d'exécution pourraient très bien voir des images différentes. Donc, le concept entier de << maintenant >> est quelque peu mal défini. Ceci n'est habituellement pas un gros problème si les applications clientes sont isolées les unes des autres, mais si les clients peuvent communiquer via des canaux externes à la base de données, de sérieuses confusions pourraient survenir.

Pour s'assurer de la validité actuelle d'une ligne et pour la protéger contre les modifications concurrentes, vous devez utiliser `SELECT FOR UPDATE` ou une instruction `LOCK TABLE` appropriée. (`SELECT FOR`

UPDATE verrouille uniquement les lignes retournées contre les mises à jours concurrentes alors que LOCK TABLE verrouille la table entière.) Ceci doit être pris en compte lors du portage d'applications vers PostgreSQL depuis d'autres environnements. (Avant la version 6.5, PostgreSQL utilisait des verrous de lecture et, du coup, cette considération est aussi valable pour les versions antérieures à la version 6.5 de PostgreSQL.)

Des vérifications globales de validité requièrent une réflexion supplémentaire avec MVCC. Par exemple, une application bancaire pourrait souhaiter vérifier que le somme de tous les crédits d'une table est équivalent à la somme des débits d'une autre table lorsque les deux tables sont activement mises à jour. Comparer les résultats de deux commandes SELECT sum(...) successives ne fonctionnera pas correctement avec le mode de lecture validée car la deuxième requête a des chances d'inclure les résultats de transactions non comptabilisées dans la première. Faire les deux sommes dans une seule transaction sérialisée donnera une image plus précise des effets des transactions validées avant le début de la transaction sérialisable — mais vous pourriez vous demander si la réponse est toujours juste au moment où elle est fournie. Si la transaction sérialisable a elle−même appliqué des modifications avant de tenter la vérification de cohérence, l'utilité de la vérification devient bien plus discutable car, maintenant, elle inclut des modifications apparues après le début de la transaction mais pas toutes. Dans de tels cas, une personne attentionnée pourrait souhaiter verrouiller toutes les tables nécessaires à la vérification pour obtenir une image indiscutable de la réalité. Un verrou de mode SHARE (ou plus important) garantit qu'il n'y a aucune modification non validée dans la table verrouillée, autres que celles de la transaction.

Il est à noter que si vous voulez vous reposer sur les verrous explicites pour éviter les changements concurrents, vous devriez utiliser le mode Read Commited, ou alors, dans le mode sérialisable, être attentif à l'obtention des verrous avant d'effectuer des requêtes. Un verrou obtenu par une transaction sérialisable garantie qu'aucune autre transaction modifiant la table n'est en cours d'exécution mais si l'image vue par la transaction est antérieure à l'obtention du verrou, elle pourrait être antérieure aux quelques modifications maintenant validées dans la table. Une image de la transaction sérialisable est généralement gelée au début de la première requête ou de la première commande de modification de données (SELECT, INSERT, UPDATE ou DELETE). Du coup, il est possible d'obtenir des verrous de façon explicite avant que l'image ne soit gelée.

## 12.5. Verrouillage et index

Bien que PostgreSQL fournisse des accès non bloquant en lecture/écriture aux données de la table, un accès non bloquant en lecture/écriture n'est pas fourni pour chaque méthode d'accès aux index implémentée dans PostgreSQL. Les différents types d'index sont gérés ainsi :

Index B−tree
: Les verrous partagés/exclusifs de court terme au niveau page sont utilisés pour les accès en lecture/écriture. Les verrous sont immédiatement relâchés après le parcours ou l'insertion de chaque ligne d'index. Les index B−tree fournissent le plus haut niveau de concurrence sans conditions de blocage.

Index GiST et R−tree
: Les verrous partagés/exclusifs de niveau index sont utilisés pour les accès en lecture/écriture. Les verrous sont relâchés à la fin de la commande.

Index hachés
: Les verrous partagés/exclusifs au niveau hash−bucket sont utilisés pour des accès en lecture/écriture. Les verrous sont relâchés après la fin des traitements sur le bucket. Les verrous au niveau bucket fournissent un meilleur accès concurrent que les verrous au niveau index mais sont sensibles aux blocages car les verrous sont détenus plus longtemps que pour une opération sur un index.

En bref, les index B−tree offrent la meilleure performance pour les applications concurrentes ; comme ils ont aussi plus de fonctionnalités que les index hachés, ils constituent le type d'index recommandé pour les applications concurrentes nécessitant des index sur des données scalaires. Pour les données non scalaires, les index B−trees ne peuvent évidemment pas être utilisés ; dans cette situation, les développeurs d'applications doivent être conscients des faibles performances de concurrence des index GiST et R−tree.

# Chapitre 13. Conseils sur les performances

La performance des requêtes peut être affectée par beaucoup d'éléments. Certains peuvent être manipulés par l'utilisateur, d'autres sont fondamentaux au concept sous−jacent du système. Ce chapitre fournit des conseils sur la compréhension et sur la configuration fine des performances de PostgreSQL.

## 13.1. Utiliser `EXPLAIN`

PostgreSQL réalise un *plan de requête* pour chaque requête qu'il reçoit. Choisir le bon plan pour correspondre à la structure de la requête et aux propriétés des données est absolument critique pour de bonnes performances. Vous pouvez utiliser la commande *EXPLAIN* pour voir quel plan de requête le système crée pour une requête particulière. La lecture du plan est un art qui mérite un tutoriel complet, ce que vous n'aurez pas là ; ici ne se trouvent que des informations de base.

Les nombres actuellement donnés par `EXPLAIN` sont :

- Le coût estimé du lancement (temps passé avant que l'affichage de la sortie ne commence, c'est−à−dire pour faire le tri dans un nœud de tri.)
- Coût total estimé (si toutes les lignes doivent être récupérées, ce qui pourrait ne pas être le cas : une requête avec une clause `LIMIT` ne paiera pas le coût total par exemple.)
- Nombre de lignes estimé en sortie par ce nœud de plan (encore une fois, seulement si exécuté jusqu'au bout)
- Largeur moyenne estimée (en octets) des lignes en sortie par ce nœud de plan

Les coûts sont mesurés en unités de récupération de page disque. (Les estimations de l'effort CPU sont converties en unités de page disque en utilisant quelques facteurs assez arbitraires. Si vous voulez expérimenter avec ces facteurs, voir la liste des paramètres de configuration en exécution dans Section 16.4.5.2.)

Il est important de noter que le coût d'un nœud de haut niveau inclut le coût de tous les nœuds fils. Il est aussi important de réaliser que le coût reflète seulement les éléments d'importance pour le planificateur/optimiseur. En particulier, le coût ne considère pas le temps dépensé dans la transmission des lignes de résultat à l'interface, qui pourrait être un facteur dominant dans le temps réellement passé ; mais le planificateur l'ignore parce qu'il ne peut pas le changer en modifiant le plan. (Chaque plan correct sortira le même ensemble de lignes.)

La sortie des lignes est un peu difficile car il ne s'agit *pas* du nombre de lignes traitées/parcourues par la requête, c'est habituellement moins, reflétant la sélectivité estimée des conditions de la clause `WHERE` qui sont appliquées à ce nœud. Idéalement, les estimations des lignes de haut niveau sera une approximation des nombres de lignes déjà renvoyés, mis à jour, supprimés par la requête.

Voici quelques exemples (utilisant la base de données des tests de régression après un `VACUUM ANALYZE` et les sources de développement de la 7.3) :

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
```

```
 Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

C'est aussi direct que ce que nous obtenons. Si vous faîtes :

```
SELECT * FROM pg_class WHERE relname = 'tenk1';
```

vous trouverez que `tenk1` a 233 pages disque et 10000 lignes. Donc, le coût est estimé à 233 lectures de page, dont le coût individuel est estimé à 1,0, plus 10000 * cpu_tuple_cost qui vaut actuellement 0,01 (essayez `SHOW cpu_tuple_cost`).

Maintenant, modifions la requête pour ajouter une condition `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..358.00 rows=1033 width=148)
   Filter: (unique1 < 1000)
```

L'estimation des lignes en sortie a baissé à cause de la clause `WHERE`. Néanmoins, le parcours devra toujours visiter les 10000 lignes, donc le coût n'a pas baissé ; en fait, il a un peu augmenté pour refléter le temps CPU supplémentaire dépensé pour vérifier la condition `WHERE`.

Le nombre réel de lignes que cette requête sélectionnera est 1000 mais l'estimation est approximative. Si vous tentez de dupliquer cette expérience, vous obtiendrez probablement une estimation légèrement différente ; de plus, elle changera après chaque commande `ANALYZE` parce que les statistiques produites par `ANALYZE` sont prises à partir d'un extrait au hasard de la table.

Modifiez la requête pour restreindre encore plus la condition :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..179.33 rows=49 width=148)
   Index Cond: (unique1 < 50)
```

et vous verrez que si nous faisons une condition `WHERE` assez sélective, le planificateur décidera éventuellement qu'un parcours d'index est moins cher qu'un parcours séquentiel. Ce plan ne visitera que 50 lignes grâce à l'index, donc il gagnera malgré le fait que chaque récupération individuelle est plus chère que la lecture séquentielle d'une page de disque complète.

Ajoutez une autre condition à la clause `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50 AND stringu1 = 'xxx';

                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..179.45 rows=1 width=148)
   Index Cond: (unique1 < 50)
   Filter: (stringu1 = 'xxx'::name)
```

La condition ajoutée `stringu1 = 'xxx'` réduit l'estimation du nombre de lignes en sortie mais pas le coût car nous devons toujours visiter le même ensemble de lignes. Notez que la clause `stringu1` ne peut pas être appliqué à une condition d'index (car cet index est seulement sur la colonne `unique1`). À la place, il est

appliqué comme un filtre sur les lignes récupérées par l'index. Du coup, le coût a un peu augmenté pour refléter cette vérification supplémentaire.

Maintenant, essayons de joindre deux tables, en utilisant les colonnes dont nous avons discuté :

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..327.02 rows=49 width=296)
   ->  Index Scan using tenk1_unique1 on tenk1 t1
                                       (cost=0.00..179.33 rows=49 width=148)
         Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2
                                       (cost=0.00..3.01 rows=1 width=148)
         Index Cond: ("outer".unique2 = t2.unique2)
```

Dans cette jointure en boucle imbriquée, le parcours externe utilise le même parcours d'index que celui utilisé dans l'avant–dernier exemple et donc son coût et le nombre de lignes sont les mêmes parce que nous appliquons la clause `WHERE unique1 < 50` à ce nœud. La clause `t1.unique2 = t2.unique2` n'a pas encore d'intérêt donc elle n'affecte pas le nombre de lignes du parcours externe. Pour le parcours interne, la valeur `unique2` de la ligne courante du parcours externe est connectée dans le parcours d'index interne pour produire une condition d'index identique à `t2.unique2 = constante`. Donc, nous obtenons le même plan de parcours interne et les coûts que nous obtenons de, disons, `EXPLAIN SELECT * FROM tenk2 WHERE unique2 = 42`. Les coûts du nœud correspondant à la boucle sont ensuite initialisés sur la base du coût du parcours externe, avec une répétition du parcours interne pour chaque ligne externe (ici, 49 * 3.01), plus un petit temps CPU pour traiter la jointure.

Dans cet exemple, le nombre de lignes en sortie de la jointure est identique aux nombres de lignes des deux parcours mais ce n'est pas vrai en règle générale car vous pouvez avoir des clauses `WHERE` mentionnant les deux tables et qui, donc, peuvent seulement être appliquées au point de jointure, non pas aux parcours d'index. Par exemple, si nous avions ajouté `WHERE ... AND t1.hundred < t2.hundred`, cela aurait diminué le nombre de lignes en sortie du nœud de jointure mais n'aurait pas changé les parcours d'index.

Une façon de rechercher des plans différents est de forcer le planificateur à oublier certaines stratégies qu'il aurait donné vainqueur en utilisant les options d'activation (enable)/désactivation (disable) pour chaque type de plan. (C'est un outil brut mais utile. Voir aussi Section 13.3.)

```
SET enable_nestloop = off;
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=179.45..563.06 rows=49 width=296)
   Hash Cond: ("outer".unique2 = "inner".unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..333.00 rows=10000 width=148)
   ->  Hash  (cost=179.33..179.33 rows=49 width=148)
         ->  Index Scan using tenk1_unique1 on tenk1 t1
                                     (cost=0.00..179.33 rows=49 width=148)
               Index Cond: (unique1 < 50)
```

Ce plan propose d'extraire les 50 lignes intéressantes de `tenk1` en utilisant le même parcours d'index, de les placer dans une table de hachage en mémoire puis de faire un parcours séquentiel de `tenk2`, en cherchant dans la table de hachage des correspondances possibles de la ligne `t1.unique2 = t2.unique2` at each `tenk2`. Le coût pour lire `tenk1` et pour initialiser la table de hachage correspond au coût de lancement

complet pour la jointure hachée car nous n'obtiendrons pas de lignes jusqu'à avoir lu `tenk2`. Le temps total estimé pour la jointure inclut aussi une charge importante du temps CPU pour requêter la table de hachage 10000 fois. Néanmoins, notez que nous ne chargeons *pas* 10000 fois 179,33 ; la configuration de la table de hachage n'est exécutée qu'une fois dans ce type de plan.

Il est possible de vérifier la précision des coûts estimés par le planificateur en utilisant `EXPLAIN ANALYZE`. Cette commande exécute réellement la requête puis affiche le vrai temps d'exécution accumulé par chaque nœud du plan, avec les mêmes coûts estimés que ceux affichés par un simple `EXPLAIN`. Par exemple, nous pourrions obtenir un résultat comme celui–ci :

```
EXPLAIN ANALYZE SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..327.02 rows=49 width=296)
                                (actual time=1.181..29.822 rows=50 loops=1)
   ->  Index Scan using tenk1_unique1 on tenk1 t1
                 (cost=0.00..179.33 rows=49 width=148)
                                (actual time=0.630..8.917 rows=50 loops=1)
         Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2
                 (cost=0.00..3.01 rows=1 width=148)
                                (actual time=0.295..0.324 rows=1 loops=50)
         Index Cond: ("outer".unique2 = t2.unique2)
 Total runtime: 31.604 ms
```

Notez que les valeurs << temps réel >> sont en millisecondes alors que les estimations de << coût >> sont exprimées dans des unités arbitraires de récupération de page disque ; donc il y a peu de chances qu'elles correspondent. L'important est de faire attention aux ratios.

Dans certains plans de requête, il est possible qu'un nœud de sous–plan soit exécuté plus d'une fois. Par exemple, le parcours d'index interne est exécuté une fois par ligne externe dans le plan de boucle imbriquée ci–dessus. Dans de tels cas, la valeur << loops >> renvoie le nombre total d'exécution du nœud, et le temps réel et les valeurs des lignes affichées sont une moyenne par exécution. Ceci est fait pour que les nombres soient comparables avec la façon dont les estimations de coûts sont affichées. Multipliez par la valeur de << loops >> pour obtenir le temps total réellement passé dans le nœud.

Le `Total runtime` (temps total d'exécution) affiché par `EXPLAIN ANALYZE` inclut les temps de lancement et d'arrêt de l'exécuteur ainsi que le temps passé lors du traitement des lignes de résultat. Il n'inclut pas le temps passé pour l'analyse, la réécriture ou la planification. Pour une requête `SELECT`, le temps total d'exécution sera juste un peu plus important que le temps total indiqué par le noœud du plan de haut niveau. Pour les commandes `INSERT`, `UPDATE` et `DELETE`, le temps total d'exécution pourrait être considérablement plus important parce qu'il inclut le temps passé au traitement des lignes de résultat. Dans ces commandes, le temps pour le nœud du plan principal est essentiellement le temps passé à calculer les nouvelles lignes et/ou l'emplacement des anciennes mais il n'inclut pas le temps passé à faire des modifications.

Il est bon de noter que les résultats de `EXPLAIN` ne devraient pas être extrapolés pour des situations autres que celles de vos tests en cours ; par exemple, les résultats sur une petite table ne peuvent être appliqués à des tables bien plus importantes. Les estimations de coût du planificateur ne sont pas linéaires et, du coup, il pourrait bien choisir un plan différent pour une table plus petite ou plus grande. Un exemple extrême est celui d'une table occupant une page disque. Vous obtiendrez pratiquement toujours un parcours séquentiel que des index soient disponibles ou non. Le planificateur réalise que cela va nécessiter la lecture d'une seule page

disque pour traiter la table dans ce cas, il n'y a donc pas d'intérêt à étendre des lectures de pages supplémentaires pour un index.

# 13.2. Statistiques utilisées par le planificateur

Comme nous avons vu dans la section précédente, le planificateur de requêtes a besoin d'estimer le nombre de lignes récupérées par une requête pour faire les bons choix dans ses plans de requêtes. Cette section fournit un aperçu rapide sur les statistiques que le système utilise pour ces estimations.

Un composant des statistiques est le nombre total d'entrées dans chaque table et index, ainsi que le nombre de blocs disque occupés par chaque table et index. Cette information est conservée dans la table `pg_class` sur les colonnes `reltuples` et `relpages`. Nous pouvons la regarder avec des requêtes comme celle–ci :

```
SELECT relname, relkind, reltuples, relpages FROM pg_class WHERE relname LIKE 'tenk1%';

    relname    | relkind | reltuples | relpages
---------------+---------+-----------+----------
 tenk1         | r       |     10000 |      233
 tenk1_hundred | i       |     10000 |       30
 tenk1_unique1 | i       |     10000 |       30
 tenk1_unique2 | i       |     10000 |       30
(4 rows)
```

Ici, nous pouvons voir que `tenk1` contient 10000 lignes, comme pour ses index, mais que les index sont bien plus petits que la table (ce qui n'est pas surprenant).

Pour des raisons d'efficacité, `reltuples` et `relpages` ne sont pas mis à jour en temps réel, et du coup, elles contiennent habituellement des valeurs un peu obsolètes. Elles sont mises à jour par les commandes `VACUUM`, `ANALYZE` et quelques commandes DDL comme `CREATE INDEX`. Un `ANALYZE` seul, donc ne faisant pas partie d'un `VACUUM`, génère une valeur approximative de `reltuples` car il ne lit pas chaque ligne de la table. Le planificateur mettra à l'échelle les valeurs qu'il aura trouver dans `pg_class` pour correspondre à la taille physique de la table, obtenant ainsi une approximation plus proche de la réalité.

La plupart des requêtes ne récupère qu'une fraction des lignes dans une table à cause de clauses `WHERE` qui restreignent les lignes à examiner. Du coup, le planificateur a besoin d'une estimation de la *sélectivité* des clauses `WHERE`, c'est–à–dire la fraction des lignes qui correspondent à chaque condition de la clause `WHERE`. L'information utilisée pour cette tâche est stockée dans le catalogue système `pg_statistic`. Les entrées de `pg_statistic` sont mises à jour par les commandes `ANALYZE` et `VACUUM ANALYZE` et sont toujours approximatives même si elles ont été mises à jour récemment.

Plutôt que de regarder directement dans `pg_statistic`, il est mieux de visualiser sa vue `pg_stats` lors de l'examen manuel des statistiques. `pg_stats` est conçu pour être plus facilement lisible. De plus, `pg_stats` est lisible par tous alors que `pg_statistic` n'est lisible que par un superutilisateur. (Ceci empêche les utilisateurs non privilégiés d'apprendre certains choses sur le contenu des tables d'autres personnes à partir des statistiques. La vue `pg_stats` est restreinte pour afficher seulement les lignes des tables lisibles par l'utilisateur courant.) Par exemple, nous pourrions lancer :

```
SELECT attname, n_distinct, most_common_vals FROM pg_stats WHERE tablename = 'road';
```

```
 attname | n_distinct |
---------+------------+-------------------------------------------------------------------------------
 name    |  -0.467008 | {"I- 580                        Ramp","I- 880                        Ramp
 thepath |         20 | {"[(-122.089,37.71),(-122.0886,37.711)]"}
(2 rows)
```

`pg_stats` est décrit en détail dans Section 41.36.

Le nombre d'informations stockées dans `pg_statistic`, en particulier le nombre maximum d'éléments dans les tableaux `most_common_vals` et `histogram_bounds` pour chaque colonne, peut être initialisé sur une base colonne–par–colonne en utilisant la commande `ALTER TABLE SET STATISTICS` ou globalement en initialisant la variable de configuration default_statistics_target. La limite par défaut est actuellement de dix entrées. Augmenter la limite pourrait permettre des estimations plus précises du planificateur, en particulier pour les colonnes ayant des distributions de données irrégulières, au prix d'un plus grand espace consommé dans `pg_statistic` et en un temps plus long pour calculer les estimations. Par contre, une limite plus basse pourrait être appropriée pour les colonnes à distributions de données simples.

## 13.3. Contrôler le planificateur avec des clauses `JOIN` explicites

Il est possible de contrôler le planificateur de requêtes à un certain point en utilisant une syntaxe `JOIN` explicite. Pour voir en quoi ceci est important, nous allons besoin de quelques connaissances.

Dans une simple requête de jointure, telle que :

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

le planificateur est libre de joindre les tables données dans n'importe quel ordre. Par exemple, il pourrait générer un plan de requête qui joint A à B en utilisant la condition `WHERE a.id = b.id`, puis joint C à cette nouvelle table jointe en utilisant l'autre condition `WHERE`. Ou il pourrait joindre B à C, puis A au résultat de cette jointure précédente. Ou il pourrait joindre A à C puis les joindre avec B mais cela pourrait ne pas être efficace car le produit cartésien complet de A et C devra être formé alors qu'il n'y a pas de condition applicable dans la clause `WHERE` pour permettre une optimisation de la jointure. (Toutes les jointures dans l'exécuteur PostgreSQL arrivent entre deux tables en entrées donc il est nécessaire de construire le résultat de l'une ou de l'autre de ces façons.) Le point important est que ces différentes possibilités de jointures donnent des résultats sémantiquement équivalents mais pourraient avoir des coûts d'exécution grandement différents. Du coup, le planificateur va toutes les explorer pour trouver le plan de requête le plus efficace.

Quand une requête implique seulement deux ou trois tables, il y a peu d'ordres de jointures à préparer. Mais le nombre d'ordres de jointures possibles grandit de façon exponentielle au fur et à mesure que le nombre de tables augmente. Au delà de dix tables en entrée, il n'est plus possible de faire une recherche exhaustive de toutes les possibilités et même la planification de six ou sept tables pourrait prendre beaucoup de temps. Quand il y a trop de tables en entrée, le planificateur PostgreSQL basculera d'une recherche exhaustive à une recherche *génétique* probabiliste via un nombre limité de possibilités. (La limite de bascule est initialisée par le paramètre en exécution geqo_threshold.) La recherche génétique prend moins de temps mais elle ne trouvera pas nécessairement le meilleur plan possible.

Quand la requête implique des jointures externes, le planificateur est moins libre qu'il ne l'est lors de jointures internes. Par exemple, considérez

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Bien que les restrictions de cette requête semblent superficiellement similaires à l'exemple précédent, les sémantiques sont différentes car une ligne doit être émise pour chaque ligne de A qui n'a pas de ligne correspondante dans la jointure entre B et C. Du coup, le planificateur n'a pas de choix dans l'ordre de la jointure ici : il doit joindre B à C puis joindre A à ce résultat. Du coup, cette requête prend moins de temps à planifier que la requête précédente.

La syntaxe de jointure interne explicite (`INNER JOIN`, `CROSS JOIN` ou `JOIN`) est sémantiquement identique à lister les relations en entrées du `FROM`, donc il n'est pas nécessaire de contenir l'ordre de la jointure. Mais il est possible d'instruire le planificateur de requêtes de PostgreSQL pour traiter les `JOIN` internes explicites comme s'ils contraignaient l'ordre de jointure. Par exemple, ces trois requêtes sont logiquement équivalentes :

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Mais si nous disons au planificateur d'honorer l'ordre des `JOIN`, la deuxième et la troisième prendront moins de temps à planifier que la première. Cet effet n'est pas inquiétant pour seulement trois tables mais cela pourrait bien nous aider avec beaucoup de tables.

Pour forcer le planificateur à suivre l'ordre `JOIN` pour les jointures internes, initialisez le paramètre en exécution join_collapse_limit à 1. (D'autres valeurs possibles sont discutées plus bas.)

Vous n'avez pas besoin de restreindre l'ordre de jointure pour diminuer le temps de recherche car il est bien d'utiliser les opérateurs `JOIN` dans les éléments d'une liste `FROM`. Par exemple, considérez

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

Avec `join_collapse_limit` = 1, ceci force le planificateur à joindre A à B avant de les joindre aux autre stables mais sans restreindre ses choix sinon. Dans cet exemple, le nombre d'ordres de jointures possibles est réduit par un facteur de 5.

Restreindre la recherche du planificateur de cette façon est une technique utile pour réduire les temps de planification et pour diriger le planificateur vers un bon plan de requêtes. Si le planificateur choisit un mauvais ordre de jointure par défaut, vous pouvez le forcer à choisir un meilleur ordre via la syntaxe `JOIN` — en supposant que vous connaissiez un meilleur ordre. Une expérimentation est recommandée.

Un problème très proche et affectant le temps de planification est le regroupement de sous−requêtes dans leur requêtes parent. Par exemple, considérez

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE quelquechose) AS ss
WHERE quelquechosedautre;
```

Cette pourrait survenir suite à l'utilisation d'une vue contenant une jointure ; la règle `SELECT` de la vue sera insérée à la place de la référence de la vue, demande une requête plutôt identique à celle ci−dessus. Normalement, le planificateur essaiera de regrouper la sous−requête avec son parent, donnant

```
SELECT * FROM x, y, a, b, c WHERE quelquechose AND quelquechosedautre;
```

Ceci résulte habituellement en un meilleur plan que de planifier séparément la sous−requête. (Par exemple, les conditions WHERE externes pourraient être telles que joindre X à A élimine en premier lieu un bon nombre de lignes de A, évitant ainsi le besoin de former la sortie complète de la sous−requête.) Mais en même temps, nous avons accru le temps de planification ; ici, nous avons une problème de jointure à cinq tables remplaçant un problème de deux jointures séparées à trois tables. À cause de l'augmentation exponentielle du nombre de possibilités, ceci fait une grande différence. Le planificateur essaie d'éviter de se retrouver coincé dans des problèmes de recherche de grosses jointures en ne regroupant pas une sous−requête sur plus de from_collapse_limit éléments sont la résultante de la requête parent. Vous pouvez comparer le temps de planification avec la qualité du plan en ajustant ce paramètre en exécution.

from_collapse_limit et join_collapse_limit sont nommés de façon similaire parce qu'ils font pratiquement la même chose : l'un d'eux contrôle le moment où le planificateur << aplatira >> les sous−requêtes et l'autre contrôle s'il y aura aplatissement des jointures internes explicites. Typiquement, vous initialiserez join_collapse_limit comme from_collapse_limit (de façon à ce que les jointures explicites et les sous−requêtes agissent de la même façon) ou vous initialiserez join_collapse_limit à 1 (si vous voulez contrôler l'ordre de jointure des jointures explicites). Mais vous pourriez les initialiser différemment si vous tentez de configurer finement la relation entre le temps de planification et le temps d'exécution.

# 13.4. Remplir une base de données

Vous pourriez avoir besoin d'insérer un grand nombre de données pour remplir une base de données au tout début. Cette section contient quelques suggestions pour réaliser cela de la façon la plus efficace.

## 13.4.1. Désactivez la validation automatique (autocommit)

Désactivez la validation automatique et faites une seule validation à la fin. (En SQL, ceci signifie de lancer BEGIN au début et COMMIT à la fin. Quelques bibliothèques client pourraient le faire derrière votre dos auquel cas vous devez vous assurer que la bibliothèque le fait quand vous le voulez.) Si vous permettez à chaque insertion d'être validée séparément, PostgreSQL fait un gros travail pour chaque ligne ajoutée. Un bénéfice supplémentaire de réaliser toutes insertions dans une transaction est que si l'insertion d'une ligne échoue alors les lignes insérées jusqu'à maintenant seront annulées. Vous ne serez donc pas bloqué avec des données partiellement chargées.

## 13.4.2. Utilisez COPY

Utilisez *COPY* pour charger toutes les lignes en une seule commande, plutôt que d'utiliser une série de commandes INSERT. La commande COPY est optimisée pour charger un grand nombre de lignes ; elle est moins flexible que INSERT mais introduit significativement moins de surcharge lors du chargement de grosses quantités de données. Comme COPY est une seule commande, il n'y a pas besoin de désactiver la validation automatique (autocommit) si vous utilisez cette méthode pour remplir une table.

Si vous ne pouvez pas utiliser COPY, utiliser *PREPARE* pourrait vous aider à créer une instruction préparée INSERT, puis utilisez EXECUTE autant de fois que nécessaire. Ceci évite certaines surcharges lors d'une analyse et d'une planification répétées de commandes INSERT.

Notez que charger un grand nombre de lignes en utilisant COPY est pratiquement toujours plus rapide que d'utiliser INSERT, même si PREPARE est utilisé lorsque de nombreuses insertions sont groupées en une seule transaction.

### 13.4.3. Supprimez les index

Si vous chargez une table tout juste créée, la façon la plus rapide est de créer la table, de charger en lot les données de cette table en utilisant `COPY`, puis de créer tous les index nécessaires pour la table. Créer un index sur des données déjà existantes est plus rapide que de mettre à jour de façon incrémentale à chaque ligne ajoutée.

Si vous augmentez une table existante, vous pouvez supprimer les index, charger la table, puis recréer l'index. Bien sûr, les performances de la base de données pour les autres utilisateurs pourraient être sévèrement affectées tout le temps où l'index sera manquant. Vous devez aussi y penser à deux fois avant de supprimer des index uniques car la vérification d'erreur apportée par la contrainte unique sera perdue tout le temps où l'index est manquant.

### 13.4.4. Augmentez `maintenance_work_mem`

Augmentez temporairement la variable maintenance_work_mem lors du chargement de grosses quantités de données peut amener une amélioration des performances. Ceci est dû au fait qu'un index B−tree est créé à partir de rien, le contenu déjà existant de la table a besoin d'être trié. Permettre au tri merge d'utiliser plus de mémoire signifie que moins de passes merge seront requises. Une grande valeur pour `maintenance_work_mem` pourrait aussi accélérer la validation des contraintes de clés étrangères.

### 13.4.5. Augmentez `checkpoint_segments`

Augmenter temporairement la variable de configuration checkpoint_segments peut aussi aider à un chargement rapide de grosses quantités de données. Ceci est dû au fait que charger une grosse quantité de données dans PostgreSQL peut causer la venue trop fréquentes de points de vérification (la fréquent de ces points de vérification est spécifiée par la variable de configuration `checkpoint_timeout`). Quand survient un point de vérification, toutes les pages modifiées sont écrites sur le disque. En augmentant `checkpoint_segments` temporairement lors du chargement des données, le nombre de points de vérification requis peut être diminué.

### 13.4.6. Lancez `ANALYZE` après

Quand vous avez changé significativement la distribution des données à l'intérieur d'une table, lancer *ANALYZE* est fortement recommandée. Ceci inclut le chargement de grosses quantités de données dans la table. Lancer `ANALYZE` (ou `VACUUM ANALYZE`) vous assure que le planificateur dispose de statistiques à jour sur la table. Sans statistiques ou avec des statistiques obsolètes, le planificateur pourrait prendre de mauvaises décisions lors de la planification de la requête, amenant des performances pauvres sur toutes les tables sans statistiques ou avec des statistiques inexactes.

# III. Administration du serveur

Cette partie couvre les thèmes qui sont intéressants pour un administrateur de bases de données PostgreSQL. Cela inclut l'installation du logiciel, la mise en place et la configuration du serveur, la gestion des utilisateurs et des bases de données, ainsi que les tâches de maintenance. Tous ceux qui font fonctionner un serveur PostgreSQL, pour un usage personnel, mais plus spécialement en production, devraient être familiers avec les thèmes couverts dans cette partie.

L'information dans cette partie est arrangée à peu près dans l'ordre dans lequel un nouvel utilisateur devrait la lire. Mais les chapitres sont indépendants et peuvent être lus individuellement, si vous le désirez. L'information dans cette partie est présentée dans un style narratif en unités thématiques. Les lecteurs qui cherchent une description complète d'une commande particulière devraient regarder dans la Partie VI.

Les tous premiers chapitres sont écrits de façon à ce qu'ils puissent être compris sans connaissances préalables, de sorte que les nouveaux utilisateurs qui ont besoin de mettre en route leur propre serveur puissent commencer leur exploration avec cette partie. Le reste de ce chapitre traite de l'optimisation (tuning) et de la gestion. Cette partie suppose que le lecteur soit familier avec l'utilisation générale du système de bases de données PostgreSQL. Les lecteurs sont encouragés à regarder la Partie I et la Partie II pour obtenir des informations supplémentaires.

**Table des matières**

# Chapitre 14. Procédure d'installation

Ce chapitre décrit l'installation de PostgreSQL à partir du code source (si vous installé une distribution préparée, tels qu'un paquetage RPM ou Debian, vous pouvez ignorer ce chapitre et lire à la place les instructions du mainteneur du paquetage).

## 14.1. Version courte

```
./configure
gmake
su
gmake install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

Le reste du chapitre est la version longue.

## 14.2. Prérequis

En général, les plateformes style unix modernes doivent être capables d'exécuter PostgreSQL. Les plateformes sur lesquelles des tests ont été effectuées sont listées dans la Section 14.7 ci−après. Dans le répertoire `doc` de la distribution, il y a plusieurs FAQ spécifiques à des plateformes particulières que vous pouvez consulter si vous avez des problèmes.

Les logiciels suivants sont nécessaires pour compiler PostgreSQL :

- GNU make est nécessaire ; les autres programmes make *ne* devraient pas fonctionner. GNU make est souvent installé sous le nom `gmake` ; ce document fera toujours référence à lui sous ce nom (sur certains système, GNU make est l'outil par défaut et est nommé `make`). Pour savoir quelle version vous utilisez, saisissez

  ```
  gmake --version
  ```

  Il est recommandé d'avoir une version postérieure à la version 3.76.1.
- Il est nécessaire d'avoir un compilateur C ISO/ANSI. Une version récente de GCC est recommandée mais PostgreSQL est connu pour être compilable avec de nombreux compilateurs de divers vendeurs.
- gzip est nécessaire pour décompresser l'archive.
- La bibliothèque GNU Readline sera utilisée par défaut (pour une édition facile des lignes et une recherche de l'historique des commandes). Si vous ne voulez pas l'utiliser, il vous faut spécifier `--without-readline` au moment d'exécuter la commande `configure` (sous NetBSD, la bibliothèque `libedit` est compatible Readline et est utilisée si le fichier `libreadline` n'est pas trouvé). Si vous utilisez une distribution Linux basée sur des paquetages, faites attention au fait que vous aurez besoin à la fois des paquetages `readline` et `readline-devel` s'ils sont séparés dans votre distribution.

- Des logiciels supplémentaires sont nécessaires pour construire PostgreSQL sur Windows. Vous pouvez construire PostgreSQL pour les versions NT de Windows (comme Windows XP et 2003) en utilisant MinGW ; voir `doc/FAQ_MINGW` pour les détails. Vous pouvez aussi construire PostgreSQL en utilisant Cygwin ; voir `doc/FAQ_CYGWIN`. Une construction basée sur Cygwin fonctionnera sur les anciennes versions de Windows mais, si vous avez le choix, nous vous recommandons l'approche de MinGW. Bien qu'il soit le seul ensemble d'outils recommandé pour une construction complète, il est possible de construire seulement la bibliothèque cliente C (libpq) et le terminal interactif (psql) en utilisant d'autres environnements de développement sous Windows. Pour des détails, voir le Chapitre 15.

Les paquetages suivants sont optionnels. Ils ne sont pas obligatoires pour une compilation par défaut mais le sont lorsque certaines options sont utilisées ainsi que c'est expliqué par la suite.

- Pour installer le langage de procédures PL/Perl, vous devez avoir une installation de Perl complète, comprenant la bibliothèque `libperl` et les fichiers d'en−tête. Comme PL/Perl est une bibliothèque partagée, la bibliothèque `libperl` doit aussi être partagée sur la plupart des plateformes. Ce qui n'est le cas que dans les versions récentes de Perl et, dans tous les cas, c'est le choix de ceux qui installent Perl.

  Si vous n'avez pas de bibliothèque partagée alors qu'il vous en faut une, un message tel que celui−ci apparaîtra durant la compilation pour vous en avertir :

  ```
  *** Cannot build PL/Perl because libperl is not a shared library.
  *** You might have to rebuild your Perl installation.  Refer to
  *** the documentation for details.
  ```

  (Si vous ne suivez pas la sortie écran, vous pourrez constater que la bibliothèque `plperl.so` de PL/Perl, ou similaire, n'est pas installée.) Si c'est le cas, il vous faudra recompiler et ré−installer Perl manuellement pour être capable de compiler PL/Perl. Lors de la phase de configuration de Perl, demandez que les bibliothèques soient partagées.
- Pour compiler le langage de procédures PL/Python, il faut que Python soit installé avec les fichiers d'en−tête et le module distutils. Le module distutils est inclus par défaut avec Python 1.6 et les versions suivantes ; les utilisateurs des versions précédentes de Python auront besoin de l'installer.

  Puisque PL/Python devra être une bibliothèque partagée, la bibliothèque `libpython` doit l'être aussi sur la plupart des plateformes. Ce n'est pas le cas pour une installation par défaut de Python. Si, après la compilation et l'installation, vous avez un fichier nommé `plpython.so` (des extensions différentes sont possibles), alors tout va pour le mieux. Sinon, vous devriez avoir vu un avertissement semblable à :

  ```
  *** Cannot build PL/Python because libpython is not a shared library.
  *** You might have to rebuild your Python installation.  Refer to
  *** the documentation for details.
  ```

  Ce qui signifie que vous devez recompiler Python (ou une partie) pour activer cette bibliothèque partagée.

  Si vous avez des problèmes, lancez le configure de Python 2.3 ou d'une version suivante en utilisant le commutateur `--enable-shared`. Sur certains systèmes d'exploitation, vous n'avez pas besoin de construire une bibliothèque partagée mais vous devrez convaincre le système de construction de PostgreSQL sur ce point. Consultez le fichier `Makefile` dans le répertoire `src/pl/plpython` pour des détails supplémentaires.

- Si vous voulez construire le langage de procédure PL/Tcl, vous avez besoin que Tcl soit installé.
- Pour activer le support de langage natif (NLS), qui permet d'afficher les messages dans une langue autre que l'anglais, vous avez besoin d'une implémentation de l'API Gettext. Certains systèmes d'exploitation l'ont intégré. (Par exemple, Linux, NetBSD, Solaris), pour les autres systèmes vous pouvez télécharger les paquetages nécessaires http://developer.postgresql.org/~petere/bsd-gettext. Si vous utilisez l'implémentation de Gettext des bibliothèques C GNU, vous devrez en plus utiliser le paquetage GNU Gettext pour certains utilitaires. Pour toutes les autres implémentations, vous n'en avez pas besoin.
- Kerberos, OpenSSL ou PAM si vous voulez une authentification ou un chiffrement en utilisant ces services.

Si vous compilez à partir d'une arborescence CVS au lieu d'utiliser un paquetage contenant les sources, ou si vous faites du développement, vous aurez aussi besoin des paquetages suivants :

- GNU Flex et Bison sont nécessaires pour compiler à partir d'une récupération du CVS ou si vous modifiez les fichiers de recherche et d'analyse. Si vous en avez besoin, vérifiez que vous avez Flex 2.5.4 ou postérieur et Bison 1.875 ou postérieur. D'autres programmes yacc peuvent parfois d'être utilisés, ce qui n'est pas recommandé vu les efforts supplémentaires que cela demande. D'autres programmes lex ne fonctionneront définitivement pas.

Si vous avez besoin de paquetages GNU, vous pourrez les trouver sur un site miroir de GNU (voir http://www.gnu.org/order/ftp.html pour en avoir la liste) ou sur ftp://ftp.gnu.org/gnu/.

Vérifiez aussi que vous avez assez d'espace disque de disponible. Il vous faudra 65 Mo pour l'espace de compilation et 15 Mo pour le répertoire d'installation. Une base de données vide en cluster nécessite 25 Mo, les fichiers de la base prenant cinq fois plus d'espace que des fichiers texte contenant les mêmes données. Si vous voulez faire des tests de régression, vous aurez besoin temporairement de 90 Mo supplémentaires. Utilisez la commande `df` pour vérifier l'espace disque.

## 14.3. Obtenir les sources

Les sources de PostgreSQL 8.0.5 peuvent être obtenues par ftp anonyme à ftp://ftp.postgresql.org/pub/source/v8.0.5/postgresql-8.0.5.tar.gz. Utilisez un site miroir si possible. Après avoir téléchargé le fichier, décompressez-le :

```
gunzip postgresql-8.0.5.tar.gz
tar xf postgresql-8.0.5.tar
```

Cela créera un répertoire `postgresql-8.0.5` contenant les sources de PostgreSQL dans le répertoire courant. Allez dans ce nouveau répertoire pour continuer la procédure d'installation.

## 14.4. Si vous effectuez une mise à jour

Le format de stockage interne des données a changé avec cette nouvelle version de PostgreSQL. Toutefois, si vous faites une mise à jour qui n'a pas un numéro de version de la forme << 8.0.x >>, vous devrez faire une sauvegarde et une restauration des données ainsi que c'est montré ici. Les instructions considèrent que votre installation existante est dans le répertoire `/usr/local/pgsql` et que la zone de données est dans `/usr/local/pgsql/data`. Remplacez les chemins de façon approprié.

1. Assurez−vous que vos données ne sont pas mises à jour pendant ou après la sauvegarde. Cela n'affecterait en rien l'intégrité de la sauvegarde mais les données modifiées ne seraient pas incluses. Si nécessaire, éditez le fichier `/usr/local/pgsql/data/pg_hba.conf` (ou équivalent) pour verrouiller tous les accès sauf les vôtres.
2. Pour effectuer une sauvegarde de votre base, saisissez :

   ```
   pg_dumpall > fichierdesortie
   ```

   Si vous souhaitez conserver les identifiants d'objets (lorsqu'ils sont utilisés comme clé étrangère), utilisez l'option `−o` lors de l'exécution de pg_dumpall.

   pg_dumpall ne sauvegarde pas les gros objets. Lisez la Section 22.1.4 si vous en avez besoin.

   Pour effectuer une sauvegarde, vous pouvez exécuter la commande pg_dumpall incluse dans la distribution de votre version actuelle. Cependant, pour de meilleurs résultats, essayez d'utiliser la commande `pg_dumpall` contenue dans PostgreSQL 8.0.5 puisqu'elle corrige les erreurs des versions précédentes. Ce conseil est absurde tant que vous n'avez pas encore installé la nouvelle version. Il devient valable si vous souhaitez installer la nouvelle version en parallèle. Dans ce cas, vous pouvez faire l'installation normalement et ne transférer les données qu'après, ce qui réduira le temps d'indisponibilité.
3. Si vous installez la nouvelle version à la place de l'ancienne, arrêtez l'ancien serveur, au plus tard avant d'installer les nouveaux fichiers :

   ```
   pg_ctl stop
   ```

   Sur les systèmes qui lancent PostgreSQL au démarrage, il y a probablement un fichier de démarrage qui peut faire la même chose. Par exemple, sur un système Red Hat Linux, la commande

   ```
   /etc/rc.d/init.d/postgresql stop
   ```

   devrait fonctionner. Une autre possibilité est **`pg_ctl stop`**.

   Les très vieilles versions pourraient ne pas disposer de pg_ctl. Si vous ne le trouvez pas ou s'il ne fonctionne pas, trouvez l'identifiant du processus du vieux serveur, en saisissant par exemple

   ```
   ps ax | grep postmaster
   ```

   et en lui envoyant un signal pour le stopper, de cette façon :

   ```
   kill −INT IDprocessus
   ```
4. Si vous faites l'installation au même endroit que l'ancienne, il peut être une bonne idée de déplacer l'ancienne version au cas où vous auriez des problèmes et que vous ayez besoin de revenir en arrière. Utilisez une commande telle que :

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

Après l'installation de PostgreSQL 8.0.5, créez un répertoire pour les données et démarrez le nouveau serveur. Rappelez−vous que vous devez exécuter ces commandes en étant connecté avec l'utilisateur dédié à la base (qui doit déjà exister si vous changez juste de version).

```
/usr/local/pgsql/bin/initdb −D /usr/local/pgsql/data
```

```
/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
```

Enfin, restaurez vos données avec

```
/usr/local/pgsql/bin/psql -d template1 -f outputfile
```

en utilisant le *nouveau* psql.

Des informations supplémentaires apparaissent dans la Section 22.4, que vous êtes encouragés à lire dans tous les cas.

## 14.5. Procédure d'installation

1. **Configuration**

   La première étape de la procédure d'installation est de configurer votre arborescence système et de choisir les options qui vous intéressent. Ce qui est fait en exécutant le script `configure`. Pour une installation par défaut, entrez simplement

   ```
   ./configure
   ```

   Ce script exécutera de nombreux tests afin de déterminer les valeurs de certaines variables dépendantes du système et de détecter certains aléas relatifs à votre système d'exploitation. Il créera divers fichiers dans l'arborescence de compilation pour enregistrer ce qui a été trouvé (vous pouvez aussi exécuter `configure` à partir d'un répertoire hors de l'arborescence des sources si vous voulez conserver l'arborescence de compilation séparé).

   La configuration par défaut compilera le serveur et les utilitaires, aussi bien que toutes les applications clientes et interfaces qui requièrent seulement un compilateur C. Tous les fichiers seront installés par défaut sous `/usr/local/pgsql`.

   Vous pouvez personnaliser les processus de compilation et d'installation en mettant une ou plusieurs options sur la ligne de commande après `configure` :

   `--prefix=`*`PREFIX`*
   : Installe tous les fichiers dans le répertoire *`PREFIX`* au lieu du répertoire `/usr/local/pgsql`. Les fichiers actuels seront installés dans divers sous−répertoires ; aucun fichier ne sera directement installés sous *`PREFIX`*.

     Si vous avez des besoins spécifiques, vous pouvez personnaliser les sous−répertoires à l'aide des options suivantes. Néanmoins, si vous les laissez vide avec leurs valeurs par défaut, l'installation sera déplaçable, ce qui signifie que vous pourrez bouger le répertoire après installation (les emplacements de `man` et `doc` ne sont pas affectés par ceci).

     Pour les installations déplaçables, vous pourriez vouloir utiliser l'option `--disable-rpath` de `configure`. De plus, vous aurez besoin d'indiquer au système d'exploitation comment trouver les bibliothèques partagées.

   `--exec-prefix=`*`EXEC-PREFIX`*

Vous pouvez installer les fichiers dépendants de l'architecture dans un répertoire différent, *EXEC-PREFIX*, de celui donné par *PREFIX*. Ce qui peut être utile pour partager les fichiers dépendants de l'architecture entre plusieurs machines. Si vous l'omettez, *EXEC-PREFIX* est égal à *PREFIX* et les fichiers dépendants seront installés sous la même arborescence que les fichiers indépendants de l'architecture, ce qui est probablement ce que vous voulez.

`--bindir=`*REPERTOIRE*
: Spécifie le répertoire des exécutables. Par défaut, il s'agit de *EXEC-PREFIX*`/bin`, ce qui signifie `/usr/local/pgsql/bin`.

`--datadir=`*REPERTOIRE*
: Prépare le répertoire pour des fichiers de données en lecture seule utilisés par les programmes d'installation. Par défaut, il s'agit de *PREFIX*`/share`. Il est bon de noter que cela n'a rien à voir avec l'emplacement des fichiers de votre base de données.

`--sysconfdir=`*REPERTOIRE*
: Le répertoire contenant divers fichiers de configuration. Par défaut, il s'agit de *PREFIX*`/etc`.

`--libdir=`*REPERTOIRE*
: L'endroit où installer les bibliothèques et les modules chargeables dynamiquement. Par défaut, il s'agit de *EXEC-PREFIX*`/lib`.

`--includedir=`*REPERTOIRE*
: Le répertoire où sont installées les en-têtes C et C++. Par défaut, il s'agit de *PREFIX*`/include`.

`--mandir=`*REPERTOIRE*
: Les pages man fournies avec PostgreSQL seront installées sous ce répertoire, dans leur sous-répertoire `man`*x* respectif. Par défaut, il s'agit de *PREFIX*`/man`.

`--with-docdir=`*REPERTOIRE*
`--without-docdir`
: Les fichiers de documentation, sauf les pages << man >>, seront installés dans ce répertoire. Par défaut, il s'agit de *PREFIX*`/doc`. Si l'option `--without-docdir` est spécifiée, la documentation ne sera pas installée par `make install`. Ceci est fait pour aider les scripts d'empaquetage ayant des méthodes particulières pour installer la documentation.

  **Note :** Une attention toute particulière a été prise afin de rendre possible l'installation de PostgreSQL dans des répertoires partagés (comme `/usr/local/include`) sans interférer avec des noms de fichiers relatifs au reste du système. En premier lieu, le mot << `/postgresql` >> est automatiquement ajouté au répertoire `datadir`, `sysconfdir` et `docdir`, à moins que le nom du répertoire à partir de la racine contienne déjà le mot << `postgres` >> où << `pgsql` >>. Par exemple, si vous choisissez `/usr/local` comme préfixe, la documentation sera installée dans `/usr/local/doc/postgresql`, mais si le préfixe est `/opt/postgres`, alors il sera dans `/opt/postgres/doc`. Les fichiers d'en-têtes publiques C de l'interface cliente seront installés sous `includedir` et sont propres par rapport aux noms de fichiers relatifs au reste du système. Les fichiers d'en-têtes privés et les fichiers d'en-têtes du serveur sont installés dans des répertoires privés sous `includedir`. Voir la documentation de chaque interface pour savoir comment obtenir ces fichiers d'en-tête. Enfin, un répertoire privé sera aussi créé si nécessaire sous `libdir` pour les modules chargeables dynamiquement.

`--with-includes=`*REPERTOIRES*
: *REPERTOIRES* est une liste de répertoires séparés par des caractères deux points (:) qui sera ajoutée à la liste de recherche des fichiers d'en-tête. Si vous avez des paquetages optionnels (tels que Readline GNU) installés dans des répertoires non conventionnels, vous pouvez

utiliser cette option et certainement l'option `--with-libraries` correspondante.

Exemple : `--with-includes=/opt/gnu/include:/usr/sup/include`.

`--with-libraries=`*`REPERTOIRES`*

*`REPERTOIRES`* est une liste de recherche de répertoires de bibliothèques séparés par des caractères deux points (:). Vous aurez probablement à utiliser cette option (et l'option correspondante `--with-includes`) si vous avez des paquetages installés dans des répertoires non conventionnels.

Exemple : `--with-libraries=/opt/gnu/lib:/usr/sup/lib`.

`--enable-nls[=`*`LANGUES`*`]`

Permet de mettre en place le support des langues natives (NLS). C'est la possibilité d'afficher les messages des programmes dans une langue autre que l'anglais. *`LANGUE`* est une liste de codes des langues que vous voulez supporter séparés par un espace. Par exemple, `--enable-nls='de fr'` (l'intersection entre votre liste et l'ensemble des langues traduites actuellement sera calculée automatiquement). Si vous ne spécifiez pas de liste, alors toutes les traductions disponibles seront installées.

Pour utiliser cette option, vous avez besoin d'une implémentation de l'API Gettext ; voir après.

`--with-pgport=`*`NUMERO`*

Positionne *`NUMERO`* comme numéro de port par défaut pour le serveur et les clients. La valeur par défaut est 5432. Le port peut toujours être changé ultérieurement mais, si vous le faites ici, alors les exécutables du serveur et des clients auront la même valeur par défaut, ce qui est vraiment très pratique. Habituellement, la seule bonne raison de choisir une valeur autre que celle par défaut est que vous souhaitez exécuter plusieurs serveurs PostgreSQL sur la même machine.

`--with-perl`

Permet l'utilisation du langage de procédures PL/Perl côté serveur.

`--with-python`

Permet la compilation du langage de procédures PL/Python.

`--with-tcl`

Permet la compilation du langage de procédures PL/Tcl.

`--with-tclconfig=`*`REPERTOIRE`*

Tcl installe les fichiers `tclConfig.sh`, contenant certaines informations de configuration nécessaires pour compiler le module d'interfaçage avec Tcl. Ce fichier est trouvé automatiquement mais, si vous voulez utiliser une version différente de Tcl, vous pouvez spécifier le répertoire où le trouver.

`--with-krb4`

`--with-krb5`

Compile le support d'authentification Kerberos. Vous pouvez utiliser soit la version 4 soit la version 5 de Kerberos, mais pas les deux. Sur beaucoup de systèmes, le système Kerberos n'est pas installé à un emplacement recherché par défaut (c'est–à–dire `/usr/include`, `/usr/lib`), donc vous devez utiliser les options `--with-includes` et `--with-libraries` en plus de cette option. `configure` vérifiera les fichiers d'en–tête et les bibliothèques requis pour s'assurer que votre installation Kerberos est suffisante avant de continuer

`--with-krb-srvnam=`*`NOM`*

Le nom du service principal de Kerberos. `postgres` est pris par défaut. Il n'y a probablement pas de raison de le changer.

`--with-openssl`

Compile le support de connexion SSL (chiffrement). Le paquetage OpenSSL doit être installé. `configure` vérifiera que les fichiers d'en−tête et les bibliothèques soient installés pour s'assurer que votre installation d'OpenSSL est suffisante avant de continuer.

`--with-pam`
: Compile le support PAM (*Modules d'Authentification Pluggable*).

`--without-readline`
: Évite l'utilisation de la bibliothèque Readline. Cela désactive l'édition de la ligne de commande et l'historique dans psql, ce n'est donc pas recommandé.

`--with-rendezvous`
: Compile le support Rendezvous. Ceci requiert le support de Rendezvous dans votre système d'exploitation. Recommandé sur Mac OS X.

`--disable-spinlocks`
: Autorise le succès de la construction y compris lorsque PostgreSQL n'a pas le support spinlock du CPU pour la plateforme. Ce manque de support résultera en des performances faibles ; du coup, cette option devra seulement être utilisée si la construction échoue et vous informe du manque de support de spinlock sur votre plateforme. Si cette option est requise pour construire PostgreSQL sur votre plateforme, merci de rapporter le problème aux développeurs de PostgreSQL.

`--enable-thread-safety`
: Rend les bibliothèques clientes compatibles avec les threads. Ceci permet des threads concurrents dans les programmes libpq et ECPG ce qui leur permet de gérer en toute sûreté leur connexions privées. Cette option requiert un support adéquat des threads sur votre système d'exploitation.

`--without-zlib`
: Évite l'utilisation de la bibliothèque Zlib. Cela désactive le support des archives compressées dans pg_dump et pg_restore. Cette option est seulement là pour les rares systèmes qui ne disposent pas de cette bibliothèque.

`--enable-debug`
: Compile tous les programmes et bibliothèques en mode de débogage. Cela signifie que vous pouvez exécuter les programmes via un débogueur pour analyser les problèmes. Cela grossit considérablement la taille des exécutables et, avec des compilateurs autres que GCC, habituellement, cela désactive les optimisations du compilateur, provoquant des ralentissements. Cependant, mettre ce mode en place est extrêmement utile pour repérer les problèmes. Actuellement, cette option est recommandée pour les installations en production seulement si vous utilisez GCC. Néanmoins, vous devriez l'utiliser si vous développez ou si vous utilisez une version béta.

`--enable-cassert`
: Permet la vérification des *assertions* par le serveur qui teste de nombreux cas de conditions << impossibles >>. Ce qui est inestimable dans le cas de développement, mais les tests ralentissent le système. Activer cette option n'influe pas sur la stabilité de votre serveur ! Les assertions vérifiées ne sont pas classées par ordre de sévérité et il se peut qu'un bogue anodin fasse redémarrer le serveur s'il y a un échec de vérification. Actuellement, cette option n'est pas recommandée dans un environnement de production mais vous devriez l'utiliser lors de développement ou pour les versions béta.

`--enable-depend`
: Active la recherche automatique des dépendances. Avec cette option, les fichiers makefile sont appelés pour recompiler les fichiers objet dès qu'un fichier d'en−tête est modifié. C'est pratique si vous faites du développement, mais inutile si vous ne voulez que compiler une fois et installer. Pour le moment, cette option ne fonctionne qu'avec GCC.

Si vous préférez utiliser un compilateur C différent de ceux listés par `configure`, positionnez la variable d'environnement `CC` pour qu'elle pointe sur le compilateur de votre choix. Par défaut, `configure` pointe sur `gcc` s'il est disponible, sinon il utilise celui par défaut de la plateforme (habituellement `cc`). De façon similaire, vous pouvez repositionner les options par défaut du compilateur à l'aide de la variable `CFLAGS`.

Vous pouvez spécifier les variables d'environnement sur la ligne de commande `configure`, par exemple :

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

2. **Compilation**

   Pour démarrer la compilation, saisissez

   ```
   gmake
   ```

   (Rappelez−vous d'utiliser GNU make). La compilation peut prendre entre cinq minutes et une demi−heure en fonction de votre matériel. La dernière ligne affichée devrait être

   ```
   All of PostgreSQL is successfully made. Ready to install.
   ```

3. **Tests de régression**

   Si vous souhaitez tester le serveur nouvellement compileé avant de l'installer, vous pouvez exécuter les tests de régression à ce moment. Les tests de régression sont une suite de tests qui vérifient que PostgreSQL fonctionne sur votre machine tel que les développeurs l'espèrent. Saisissez

   ```
   gmake check
   ```

   (Cela ne fonctionne pas en tant que root ; faites−le en tant qu'utilisateur sans droits.) Le Chapitre 26 contient des détails sur l'interprétation des résultats de ces tests. Vous pouvez les répéter autant de fois que vous le voulez en utilisant la même commande.

4. **Installer les fichiers**

   > **Note :** Si vous mettez à jour une version existante et que vous placez les fichiers au même endroit que les anciens, assurez−vous d'avoir sauvegardé vos données et arrêté l'ancien serveur avant de continuer, comme l'explique la Section 14.4 ci−après.

   Pour installer PostgreSQL, saisissez

   ```
   gmake install
   ```

   Cela installera les fichiers dans les répertoires spécifiés dans l'étape 1. Assurez−vous d'avoir les droits appropriés pour écrire dans ces répertoires. Normalement, vous avez besoin d'être superutilisateur pour cette étape. Une alternative consiste à créer les répertoires cibles à l'avance et à leur donner les droits appropriées.

   Vous pouvez utiliser `gmake install-strip` en lieu et place de `gmake install` pour dépouiller l'installation des exécutables et des bibliothèques. Cela économise un peu d'espace disque. Si vous avez effectué la compilation en mode de débogage, ce dépouillage l'enlèvera, donc ce n'est à faire seulement si ce mode n'est plus nécessaire. `install-strip` essaie d'être raisonnable en

sauvegardant de l'espace disque mais il n'a pas une connaissance parfaite de la façon de dépouiller un exécutable de tous les octets inutiles. Ainsi, si vous voulez sauvegarder le maximum d'espace disque, vous devrez faire le travail à la main.

L'installation standard fournit seulement les fichiers en−têtes nécessaires pour le développement d'applications clientes ainsi que pour le développement de programmes côté serveur comme des fonction personnelles ou des types de données écrits en C (avant PostgreSQL 8.0, une commande `gmake install-all-headers` séparée était nécessaire pour ce dernier point mais cette étape a été intégrée à l'installation standard).

**Installation du client uniquement :** Si vous voulez uniquement installer les applications clientes et les bibliothèques d'interface, alors vous pouvez utilisez ces commandes :

```
gmake -C src/bin install
gmake -C src/include install
gmake -C src/interfaces install
gmake -C doc install
```

**Enregistrer eventlog sur Windows :** Pour enregistrer une bibliothèque eventlog sur Windows grâce au système d'exploitation, exécutez cette commande après l'installation :

```
regsvr32 pgsql_library_directory/pgevent.dll
```

Ceci crée les entrées du registre utilisées par le visualisateur d'événements.

**Désinstallation :** Pour désinstaller, utilisez la commande `gmake uninstall`. Cependant, cela ne supprimera pas les répertoires créés.

**Ménage :** Après l'installation, vous pouvez faire le ménage en supprimant les fichiers issus de la compilation des répertoires sources à l'aide de la commande `gmake clean`. Cela conservera les fichiers créés par la commande `configure`, ainsi vous pourrez tout recompiler ultérieurement avec `gmake`. Pour remettre l'arborescence source dans l'état initial, utilisez `gmake distclean`. Si vous voulez effectuer la compilation pour diverses plateformes à partir des mêmes sources vous devrez d'abord refaire la configuration à chaque fois (autrement, utilisez un répertoire de construction séparé pour chaque plateforme, de façon à ce que le répertoire des sources reste inchangé).

Si vous avez compilé et que vous vous êtes rendu compte que les options de `configure` sont fausses ou si vous changez quoique ce soit que `configure` prenne en compte (par exemple, la mise à jour d'applications), alors faire un `gmake distclean` avant de reconfigurer et recompiler est une bonne idée. Sans ça, vos changements dans la configuration ne seront pas répercutés partout où il faut.

# 14.6. Initialisation post−installation

## 14.6.1. Bibliothèques partagées

Sur certains systèmes qui utilisent les bibliothèques partagées (ce que font de nombreux systèmes), vous avez besoin de leurs spécifier comment trouver les nouvelles bibliothèques partagées. Les systèmes sur lesquels ce *n'est* pas nécessaire comprennent BSD/OS, FreeBSD, HP−UX, IRIX, Linux, NetBSD, OpenBSD, Tru64 UNIX (formellement Digital UNIX) et Solaris.

La méthode pour le faire varie selon la plateforme, mais la méthode la plus répandue consiste à positionner des variables d'environnement comme LD_LIBRARY_PATH : avec les shells Bourne (sh, ksh, bash, zsh)

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

ou en csh ou tcsh

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Remplacez /usr/local/pgsql/lib par la valeur donnée à --libdir dans l'étape 1. Vous pouvez mettre ces commandes dans un script de démarrage tel que /etc/profile ou ~/.bash_profile. Certaines informations pertinentes au sujet de mises en garde associées à cette méthode peuvent être trouvées sur http://www.visi.com/~barr/ldpath.html.

Sur certains systèmes, il peut être préférable de renseigner la variable d'environnement LD_RUN_PATH *avant* la compilation.

Avec Cygwin, placez le répertoire des bibliothèques dans la variable PATH ou déplacez les fichiers .dll dans le répertoire bin.

En cas de doute, référez–vous aux pages de man de votre système (peut–être ld.so ou rld). Si vous avez ultérieurement un message tel que

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

alors cette étape est vraiment nécessaire. Faites–y attention.

Si votre système d'exploitation est BSD/OS, Linux ou SunOS 4 et que vous avez les accès de superutilisateur, vous pouvez exécuter

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(ou le répertoire équivalent) après l'installation pour permettre à l'éditeur de liens de trouver les bibliothèques partagées plus rapidement. Référez–vous aux pages man portant sur ldconfig pour plus d'informations. Pour les systèmes d'exploitation FreeBSD, NetBSD et OpenBSD, la commande est

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

Les autres systèmes d'exploitation ne sont pas connus pour avoir de commande équivalente.

## 14.6.2. Variables d'environnement

Si l'installation a été réalisée dans /usr/local/pgsql ou à un autre endroit qui n'est pas dans les répertoires contenant les exécutables par défaut, vous devez ajouter /usr/local/pgsql/bin (ou le répertoire fourni à --bindir au moment de l'étape 1) dans votre PATH. Techniquement, ce n'est pas une obligation mais cela rendra l'utilisation de PostgreSQL plus confortable.

Pour ce faire, ajoutez ce qui suit dans le fichier d'initialisation de votre shell, par exemple ~/.bash_profile (ou /etc/profile, si vous voulez que tous les utilisateurs l'aient) :

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

Si vous utilisez le `csh` ou le `tcsh`, alors utilisez la commande :

```
set path = ( /usr/local/pgsql/bin $path )
```

Pour que votre système trouve la documentation man, il vous faut ajouter des lignes telles que celles qui suivent à votre fichier d'initialisation du shell, à moins que vous installiez ces pages dans un répertoire où elles sont mises normalement.

```
MANPATH=/usr/local/pgsql/man:$MANPATH
export MANPATH
```

Les variables d'environnement `PGHOST` et `PGPORT` indiquent aux applications clientes l'hôte et le port du serveur de base. Elles surchargent les valeurs utilisées lors de la compilation. Si vous exécutez des applications clientes à distance, alors c'est plus pratique si tous les utilisateurs peuvent paramétrer `PGHOST`. Ce n'est pas une obligation, cependant, la configuration peut être communiquée via les options de lignes de commande à la plupart des programmes clients.

## 14.7. Plateformes supportées

La communauté de développeurs a vérifié que PostgreSQL fonctionne sur les plateformes listées ci−après. Dire qu'une plateforme est supportée signifie en général que la compilation et l'installation de PostgreSQL peuvent s'exécuter en suivant les instructions et que les tests de régression sont valides. Les entrées << Ferme de construction >> font référence à la ferme de construction PostgreSQL (*PostgreSQL Ferme de construction*). Les entrées de plateformes affichant une version plus ancienne de PostgreSQL sont celles qui n'ont pas reçu de tests explicites au moment de la sortie de la version 8.0 mais qui doivent toujours fonctionner.

> **Note :** Si vous avez des problèmes avec une installation sur une plateforme supportée, veuillez écrire à <pgsql−bogues@postgresql.org> ou à <pgsql−ports@postgresql.org> et non aux personnes listées dans ce document.

| Système d'exploitation | Processeur | Version | Date du rapport | Remarques |
|---|---|---|---|---|
| AIX | PowerPC | 8.0.0 | Travis P (<twp@castle.fastmail.fm>), 2004−12−12 | voir aussi `doc/FAQ_AIX` |
| AIX | RS6000 | 8.0.0 | Hans−Jürgen Schönig (<hs@cybertec.at>), 2004−12−06 | voir aussi `doc/FAQ_AIX` |
| BSD/OS | x86 | 8.0.0 | Bruce Momjian (<pgman@candle.pha.pa.us>), 2004−12−07 | 4.3.1 |
| Debian GNU/Linux | Alpha | 7.4 | Noèl Köthe (<noel@debian.org>), 2003−10−25 | |
| Debian GNU/Linux | AMD64 | 8.0.0 | Ferme de construction panda, snapshot 2004−12−06 01:20:02 | sid, kernel 2.6 |

| | | | | |
|---|---|---|---|---|
| Debian GNU/Linux | ARM | 8.0.0 | Jim Buttafuoco (<jim@contactbda.com>), 2005−01−06 | |
| Debian GNU/Linux | IA64 | 7.4 | Noèl Köthe (<noel@debian.org>), 2003−10−25 | |
| Debian GNU/Linux | m68k | 8.0.0 | Noèl Köthe (<noel@debian.org>), 2004−12−09 | sid |
| Debian GNU/Linux | MIPS | 8.0.0 | Ferme de construction lionfish, snapshot 2004−12−06 11:00:08 | 3.1 (sarge), kernel 2.4 |
| Debian GNU/Linux | PA−RISC | 8.0.0 | Noèl Köthe (<noel@debian.org>), 2004−12−07 | sid |
| Debian GNU/Linux | PowerPC | 8.0.0 | Noèl Köthe (<noel@debian.org>), 2004−12−15 | sid |
| Debian GNU/Linux | S/390 | 7.4 | Noèl Köthe (<noel@debian.org>), 2003−10−25 | |
| Debian GNU/Linux | Sparc | 8.0.0 | Noèl Köthe (<noel@debian.org>), 2004−12−09 | sid, 32−bit |
| Debian GNU/Linux | x86 | 8.0.0 | Peter Eisentraut (<peter_e@gmx.net>), 2004−12−06 | 3.1 (sarge), kernel 2.6 |
| Fedora | AMD64 | 8.0.0 | John Gray (<jgray@azuli.co.uk>), 2004−12−12 | FC3 |
| Fedora | x86 | 8.0.0 | Ferme de construction dog, snapshot 2004−12−06 02:06:01 | FC1 |
| FreeBSD | Alpha | 7.4 | Peter Eisentraut (<peter_e@gmx.net>), 2003−10−25 | 4.8 |
| FreeBSD | x86 | 8.0.0 | Ferme de construction cockatoo, snapshot 2004−12−06 14:10:01 (4.10); Marc Fournier (<scrappy@postgresql.org>), 2004−12−07 (5.3) | |
| Gentoo Linux | AMD64 | 8.0.0 | Jani Averbach (<jaa@jaa.iki.fi>), 2005−01−13 | |
| Gentoo Linux | x86 | 8.0.0 | Paul Bort (<pbort@tmwsystems.com>), 2004−12−07 | |
| HP−UX | IA64 | 8.0.0 | Tom Lane (<tgl@sss.pgh.pa.us>), 2005−01−06 | 11.23, `gcc` et `cc` ; voir aussi `doc/FAQ_HPUX` |
| HP−UX | PA−RISC | 8.0.0 | Tom Lane (<tgl@sss.pgh.pa.us>), 2005−01−06 | 10.20 et 11.11, `gcc` et `cc` ; voir aussi `doc/FAQ_HPUX` |
| IRIX | MIPS | 7.4 | Robert E. Bruccoleri (<bruc@stone.congenomics.com>), 2003−11−12 | 6.5.20, `cc` only |
| Mac OS X | PowerPC | 8.0.0 | Andrew Rawnsley (<ronz@ravensfield.com>), 2004−12−07 | 10.3.5 |
| Mandrakelinux | x86 | 8.0.0 | Ferme de construction shrew, snapshot 2004−12−06 02:02:01 | 10.0 |
| NetBSD | arm32 | 7.4 | | 1.6ZE/acorn32 |

| | | | | |
|---|---|---|---|---|
| | | | Patrick Welche (<prlw1@newn.cam.ac.uk>), 2003−11−12 | |
| NetBSD | m68k | 8.0.0 | Rémi Zara (<remi_zara@mac.com>), 2004−12−14 | 2.0 |
| NetBSD | Sparc | 7.4.1 | Peter Eisentraut (<peter_e@gmx.net>), 2003−11−26 | 1.6.1, 32−bit |
| NetBSD | x86 | 8.0.0 | Ferme de construction canary, snapshot 2004−12−06 03:30:00 | 1.6 |
| OpenBSD | Sparc | 8.0.0 | Chris Mair (<list@1006.org>), 2005−01−10 | 3.3 |
| OpenBSD | Sparc64 | 8.0.0 | Ferme de construction spoonbill, snapshot 2005−01−06 00:50:05 | 3.6 |
| OpenBSD | x86 | 8.0.0 | Ferme de construction emu, snapshot 2004−12−06 11:35:03 | 3.6 |
| Red Hat Linux | AMD64 | 8.0.0 | Tom Lane (<tgl@sss.pgh.pa.us>), 2004−12−07 | RHEL 3AS |
| Red Hat Linux | IA64 | 8.0.0 | Tom Lane (<tgl@sss.pgh.pa.us>), 2004−12−07 | RHEL 3AS |
| Red Hat Linux | PowerPC | 8.0.0 | Tom Lane (<tgl@sss.pgh.pa.us>), 2004−12−07 | RHEL 3AS |
| Red Hat Linux | PowerPC 64 | 8.0.0 | Tom Lane (<tgl@sss.pgh.pa.us>), 2004−12−07 | RHEL 3AS |
| Red Hat Linux | S/390 | 8.0.0 | Tom Lane (<tgl@sss.pgh.pa.us>), 2004−12−07 | RHEL 3AS |
| Red Hat Linux | S/390x | 8.0.0 | Tom Lane (<tgl@sss.pgh.pa.us>), 2004−12−07 | RHEL 3AS |
| Red Hat Linux | x86 | 8.0.0 | Tom Lane (<tgl@sss.pgh.pa.us>), 2004−12−07 | RHEL 3AS |
| Solaris | Sparc | 8.0.0 | Kenneth Marshall (<ktm@is.rice.edu>), 2004−12−07 | Solaris 8 ; voir aussi doc/FAQ_Solaris |
| Solaris | x86 | 8.0.0 | Ferme de construction kudu, snapshot 2004−12−10 02:30:04 (cc); dragonfly, snapshot 2004−12−09 04:30:00 (gcc) | Solaris 9 ; voir aussi doc/FAQ_Solaris |
| SUSE Linux | AMD64 | 8.0.0 | Reinhard Max (<max@suse.de>), 2005−01−03 | 9.0, 9.1, 9.2, SLES 9 |
| SUSE Linux | IA64 | 8.0.0 | Reinhard Max (<max@suse.de>), 2005−01−03 | SLES 9 |
| SUSE Linux | PowerPC | 8.0.0 | Reinhard Max (<max@suse.de>), 2005−01−03 | SLES 9 |
| SUSE Linux | PowerPC 64 | 8.0.0 | Reinhard Max (<max@suse.de>), 2005−01−03 | SLES 9 |
| SUSE Linux | S/390 | 8.0.0 | Reinhard Max (<max@suse.de>), 2005−01−03 | SLES 9 |
| SUSE Linux | S/390x | 8.0.0 | Reinhard Max (<max@suse.de>), 2005−01−03 | SLES 9 |

| | | | | |
|---|---|---|---|---|
| SUSE Linux | x86 | 8.0.0 | Reinhard Max (<max@suse.de>), 2005−01−03 | 9.0, 9.1, 9.2, SLES 9 |
| Tru64 UNIX | Alpha | 8.0.0 | Honda Shigehiro (<fwif0083@mb.infoweb.ne.jp>), 2005−01−07 | 5.0 |
| UnixWare | x86 | 8.0.0 | Peter Eisentraut (<peter_e@gmx.net>), 2004−12−14 | cc, 7.1.4 ; voir aussi doc/FAQ_SCO |
| Windows | x86 | 8.0.0 | Dave Page (<dpage@vale-housing.co.uk>), 2004−12−07 | XP Pro ; voir doc/FAQ_MINGW |
| Windows with Cygwin | x86 | 8.0.0 | Ferme de construction gibbon, snapshot 2004−12−11 01:33:01 | voir doc/FAQ_CYGWIN |

**Plateformes non supportées :** Les plateformes suivantes sont connues pour ne pas fonctionner, ou fonctionnaient avec des versions très antérieures. Nous les plaçons ici afin de vous prévenir qu'elles *peuvent* être supportées avec quelques efforts.

| Système d'exploitation | Processeur | Version | Date du rapport | Remarques |
|---|---|---|---|---|
| BeOS | x86 | 7.2 | Cyril Velter (<cyril.velter@libertysurf.fr>), 2001−11−29 | Besoin de mises à jour pour le code sur les sémaphores |
| Linux | PlayStation 2 | 8.0.0 | Chris Mair (<list@1006.org>), 2005−01−09 | Requiert --disable-spinlocks (fonctionne bien que lent) |
| NetBSD | Alpha | 7.2 | Thomas Thai (<tom@minnesota.com>), 2001−11−20 | 1.5W |
| NetBSD | MIPS | 7.2.1 | Warwick Hunter (<whunter@agile.tv>), 2002−06−13 | 1.5.3 |
| NetBSD | PowerPC | 7.2 | Bill Studenmund (<wrstuden@netbsd.org>), 2001−11−28 | 1.5 |
| NetBSD | VAX | 7.1 | Tom I. Helbekkmo (<tih@kpnQwest.no>), 2001−03−30 | 1.5 |
| QNX 4 RTOS | x86 | 7.2 | Bernd Tegge (<tegge@repas-aeg.de>), 2001−12−10 | Nécessite des mises à jour pour le code des sémaphores ; voir aussi doc/FAQ_QNX4 |
| QNX RTOS v6 | x86 | 7.2 | Igor Kovalenko (<Igor.Kovalenko@motorola.com>), 2001−11−20 | Correctifs disponibles dans les archives, mais trop tard pour la 7.2 |
| SCO OpenServer | x86 | 7.3.1 | Shibashish Satpathy (<shib@postmark.net>), 2002−12−11 | 5.0.4, gcc ; voir aussi doc/FAQ_SCO |

| | | | | |
|---|---|---|---|---|
| SunOS 4 | Sparc | 7.2 | Tatsuo Ishii (<t-ishii@sra.co.jp>), 2001-12-04 | |

# Chapitre 15. Installation sur Windows du client uniquement

Bien qu'une installation complète de PostgreSQL peut seulement être construit avec MinGW ou Cygwin, la bibliothèque cliente en C (libpq) ainsi que le terminal interactif (psql) peuvent être compilés en utilisant d'autres outils Windows. Les fichiers makefile sont inclus dans la distribution source pour Microsoft Visual C++ et Borland C++. Il doit être possible de compiler les bibliothèques manuellement dans d'autres configurations.

> **Astuce :** Utiliser Microsoft Visual C++ ou Borland C++ est la solution préférée. Si vous utilisez un de ces ensembles d'outils, référez–vous au Chapitre 14.

Pour compiler tous les utilitaires disponibles pour Windows en utilisant Microsoft Visual C++, placez–vous dans le répertoire `src` et saisissez la commande :

```
nmake /f win32.mak
```

Cette action ne peut être accomplie que si vous avez Visual C++ dans la variable d'environnement path.

Pour tout construire en utilisant Borland C++, placez–vous dans le répertoire `src` et saisissez la commande :

```
make -N -DCFG=Release /f bcc32.mak
```

Les fichiers suivants seront produits :

`interfaces\libpq\Release\libpq.dll`
: La bibliothèque dynamique d'interface client

`interfaces\libpq\Release\libpqdll.lib`
: La bibliothéque d'import nécessaire pour lier vos programmes à `libpq.dll`

`interfaces\libpq\Release\libpq.lib`
: La version statique de la bibliothèque d'interface client

`bin\psql\Release\psql.exe`
: Le terminal interactif de PostgreSQL

Le seul fichier devant réellement être installé est la bibliothèque dynamique `libpq.dll`. Ce fichier doit être placé dans la plupart des cas dans le répertoire `WINNT\SYSTEM32` (ou dans `WINDOWS\SYSTEM` sur des systèmes Windows 95/98/ME). Si le fichier est installé par le biais d'un programme d'installation, il doit être installé avec un contrôle de version utilisant la ressource `VERSIONINFO` incluse dans le fichier, afin d'assurer qu'une nouvelle version de la bibliothèque ne sera pas écrasée.

Si vous envisagez d'effectuer le développement d'une application utilisant libpq sur cette machine, vous devrez ajouter les sous–répertoires `src\include` et `src\interfaces\libpq` de l'ensemble des source dans le chemin des fichiers d'inclusion de votre compilateur.

Afin d'utiliser la bibliothèque, vous devrez ajouter `libpqdll.lib` dans votre projet. (Dans Visual C++, vous n'aurez qu'à cliquer avec le bouton droit sur le projet et choisir de l'ajouter.).

# Chapitre 16. Environnement d'exécution du serveur

Ce chapitre discute de la configuration, du lancement du serveur de bases de données et de ses interactions avec le système d'exploitation.

## 16.1. Compte utilisateur PostgreSQL

Comme avec tout autre démon serveur accessible au monde externe, il est conseillé de lancer PostgreSQL sous un compte utilisateur séparé. Ce compte devrait seulement être le propriétaire des données gérées par le serveur et ne devrait pas être partagé avec d'autres démons (par exemple, utiliser l'utilisateur `nobody` est une mauvaise idée). Il n'est pas conseillé de changer le propriétaire des exécutables par cet utilisateur car les systèmes compromis pourraient alors se voir modifier leur propres binaires.

Pour ajouter un compte utilisateur Unix, jetez un œil à la commande `useradd` ou `adduser` de votre système. Le nom de l'utilisateur postgres est souvent utilisé et l'est sur tout le livre, mais vous pouvez utiliser un autre nom si vous le souhaitez.

## 16.2. Créer un groupe de base de données

Avant de faire quoi que ce soit, vous devez initialiser un emplacement de stockage de la base de données. Nous appelons ceci un *groupe de bases de données*. (SQL utilise le terme de groupe de catalogues.) Un groupe de bases de données est une collection de bases données et est géré par une seule instance d'un serveur de bases de données en cours d'exécution. Après initialisation, un groupe de bases de données contiendra une base de données nommée `template1`. Comme le nom le suggère, elle sera utilisée comme modèle pour les bases de données créées après ; elle ne devrait pas être utilisée pour un vrai travail. (Voir Chapitre 18 pour des informations sur la création de nouvelles bases de données dans le groupe.)

En termes de système de fichiers, un groupe de bases de données sera un simple répertoire sous lequel les données seront stockées. Nous l'appelons le *répertoire de données* ou l'*emplacement des données*. Le choix de cet emplacement vous appartient complètement. Il n'existe pas de valeur par défaut bien que les emplacements tels que `/usr/local/pgsql/data` ou `/var/lib/pgsql/data` sont populaires. Pour initialiser un groupe de bases de données, utilisez la commande initdb, installée avec PostgreSQL. L'emplacement désiré sur le groupe de fichier est indiqué par l'option `-D`, par exemple

```
$ initdb -D /usr/local/pgsql/data
```

Notez que vous devez exécuter cette commande en étant connecté sous le compte de l'utilisateur PostgreSQL décrit dans la section précédente.

> **Astuce :** Comme alternative à l'option `-D`, vous pouvez initialiser la variable d'environnement `PGDATA`.

`initdb` tentera de créer le répertoire que vous avez spécifié si celui-ci n'existe pas déjà. Il est possible qu'il n'ait pas le droit de le faire (si vous avez suivi notre conseil et créé un compte sans droits). Dans ce cas, vous devez créer le répertoire vous-même (en tant que root) et modifier le propriétaire pour qu'il corresponde à

l'utilisateur PostgreSQL. Voici comment réaliser ceci :

```
root# mkdir /usr/local/pgsql/data
root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

`initdb` refusera de s'exécuter si le répertoire des données semble être déjà initialisé.

Comme le répertoire des données contient toutes les données stockées par le système de bases de données, il est essentiel qu'il soit sécurisé par rapport à des accès non autorisés. Du coup, `initdb` supprimera les droits d'accès à tout le monde sauf l'utilisateur PostgreSQL.

Néanmoins, bien que le contenu du répertoire soit sécurisé, la configuration d'authentification du client par défaut permet à tout utilisateur local de se connecter à la base de données et même à devenir le super–utilisateur de la base de données. Si vous ne faites pas confiance aux utilisateurs locaux, nous vous recommandons d'utiliser une des options `-W` ou `--pwprompt` de la commande `initdb` pour affecter un mot de passe au super–utilisateur de la base de données.. De plus, spécifiez `-A md5` ou `-A mot_de_passe` de façon à ce que la méthode d'authentification `trust` par défaut ne soit pas utilisée ; ou modifiez le fichier `pg_hba.conf` généré après l'exécution d'`initdb` (d'autres approches raisonnables incluent l'utilisation de l'authentification `ident` ou les droits du système de fichiers pour restreindre les connexions. Voir le Chapitre 19 pour plus d'informations).

`initdb` initialise aussi la locale par défaut du groupe de bases de données. Normalement, elle prends seulement le paramétrage local dans l'environnement et l'applique à la base de données initialisée. Il est possible de spécifier une locale différente pour la base de données ; la Section 20.1 propose plus d'informations là–dessus. L'ordre de tri utilisé à l'intérieur du groupe de bases de données est initialisé par `initdb` et ne peut pas être modifié après, en dehors de la sauvegarde des données, du redémarrage de `initdb` et du rechargement des données. Il y a aussi un impact sur les performances lors de l'utilisation de locales autres que `C` ou `POSIX`. Du coup, il est important de faire ce choix correctement la première fois.

`initdb` configure aussi le codage par défaut de l'ensemble de caractères pour le groupe de bases de données. Normalement, cela doit été choisi pour correspondre au paramétrage de la locale. Pour les détails, voir la Section 20.2.

## 16.3. Lancer le serveur de bases de données

Avant qu'une personne ait accès à la base de données, vous devez démarrer le serveur de bases de données. Le programme serveur est appelé `postmaster`.. Le `postmaster` doit savoir où trouver les données qu'il est supposé utiliser. Ceci se fait avec l'option `-D`. Du coup, la façon la plus simple de lancer le serveur est :

```
$ postmaster -D /usr/local/pgsql/data
```

qui laissera le serveur s'exécuter en avant plan. Pour cela, vous devez être connecté en utilisant le compte de l'utilisateur PostgreSQL. Sans `-D`, le serveur essaiera d'utiliser le répertoire de données nommé par la variable d'environnement `PGDATA`. Si cette variable ne le fournit pas non plus, le lancement échouera.

Habituellement, il est préférable de lancer `postmaster` en tâche de fond. Pour cela, utilisez la syntaxe shell habituelle :

```
$ postmaster -D /usr/local/pgsql/data >journaux_trace 2>&1 &
```

Il est important de sauvegarder les sorties stdout et stderr du serveur quelque part, comme montré ci−dessus. Cela vous aidera dans des buts d'audits ou pour diagnostiquer des problèmes (voir la Section 21.3 pour une discussion plus détaillée de la gestion de journaux de trace).

Le `postmaster` prend aussi un certain nombre d'autres options en ligne de commande. Pour plus d'informations, voir la page de référence postmaster ainsi que le Section 16.4 ci−dessous.

Cette syntaxe shell peut rapidement devenir ennuyante. Donc, le programme d'emballage pg_ctl est fourni pour simplifier certaines tâches. Par exemple :

```
pg_ctl start −l journaux_trace
```

lancera le serveur en tâche de fond et placera les sorties dans le journal de trace indiqué. L'option `−D` a la même signification ici qu'avec `postmaster`. `pg_ctl` est aussi capable d'arrêter le serveur.

Normalement, vous lancerez le serveur de bases de données lors du démarrage de l'ordinateur. Les scripts de lancement automatique sont spécifiques au système d'exploitation. Certains sont distribués avec PostgreSQL dans le répertoire `contrib/start−scripts`. En installer un demandera les droits de root.

Différents systèmes ont différentes conventions pour lancer les démons au démarrage. La plupart des systèmes ont un fichier `/etc/rc.local` ou `/etc/rc.d/rc.local`. D'autres utilisent les répertoires `rc.d`. Quoi que vous fassiez, le serveur doit être exécuté par le compte utilisateur PostgreSQL *et non pas par root* ou tout autre utilisateur. Donc, vous devriez probablement former vos commandes en utilisant `su −c '...' postgres`. Par exemple :

```
su −c 'pg_ctl start −D /usr/local/pgsql/data −l serverlog' postgres
```

Voici quelques suggestions supplémentaires par système d'exploitation (dans chaque cas, assurez−vous d'utiliser le bon répertoire d'installation et le bon nom de l'utilisateur où nous montrons des valeurs génériques).

- Pour FreeBSD, regardez le fichier `contrib/start−scripts/freebsd` du répertoire des sources de PostgreSQL.
- Sur OpenBSD, ajoutez les lignes suivantes à votre fichier `/etc/rc.local` :

  ```
  if [ −x /usr/local/pgsql/bin/pg_ctl −a −x /usr/local/pgsql/bin/postmaster ]; then
      su − −c '/usr/local/pgsql/bin/pg_ctl start −l /var/postgresql/log −s' postgres
      echo −n ' postgresql'
  fi
  ```
- Sur les systèmes Linux, soit vous ajoutez

  ```
  /usr/local/pgsql/bin/pg_ctl start −l journaux_trace −D /usr/local/pgsql/data
  ```

  à `/etc/rc.d/rc.local` soit vous jetez un œil à `contrib/start−scripts/linux` dans le répertoire des sources de PostgreSQL.
- Sur NetBSD, vous pouvez utiliser les scripts de lancement de FreeBSD ou de Linux suivant vos préférences.
- Sur Solaris, créez un fichier appelé `/etc/init.d/postgresql` et contenant la ligne suivante :

  ```
  su − postgres −c "/usr/local/pgsql/bin/pg_ctl start −l journaux_trace −D /usr/local/pgsql/
  ```

  Puis, créez un lien symbolique vers lui dans `/etc/rc3.d` de nom `S99postgresql`.

Tant que `postmaster` est lancé, son PID est stocké dans le fichier `postmaster.pid` du répertoire de données. C'est utilisé pour empêcher les nombreux processus `postmaster` d'être exécuté dans le même répertoire de données et peut aussi être utilisé pour arrêter le processus `postmaster`.

## 16.3.1. Échecs de lancement

Il existe de nombreuses raisons habituelles pour lesquelles le serveur échouerait au lancement. Vérifiez le journal des traces du serveur ou lancez–le manuellement (sans redirection des sorties standard et d'erreur) et regardez les messages d'erreurs qui apparaissent. Nous en expliquons certains ci–dessous parmi les messages d'erreurs les plus communs.

```
LOG:  could not bind IPv4 socket: Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds and retry.
FATAL:  could not create TCP/IP listen socket
```

Ceci signifie seulement ce que cela suggère : vous avez essayé de lancer un autre `postmaster` sur le même port où un autre est en cours d'exécution. Néanmoins, si le message d'erreur du noyau n'est pas `Address already in use` ou une quelconque variante, il pourrait y avoir un autre problème. Par exemple, essayer de lancer un `postmaster` sur un numéro de port réservé pourrait avoir ce résultat :

```
$ postmaster -p 666
LOG:  could not bind IPv4 socket: Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds and retry.
FATAL:  could not create TCP/IP listen socket
```

Un message du type

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

signifie probablement que les limites de votre noyau sur la taille de la mémoire partagée est plus petite que l'aire de fonctionnement que PostgreSQL essaie de créer (4011376640 octets dans cet exemple). Ou il pourrait signifier que vous n'avez pas du tout configuré le support de la mémoire partagée de type System–V dans votre noyau. Comme contournement temporaire, vous pouvez essayer de lancer le serveur avec un nombre de tampons plus petit que la normale (option `-B`). Vous voudrez éventuellement reconfigurer votre noyau pour accroître la taille de mémoire partagée autorisée. Vous pourriez voir aussi ce message en essayant de lancer plusieurs serveurs sur la même machine si le total de l'espace qu'ils requièrent dépasse la limite du noyau.

Une erreur du type

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

ne signifie *pas* qu'il vous manque de l'espace disque. Elle signifie que la limite de votre noyau sur le nombre de sémaphores System V est inférieur au nombre que PostgreSQL veut créer. Comme ci–dessus, vous pourriez contourner le problème en lançant le serveur avec un nombre réduit de connexions autorisées (option `-N`) mais vous voudrez éventuellement augmenter la limite du noyau.

Si vous obtenez une erreur << illegal system call >>, il est probable que la mémoire partagée ou les sémaphores ne sont pas du tout supportés par votre noyau. Dans ce cas, votre seule option est de reconfigurer le noyau pour activer ces fonctionnalités.

Des détails sur la configuration des capacités IPC System V sont donnés dans la Section 16.5.1.

### 16.3.2. Problèmes de connexion du client

Bien que les conditions d'erreurs possibles du côté client sont assez variées et dépendantes de l'application, certaines pourraient être en relation direct avec la façon dont le serveur a été lancé. Les conditions autres que celles montrées ici devraient être documentées avec l'application client respective.

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

Ceci est l'échec générique << Je n'ai pas trouvé de serveur à qui parler >>. Cela ressemble au message ci−dessus lorsqu'une connexion TCP/IP est tentée. Une erreur commune est d'oublier de configurer le serveur pour qu'il autorise les connexions TCP/IP.

Autrement, vous obtiendrez ceci en essayant une communication de type socket de domaine Unix vers un serveur local :

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

La dernière ligne est utile pour vérifier si le client essaie de se connecter au bon endroit. Si aucun serveur n'est exécuté ici, le message d'erreur du noyau sera typiquement soit `Connection refused` soit `No such file or directory`, comme ce qui est illustré (il est important de réaliser que `Connection refused`, dans ce contexte, ne signifie *pas* que le serveur a obtenu une demande de connexion et l'a refusé. Ce cas produira un message différent comme indiqué dans la Section 19.3). D'autres messages d'erreurs tel que `Connection timed out` pourraient indiquer des problèmes plus fondamentaux comme un manque de connexion réseau.

## 16.4. Configuration à l'exécution

Il existe un grand nombre de paramètres de configuration affectant le comportement du système de bases de données. Dans cette sous−section, nous décrivons comment configurer ces paramètres ; les sections suivantes discutent de chaque paramètre en détail.

Tous les noms de paramètres ne sont pas sensibles à la casse. Chaque paramètre prend une valeur d'un de ces quatre types : booléen, entier, nombre à virgule flottante ou chaîne de caractères. Les valeurs booléennes peuvent être saisies comme `ON`, `OFF`, `TRUE`, `FALSE`, `YES`, `NO`, `1`, `0` (quelque soit la casse) ou tout préfixe non ambigu de ceux−ci.

Une façon d'initialiser ces paramètres est d'éditer le fichier `postgresql.conf` qui est normalement placé dans le répertoire data (`initdb` y installe une copie par défaut). Voici un exemple de ce que ce fichier peut contenir :

```
# Ceci est un commentaire
log_connections = yes
log_destination = 'syslog'
search_path = '$user, public'
```

Un seul paramètre est spécifié par ligne. Le signe égal entre le nom et la valeur est optionnel. Les espaces blancs n'ont pas de signification et les lignes blanches sont ignorées. Les marques de hachage (#) introduisent des commentaires. Les valeurs des paramètres qui ne sont pas des identifieurs simples ou des nombres doivent être entre des guillemets simples.

Le fichier de configuration est relu à chaque fois que le processus `postmaster` reçoit un signal SIGHUP (qui est envoyé par un simple appel à `pg_ctl reload`). Le `postmaster` propage aussi ce signal aux processus serveur en cours d'exécution de façon à ce que les sessions existantes obtiennent aussi la nouvelle valeur. Autrement, vous pouvez envoyer le signal directement à un seul processus serveur. Quelques paramètres peuvent seulement être initialisés au lancement du serveur ; tout changement dans leur entrée du fichier de configuration sera ignoré jusqu'au prochain lancement du serveur.

Une autre façon de configurer ces paramètres est de les donner en option sur la ligne de commande de `postmaster` ainsi :

```
postmaster -c log_connections=yes -c log_destination='syslog'
```

Les options de la ligne de commande surchargent tout paramétrage en conflit avec ceux du fichier `postgresql.conf`. Notez que cela signifie que vous ne pourrez pas modifier la valeur en direct en éditant simplement le fichier `postgresql.conf`, donc bien que la méthode de la ligne de commande soit agréable, cela peut vous coûter cher plus tard en flexibilité.

Occasionnellement, il est utile de donner une option en ligne de commande à une session particulière seulement. La variable d'environnement `PGOPTIONS` peut être utilisée dans ce but du côté client :

```
env PGOPTIONS='-c geqo=off' psql
```

(Ceci fonctionne pour toute application client basée sur libpq, et non pas seulement pour psql.) Notez que ceci ne fonctionnera pas pour les paramètres fixes lorsque le serveur est lancé ou qui doivent être spécifiés dans `postgresql.conf`.

De plus, il est possible d'affecter un ensemble de paramètres à un utilisateur d'une base de données. Quand une session est lancée, les paramétrages par défaut de l'utilisateur et de la base de données impliqués sont chargés. Les commandes *ALTER USER* et *ALTER DATABASE*, respectivement, sont utilisées pour configurer ces paramétrages. Les paramètres par base de données surcharge tous ceux reçus de la ligne de commande de `postmaster` ou du fichier de configuration, et sont aussi surchargés par ceux de l'utilisateur ; les deux sont surchargés par les options par session.

Quelques paramètres peuvent être changés dans des sessions SQL individuelles avec la commande *SET*, par exemple :

```
SET ENABLE_SEQSCAN TO OFF;
```

Si `SET` est autorisé, il surcharge toutes les autres sources de valeurs pour le paramètre. Quelques paramètres ne peuvent pas être changés via `SET` : par exemple, s'ils contrôlent un comportement qui ne peut pas être changé raisonnablement sans relancer PostgreSQL. De plus, quelques paramètres peuvent être modifiés via `SET` ou `ALTER` par les superutilisateurs, mais pas par des utilisateurs ordinaires.

La commande *SHOW* permet une inspection des valeurs actuelles de tous les paramètres.

La table virtuelle pg_settings (décrite dans la Section 41.35) autorise aussi l'affichage et la mise à jour de paramètres de session à l'exécution. Elle est équivalente à SHOW et SET mais peut être plus agréable à utiliser parce qu'elle peut être jointe avec d'autres tables ou sélectionnée avec l'utilisation des conditions de sélection désirées.

## 16.4.1. Emplacement des fichiers

En plus du fichier postgresql.conf déjà mentionné, PostgreSQL utilise deux autres fichiers de configuration édités manuellement, contrôlant l'authentification du client (leur utilisation est discuté dans Chapitre 19). Par défaut, les trois fichiers de configuration sont stockés dans le répertoire data du groupe des bases de données. Les options décrites dans cette sous−section permettent de déplacer les fichiers de configuration. (Faire ceci peut faciliter l'administration. En particulier, il est souvent plus simple de s'assurer que les fichiers de configuration sont proprement sauvegardés quand ils sont conservés à part.)

data_directory (string)
: Spécifie le répertoire à utiliser pour le stockage des données. Cette option peut seulement être initialisée au lancement du serveur.

config_file (string)
: Spécifie le fichier de configuration principal du serveur (appelé postgresql.conf). Cette option peut seulement être initialisée sur la ligne de commande de postmaster.

hba_file (string)
: Spécifie le fichier de configuration pour l'authentification basée sur l'hôte (appelé pg_hba.conf). Cette option peut seulement être initialisée au lancement du serveur.

ident_file (string)
: Spécifie le fichier de configuration pour l'authentification ident (appelé pg_ident.conf). Cette option peut seulement être initialisée au lancement du serveur.

external_pid_file (string)
: Spécifie le nom d'un fichier supplémentaire par identifiant de processus (PID) que postmaster doit créer à l'intention des programmes d'administration du serveur. Cette option peut seulement être initialisée au lancement du serveur.

Dans une installation par défaut, aucune des options ci−dessus n'est configurée explicitement. À la place, le répertoire des données est spécifié par l'option −D en ligne de commande ou par la variable d'environnement PGDATA, et les fichiers de configuration sont tous disponibles dans le répertoire des données.

Si vous souhaitez conserver les fichiers de configuration ailleurs, l'option −D en ligne de commande du postmaster ou la variable d'environnement PGDATA doit pointer sur le répertoire contenant les fichiers de configuration. L'option data_directory doit être configurée à postgresql.conf (ou sur la ligne de commande) pour montrer où est situé le répertoire des données. Notez que data_directory surcharge −D et PGDATA quant à l'emplacement du répertoire des données, mais pas pour l'emplacement des fichiers de configuration.

Si vous le souhaitez, vous pouvez spécifier les noms des fichiers de configuration et leur emplacement individuellement en utilisant les options config_file, hba_file et/ou ident_file. config_file peut seulement être spécifié sur la ligne de commande de postmaster mais les autres peuvent être placés dans le fichier de configuration principal. Si les trois options et data_directory sont configurés explicitement, alors il n'est pas nécessaire de spécifier −D ou PGDATA.

Lors de la configuration de ces options, un chemin relatif sera interprété suivant le répertoire d'où est lancé postmaster.

## 16.4.2. Connexions et authentification

### 16.4.2.1. Paramétrages de connexion

`listen_addresses` (`string`)
: Spécifie les adresses TCP/IP sur lesquelles le serveur écoute les connexions des applications client. La valeur prend la forme d'une liste de noms d'hôte ou d'adresses IP numériques séparés par des virgules. L'entrée spéciale `*` correspond à toutes les interfaces IP disponibles. Si la liste est vide, le serveur n'écoute aucune interface IP, auquel cas seuls les sockets de domaine Unix peuvent être utilisés pour s'y connecter. La valeur par défaut est localhost, ce qui autorise seulement les connexions locales de type << loopback >>. Ce paramètre peut seulement être configuré au lancement du serveur.

`port` (`integer`)
: Le port TCP sur lequel le serveur écoute ; 5432 par défaut. Notez que le même numéro de port est utilisé pour toutes les adresses IP où le serveur écoute. Ce paramètre peut seulement être configuré au lancement du serveur.

`max_connections` (`integer`)
: Détermine le nombre maximum de connexions concurrentes au serveur de la base de données. La valeur par défaut typique est 100 mais pourrait être bien moindre si vos paramétrages du noyau ne le supportent pas (ce qui est déterminé lors du initdb). Ce paramètre peut seulement être initialisé au lancement du serveur.

  Augmenter ce paramètre pourrait faire que PostgreSQL réclame plus de mémoire partagée System V ou de sémaphores que ne le permet la configuration par défaut de votre système d'exploitation. Voir la Section 16.5.1 pour plus d'informations sur la façon d'ajuster ces paramètres si nécessaire.

`superuser_reserved_connections` (`integer`)
: Détermine le nombre de connexions (<< slots >>) réservés aux connexions des superutilisateurs PostgreSQL. Au plus max_connections connexions peuvent être actives simultanément. À chaque fois que le nombre de connexions actives en même temps est d'au moins `max_connections` moins `superuser_reserved_connections`, les nouvelles connexions ne seront acceptées que pour les superutilisateurs.

  La valeur par défaut est de 2. La valeur doit être plus petite que la valeur de `max_connections`. Ce paramètre peut seulement être configuré uniquement au lancement du serveur.

`unix_socket_directory` (`string`)
: Spécifie le répertoire du socket de domaine Unix sur lequel le serveur écoute les connexions des applications clients. Par défaut, il s'agit de `/tmp` mais cela peut être modifié au moment de la construction. Ce paramètre peut seulement être configuré au lancement du serveur.

`unix_socket_group` (`string`)
: Configure le groupe propriétaire du socket de domaine Unix (l'utilisateur propriétaire de la socket est toujours l'utilisateur qui lance le serveur). En combinaison avec l'option `unix_socket_permissions`, ceci peut être utilisé comme un mécanisme de contrôle d'accès supplémentaire pour les connexions de domaine Unix. Par défaut, il s'agit d'une chaîne vide utilisant le groupe par défaut pour l'utilisateur en cours. Cette option est seulement disponible au lancement du serveur.

`unix_socket_permissions` (`integer`)
: Configure les droits d'accès au socket de domaine Unix. Ce socket utilise l'ensemble habituel des droits du système de fichiers Unix. La spécification de l'option est attendue dans le mode numérique avec la forme acceptée par les appels système `chmod` et `umask` (pour utiliser le format octal, ce nombre doit commencer avec un `0` (zéro)).

Les droits par défaut sont `0777`, signifiant que tout le monde peut se connecter. Les alternatives raisonnables sont `0770` (seul l'utilisateur et le groupe, voir aussi `unix_socket_group`) et `0700` (seul l'utilisateur) (notez, que pour un socket de domaine Unix, seul le droit d'accès en écriture importe et, donc, il n'est pas nécessaire de donner ou de révoquer les droits de lecture ou d'exécution).

Ce mécanisme de contrôle d'accès est indépendant de celui décrit dans le Chapitre 19.

Cette option est seulement disponible au lancement du serveur.

`rendezvous_name` (`string`)
Spécifie le nom broadcast de Rendezvous. Par défaut, le nom de l'ordinateur est utilisé, spécifié comme une chaîne vide. Cette option est ignorée si le serveur n'était pas compilé avec le support Rendezvous. Cette option est seulement configurable au lancement du serveur.

#### 16.4.2.2. Sécurité et authentification

`authentication_timeout` (`integer`)
Temps maximum pour terminer l'authentification du client en secondes. Si un client n'a pas terminé le protocole d'authentification dans ce délai, le serveur rompt la connexion. Ceci protège le serveur des clients bloqués occupant une connexion indéfiniment. Cette option n'est configurable qu'au lancement du serveur ou dans le fichier `postgresql.conf`. La valeur par défaut est de 60 secondes.

`ssl` (`boolean`)
Active les connexions SSL. Merci de lire Section 16.8 avant d'utiliser ceci. C'est désactivé par défaut. Ce paramètre est seulement disponible au lancement du serveur.

`password_encryption` (`boolean`)
Cette option détermine si le mot de passe doit être crypté quand un mot de passe est spécifié dans *CREATE USER* ou *ALTER USER* sans écrire soit `ENCRYPTED` soit `UNENCRYPTED`. Actif par défaut (crypte le mot de passe).

`krb_server_keyfile` (`string`)
Configure l'emplacement du fichier contenant la clé secrète du serveur Kerberos. Voir Section 19.2.3 pour des détails.

`db_user_namespace` (`boolean`)
Ceci autorise les noms d'utilisateur par base de données. Désactivé par défaut.

Si cette option est activée, vous devrez créer des utilisateurs de nom `nomutilisateur@nom_base`. Quand `nomutilisateur` est passé pour un client en cours de connexion, `@` et le nom de la base de données est ajouté au nom de l'utilisateur et ce nom d'utilisateur spécifique à la base de données est recherché dans le serveur. Notez que lorsque vous créez des utilisateurs dont le nom contient un `@` dans l'environnement SQL, vous devez mettre le nom entre guillemets.

Cette option activée, vous pouvez toujours créer des utilisateurs globaux ordinaires. Ajoutez simplement `@` lors de la spécification du nom du client. Le `@` sera supprimé avant de chercher le nom de l'utilisateur dans le serveur.

> **Note :** Cette fonctionnalité est temporaire jusqu'à ce qu'une solution complète soit trouvée. Cette option sera supprimée.

## 16.4.3. Consomnation de ressources

### 16.4.3.1. Mémoire

`shared_buffers` (`integer`)
: Initialise le nombre de tampons en mémoire partagée utilisés par le serveur de bases de données. La valeur par défaut est de 1000 mais pourrait être moindre si la configuration de votre noyau ne le supporte pas (c'est déterminé lors de l'exécution d'initdb). Chaque tampon fait 8192 octets sauf si une valeur différente de `BLCKSZ` a été choisie lors de la construction du serveur. Ce paramétrage doit valoir au moins 16, mais aussi au moins deux fois la valeur de max_connections ; néanmoins, des valeurs significativement plus importantes que ce minimum sont généralement nécessaires pour de bonnes performances. Des valeurs de quelques milliers sont recommandées pour des installations de production. Cette option n'est initialisable qu'au lancement du serveur.

  Augmenter ce paramètre pourrait faire en sorte que PostgreSQL réclame plus de mémoire partagé System V que ce que la configuration par défaut de votre système d'exploitation ne peut gérer. Voir la Section 16.5.1 pour plus d'informations sur l'ajustement de ces paramètres si nécessaire.

`work_mem` (`integer`)
: Spécifie la mémoire à utiliser pour les opérations de tri interne et pour les tables de découpage avant de basculer sur des fichiers temporaires sur disque. La valeur est spécifiée en Ko et vaut par défaut 1024 (soit 1 Mo). Notez que pour une requête complexe, plusieurs tris ou opérations de hachage pourraient être exécutés en parallèle ; chacun d'entre eux se verra autorisé à utiliser autant de mémoire que cette valeur indique avant de commencer à placer des données dans des fichiers temporaires. De plus, plusieurs sessions en cours d'exécution pourraient exécuter des opérations simultanément. Donc, la mémoire totale utilisée pourrait être plusieurs fois la valeur de `work_mem` ; il est nécessaire de conserver ce fait en tête lors du choix de cette valeur. Les opérations de tri sont utilisées pour `ORDER BY`, `DISTINCT` et les jointures d'assemblage. Les tables de découpage sont utilisées dans les jointures de découpage, les agrégations basées sur le découpage et le traitement des sous−requêtes `IN`.

`maintenance_work_mem` (`integer`)
: Spécifie la mémoire maximum utilisée dans les opérations de maintenance telles que `VACUUM`, `CREATE INDEX` et `ALTER TABLE ADD FOREIGN KEY`. La valeur est spécifiée en Ko et vaut par défaut 16384 (soit 16 Mo). Comme une seule de ces opérations peut être exécutée à un moment donné sur une session de la base de données et qu'une installation n'en exécute pas beaucoup en même temps, il est possible d'initialiser cette variable à une valeur bien plus importante que `work_mem`. De gros paramétrages pourraient améliorer les performances sur les opérations VACUUM et pour la restauration des sauvegardes de bases de données.

`max_stack_depth` (`integer`)
: Spécifie la profondeur maximum de la pile d'exécution du serveur. La configuration idéale pour ce paramètre est la limite actuelle de la pile forcée par le noyau (configurée par `ulimit -s` ou une commande locale équivalente), moins une marge de sécurité d'un Mo ou plus. La marge de sécurité est nécessaire parce que la profondeur de la pile n'est pas vérifiée dans chaque routine du serveur mais seulement dans les routines clés potentiellement récursives telles que l'évaluation d'une expression. Configurer ce paramètre à une valeur plus importante que la limite réelle du noyau signifiera qu'une fonction récursive peut arrêter brutalement un processus serveur individuel. Le paramétrage par défaut est de 2048 Ko (soit 2 Mo), ce qui est très petit et comporte peu de risques. Néanmoins, cela pourrait être trop petit pour autoriser l'exécution de fonctions complexes.

#### 16.4.3.2. Carte de l'espace libre (*Free Space Map*)

`max_fsm_pages` (`integer`)
: Initialise le nombre maximum de pages disque pour lesquels les espaces libres seront tracés dans la carte partagée des espaces libres. Six octets de mémoire partagée sont consommés pour chaque emplacement de page. Ce paramétrage doit être supérieur à 16 * `max_fsm_relations`. Par défaut, il est à 20000. Cette option n'est configurable qu'au lancement du serveur.

`max_fsm_relations` (`integer`)
: Initialise le nombre maximum de relations (tables et index) pour qui les espaces libres seront tracés dans la carte partagée de l'espace libre. En gros, 50 octets de mémoire partagée sont consommés par emplacement. La valeur par défaut est de 1000. Cette option n'est configurable qu'au lancement du serveur.

---

#### 16.4.3.3. Usage des ressources du noyau

`max_files_per_process` (`integer`)
: Initialise le nombre maximum de fichiers ouverts simultanément permis pour chaque sous−processus serveur. La valeur par défaut est de 1000. Si le noyau force une limite par processus, vous n'avez pas besoin de vous inquiéter de ce paramétrage. Mais, sur certaines plateformes (notamment la plupart des systèmes BSD), le noyau autorisera des processus individuels à ouvrir beaucoup plus de fichiers que le système ne peut réellement supporter quand un grand nombre de processus essaient d'ouvrir autant de fichiers. Si vous récupérez des erreurs du type << Too many open files >> (trop de fichiers ouverts), essayez de réduire ce paramètre. Cette option peut aussi être configurée au lancement du serveur.

`preload_libraries` (`string`)
: Cette variable spécifie une ou plusieurs bibliothèques préchargées au lancement du serveur. Une fonction d'initialisation sans paramètre peut être appelée pour chaque bibliothèque. Pour cela, ajouter un caractère deux points et le nom de la fonction d'initialisation après le nom de la bibliothèque. Par exemple, `'$libdir/malib:malib_init'` causerait le préchargement de `malib` et l'exécution de `malib_init`. Si plus d'une bibliothèque doit être chargée, séparez leur nom par des virgules.

  Si une bibliothèque spécifique ou une fonction d'initialisation sont introuvables, le serveur échouera au lancement.

  Les bibliothèques des langages de procédure de PostgreSQL peuvent être préchargées de cette façon, typiquement en utilisant la syntaxe `'$libdir/plXXX:plXXX_init'` où `XXX` est soit `pgsql` soit `perl` soit `tcl` soit `python`.

  En préchargeant une bibliothèque partagée (et en l'initialisant dans les cas applicables), le temps de lancement de la bibliothèque est évité à la première utilisation de la bibliothèque. Néanmoins, le temps de lancer chaque nouveau processus pourrait augmenter légèrement même si le processus n'utilise pas la bibliothèque. Donc, cette option est seulement recommandée pour les bibliothèques qui seront utilisées par la majorité des sessions.

---

#### 16.4.3.4. Délais du VACUUM basé sur le coût

Lors de l'exécution des commandes *VACUUM* et de *ANALYZE*, le système maintient un compteur interne conservant la trace du coût estimé des différentes opérations d'entrées/sorties réalisées. Quand le coût accumulé atteint une limite (spécifiée par `vacuum_cost_limit`), le processus traitant l'opération s'arrêtera un moment (spécifié par `vacuum_cost_delay`). Puis, il réinitialisera le compteur et continuera

l'exécution.

Le but de cette fonctionnalité est d'autoriser les administrateurs à réduire l'impact des entrées/sorties de ces commandes suivant l'activité des bases de données. Il existe un grand nombre de situations pour lesquelles ceci n'est pas très important mais les commandes de maintenance quand `VACUUM` et `ANALYZE` se finissent rapidement ; néanmoins, il est généralement très important que ces commandes n'interfèrent pas de façon significative sur la capacité du système à réaliser d'autres opérations sur les bases de données. Un délai du VACUUM basé sur le coût fournit un moyen aux administrateurs pour y parvenir.

Cette fonctionnalité est désactivée par défaut. Pour l'activer, initialisez la variable `vacuum_cost_delay` à une valeur différente de zéro.

`vacuum_cost_delay` (`integer`)
: Le temps, en millisecondes, de repos du processus quand la limite de coût a été atteinte. La valeur par défaut vaut 0, ce qui désactive la fonctionnalité du délai du VACUUM basé sur le coût. Une valeur positive active cette fonctionnalité. Notez que, sur plusieurs systèmes, la résolution réelle des délais de repos est de 10 millisecondes ; configurer `vacuum_cost_delay` à une valeur qui n'est pas un multiple de 10 pourrait avoir le même résultat que de le configurer au prochain multiple de 10.

`vacuum_cost_page_hit` (`integer`)
: Le coût estimé pour nettoyer via VACUUM un tampon trouvé dans le cache des tampons partagés. Cela représente le coût pour verrouiller le lot de tampons, la recherche dans la table de découpage partagée et le parcours du contenu de la page. La valeur par défaut vaut 1.

`vacuum_cost_page_miss` (`integer`)
: Le coût estimé pour nettoyer via VACUUM un tampon qui doit être lu sur le disque. Ceci représente l'effort pour verrouiller le lot de tampons, la recherche dans la table de découpage partagée, la lecture du bloc désiré du disque et le parcours de son contenu. La valeur par défaut vaut 10.

`vacuum_cost_page_dirty` (`integer`)
: Le coût estimé chargé quand VACUUM modifie un bloc qui était précédemment propre. Cela représente les entrées/sorties supplémentaires requis pour vider le bloc sale de nouveau sur le disque. La valeur par défaut vaut 20.

`vacuum_cost_limit` (`integer`)
: Le coût accumulé qui causera l'endormissement du processus de VACUUM. La valeur par défaut vaut 200.

  **Note :** Certaines opérations contenant des verrous critiques devraient se terminer aussi rapidement que possible. Les délais de VACUUM basés sur le coût ne surviennent pas pendant ces opérations. Du coup, il est possible que le coût accumulé soit bien plus important que la limite spécifiée. Pour éviter des délais inutilement longs dans de tels cas, le délai réel est calculé de la façon suivante : `vacuum_cost_delay` * `accumulated_balance` / `vacuum_cost_limit` avec un maximum de `vacuum_cost_delay` * 4.

### 16.4.3.5. Écriture en tâche de fond

À partir de PostgreSQL 8.0, il existe un processus serveur séparé pour l'*écriture en tâche de fond*, dont la seule fonction est de lancer les écritures des tampons partagés << sales >>. Le but est que les processus serveur gérant les requêtes des utilisateurs doivent déléguer ou n'avoir jamais à attendre la fin d'une écriture car le processus d'écriture en tâche de fond s'en chargera. Cet arrangement réduit aussi la pénalité de performance associée avec les points de vérification. Le processus d'écriture en tâche de fond vérifiera en contenu les pages sales pour les écrire sur le disque, de façon à ce que seules quelques pages doivent être forcées en écriture lorsque survient le point de vérification, à la place d'un déluge d'écritures de tampons sales ne se faisant qu'à chaque point de vérification. Néanmoins, il y a une nette augmentation dans la charge des

entrées/sorties parce que, là où une page souvent sale n'aurait été écrite qu'une seule fois par intervalle de point de vérification, le processus d'écriture en tâche de fond l'aurait écrite plusieurs fois dans la même période. Dans la plupart des situations, un chargement lent continu est préférable à des pointes périodiques mais les paramètres discutés dans cette section peuvent être utilisés pour configurer finement le comportement pour les besoins locaux.

`bgwriter_delay` (`integer`)
: Spécifie le délai entre les tours d'activité du processus d'écriture en tâche de fond. À chaque tour, le processus écrit un certain nombre de tampons sales (contrôlable par les paramètres suivants). Les tampons sélectionnées seront toujours ceux les moins récemment utilisés parmi les tampons sales en cours. Puis, il s'endort pour `bgwriter_delay` millisecondes et recommande. La valeur par défaut est de 200. Notez que sur de nombreux systèmes, la résolution réelle des délais de sommeil est de 10 millisecondes; configurer `bgwriter_delay` à une valeur qui n'est pas un multiple de 10 pourrait avoir le même résultat que de le configurer au multiple de 10 supérieur. Cette option est seulement disponible au lancement du serveur ou dans le fichier `postgresql.conf`.

`bgwriter_percent` (`integer`)
: À chaque tour, pas plus que ce pourcentage de tampons sales ne seront écrits (ne terminant tout fraction sur le prochain nombre de tampons). La valeur par défaut est 1. Cette option est seulement disponible au lancement du serveur ou dans le fichier `postgresql.conf`.

`bgwriter_maxpages` (`integer`)
: À chaque tour, pas plus que ce nombre de tampons partagés sera écrit. La valeur par défaut vaut 100. Cette option est seulement disponible au lancement du serveur ou dans le fichier `postgresql.conf`.

Des valeurs plus petites de `bgwriter_percent` et `bgwriter_maxpages` réduiront la charge supplémentaire en entrées/sorties causée par le processus d'écriture en tâche de fond mais laisseront plus de travail au point de vérification. Pour réduire les pointes de chargements aux points de vérification, augmentez les valeurs. Pour désactiver totalement le processus en d'écriture en tâche de fond, configurez `bgwriter_percent` et/ou `bgwriter_maxpages` à zéro.

## 16.4.4. Write Ahead Log

Voir aussi la Section 25.2 pour des détails sur la configuration pointue des WAL.

### 16.4.4.1. Paramétrages

`fsync` (`boolean`)
: Si cette option est activée, le serveur PostgreSQL utilisera l'appel système `fsync()` à plusieurs endroits pour s'assurer que les mises à jour sont écrites physiquement sur le disque. Ceci vous assure que le groupe de bases de données retournera à un état cohérent après un arrêt brutal du système d'exploitation ou suite à un problème matériel.

  Néanmoins, utiliser `fsync()` implique des coûts au niveau performance : lorsqu'une transaction est validée, PostgreSQL doit attendre que le système d'exploitation vide les WAL sur disque. Lorsque `fsync` est désactivé, le système d'exploitation est autorisé à faire de son mieux en utilisant des tampons pour les écritures, en ordonnant et en ajoutant des délais aux écritures. Néanmoins, si le système s'arrête brutalement, les résultats des dernières transactions validées pourraient être perdus en partie ou complètement. Dans le pire des cas, une corruption non récupérable des données pourrait survenir (les arrêts brutaux du serveur de bases de données lui−même ne sont *pas* un facteur de risque ici. Seul l'arrêt brutal au niveau du système d'exploitation crée un risque de corruption).

À cause des risques encourus, il n'existe pas de paramétrage universel pour `fsync`. Certains administrateurs désactivent en permanence `fsync` alors que d'autres ne le désactivent que pour les charges importantes s'il existe un moyen de recommencer proprement si quelque chose se passe mal. Mais d'autres administrateurs laissent toujours `fsync` activé. La valeur par défaut est d'activer `fsync` pour une confiance maximale. Si vous avez confiance en votre système d'exploitation (ou sur la sauvegarde sur batterie), vous pouvez considérer la désactivation de `fsync`.

Cette option n'est configurable qu'au lancement du serveur ou dans le fichier `postgresql.conf` file.

`wal_sync_method` (`string`)
: Méthode utilisée pour forcer les mises à jour des WAL sur le disque. Les valeurs possibles sont `fsync` (appel de `fsync()` à chaque validation), `fdatasync` (appel de `fdatasync()` à chaque validation), `fsync_writethrough` (appel de `_commit()` à chaque validation sur Windows), `open_sync` (écriture des fichiers WAL avec l'option `O_SYNC` de `open()`) et `open_datasync` (écriture des fichiers WAL avec l'option `O_DSYNC` de `open()`). Toutes ces options ne sont pas disponibles sur toutes les plateformes. Si `fsync` est désactivée, alors cette configuration n'a pas d'intérêt. Cette option n'est configurable qu'au lancement du serveur ou dans le fichier `postgresql.conf`.

`wal_buffers` (`integer`)
: Nombre de tampons de pages disque alloués en mémoire partagée pour les données WAL. La valeur par défaut vaut 8. Ce paramétrage a seulement besoin d'être assez important pour contenir toutes les données WAL générées par une transaction typique. Cette option est seulement disponible au lancement du serveur.

`commit_delay` (`integer`)
: Délai entre l'enregistrement d'une validation dans le tampon WAL et le vidage du tampon sur le disque, en microsecondes. Un délai différent de zéro peut autoriser la validation de plusieurs transactions sur un seul appel système `fsync()` si la charge système est assez importante pour que des transactions supplémentaires soient prêtes dans l'intervalle donné. Mais le délai est perdu si aucune autre transaction n'est prête à être validée. Du coup, le délai est seulement traitée si au moins `commit_siblings` autres transactions sont actives au moment où le processus serveur a écrit son enregistrement de validation. La valeur par défaut est zéro, donc pas de délai.

`commit_siblings` (`integer`)
: Nombre minimum de transactions ouvertes en même temps avant d'attendre le délai `commit_delay`. Une valeur plus importante rend plus probable le fait qu'au moins une autre transaction sera prête à valider pendant la durée du délai. La valeur par défaut est de cinq.

---

#### 16.4.4.2. Points de vérification

`checkpoint_segments` (`integer`)
: Distance maximum entre des points de vérifications automatiques des WAL, dans les segments des journaux de traces (chaque segment fait normalement 16 Mo). Par défaut, il y en a trois. Cette option n'est configurable qu'au lancement du serveur ou dans le fichier `postgresql.conf`.

`checkpoint_timeout` (`integer`)
: Temps maximum entre des points de vérification automatiques des WAL en secondes. La valeur par défaut est de 300 secondes. Cette option n'est configurable qu'au lancement du serveur ou dans le fichier `postgresql.conf`.

`checkpoint_warning` (`integer`)
: Écrit un message dans les journaux du serveur si les points de vérification causées par le remplissage des fichiers segment sont arrivés dans un intervalle plus rapide que ce nombre de secondes. Vaut par défaut 30 secondes. Une valeur de zéro désactive cet avertissement.

#### 16.4.4.3. Archivage

`archive_command` (`string`)
: La commande shell pour exécuter l'archivage d'un segment complet de fichier WAL. Si cette chaîne est vide (valeur par défaut), l'archivage des WAL est désactivé. Tout `%p` dans la chaîne est remplacé par le chemin absolu vers le fichier à archiver et tout `%f` est seulement remplacé par le nom du fichier. Utilisez `%%` pour intégrer un vrai caractère `%` dans la commande. Pour plus d'informations, voir la Section 22.3.1. Cette option est seulement disponible au lancement du serveur ou dans le fichier `postgresql.conf`.

Il est important que la commande renvoie un code de sortie zéro si et seulement si elle a réussi. Exemples :

```
archive_command = 'cp "%p" /mnt/server/archivedir/"%f"'
archive_command = 'copy "%p" /mnt/server/archivedir/"%f"'  # Windows
```

## 16.4.5. Planification des requêtes

#### 16.4.5.1. Configuration de la méthode du planificateur

Ces paramètres de configuration fournissent une méthode dure pour influencer les plans de requête choisis par l'optimiseur de requêtes. Si le plan choisi par défaut par l'optimiseur pour une requête particulière n'est pas optimale, une solution temporaire pourrait être découverte en utilisant un de ces paramètres de configuration pour forcer l'optimiseur à choisir un meilleur plan. Désactiver un de ces paramétrages de façon permanente est néanmoins quelque fois une bonne idée. De meilleures façons d'améliorer la qualité des plans choisis par l'optimiseur incluent l'ajustement de *Constantes de coût du planificateur*, le lancement plus fréquent de *ANALYZE*, l'augmentation de la valeur du paramètre de configuration default_statistics_target et l'augmentation du nombre de statistiques récupérées pour des colonnes spécifiques en utilisant `ALTER TABLE SET STATISTICS`.

`enable_hashagg` (`boolean`)
: Active ou désactive l'utilisation des agrégats hachés par le planificateur. Actif par défaut.

`enable_hashjoin` (`boolean`)
: Active ou désactive l'utilisation des jointures hachées par le planificateur. Actif par défaut.

`enable_indexscan` (`boolean`)
: Active ou désactive l'utilisation des parcours d'index par le planificateur. Actif par défaut.

`enable_mergejoin` (`boolean`)
: Active ou désactive l'utilisation des jointures de fusion par le planificateur. Actif par défaut.

`enable_nestloop` (`boolean`)
: Active ou désactive l'utilisation des jointures de boucles imbriquées par le planificateur. Il n'est pas possible de supprimer les jointures de boucles imbriquées complètement mais désactiver cette variable décourage le planificateur de l'utiliser si d'autres méthodes sont disponibles. Actif par défaut.

`enable_seqscan` (`boolean`)
: Active ou désactive l'utilisation des parcours séquentiel par le planificateur. Il n'est pas possible de supprimer complètement les parcours séquentiels mais désactiver cette variable décourage le planificateur de l'utiliser si d'autres méthodes sont disponibles. Actif par défaut.

`enable_sort` (`boolean`)
: Active ou désactive l'utilisation des étapes de tri explicite par le planificateur. Il n'est pas possible de supprimer complètement ces tris mais désactiver cette variable décourage le planificateur de l'utiliser

si d'autres méthodes sont disponibles. Actif par défaut.

`enable_tidscan` (`boolean`)
Active ou désactive l'utilisation des parcours de TID par le planificateur. Actif par défaut.

### 16.4.5.2. Constantes de coût du planificateur

**Note :** Malheureusement, il n'existe pas de méthode bien définie pour déterminer les valeurs idéales pour la famille des variables de coût (<< cost >>) qui apparaissent ci−dessous. Vous êtes encouragés à expérimenter et à partager vos découvertes.

`effective_cache_size` (`floating point`)
Configure l'idée du planificateur sur la taille réelle du cache disque disponible pour un simple parcours d'index. Ceci est factorisé en estimation du coût d'utilisation d'un index ; une valeur plus grande rend un parcours d'index plus probable, une valeur plus basse favorise le parcours séquentiel. Lors de la modification de ce paramètre, vous devez considérer les tampons partagés de PostgreSQL et la partie cache disque du noyau qui seront utilisés pour les fichiers de données de PostgreSQL. De plus, prenez en compte le nombre attendu de requêtes concurrentes utilisant différents index car elles devront partager l'espace disponible. Ce paramètre n'a pas d'effet sur la taille de la mémoire partagée allouée par PostgreSQL, pas plus qu'il ne réserve de cache disque du noyau ; c'est utilisé uniquement dans un but d'estimation. La valeur est mesurée en pages disque, qui sont normalement de 8192 octets chaque. La valeur par défaut est de 1000.

`random_page_cost` (`floating point`)
Initialise l'estimation du coût du planificateur pour une page disque récupérée de façon non séquentielle. C'est mesuré comme un multiple du coût de récupération d'une page séquentielle. Une valeur plus haute rend plus probable l'utilisation d'un parcours séquentiel, une valeur basse l'utilisation d'un parcours d'index. La valeur par défaut est quatre.

`cpu_tuple_cost` (`floating point`)
Initialise l'estimation du coût du planificateur pour le traitement de chaque ligne lors d'une requête. C'est mesuré comme une fraction du coût de la récupération séquentielle d'une page. La valeur par défaut est 0,01.

`cpu_index_tuple_cost` (`floating point`)
Initialise l'estimation du coût du planificateur pour le traitement de chaque ligne lors d'un parcours d'index. C'est mesuré comme une fraction du coût de la récupération séquentielle d'une page. La valeur par défaut est 0,001.

`cpu_operator_cost` (`floating point`)
Initialise l'estimation du coût du planificateur de requêtes pour le traitement de chaque opérateur dans une clause `WHERE`. C'est mesuré comme une fraction du coût de récupération séquentielle d'une page. La valeur par défaut est 0,025.

### 16.4.5.3. Optimiseur génétique de requêtes

`geqo` (`boolean`)
Active ou désactive l'optimisation génétique des requêtes, algorithme tentant de faire de la planification de requêtes sans recherche exhaustive. Activé par défaut. La variable `geqo_threshold` fournit un moyen plus fin pour désactiver certaines classes de requêtes.

`geqo_threshold` (`integer`)
Utilise l'optimisation génétique des requêtes pour planifier les requêtes avec au moins ce nombre d'éléments impliqués dans la clause `FROM` (notez qu'une construction `JOIN` externe compte seulement comme un élément du `FROM`). La valeur par défaut est de 12. Pour des requêtes simples, il est généralement préférable d'utiliser le planificateur déterministe, exhaustif mais pour les requêtes

comprenant beaucoup de tables, le planificateur déterministe prendrait trop de temps.

`geqo_effort` (`integer`)

Contrôle l'équité entre le temps de planification et l'efficacité du plan de requête dans GEQO. Cette variable doit être un entier entre 1 et 10. La valeur par défaut est 5. Les valeurs plus importantes augmentent le temps passé pendant la planification de la requête mais augmentent aussi la possibilité qu'un plan de requête efficace soit choisi.

`geqo_effort` ne fait réellement rien directement ; c'est seulement utilisé pour calculer les valeurs par défaut des autres variables qui influencent le comportement de GEQO (décrit ci−dessous). Si vous préférez, vous pouvez configurer les autres paramètres manuellement.

`geqo_pool_size` (`integer`)

Contrôle la taille de la queue utilisée par GEQO. Cette taille est le nombre d'individus dans une population génétique. Elle doit être d'au moins deux, et les valeurs utiles sont typiquement 100 et 1000. Si elle est configurée à zéro (la valeur par défaut), alors une valeur par défaut convenable est choisie suivant `geqo_effort` et le nombre de tables dans la requête.

`geqo_generations` (`integer`)

Contrôle le nombre de générations utilisé par GEQO. Les générations spécifient le nombre d'itérations de l'algorithme. Il doit être au moins un, et les valeurs utiles sont dans la même échelle que la taille de la queue. S'il est configuré à zéro (la valeur par défaut), alors une version par défaut convenable est choisie suivant `geqo_pool_size`.

`geqo_selection_bias` (`floating point`)

Contrôle le biais de sélection utilisé par GEQO. C'est une pression sélective à l'intérieur de la population. Les valeurs peuvent aller de 1,50 à 2,00 ; ce dernier est la valeur par défaut.

---

#### 16.4.5.4. Autres options du planificateur

`default_statistics_target` (`integer`)

Initialise la cible par défaut des statistiques pour les colonnes de table qui n'ont pas une cible spécifique de colonne configurée via `ALTER TABLE SET STATISTICS`. Des valeurs plus importantes accroissent le temps nécessaire à exécuter `ANALYZE` mais pourrait améliorer les estimations du planificateurs. La valeur par défaut est de 10. Pour plus d'informations sur l'utilisation des statistiques par le planificateur de requêtes de PostgreSQL, référez−vous à la Section 13.2.

`from_collapse_limit` (`integer`)

Le planificateur assemblera les sous−requêtes dans des requêtes supérieures si la liste `FROM` résultante n'aurait pas plus de ce nombre d'éléments. Des valeurs plus petites réduisent le temps de planification mais ramènent des plans de requêtes inférieurs. La valeur par défaut est huit. Il est généralement conseillé de conserver cette valeur inférieure à geqo_threshold.

`join_collapse_limit` (`integer`)

Le planificateur réécrira les constructions `JOIN` internes explicites en listes d'éléments `FROM` à chaque fois qu'une liste d'au moins ce nombre d'éléments au total en résulterait. Avant PostgreSQL 7.4, les jointures spécifiées via la construction `JOIN` ne seraient jamais réordonnées. Le planificateur de la requête a du coup été amélioré pour que les jointures internes écrites de cette forme puissent être ré−ordonnées ; ce paramètre de configuration contrôle à quel point ce ré−ordonnancement sera réalisé.

> **Note :** À présent, l'ordre des jointures externes spécifiée via la construction `JOIN` n'est jamais ajusté par le planificateur de requêtes ; du coup, `join_collapse_limit` n'a aucun effet sur ce comportement. Le planificateur pourrait être amélioré pour ré−ordonner certaines classes de jointures externes dans une prochaine version de PostgreSQL.

Par défaut, cette variable est configurée de la même façon que `from_collapse_limit`, qui est approprié pour la plupart des utilisations. Configurer cette variable à 1 empêche le réordonnancement des `JOIN`tures internes. Du coup, l'ordre de jointure explicite spécifiée dans la requête sera l'ordre réel dans lequel les relations sont jointes. Le planificateur de la requête ne choisit pas toujours l'ordre de jointure optimal ; les utilisateurs avancées pourraient choisir d'initialiser temporairement cette variable à 1, puis de spécifier l'ordre de jointure qu'ils désirent explicitement. Une autre conséquence en l'initialisant à 1 est que le planificateur de requêtes se comporte plus comme le planificateur de requête de PostgreSQL 7.3, ce que certains utilisateurs trouvent utile pour des raisons de compatibilités descendantes.

Configurer cette variable à une valeur entre 1 et `from_collapse_limit` pourrait être utile pour négocier entre le temps de planification et la qualité du plan choisie (les grandes valeurs produisent les meilleures plans).

## 16.4.6. Rapports d'erreur et traces

### 16.4.6.1. Où tracer

`log_destination` (`string`)
: PostgreSQL supporte plusieurs méthodes pour la journalisation des messages du serveur, dont stderr et syslog. Sur Windows, eventlog est aussi supporté. Configurez cette option avec une liste de destinations désirées pour les journaux, séparées par des virgules. Par défaut, les traces vont seulement sur stderr. Cette option est seulement disponible au lancement du serveur et dans le fichier de configuration de `postgresql.conf`.

`redirect_stderr` (`boolean`)
: Cette option autorise la capture et la redirection des messages envoyés vers stderr dans des journaux de traces. Cette option, en combinaison avec les traces stderr, est souvent plus utile que de tracer dans syslog car certains types de messages pourraient ne pas apparaître dans la sortie syslog (un exemple habituel concerne les messages d'échecs de l'éditeur de liens). Cette option est seulement disponible au lancement du serveur.

`log_directory` (`string`)
: Quand `redirect_stderr` est activé, cette option détermine le répertoire dans lequel les fichiers de trace seront créés. Elle pourrait être spécifiée avec un chemin absolu ou relatif au répertoire des données du groupe. Cette option est seulement disponible au lancement du serveur ou dans le fichier `postgresql.conf`.

`log_filename` (`string`)
: Quand `redirect_stderr` est activé, cette option initialise les noms des fichiers des journaux créés. La valeur est traitée comme un modèle strftime, du coup les échappements `%` peuvent être utilisés pour spécifier des noms de fichiers dépendant de l'heure. Si aucun échappement `%` n'est présent, PostgreSQL ajoutera l'époque de l'heure d'ouverture du nouveau journal. Par exemple, si `log_filename` valait `server_log`, alors le nom de fichier choisi serait `server_log.1093827753` pour un journal commençant le dimanche 29 août 19:02:33 2004 MST. Cette option est seulement disponible au lancement du serveur ou dans le fichier de configuration `postgresql.conf`.

`log_rotation_age` (`integer`)
: Quand `redirect_stderr` est activé, cette option détermine la durée de vie maximum d'un journal individuel. Après que cette durée se soit écoulée, un nouveau journal sera créé. Initialisez–la à zéro pour désactiver la création des nouveaux journaux suivant un délai. Cette option est seulement

disponible au lancement du serveur ou dans le fichier de configuration `postgresql.conf`.

`log_rotation_size` (`integer`)

Quand `redirect_stderr` est activé, cette option détermine la taille maximum d'un journal individuel. Après ce nombre d'octets, un nouveau journal est créé. Initialiser cette taille à zéro désactive la création de nouveaux journaux suivant leur taille. Cette option est seulement disponible au lancement du serveur ou dans le fichier de configuration `postgresql.conf`.

`log_truncate_on_rotation` (`boolean`)

Quand `redirect_stderr` est activé, cette option fera que PostgreSQL tronque (surcharge), plutôt qu'il n'ajoute à un journal de même nom. Néanmoins, ce tronquage ne surviendra qu'à partir du moment où un nouveau fichier sera ouvert à cause d'une rotation basée sur le temps, et non pas suite au lancement du serveur ou suite à une rotation basée sur la taille. Si false, les fichiers déjà existants se verront ajoutés les nouvelles traces dans tous les cas. Par exemple, en utilisant cette option en combinaison avec un `log_filename` comme `postgresql-%H.log` résulterait en la génération de 24 journaux (un pour chaque heure) puis les surchargera cycliquement. Cette option est seulement disponible au lancement du serveur ou dans le fichier de configuration `postgresql.conf`.

Exemple : pour conserver sept jours de traces, un fichier par jour nommé `server_log.Mon`, `server_log.Tue`, etc. et surcharger automatiquement les traces de la semaine dernière avec ceux de cette semaine, configurez `log_filename` à `server_log.%a`, `log_truncate_on_rotation` à `true` et `log_rotation_age` à `1440`.

Exemple : pour conserver 24 heures de traces, un journal par heure mais aussi, faire une rotation plus souvent si le journal dépasse 1 Go, configurez `log_filename` à `server_log.%H%M`, `log_truncate_on_rotation` à `true`, `log_rotation_age` à `60` et `log_rotation_size` à `1000000`. Inclure `%M` dans `log_filename` permet des rotations conduites par la taille qui pourraient survenir pour sélectionner un nom de fichier différent du nom de fichier initial.

`syslog_facility` (`string`)

Lors que les traces syslog sont activées, cette option détermine le niveau (<< facility >>) utilisé par syslog. Vous pouvez choisir entre `LOCAL0`, `LOCAL1`, `LOCAL2`, `LOCAL3`, `LOCAL4`, `LOCAL5`, `LOCAL6`, `LOCAL7` ; la valeur par défaut est `LOCAL0`. Voir aussi la documentation du démon syslog de votre serveur. Cette option ne se configure qu'au lancement du système.

`syslog_ident` (`string`)

Si syslog est activé, cette option détermine le nom du programme utilisé pour identifier les messages de PostgreSQL dans les journaux de traces de syslog. La valeur par défaut est `postgres`. Cette option ne se configure qu'au lancement du système.

---

#### 16.4.6.2. Quand tracer

`client_min_messages` (`string`)

Contrôle les niveaux des messages envoyés au client. Les valeurs valides peuvent être `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `LOG`, `NOTICE`, `WARNING` et `ERROR`. Chaque niveau inclut tous les niveaux qui le suivent. La valeur par défaut est `NOTICE`. Notez que `LOG` a un niveau différent que dans `log_min_messages`.

`log_min_messages` (`string`)

Contrôle les niveaux des messages écrits dans les journaux de traces. Les valeurs valides sont `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` et `PANIC`. Chaque niveau inclut tous les niveaux qui le suivent. Le niveau le plus bas obtient le plus petit nombre de messages. La valeur par défaut est `NOTICE`. Notez que `LOG` a un niveau différent que dans `client_min_messages`. Seuls les superutilisateurs peuvent modifier ce paramétrage.

`log_error_verbosity` (`string`)

Contrôle le nombre de détails écrits dans les journaux de traces pour chaque message tracé. Les valeurs valides sont `TERSE`, `DEFAULT` et `VERBOSE`, chacune ajoutant plus de champs aux messages affichés. Seuls les superutilisateurs peuvent modifier ce paramétrage.

`log_min_error_statement` (`string`)
Contrôle si l'instruction SQL ayant causé une erreur sera aussi enregistrée dans le journal des traces. Toutes les instructions SQL, qui causent une erreur du niveau spécifié ou d'un niveau plus haut, seront tracées. La valeur par défaut est `PANIC` (désactivant réellement cette fonctionnalité en production). Les valeurs valides sont `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `FATAL` et `PANIC`. Par exemple, si vous l'initialisez à `ERROR`, alors toutes les instructions SQL causant des erreurs, fatales ou non, ou des paniques seront tracées. Activer cette option est utile pour localiser la source de toute erreur apparaissant dans le journal des traces. Seuls les superutilisateurs peuvent modifier ce paramétrage.

`log_min_duration_statement` (`integer`)
Initialise un temps d'exécution minimum (en millisecondes) pour que l'instruction soit tracée. Toutes les instructions SQL exécutées dans le temps imparti ou prenant plus de temps seront tracées avec leur durée. L'initialiser à zéro tracera toutes les requêtes avec leur durée. −1 (la valeur par défaut) désactive cette fonctionnalité. Par exemple, si vous la configurez à `250`, toutes les instructions SQL s'exécutant en au moins 250 ms seront tracées. Activer cette option peut se révéler utile pour tracer les requêtes non optimisées dans vos applications. Seuls les superutilisateurs peuvent modifier ce paramétrage.

`silent_mode` (`boolean`)
Exécute le serveur silencieusement. Si cette option est configurée, le serveur se lancera automatiquement en tâche de fond et tout terminal de contrôle sera dissocié (même effet que l'option `-S` de `postmaster`). La sortie standard et la sortie standard des erreurs du serveur sont redirigées vers `/dev/null`, donc tout message qui leur est adressé sera perdu. Sauf si le journal syslog est sélectionné ou que `redirect_stderr` est activé, utiliser cette option n'est pas encouragé car il empêche de voir les messages d'erreurs.

Voici une liste des niveaux de sévérité utilisés dans ces paramétrages :

`DEBUG[1-5]`
Fournit des informations utiles aux développeurs.

`INFO`
Fournit des informations implicitement demandées par l'utilisateur, par exemple lors d'un `VACUUM VERBOSE`.

`NOTICE`
Fournit des informations qui pourraient être utiles aux utilisateurs, par exemple lors du tronquage d'identifiants longs ou la création d'index faisant partie de clés primaires.

`WARNING`
Fournit des avertissements aux utilisateurs, par exemple un `COMMIT` en dehors d'un bloc de transactions.

`ERROR`
Rapporte une erreur qui a annulé la commande en cours.

`LOG`
Rapporte des informations intéressant les administrateurs, par exemple l'activité des points de vérification.

`FATAL`
Rapporte une erreur qui a causé l'annulation de la session courante.

`PANIC`
Rapporte une erreur qui a causé l'annulation de toutes les sessions.

#### 16.4.6.3. Que tracer

`debug_print_parse` (`boolean`)
`debug_print_rewritten` (`boolean`)
`debug_print_plan` (`boolean`)
`debug_pretty_print` (`boolean`)
Ces options activent plusieurs sorties de débogage. Pour chaque requête exécutée, elles affichent l'arbre d'analyse résultant. `debug_pretty_print` indente ces affichages pour produire un format de sortie plus lisible mais plus long. `client_min_messages` ou `log_min_messages` doivent valoir `DEBUG1` ou plus bas pour réellement envoyer cette sortie vers le client ou les traces du serveur, respectivement. Ces options sont désactivées par défaut.

`log_connections` (`boolean`)
Ceci affiche une ligne dans les traces du serveur détaillant chaque connexion réussie. Désactivée par défaut, elle est probablement très utile. Cette option peut seulement être configurée au lancement du serveur ou dans le fichier de configuration `postgresql.conf`.

`log_disconnections` (`boolean`)
Ceci affiche dans les journaux du serveur une ligne similaire à `log_connections` mais à la fin d'une session et inclut la durée de la session. Elle est désactivée par défaut. Cette option est seulement disponible au lancement et dans le fichier de configuration `postgresql.conf`.

`log_duration` (`boolean`)
Fait que la durée d'une instruction terminée satisfaisant `log_statement` est tracée. En utilisant cette option, si vous n'utilisez pas syslog, il est recommandé que vous traciez le PID ou l'ID de la session en utilisant `log_line_prefix`, de façon à ce que vous puissiez lier l'instruction à la durée en utilisant l'ID du processus ou de la session. Cette variable est désactivée par défaut. Seuls les superutilisateurs peuvent modifier ce paramétrage.

`log_line_prefix` (`string`)
Ceci est une chaîne style `printf` qui est affichée au début de chaque ligne de trace. La valeur par défaut est une chaîne vide. Chaque échappement reconnu est remplacé comme indiqué ci−dessous − tout ce qui reste et qui ressemble à un échappement est ignoré. Les autres caractères sont copiés directement sur la ligne. Certains échappements sont seulement reconnus par les processus de session et ne s'appliquent pas aux processus en tâche de fond comme le postmaster. Syslog produit son propre marquage horaire et informations d'ID sur le processus, donc vous ne voudrez probablement pas utiliser ces échappements si vous utilisez syslog. Cette option est seulement disponible au lancement du serveur et dans le fichier de configuration `postgresql.conf`.

| Échappement | Effet | Session seulement |
|---|---|---|
| `%u` | Nom de l'utilisateur | oui |
| `%d` | Nom de la base de données | oui |
| `%r` | Nom ou adresse IP de l'hôte distant et port distant | oui |
| `%p` | ID du processus | non |
| `%t` | Marquage horaire | non |
| `%i` | Balise de commande : la commande qui a généré cette trace. | oui |
| `%c` | ID de session : un identifiant unique pour chaque session. Ce sont deux numéros hexadécimaux sur quatre octets (sans zéros devant) séparés par un point. Les nombres sont l'heure de début de la session et son ID de processus, donc ceci peut aussi être utilisé comme moyen de sauvegarde pour ces éléments. | oui |

| | | |
|---|---|---|
| %l | Numéro de la ligne de traces pour chaque processus, commençant à 1 | non |
| %s | Marquage horaire du début de session | oui |
| %x | ID de la transaction | oui |
| %q | Ne produit aucune sortie, mais indique aux autres processus de stopper à cet endroit dans la chaîne. Ignoré par les processus de session. | non |
| %% | Littéral % | non |

log_statement (string)
: Contrôle les instructions SQL tracées. Les valeurs valides sont none, ddl, mod et all. ddl trace toutes les commandes de définition comme CREATE, ALTER et DROP. mod trace toutes les instructions ddl, plus INSERT, UPDATE, DELETE, TRUNCATE et COPY FROM. Les instructions PREPARE et EXPLAIN ANALYZE sont aussi tracées si leur commande contenue est d'un type approprié.

La valeur par défaut est none. Seuls les superutilisateurs peuvent changer ce paramétrage.

**Note :** L'instruction EXECUTE n'est pas considérée comme une instruction ddl ou mod. Quand elle est tracée, seul le nom de l'instruction préparée est rapporté, pas l'instruction préparée réelle.

Quand une fonction est définie dans le langage côté serveur PL/pgSQL, toute requête exécutée par la fonction sera seulement tracée la première fois que la fonction est appelée d'une session particulière. Ceci est dû au fait que PL/pgSQL conserve un cache des plans de requête produits par les instructions SQL dans la fonction.

log_hostname (boolean)
: Par défaut, les messages de traces de connexion affichent seulement l'adresse IP de l'hôte se connectant. Activer cette option causera la trace du nom de l'hôte. Notez que, suivant votre configuration de résolution de nom d'hôte, ceci pourrait imposer une pénalité de performance non négligeable. Cette option est seulement disponible au lancement du serveur ou dans le fichier postgresql.conf.

## 16.4.7. Statistiques d'exécution

### 16.4.7.1. Surveillance des statistiques

log_statement_stats (boolean)
log_parser_stats (boolean)
log_planner_stats (boolean)
log_executor_stats (boolean)
: Pour chaque requête, écrit les statistiques de performance du module respectif dans les journaux de trace. Ceci est un outil brut de profilage. log_statement_stats rapporte les statistiques totales sur les instructions alors que les autres rapportent des statistiques par module. log_statement_stats ne peut pas être activé avec toute option par module. Toutes ces options sont désactivées par défaut. Seuls les superutilisateurs peuvent modifier ces paramétrages.

### 16.4.7.2. Collecteur des statistiques sur les requêtes et les index

stats_start_collector (boolean)
: Contrôle si le serveur doit lancer le sous−processus de récupération de statistiques. Il est activé par défaut mais pourrait être enlevé si vous n'avez aucun intérêt dans la récupération de statistiques.

`stats_command_string` (`boolean`)
: Active la récupération de statistiques sur les commandes en cours d'exécution par chaque session, avec le moment de l'exécution de la commande. Cette option est désactivée par défaut. Notez que, même après son activation, cette information n'est pas visible par tous les utilisateurs, mais seulement par les superutilisateurs et l'utilisateur possédant la session. Donc, cela ne devrait pas représenter un risque de sécurité. La donnée est accessible via la vue système `pg_stat_activity` ; référez−vous au Chapitre 23 pour plus d'informations.

`stats_block_level` (`boolean`)
: Active la récupération des statistiques au niveau bloc sur l'activité de la base de données. Cette option est désactivée par défaut. Si cette option est activée, les données produites sont accessibles via la famille de vues système `pg_stat` et `pg_statio` ; référez−vous au Chapitre 23 pour plus d'informations.

`stats_row_level` (`boolean`)
: Active la récupération de statistiques au niveau ligne sur l'activité de la base de données. Cette option est désactivée par défaut. Si cette option est activée, les données produites sont accessibles via la famille de vues système `pg_stat` et `pg_statio` ; référez−vous au Chapitre 23 pour plus d'informations.

`stats_reset_on_server_start` (`boolean`)
: Si elle est activée, les statistiques récupérées sont vidées à chaque fois que le serveur est redémarré. Dans le cas contraire, les statistiques sont cumulées après les redémarrages de serveur. Actif par défaut. Cette option peut seulement être configuré au lancement du serveur.

## 16.4.8. Valeurs par défaut des connexions client

### 16.4.8.1. Comportement des instructions

`search_path` (`string`)
: Cette variable spécifie l'ordre dans lequel les schémas sont recherchés lorsqu'un objet (table, type de données, fonction, etc.) est référencé par un simple nom sans son composant schéma. Quand il existe des noms identiques dans différents schémas, le premier trouvé dans le chemin de recherche est utilisé. Un objet qui ne fait partie d'aucun des schémas du chemin de recherche peut seulement être référencé en spécifiant son schéma conteneur avec un nom qualifié.

  La valeur de `search_path` doit être une liste de noms de schémas séparés par des virgules. Si un des éléments de la liste est la valeur spéciale `$user`, alors le schéma ayant le nom renvoyé par `SESSION_USER` est substitué, si un tel schéma existe (sinon `$user` est ignoré).

  Le schéma du catalogue système, `pg_catalog`, est toujours recherché, qu'il soit ou non mentionné dans le chemin. S'il est mentionné, alors il sera cherché dans l'ordre spécifié. Si `pg_catalog` ne fait pas partie du chemin, alors il sera cherché *avant* tout élément du chemin. Il devrait aussi être noté que le schéma des tables temporaires, `pg_temp_`*`nnn`*, est cherché implicitement avant tout autre.

  Lorsque des objets sont créés sans spécifier un schéma cible particulier, ils sont placés dans le premier schéma listé dans le chemin de recherche. Une erreur est rapportée si le chemin de recherche est vide.

  La valeur par défaut de ce paramètre est `'$user, public'` (où la deuxième partie sera ignorée s'il n'existe pas de schéma nommé `public`). Elle supporte l'utilisation partagée d'une base de données (où aucun utilisateur n'a de schémas privés et tous partagent l'utilisation de `public`), de schémas privés par utilisateur et une combinaison des deux. D'autres effets peuvent être obtenus en modifiant le chemin de recherche par défaut soit globalement soit par utilisateur.

La valeur réelle actuelle du chemin de recherche peut être examinée via la fonction SQL `current_schemas()`. Elle n'est pas identique à la valeur de `search_path`, car `current_schemas()` affiche comment la requête apparaissant dans `search_path` sera résolue.

Pour plus d'informations sur la gestion des schémas, voir la Section 5.8.

`default_tablespace` (`string`)
Cette variable spécifie l'espace logique par défaut dans lequel créé les objets (tables et index) quand une commande `CREATE` ne spécifie par explicitement un espace logique.

La valeur est soit le nom de l'espace logique soit une chaîne vide pour indiquer l'utilisation de l'espace logique par défaut de la base de données en cours. Si la valeur ne correspond pas au nom d'une base de données existantes, PostgreSQL utilisera automatiquement l'espace logique par défaut de la base de données en cours.

Pour plus d'informations sur les espaces logiques, voir la Section 18.6.

`check_function_bodies` (`boolean`)
Ce paramètre est habituellement à true. Lorsqu'il vaut false, il désactive la validation de la chaîne du corps de la fonction lors de *CREATE FUNCTION*. Désactiver la validation est quelque fois utile pour éviter des problèmes tels que les références avant lors de la restauration des définitions de fonctions à partir d'une sauvegarde.

`default_transaction_isolation` (`string`)
Chaque transaction SQL a un niveau d'isolation, qui peut être soit << read uncommitted >>, soit << read committed >>, soit << repeatable read >> soit << serializable >>. Ce paramètre contrôle le niveau d'isolation par défaut de chaque nouvelle transaction. Par défaut, << read committed >>.

Consultez Chapitre 12 et SET TRANSACTION pour plus d'informations.

`default_transaction_read_only` (`boolean`)
Une transaction SQL en lecture seule ne pourrait pas modifier les tables non temporaires. Ce paramètre contrôle le statut de lecture seule par défaut de chaque nouvelle transaction. La valeur par défaut est faux (lecture/écriture).

Consultez SET TRANSACTION pour plus d'informations.

`statement_timeout` (`integer`)
Annule toute instruction prenant plus que le nombre spécifié de millisecondes. Une valeur de zéro désactive le chronomètre et est la valeur par défaut.

---

### 16.4.8.2. Locale et formatage

`DateStyle` (`string`)
Configure le format d'affichage pour les valeurs de type date et heure, ainsi que les règles d'interprétation des valeurs de saisie des dates ambiguës. Pour des raisons historiques, cette variable contient deux composants indépendants : la spécification du format en sortie (`ISO`, `Postgres`, `SQL` ou `German`) et la spécification en entrée/sortie pour l'ordre année/mois/jour (`DMY`, `MDY` ou `YMD`). Elles peuvent être configurées séparément ou ensemble. Les mots clés `Euro` et `European` sont un synonyme pour `DMY` ; les mots clés `US`, `NonEuro` et `NonEuropean` sont des synonymes pour `MDY`. Voir la Section 8.5 pour plus d'informations. La valeur par défaut est `ISO, MDY`.

`timezone` (`string`)
Configure le fuseau horaire pour l'affichage et l'interprétation de la date et de l'heure. Par défaut, vaut 'unknown', ce qui signifie qu'il utilise ce que l'environnement système spécifie comme fuseau horaire. Voir la Section 8.5 pour plus d'informations.

`australian_timezones` (`boolean`)

Si vrai, ACST, CST, EST et SAT sont interprétés comme des fuseaux horaires australiens plutôt que comme des fuseaux horaires d'Amérique Nord/Sud. Par défaut à faux.

extra_float_digits (integer)

Ce paramètre ajuste le nombre de chiffres affichés par les valeurs à virgule flottante, ceci incluant float4, float8 et les types de données géométriques. La valeur du paramètre est ajouté au nombre standard de chiffres (FLT_DIG ou DBL_DIG comme approprié). La valeur peut être initialisée à une valeur maximum de 2 pour inclure les chiffres partiellement significatifs ; c'est tout spécialement utile pour sauvegarder les données à virgule flottante qui ont besoin d'être restaurées exactement. Cette variable peut aussi être négative pour supprimer les chiffres non souhaités.

client_encoding (string)

Initialise le codage côté client (ensemble de caractères). Par défaut, utilise le codage de la base de données.

lc_messages (string)

Initialise la langue dans laquelle les messages sont affichés. Les valeurs acceptables sont dépendantes du système ; voir Section 20.1 pour plus d'informations. Si cette variable est initialisée en tant que chaîne vide (ce qui est la valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur.

Avec certains systèmes, cette catégorie de locale n'existe pas. Initialiser cette variable fonctionnera toujours mais n'aura aucun effet. De même, il existe une chance pour qu'aucun message traduit n'existe pour la langue sélectionnée. Dans ce cas, vous continuerez de voir les messages en anglais.

lc_monetary (string)

Initialise la locale à utiliser pour formater les montants de monnaie, par exemple avec la famille de fonctions to_char. Les valeurs acceptables dépendent du système ; voir la Section 20.1 pour plus d'informations. Si cette variable est une chaîne vide (la valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur.

lc_numeric (string)

Initialise la locale à utiliser pour formater les nombres, par exemple avec la famille de fonctions to_char. Les valeurs acceptables dépendent du système ; voir la Section 20.1 pour plus d'informations. Si cette variable est une chaîne vide (valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur.

lc_time (string)

Initialise la locale à utiliser pour formater les valeurs de date et d'heure (actuellement, ce paramétrage ne fait rien mais il le pourrait dans le futur). Les valeurs acceptables dépendent du système ; voir la Section 20.1 pour plus d'informations. Si cette variable est une chaîne vide (la valeur par défaut), alors la valeur est héritée de l'environnement système du serveur.

### 16.4.8.3. Autres valeurs par défaut

explain_pretty_print (boolean)

Détermine si EXPLAIN VERBOSE utilise le format indenté ou non pour l'affichage des arbres détaillés des requêtes. Activé par défaut.

dynamic_library_path (string)

Si un module chargeable dynamiquement a besoin d'être ouvert et que le nom de fichier spécifié dans la commande CREATE FUNCTION ou LOAD ne contient pas de répertoire (c'est−à−dire que le nom ne contient pas de slash), le système cherchera ce chemin pour le fichier requis.

La valeur pour dynamic_library_path doit être une liste de chemins absolus séparés par des virgules (ou des points virgules sous Windows). Si un élément de la liste commence avec la chaîne spéciale $libdir, le répertoire des bibliothèques internes du paquetage PostgreSQL est substitué à $libdir. C'est l'emplacement où sont installés les modules fournis par la distribution PostgreSQL

standard (utilisez `pg_config --pkglibdir` pour connaître le nom de ce répertoire). Par exemple :

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

ou dans un environnement Windows :

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

La valeur par défaut de ce paramètre est `'$libdir'`. Si la valeur est une chaîne vide, la recherche automatique du chemin est désactivée.

Ce paramètre peut être modifié à l'exécution par les superutilisateurs mais un paramétrage réalisé de cette façon ne persistera que pendant la durée de la connexion du client, donc cette méthode devrait être réservée à des buts de développements. La façon recommandée pour initialiser ce paramètre est d'utiliser le fichier de configuration `postgresql.conf`.

## 16.4.9. Gestion des verrous

`deadlock_timeout` (`integer`)

Temps total, en millisecondes, d'attente d'un verrou avant de vérifier s'il s'agit d'une condition de verrous morts (deadlock condition). La vérification d'un verrou mort est assez lente, donc le serveur ne le fait pas à chaque fois qu'il attend pour un verrou. Nous supposons (de façon optimiste) que les verrous morts ne sont pas communs pour les applications de production et nous attendons simplement un verrou pendant un certain temps avant de lancer une recherche de blocage. Augmenter cette valeur réduit le temps perdu en recherche inutile de verrous morts mais ralentit la détection de vraies erreurs de verrous morts. La valeur par défaut est de 1000 (c'est–à–dire une par seconde), ce qui est probablement la plus petite valeur que vous pourriez vouloir en pratique. Sur un serveur déjà chargé, vous pouvez l'augmenter. Idéalement, ce paramétrage devrait dépasser le temps typique d'une transaction de façon à améliorer la probabilité qu'un verrou soit abandonné avant que le processus en attente ne décide de lancer une recherche de verrous morts.

`max_locks_per_transaction` (`integer`)

La table des verrous partagés a une taille basée sur la supposition que au moins `max_locks_per_transaction` * `max_connections` objets distincts seront nécessaires pour être verrouillés en même temps. (De coup, le nom de ce paramètre pourrait rendre confus : il n'y a pas de limite dur sur le nombre de verrous pris par une transaction, mais plutôt une valeur moyenne oui maximum.) Par défaut, 64, qui a prouvé son adéquation historiquement mais vous pourriez avoir besoin d'augmenter cette valeur si vous avez des clients qui touchent beaucoup de tables différentes dans une seule transaction. Cette option est initialisée au lancement du serveur uniquement.

## 16.4.10. Compatibilité de version et de plateforme

### 16.4.10.1. Précédentes versions de PostgreSQL

`add_missing_from` (`boolean`)

Si `true`, les tables qui sont référencées par une requête seront automatiquement ajoutées dans la clause `FROM` si elles n'y sont pas déjà présentes. La valeur par défaut est `true` pour des raisons de compatibilité avec les versions précédentes de PostgreSQL. Néanmoins, ce comportement n'appartient pas au standard SQL et beaucoup de personnes ne l'aiment pas car elle peut masquer les erreurs (comme référencer une table où vous devriez avoir référencé son alias). L'initialiser à `false`

pour avoir un comportement en compatibilité avec le standard SQL permet de rejeter les références non listés dans la clause FROM.

regex_flavor (string)
regex_flavor (string)

La << saveur >> des expressions rationnelles peut être advanced (avancée), extended (étendue) ou basic (basique). La valeur par défaut est advanced. Le paramétrage extended peut être utile pour une compatibilité ascendante exacte avec les versions précédant la 7.4 de PostgreSQL. Voir la Section 9.7.3.1 pour les détails.

sql_inheritance (boolean)

Ceci contrôle la sémantique de l'héritage, en particulier si les sous−tables sont incluses par les différentes commandes par défaut. Elles ne l'étaient pas dans les versions antérieures à la 7.1. Si vous avez besoin de l'ancien comportement, vous pouvez désactiver cette variable mais, à long terme, vous êtes encouragé à changer vos applications pour utiliser le mot clé ONLY qui exclut les sous−tables. Voir la Section 5.5 pour plus d'informations sur les héritages.

default_with_oids (boolean)

Elle contrôle si les commandes CREATE TABLE et CREATE TABLE AS incluent une colonne OID dans les tables nouvellement créées, si ni WITH OIDS ni WITHOUT OIDS ne sont spécifiées. Elle détermine aussi si les OID seront inclus dans les tables créés par SELECT INTO. Dans PostgreSQL 8.0.5, la valeur par défaut de default_with_oids est true. C'est aussi le comportement des versions précédentes de PostgreSQL. Néanmoins, supposer que les tables contiendront des OID par défaut n'est pas encouragé. Cette option aura probablement la valeur false par défaut dans une future version de PostgreSQL.

Pour faciliter la compatibilité avec les applications utilisant les OID, cette option devrait être activée. Pour faciliter la compatibilité avec les futures versions de PostgreSQL, cette option devrait être désactivée et les applications réclamant des OID sur certaines tables devraient spécifier explicitement WITH OIDS lors de la création de ces tables.

### 16.4.10.2. Compatibilité de la plateforme et du client

transform_NULL_equals (boolean)

Une fois activée, les expressions de la forme *expr* = NULL (ou NULL = *expr*) sont traitées comme *expr* IS NULL, c'est−à−dire qu'elles renvoient vrai si *expr* s'évalue par la valeur NULL, et faux sinon. Le bon comportement, compatible avec le standard SQL, de *expr* = NULL est de toujours renvoyer NULL (inconnu). Du coup, cette option est désactivée par défaut.

Néanmoins, les formulaires filtrés dans Microsoft Access génèrent des requêtes qui utilisent *expr* = NULL pour tester les valeurs NULL, donc si vous utilisez cette interface pour accéder à une base de données, vous souhaiterez activer cette option. Comme les expressions de la forme *expr* = NULL renvoient toujours la valeur NULL (en utilisant la bonne interprétation), elles ne sont pas très utiles et n'apparaissent pas souvent dans les applications normales, donc cette option a peu d'utilité en pratique. Mais les nouveaux utilisateurs confondent fréquemment la sémantique des expressions impliquant des valeurs NULL. Du coup, cette option n'est pas activée par défaut.

Notez que cette option affecte seulement la forme exacte = NULL, aucun autre opérateur de comparaison ou aucune autre expression qui sont équivalentes en terme de calcul à des expressions impliquant l'opérateur égal (telles que IN). Du coup, cette option n'est pas un correctif général pour une mauvaise programmation.

Référez−vous à la Section 9.2 pour des informations en relation.

## 16.4.11. Options préconfigurées

Les << paramètres >> suivant sont en lecture seule et sont déterminés lors de la compilation ou de l'installation de PostgreSQL. Ainsi, ils ont été exclus du fichier `postgresql.conf` d'exemple. Ces options rapportent différents aspects du comportement de PostgreSQL qui pourraient avoir un intérêt pour certaines applications, particulièrement pour les interfaces d'administration.

`block_size` (`integer`)
: Affiche la taille d'un bloc disque. Elle est déterminée par la valeur de `BLCKSZ` à la construction du serveur. La valeur par défaut est de 8192 octets. La signification de quelques variables de configuration (comme shared_buffers) est influencée par `block_size`. Voir la Section 16.4.3 pour plus d'informations.

`integer_datetimes` (`boolean`)
: Affiche si PostgreSQL a été construit avec le support des dates et heures sur des entiers de 64 bits. Il est configuré en utilisant `--enable-integer-datetimes` au moment de la construction de PostgreSQL. La valeur par défaut est `off`.

`lc_collate` (`string`)
: Affiche la locale utilisée pour le tri des données de type texte. Voir la Section 20.1 pour plus d'informations. La valeur est déterminée lors de l'initialisation du groupe de bases de données.

`lc_ctype` (`string`)
: Affiche la locale déterminant les classifications de caractères. Voir la Section 20.1 pour plus d'informations. La valeur est déterminée au moment de l'initialisation du groupe de bases de données. D'habitude, elle sera identique à `lc_collate` mais, pour des applications particulières, elle pourrait être configurée différemment.

`max_function_args` (`integer`)
: Affiche le nombre maximum d'arguments des fonctions. Ce nombre est déterminé par la valeur de `FUNC_MAX_ARGS` lors de la construction du serveur. La valeur par défaut est de 32.

`max_identifier_length` (`integer`)
: Affiche la longueur maximum d'un identifiant. Elle est déterminée comme valant `NAMEDATALEN` moins un lors de la construction du serveur. La valeur par défaut de `NAMEDATALEN` est 64 ; du coup, la valeur par défaut de `max_identifier_length` est 63.

`max_index_keys` (`integer`)
: Affiche le nombre maximum de clés d'index. Ce nombre est déterminé par la valeur de `INDEX_MAX_KEYS` lors de la construction du serveur. La valeur par défaut est 32.

`server_encoding` (`string`)
: Affiche le codage de la base de données (ensemble de caractères). Il est déterminé lors de la création de la base de données. Habituellement, les clients sont seulement concernés par la valeur de client_encoding.

`server_version` (`string`)
: Affiche le numéro de version du serveur. Il est déterminé par la valeur de `PG_VERSION` lors de la construction du serveur.

## 16.4.12. Options personnalisées

Cette fonctionnalité a été conçue pour permettre l'ajout d'options habituellement inconnues de PostgreSQL car ajoutées par des modules supplémentaires (comme les langages de procédures). Ceci autorise les modules à être configuré de la façon standard.

`custom_variable_classes` (`string`)

Cette variable spécifie un ou plusieurs noms de classe à utiliser par des variables personnalisées, de la forme d'une liste séparée par des virgules. Une variable personnalisée est une variable habituellement inconnue de PostgreSQL lui−même mais utilisée par certains modules supplémentaires. Ces variables doivent avoir des noms consistant en un nom de classe, un point et un nom de variable. `custom_variable_classes` spécifie tous les noms de classe utilisés dans une installation particulière. Cette option est seulement disponible au lancement du serveur ou dans le fichier de configuration `postgresql.conf`.

La difficulté avec la configuration de variables personnalisées dans `postgresql.conf` est que le fichier doit être lu avant que les modules supplémentaires ne soient chargés et, du coup, les variables seraient habituellement rejetées comme étant inconnues. Lorsque `custom_variable_classes` est initialisé, le serveur acceptera les définitions de variables arbitraires à l'intérieur de chaque classe spécifiée. Ces variables seront traitées comme des emplacements vides et n'auront aucune fonction tant que le module qui les définit n'est pas chargé. Quand un module d'une classe spécifique est chargé, il ajoute les bonnes définitions de variables pour son nom de classe, convertit les valeurs des emplacements suivant les définitions et émet des messages d'avertissement pour tout emplacement restant dans sa classe (qui seraient à priori des noms de variables mal saisis).

Voici un exemple de ce que pourrait contenir `postgresql.conf` en utilisant des variables personnalisées :

```
custom_variable_classes = 'plr,pljava'
plr.path = '/usr/lib/R'
pljava.foo = 1
plruby.bar = true        # generates error, unknown class name
```

## 16.4.13. Options pour les développeurs

Les options suivantes ont pour but de travailler sur les sources de PostgreSQL et, dans certains cas, d'assister à une récupération sur des bases de données sévèrement endommagées. Il n'y a aucune raison pour les utiliser dans la configuration d'un système de production. En tant que tel, elles ont été exclues du fichier d'exemple de `postgresql.conf`. Notez qu'un certain nombre d'entre elles requièrent des options de compilation spéciales pour fonctionner.

`debug_assertions` (`boolean`)
: Active différentes vérifications des affectations. C'est une aide au débogage. Si vous expérimentez des problèmes étranges ou des arrêts brutaux, vous pouvez l'activer car cette option pourrait exposer des erreurs de programmation. Pour utiliser cette option, la macro `USE_ASSERT_CHECKING` doit être définie lors de la construction de PostgreSQL (ceci s'accomplit par l'option `--enable-cassert` de `configure`). Notez que la valeur de `debug_assertions` est par défaut active si PostgreSQL a été construit après ajout de l'option ci−dessus.

`debug_shared_buffers` (`integer`)
: Nombre de secondes entre les rapports des tampons freelist. Si plus grand que zéro, émet des statistiques freelist sur le journal toutes les `debug_shared_buffers` secondes. Zéro (par défaut) désactive les rapports.

`pre_auth_delay` (`integer`)
: Si différent de zéro, un délai de ce nombre de secondes arrive juste après qu'un nouveau processus ne soit créé, avant le processus d'authentification. Ceci a pour but de donner une opportunité d'attacher un débogueur au processus serveur pour tracer les mauvais comportements pendant l'authentification.

`trace_notify` (`boolean`)
: Génère un grand nombre de sorties de débogage pour les commandes `LISTEN` et `NOTIFY`. client_min_messages ou log_min_messages doivent être `DEBUG1` ou plus bas pour envoyer cette

sortie sur les traces client ou serveur, respectivement.

`trace_locks` (`boolean`)
`trace_lwlocks` (`boolean`)
`trace_userlocks` (`boolean`)
`trace_lock_oidmin` (`boolean`)
`trace_lock_table` (`boolean`)
`debug_deadlocks` (`boolean`)
`log_btree_build_stats` (`boolean`)

Différentes autres options de trace et de débogage.

`wal_debug` (`boolean`)

Si true, émet une sortie de débogage relative aux WAL. Cette option est seulement disponible si la macro `WAL_DEBUG` était définie au moment de la compilation de PostgreSQL.

`zero_damaged_pages` (`boolean`)

La détection d'une en−tête de page endommagée cause généralement le rapport d'une erreur par PostgreSQL, l'annulation de la commande courante. Initialiser `zero_damaged_pages` à vrai fait que le système rapporte à la place un avertissement, supprime la page endommagée et continue son traitement. Ce comportement *détruira les données*, nommément toutes les lignes de la page endommagée. Mais cela vous permettra de passer l'erreur et de récupérer les lignes des pages non endommagées qui pourraient être présentes dans la table. Donc, c'est utile pour récupérer les données si la corruption est dûe à une erreur matérielle ou logicielle. Vous ne devriez généralement pas configurer cette option à vrai sauf si vous abandonnez l'espoir de récupérer les données des pages endommagées d'une table. Le paramétrage par défaut est désactivé, et ne peut être modifié que par un superutilisateur.

## 16.4.14. Options courtes

Pour le côté pratique, voici les options sur une lettre en ligne de commande disponibles pour certains paramètres. Elles sont décrites dans le Tableau 16−1.

**Tableau 16−1. Clés des options courtes**

| Option courte | Équivalent |
|---|---|
| `-B` *x* | `shared_buffers = x` |
| `-d` *x* | `log_min_messages = DEBUG`*x* |
| `-F` | `fsync = off` |
| `-h` *x* | `listen_addresses = x` |
| `-i` | `listen_addresses = '*'` |
| `-k` *x* | `unix_socket_directory = x` |
| `-l` | `ssl = on` |
| `-N` *x* | `max_connections = x` |
| `-p` *x* | `port = x` |
| `-fi`, `-fh`, `-fm`, `-fn`, `-fs`, `-ft`[a] | `enable_indexscan=off`, `enable_hashjoin=off`, `enable_mergejoin=off`, `enable_nestloop=off`, `enable_seqscan=off`, `enable_tidscan=off` |
| `-s`[a] | `log_statement_stats = on` |
| `-S` *x*[a] | `work_mem = x` |

| | |
|---|---|
| -tpa, -tpl, -te[a] | log_parser_stats=on, log_planner_stats=on, log_executor_stats=on |

Remarques :
a. Pour des raisons historiques, ces options doivent êtres passées au processus serveur individuel via l'option -o de postmaster, par exemple,

```
$ postmaster -o '-S 1024 -s'
```

ou via PGOPTIONS du côté client, comme expliqué ci-dessous.

# 16.5. Gérer les ressources du noyau

Une installation importante de PostgreSQL peut rapidement épuiser les limites des ressources du système d'exploitation. (Sur certains systèmes, les valeurs par défaut sont trop basses que vous n'avez même pas besoin d'une installation << importante >>.) Si vous avez rencontré ce type de problème, continuez votre lecture.

## 16.5.1. Mémoire partagée et sémaphore

La mémoire partagée et les sémaphores sont nommés collectivement << IPCSystem V >> (ensemble avec les queues de messages, qui n'ont pas d'importance pour PostgreSQL). Pratiquement tous les systèmes d'exploitation modernes fournissent ces fonctionnalités mais parmi elles toutes ne sont pas activées ou dimensionnées suffisamment par défaut, spécialement les systèmes ayant l'héritage BSD. (Pour les ports QNX et BeOS, PostgreSQL fournit sa propre implémentation de remplacement de ces fonctionnalités.)

Le manque complet de fonctionnalités est généralement manifesté par une erreur Illegal system call au lancement du serveur. Dans ce cas, il n'y a rien à faire à part reconfigurer votre noyau. PostgreSQL ne fonctionnera pas sans.

Quand PostgreSQL dépasse une des nombreuses limites IPC, le serveur refusera de s'exécuter et lèvera un message d'erreur instructif décrivant le problème rencontré et que faire avec (voir aussi la Section 16.3.1). Les paramètres adéquats du noyau sont nommés de façon cohérente parmi les différents systèmes ; le Tableau 16-2 donne un aperçu. Néanmoins, les méthodes pour les obtenir varient. Les suggestions pour quelques plateformes sont données ci-dessous. Attention, il est souvent nécessaire de redémarrer votre machine, voire même de recompiler le noyau, pour changer ces paramétrages.

**Tableau 16-2. Paramètres System V IPC**

| Name | Description | Valeurs raisonnables |
|---|---|---|
| SHMMAX | Taille maximum du segment de mémoire partagée (octets) | 250 Ko + 8.2 Ko * shared_buffers + 14.2 Ko * max_connections jusqu'à l'infini |
| SHMMIN | Taille minimum du segment de mémoire partagée (octets) | 1 |
| SHMALL | Total de la mémoire partagée disponible (octets ou pages) | Si octets, identique à SHMMAX ; si pages, ceil(SHMMAX/PAGE_SIZE) |
| SHMSEG | Nombre maximum de segments de mémoire partagée par processus | Seul un segment est nécessaire mais la valeur par défaut est bien plus importante |

| | | |
|---|---|---|
| SHMMNI | Nombre maximum de segments de mémoire partagée pour tout le système | Comme SHMSEG plus la place pour les autres applications |
| SEMMNI | Nombre maximum d'identifiants de sémaphores (c'est−à−dire d'ensembles) | Au moins ceil(max_connections / 16) |
| SEMMNS | Nombre maximum de sémaphores rapartis dans le système | ceil(max_connections / 16) * 17 plus la place pour les autres applications |
| SEMMSL | Nombre maximum de sémaphores par ensemble | Au moins 17 |
| SEMMAP | Nombre d'entrées dans la carte des sémaphores | Voir le texte |
| SEMVMX | Valeur maximum d'une sémaphore | Au moins 1000 (vaut souvent par défaut 32767, ne pas changer sauf si vous êtes forcé.) |

Le paramètre de mémoire partagé le plus important est SHMMAX, la taille maximum, en octets, d'un segment de mémoire partagée. Si vous obtenez un message d'erreur à partir de shmget comme Invalid argument, il est possible que cette limite soit dépassée. La taille du segment de mémoire partagée requise varie à la fois avec le nombre de tampons requis (option −B) et le nombre de connexions autorisées (option −N), bien que le premier soir le plus important. (Vous pouvez, en solution temporaire, baisser ces paramétrages pour supprimer l'échec.) Tout message d'erreur que vous obtiendriez contiendra la taille de la demande d'allocation échouée.

Certains systèmes ont aussi une limite sur le nombre total de mémoire partagée dans le système (SHMALL). Assurez−vous que cela soit suffisamment important pour PostgreSQL et quelque autres applications utilisant des segments de mémoire partagée. (Attention : SHMALL est mesuré en pages plutôt qu'en octets sur beaucoup de systèmes.)

Moins sensible aux problèmes est la taille minimum des segments de mémoire partagée (SHMMIN), qui devrait être au plus à environ 256 Ko pour PostgreSQL (il est habituellement à 1). Le nombre maximum de segments au travers du système (SHMMNI) ou par processus (SHMSEG) a peu de chances de causer un problème sauf s'ils sont configurés à zéro sur votre système.

PostgreSQL utilise une sémaphore par connexion autorisée (option −N), par ensemble de 16. Chacun de ces ensembles contiendra aussi une 17è sémaphore qui contient un << nombre magique >>, pour détecter la collision avec des ensembles de sémaphore utilisé par les autres applications. Le nombre maximum de sémaphores dans le système est initialisé par SEMMNS, qui en conséquence doit être au moins aussi haut que max_connections plus un extra de chacune des 16 connexions autorisées (voir la formule dans Tableau 16−2). Le paramètre SEMMNI détermine la limite sur le nombre d'ensembles de sémaphores qui peuvent exister sur le système à un instant précis. Donc, ce paramètre doit être au moins égal à ceil(max_connections / 16). Baisser le nombre de connexions autorisées est un contournement temporaire pour les échecs qui sont habituellement indiqués par le message No space left on device, à partir de la fonction semget.

Dans certains cas, il pourrait être nécessaire d'augmenter SEMMAP pour être au moins dans l'ordre de SEMMNS. Ce paramètre définit la taille de la carte de ressources de sémaphores, dans laquelle chaque bloc contigu de sémaphores disponibles ont besoin d'une entrée. Lorsqu'un ensemble de sémaphore est libéré ou qu'il est enregistré sous une nouvelle entrée de carte. Si la carte est pleine, les sémaphores libérées sont perdues (jusqu'au redémarrage). La fragmentation de l'espace des sémaphores pourrait amener dans le temps à moins de sémaphores disponibles.

La paramètre SEMMSL, qui détermine le nombre de sémaphores dans un ensemble, pourrait valoir au moins 17 pour PostgreSQL.

D'autres paramètres en relation avec l'<< annulation de sémaphores >>, tels que SEMMNU et SEMUME, ne concernent pas PostgreSQL.

BSD/OS

**Mémoire partagée.** Par défaut, seulement 4 Mo de mémoire partagée est supportée. Gardez en tête que la mémoire partagée n'est pas paginable ; elle est verrouillée en RAM. Pour accroître la mémoire partagée supportée par votre système, ajoutez ce qui suit à la configuration de votre noyau. Une valeur de 1024 pour SHMALL représente 4 Mo de mémoire partagée. Pour argumenter la mémoire partagée supportée par votre système, ajoutez quelque chose comme ceci à votre configuration du noyau :

```
options "SHMALL=8192"
options "SHMMAX=\(SHMALL*PAGE_SIZE\)"
```

SHMALL est mesuré en pages de 4 Ko, donc une valeur de 1024 représente 4 Mo de mémoire partagée. Du coup, la configuration ci–dessus augmente l'aire de mémoire partagée à 32 Mo. Pour ceux utilisant une version 4.3 ou ultérieure, vous aurez probablement besoin d'augmenter KERNEL_VIRTUAL_MB au–dessus de la valeur par défaut, 248. Une fois tous les changements effectués, recompilez le noyau et redémarrez.

Pour ceux utilisant une version 4.0 ou antérieures, utilisez bpatch pour connaître la valeur sysptsize dans le noyau actuel. Elle est calculée dynamiquement au démarrage.

```
$ bpatch -r sysptsize
0x9 = 9
```

Ensuite, ajoutez SYSPTSIZE comme valeur codée en dur dans le fichier de configuration du noyau. Augmentez la valeur que vous trouvez en utilisant bpatch. Ajoutez 1 pour chaque 4 Mo supplémentaire de mémoire partagée que vous souhaitez.

```
options "SYSPTSIZE=16"
```

sysptsize ne peut pas être modifié avec sysctl.

**Sémaphores.** Vous voudrez probablement aussi augmenter le nombre de sémaphores ; la somme totale par défaut du système (60) n'autorisera seulement que 50 connexions PostgreSQL. Initialisez les valeurs que vous souhaitez dans le fichier de configuration du noyau :

```
options "SEMMNI=40"
options "SEMMNS=240"
```

FreeBSD

NetBSD

OpenBSD

Les options SYSVSHM et SYSVSEM doivent être activées à la compilation du noyau. (Ils le sont par défaut.) La taille maximum de mémoire partagée est déterminée par l'option SHMMAXPGS (en pages). Ce qui suit montre un exemple de l'initialisation des différents paramètres :

```
options         SYSVSHM
options         SHMMAXPGS=4096
options         SHMSEG=256

options         SYSVSEM
options         SEMMNI=256
options         SEMMNS=512
options         SEMMNU=256
```

```
options         SEMMAP=256
```

(Sur OpenBSD, le mot clé est en fait `option` au singulier.)

Vous pourriez aussi vouloir configurer votre noyau pour verrouiller la mémoire partagée en RAM et l'empêcher d'être paginée en swap. Utilisez le paramétrage `kern.ipc.shm_use_phys` de `sysctl`.

HP–UX

Les paramètres par défaut tendent à suffire pour des installations normales. Sur HP–UX 10, la valeur par défaut de `SEMMNS` est 128, qui pourrait être trop basse pour de gros sites de bases de données.

Les paramètres IPC peuvent être initialisés dans System Administration Manager (SAM) sous Kernel Configuration–>Configurable Parameters. Allez sur Create A New Kernel une fois terminée.

Linux

La limite de mémoire partagée par défaut (à la fois `SHMMAX` et `SHMALL`) est de 32 Mo pour les noyaux 2.2 mais cela peut se changer dans le système de fichiers `proc` (sans redémarrage). Par exemple, pour permettre 128 Mo :

```
$ echo 134217728 >/proc/sys/kernel/shmall
$ echo 134217728 >/proc/sys/kernel/shmmax
```

Vous pouvez placer ces commandes dans un script exécuté au démarrage.

Autrement, vous pouvez utiliser `sysctl`, si cette commande est disponible, pour contrôler ces paramètres. Cherchez un fichier nommé `/etc/sysctl.conf` et ajoutez les lignes qui suivent :

```
kernel.shmall = 134217728
kernel.shmmax = 134217728
```

Ce fichier est habituellement traité au démarrage mais `sysctl` peut aussi être appelé explicitement plus tard.

D'autres paramètres ont une taille suffisante pour toute application. Si vous voulez voir par vous–même, jetez un œil dans `/usr/src/linux/include/asm-`*`xxx`*`/shmparam.h` et `/usr/src/linux/include/linux/sem.h`.

MacOS X

Avec OS X 10.2 et antérieures, éditez le fichier `/System/Library/StartupItems/SystemTuning/SystemTuning` et modifiez les valeurs avec les commandes suivantes :

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

Avec OS X 10.3, ces commandes ont été déplacées dans `/etc/rc` et doivent être éditées là–bas. Vous aurez besoin de redémarrer pour que les modifications prennent effet. Notez que `/etc/rc` est habituellement surchargé par les mises à jour d'OS X (comme celle de 10.3.6 à 10.3.7) donc vous devez vous attendre à avoir à refaire votre édition après chaque mise à jour.

`SHMALL` est mesuré en pages de 4 Ko sur cette plateforme.

SCO OpenServer

Dans la configuration par défaut, seuls 512 Ko de mémoire partagée par segment est autorisé, ce qui est assez pour `-B 24 -N 12`. Pour augmenter ce paramétrage, allez tout d'abord dans le répertoire `/etc/conf/cf.d`. Pour afficher la valeur courante de `SHMMAX`, lancez

```
./configure -y SHMMAX
```

Pour configurer une nouvelle valeur de `SHMMAX`, lancez

```
./configure SHMMAX=valeur
```

où *`value`* est la nouvelle valeur que vous voulez utiliser (en octets). Après avoir configuré `SHMMAX`, reconstruisez le noyau :

```
./link_unix
```

et redémarrez.

AIX

Au moins à partir de la version 5.1, il ne devrait plus être nécessaire de faire une configuration spéciale pour tel paramètre comme `SHMMAX` car il apparaît qu'il est configuré pour autoriser l'utilisation de toute la mémoire comme mémoire partagée. C'est la sorte de configuration utilisée habituellement pour les autres bases de données comme DB/2.

Néanmoins, cela pourrait être nécessaire pour modifier l'information globale `ulimit` dans `/etc/security/limits` car les limites dures par défaut pour les tailles de fichiers (`fsize`) et les nombres de fichiers (`nofiles`) pourraient être trop bas.

Solaris

Au moins dans la version 2.6, la taille maximum par défaut des segments de mémoire partagée est trop basse pour PostgreSQL. Le paramétrage adéquat peut être modifié dans `/etc/system`, par exemple :

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

Vous avez besoin de redémarrer pour que les modifications prennent effet.

Voir aussi http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html pour des informations sur la mémoire partagée sous Solaris.

UnixWare

Avec UnixWare 7, la taille maximum des segments de mémoire partagée est de 512 Ko dans la configuration par défaut. Ceci est suffisant pour `-B 24 -N 12`. Pour afficher la valeur courante de `SHMMAX`, lancez

```
/etc/conf/bin/idtune -g SHMMAX
```

qui affiche la valeur courante, par défaut, minimum et maximum. Pour configurer une nouvelle valeur de `SHMMAX`, lancez

```
/etc/conf/bin/idtune SHMMAX valeur
```

où *valeur* est la nouvelle valeur que vous voulez utiliser (en octets). Après avoir initialisé `SHMMAX`, reconstruisez le noyau :

```
/etc/conf/bin/idbuild -B
```

et relancez.

## 16.5.2. Limites de ressources

Les systèmes d'exploitation style Unix renforcent différents types de limites de ressources qui pourraient interférer avec les opérations de votre serveur PostgreSQL. Les limites sur le nombre de processus par utilisateur, le nombre de fichiers ouverts par un processus et la taille mémoire disponible pour chaque processus sont d'une grande importance. Chacun d'entre elles ont une limite << dure >> et une limite << souple >>. La limite souple est réellement ce qui compte mais cela pourrait être changé par l'utilisateur jusqu'à la limite dure. La limite dure pourrait seulement être modifiée par l'utilisateur root. L'appel système `setrlimit` est responsable de l'initialisation de ces paramètres. La commande interne du shell `ulimit` (shells Bourne) ou `limit` (csh) est utilisé pour contrôler les limites de ressource à partir de la ligne de commande. Sur les systèmes dérivés BSD, le fichier `/etc/login.conf` contrôle les différentes limites de ressource initialisées à la connexion. Voir la documentation du système d'exploitation pour les détails. Les paramètres en question sont `maxproc`, `openfiles` et `datasize`. Par exemple :

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

(`-cur` est la limite douce. Ajoutez `-max` pour configurer la limite dure.)

Les noyaux peuvent aussi avoir des limites sur le système complet pour certaines ressources.

- Sur Linux, `/proc/sys/fs/file-max` détermine le nombre maximum de fichiers ouverts que le noyau supportera. Ce nombre est modifiable en écrivant un autre nombre dans le fichier ou en ajoutant une affectation dans `/etc/sysctl.conf`. La limite des fichiers par processus est fixée lors de la compilation du noyau ; voir `/usr/src/linux/Documentation/proc.txt` pour plus d'informations.

Le serveur PostgreSQL utilise un processus par connexion de façon à ce que vous puissiez fournir au moins autant de processus que de connexions autorisées, en plus de ce dont vous avez besoin pour le reste de votre système. Ceci n'est habituellement pas un problème mais si vous exécutez plusieurs serveurs sur une seule machine, cela pourrait devenir étroit.

La limite par défaut des fichiers ouverts est souvent initialisée pour être << amicalement sociale >>, pour permettre à de nombreux utilisateurs de coexister sur une machine sans utiliser une fraction inappropriée des ressources du système. Si vous lancez un grand nombre de serveurs sur une machine, cela pourrait être quelque chose que vous souhaitez mais sur les serveurs dédiés, vous pourriez vouloir augmenter cette limite.

D'un autre côté, certains systèmes autorisent l'ouverture d'un grand nombre de fichiers à des processus individuels ; si un plus grand nombre le font, alors les limites du système peuvent facilement être dépassées.

Si vous rencontrez ce cas et que vous ne voulez pas modifier la limite du système, vous pouvez initialiser le paramètre de configuration max_files_per_process de PostgreSQL pour limiter la consommation de fichiers ouverts.

### 16.5.3. Linux Memory Overcommit

Dans Linux 2.4 et ultérieurs, le comportement par défaut de la mémoire virtuelle n'est pas optimale pour PostgreSQL. À cause de la façon dont le noyau implémente le << memory overcommit >>, le noyau pourrait arrêter le serveur PostgreSQL (le processus `postmaster`) si la mémoire demande un autre processus et que le système est en manque de mémoire virtuelle.

Si ceci arrive, vous verrez un message du noyau qui ressemble à ceci (consultez la documentation et la configuration de votre système pour savoir où chercher un tel message) :

```
Out of Memory: Killed process 12345 (postmaster).
```

Ceci indique que le processus `postmaster` a été terminé à cause d'un problème de mémoire. Bien que les connexions en cours continueront de fonctionner normalement, aucune nouvelle connexion ne sera acceptée. Pour revenir à un état normal, PostgreSQL devra être relancé.

Une façon d'éviter ce problème revient à lancer PostgreSQL sur une machine où vous pouvez vous assurer que les autres processus ne mettront pas la machine en manque de mémoire.

Sur Linux 2.6 et ultérieure, une meilleure solution est de modifier le comportement du noyau de façon à ce qu'il n'<< overcommit >> pas la mémoire. Ceci se fait en sélectionnant le mode overcommit strict via `sysctl` :

```
sysctl -w vm.overcommit_memory=2
```

ou en plaçant une entrée équivalente dans `/etc/sysctl.conf`. Vous pourriez souhaiter modifier le paramétrage relatif `vm.overcommit_ratio`. Pour les détails, voir la documentation du noyau (`Documentation/vm/overcommit-accounting`).

Quelques noyaux 2.4 de vendeurs ont des pré−versions de l'overcommit du 2.6. Néanmoins, configurer `vm.overcommit_memory` à 2 sur un noyau qui n'a pas le code correspondant rendra les choses pires qu'elles n'étaient. Il est recommandé d'inspecter le code source du noyau (voir la fonction `vm_enough_memory` dans le fichier `mm/mmap.c`) pour vérifier ce qui est supporté dans votre copie avant d'essayer ceci avec une installation 2.4. La présence du fichier de documentation `overcommit-accounting` ne devrait *pas* être pris comme une preuve de la présence de cette fonctionnalité. En cas de doute, consultez un expert du noyau ou le vendeur de votre noyau.

## 16.6. Arrêter le serveur

Il existe plusieurs façons d'arrêter le serveur de bases de données. Vous contrôlez le type d'arrêt en envoyant différents signaux au processus `postmaster`.

SIGTERM
: Après réception de SIGTERM, le serveur désactive les nouvelles connexions mais permet aux sessions en cours de terminer leur travail normalement. Il s'arrête seulement après que toutes les

sessions se sont terminées normalement. C'est l'arrêt intelligent (*Smart Shutdown*).

SIGINT
: Le serveur désactive les nouvelles connexions et envoie à tous les processus serveur le signal SIGTERM, qui les fera annuler leurs transactions courantes pour quitter rapidement. Il attend ensuite la fin des processus serveur et s'arrête finalement. C'est l'arrêt rapide (*Fast Shutdown*).

SIGQUIT
: Voici l'arrêt immédiat (*Immediate Shutdown*), qui demande au processus `postmaster` d'envoyer un signal SIGQUIT à tous les processus fils et à quitter immédiatement non proprement. Les processus fils quittent immédiatement à réception du signal SIGQUIT. Ceci amènera une tentative de récupération (en rejouant les traces WAL) au prochain lancement. Ceci n'est recommandé que dans les cas d'urgence.

Le programme pg_ctl fournit une interface agréable pour envoyer ces signaux dans le but d'arrêter le serveur.

Autrement, vous pouvez envoyer le signal directement en utilisant `kill`. Le PID du processus `postmaster` peut être trouvé en utilisant le programme `ps` ou à partir du fichier `postmaster.pid` dans le répertoire des données. Par exemple, pour exécuter un arrêt rapide :

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

**Important :** Il est mieux de ne pas utiliser SIGKILL pour arrêter le serveur. Le faire empêchera le serveur de libérer la mémoire partagée et les sémaphores, ce qui pourrait devoir être fait manuellement avant qu'un nouveau serveur ne soit lancé. De plus, SIGKILL tue le processus `postmaster` sans que celui–ci ait le temps de relayer ce signal à ses sous–processus, donc il sera aussi nécessaire de tuer les sous–processus individuels à la main.

## 16.7. Options de cryptage

PostgreSQL offre du cryptage sur plusieurs niveaux et fournit une flexibilité pour protéger les données d'être révélées suite à un vol du serveur de la base de données, des administrateurs non scrupuleux et des réseaux non sécurisés. Le cryptage pourrait aussi être requis par des demandes du gouvernement, par exemple pour des informations médicales ou des transactions financières.

Chiffrement du mot de passe stocké
: Par défaut, les mots de passe des utilisateurs de la base de données sont stockées suivant des hachages MD5, donc l'administrateur ne peut pas déterminer le mot de passe affecté à l'utilisateur. Si le cryptage MD5 est utilisé pour l'authentification du client, le mot de passe non crypté n'est jamais présent temporairement sur le serveur parce que le client le crypte en MD5 avant de l'envoyer sur le réseau. MD5 est un cryptage à sens unique — il n'existe pas d'algorithme de décryptage.

Chiffrement de colonnes spécifiques
: La bibliothèque de fonctions `/contrib` `pgcrypto` autorise le stockage crypté de certains champs. Ceci est utile si seulement certaines données sont sensibles. Le client fournit la clé de décryptage et la donnée est décryptée sur le serveur puis elle est envoyée au client.

  La donnée décryptée et la clé de déchiffrement sont présente sur le serveur pendant un bref moment où la donnée est décryptée, puis envoyée entre le client et le serveur. Ceci présente un bref moment où la données et les clés peuvent être interceptées par quelqu'un ayant un accès complet au serveur de bases de données, tel que l'administrateur du système.

Chiffrement de la partition de données

Sur Linux, le chiffrement peut se faire au niveau du montage d'un système de fichiers en utilisant un << périphérique loopback >>. Ceci permet à une partition entière du système de fichiers d'être cryptée et décryptée par le système d'exploitation. Sur FreeBSD, la fonctionnalité équivalent est appelé << GEOM Based Disk Encryption >>, ou gbde.

Ce mécanisme empêche la lecture de données non cryptées à partir des disques si ceux−ci ou le serveur sont volés. Ce mécanisme ne protège en rien contre les attaques quand le système de fichiers est monté parce que, dans ce cas, le système d'exploitation fournit une vue non cryptée des données. Néanmoins, pour monter le système de fichiers, vous avez besoin de passer la clé de cryptage au système d'exploitation et, quelque fois, cette clé est stocké quelque part sur l'hôte qui monte le disque.

Cryptage des mots de passe sur le réseau

La méthode d'authentification `MD5` crypte deux fois le mot de passe sur le client avant de l'envoyer au serveur. Il le crypte tout d'abord à partir du nom de l'utilisateur puis il le crypte à partir d'un élément du hasard envoyé par le serveur au moment de la connexion. Cette valeur, deux fois cryptée, est envoyée sur le réseau au serveur. Le double cryptage empêche non seulement la découverte du mot de passe, il empêche aussi une autre connexion en rejouant la même valeur de double cryptage dans une connexion future.

Chiffrement des données sur le réseau

Les connexions SSL cryptent toutes les données envoyées sur le réseau : le mot de passe, les requêtes et les données renvoyées. Le fichier `pg_hba.conf` permet aux administrateurs de spécifier quels hôtes peuvent utiliser des connexions non cryptées (`host`) et lesquels requièrent des connexions SSL (`hostssl`). De plus, les clients peuvent spécifier qu'ils se connectent aux serveurs seulement via SSL. Stunnel ou SSH peuvent aussi être utilisés pour crypter les transmissions.

Authentification de l'hôte SSL

Il est possible que le client et le serveur fournissent des clés SSL ou des certificats à l'autre. Cela demande une configuration supplémentaire de chaque côté mais cela fournit une vérification plus forte de l'identité que la simple utilisation de mots de passe. Cela empêche un ordinateur de se faire passer pour le serveur assez longtemps pour lire le mot de passe envoyé par le client. Cela empêche aussi les attaques du type << man in the middle >> où un ordinateur, entre le client et le serveur, prétend être le serveur, lit et envoie les données entre le client et le serveur.

Chiffrement côté client

Si vous n'avez pas confiance en l'administrateur système, il est nécessaire que le client crypte les données ; de cette façon, les données non cryptées n'apparaissent jamais sur le serveur de la base de données. Les données sont cryptées sur le client avant d'être envoyé au serveur, et les résultats de la base de données doivent être décryptés sur le client avant d'être utilisés. Le livre de Peter Wayner, [Translucent Databases], parle de cela de façon très détaillé.

## 16.8. Connexions TCP/IP sécurisées avec SSL

PostgreSQL dispose d'un support natif pour l'utilisation de connexions SSL, cryptant ainsi les communications clients/serveurs pour une sécurité améliorée. Ceci requiert l'installation d'OpenSSL à la fois sur le système client et sur le système serveur et que ce support soit activé au moment de la construction de PostgreSQL (voir le Chapitre 14).

Avec le support SSL compilé, le serveur PostgreSQL peut être lancé avec SSL activé en activant `ssl` dans `postgresql.conf`. Lors d'un démarrage en mode SSL, le serveur cherchera les fichiers `server.key` et `server.crt` dans le répertoire des données, qui doivent contenir respectivement la clé privée du serveur et le certificat. Ces fichiers doivent être configurés correctement avant qu'un serveur dont le mode SSL est activé puisse démarrer. Si la clé privée est protégée avec une phrase, le serveur la demandera et ne se lancera pas tant

que celle−ci n'aura pas été saisie.

Le serveur écoutera les connexions SSL et standard sur le même port TCP et négociera avec tout client se connectant qu'il utilise ou non SSL. Voir le Chapitre 19 pour savoir comment forcer l'utilisation de SSL pour certaines connexions.

Pour les détails sur la création de la clé privé et du certificat du serveur, référez−vous à la documentation d'OpenSSL. Un simple certificat signé par soi−même peut être utilisé pour des tests mais un certificat signé par une autorité (CA) (soit un des CA globaux soit un local) devrait être utilisé en production de façon à ce que le client puisse vérifier l'identité du serveur. Pour créer rapidement un certificat signé soi−même, utilisez la commande OpenSSL suivante :

```
openssl req −new −text −out server.req
```

Remplissez les informations que `openssl` réclame. Assurez−vous que vous entrez le nom local de l'hôte sur << Common Name >> ; le mot de passe de challenge peut être laissé vide. Le programme générera une clé qui est protégée par une phrase ; elle n'acceptera pas une phrase qui fait moins de quatre caractères. Pour supprimer la phrase (ce que vous devez faire si vous voulez automatiser le lancement du serveur), lancez les commandes

```
openssl rsa −in privkey.pem −out server.key
rm privkey.pem
```

Saisissez l'ancienne phrase pour débloquer la clé existante. Maintenant, saisissez

```
openssl req −x509 −in server.req −text −key server.key −out server.crt
chmod og−rwx server.key
```

pour remplacer le certificat en un certificat signé par soi−même et copiez la clé et le certificat là où le serveur les cherchera.

Si la vérification des certificats du client est requise, placez les certificats du CA que vous souhaitez vérifier dans le fichier `root.crt` du répertoire des données. S'il est présent, un certificat client sera demandé à partir du client lors du lancement d'une connexion SSL et il doit y avoir des certificats présent dans `root.crt`.

Quand le fichier `root.crt` est absent, les certificats du client ne seront ni demandés ni vérifiés. Dans ce mode, SSL fournit la sécurité de la communication pas l'authentification.

Les fichiers `server.key`, `server.crt`, et `root.crt` sont seulement examinés lors du lancement du serveur ; donc vous devez relancer le serveur pour prendre en compte les modifications.

## 16.9. Connexions TCP/IP sécurisées avec des tunnels SSH Tunnels

Vous pouvez utiliser SSH pour crypter la connexion réseau entre les clients et un serveur PostgreSQL. Réalisé correctement, ceci fournit une connexion réseau sécurisée, y compris pour les clients non SSL.

Tout d'abord, assurez−vous qu'un serveur SSH est en cours d'exécution sur la même machine que le serveur PostgreSQL et que vous pouvez vous connecter via `ssh` en tant qu'un utilisateur quelconque. Ensuite, vous

pouvez établir un tunnel sécurisé avec une commande comme ceci sur la machine cliente :

```
ssh -L 3333:foo.com:5432 joe@foo.com
```

Le premier numéro de l'argument `-L`, 3333, est le numéro de port de votre bout du tunnel ; il peut être choisi librement. Le second numéro, 5432, est le bout distant du tunnel : le numéro de port que votre serveur utilise. Le nom ou l'adresse entre les numéros de port est l'hôte disposant du serveur de bases de données auquel vous souhaitez vous connecter. Pour cela, vous vous connectez au port 3333 de votre machine locale :

```
psql -h localhost -p 3333 template1
```

Sur le serveur de bases de données, il semblera que vous êtes réellement l'utilisateur `joe@foo.com` et il utilisera la procédure d'authentification adéquate pour cet utilisateur. Pour que la configuration du serveur réussisse, vous devez être autorisé à vous connecter via `ssh` en tant que `joe@foo.com`, comme si vous essayez d'utiliser `ssh` pour configurer une session terminal.

> **Astuce :** Plusieurs autres applications existantes peuvent fournir des tunnels sécurisés en utilisant une procédure similaire dans le concept à celle que nous venons de décrire.

# Chapitre 17. Utilisateurs et droits de la base de données

Chaque groupe de bases de données contient un ensemble d'utilisateurs. Ces utilisateurs sont différents des utilisateurs gérés par le système d'exploitation sur lequel le serveur tourne. Les utilisateurs possèdent des objets de la base (par exemple des tables) et peuvent affecter des droits sur ces objets à d'autres utilisateurs pour contrôler qui a accès à quel objet.

Ce chapitre décrit comment créer et gérer des utilisateurs et introduit le système de droits. Plus d'informations sur les différents types d'objets de la base de données et les effets des droits sont disponibles dans le Chapitre 5.

## 17.1. Utilisateurs de la base de données

Conceptuellement, les utilisateurs de la base sont totalement séparés des utilisateurs du système d'exploitation. En pratique, il peut être commode de maintenir une correspondance mais cela n'est pas requis. Le nom des utilisateurs est global à toute une installation de groupe de bases de données (et non individuelle pour chaque base). Pour créer un utilisateur, utilisez la commande SQL *CREATE USER* :

```
CREATE USER nom_utilisateur;
```

*nom_utilisateur* suit les règles des identifiants SQL : soit sans guillemets et sans caractères spéciaux, soit entre double−guillemets. Pour supprimer un utilisateur existant, utilisez la commande analogue *DROP USER* :

```
DROP USER nom_utilisateur;
```

Pour une certaine facilité d'utilisation, les programmes createuser et dropuser sont fournis comme emballage de ces commandes SQL et peuvent être appelés depuis la ligne de commande du shell :

```
createuser nom_utilisateur
dropuser nom_utilisateur
```

Pour déterminer l'ensemble des utilisateurs existants, examinez le catalogue système `pg_user` existant, par exemple

```
SELECT usename FROM pg_user;
```

La méta−commande `\du` du programme psql est aussi utile pour lister les utilisateurs existants.

Afin d'amorcer le système de base de données, un système récemment installé contient toujours un utilisateur prédéfini. Cet utilisateur aura l'identifiant fixe 1 et aura par défaut (à moins que cela ne soit modifié en lançant la commande `initdb`) le même nom que l'utilisateur du système d'exploitation qui a initialisé le groupe de bases de données. Pour créer plus d'utilisateurs, vous devez d'abord vous connecter en tant que cet utilisateur initial.

Une identité utilisateur, et uniquement une, est active par connexion au serveur de bases de données. Le nom d'utilisateur à employer pour une connexion à une base particulière est indiqué par le client initialisant la demande de connexion et ce, de la manière qui lui est propre. Par exemple, le programme `psql` utilise l'option de ligne de commandes `-U` pour préciser sous quel utilisateur il se connecte. Beaucoup d'applications (incluant `createuser` et `psql`) utilisent par défaut le nom courant de l'utilisateur du système d'exploitation. Par conséquence, il peut être pratique de maintenir une correspondance de nommage entre les deux ensembles d'utilisateurs.

La configuration de l'authentification du client détermine avec quel utilisateur de la base, la connexion cliente donnée se connectera, comme cela est expliqué dans le Chapitre 19. (Donc, un client n'est pas nécessairement obligé de se connecter avec le même nom d'utilisateur que celui qu'il a dans le système d'exploitation ; de la même façon que le nom de connexion d'un utilisateur peut ne pas correspondre à son vrai nom.) Comme l'identité de l'utilisateur détermine l'ensemble des droits disponibles pour le client connecté, il est important de configurer cela soigneusement quand un environnement multi−utilisateurs est mis en place.

## 17.2. Attributs utilisateurs

Un utilisateur de bases de données peut avoir un certain nombre d'attributs qui définissent ses droits et interagissent avec le système d'authentification du client.

super−utilisateur (superuser)
: Un super−utilisateur d'une base passe au travers de toutes les vérifications de droits. De plus, seul un super−utilisateur peut créer de nouveaux utilisateurs. Pour créer un super−utilisateur de la base de données, utilisez `CREATE USER` *`nom_utilisateur`* `CREATEUSER`.

création de bases de données
: Les droits de création de bases doivent être explicitement données à un utilisateur (à l'exception des super−utilisateurs qui passent au travers de toutes vérifications de droits). Pour créer un tel utilisateur, utilisez `CREATE USER` *`nom_utilisateur`* `CREATEDB`.

mot de passe
: Un mot de passe est seulement significatif si la méthode d'authentification du client exige que le client fournisse un mot de passe quand il se connecte à la base. Les méthodes d'authentification par `mot de passe`, `md5` et `crypt` utilisent des mots de passe. Les mots de passe de la base de données ne sont pas les mêmes que ceux du système d'exploitation. Indiquez un mots de passe lors de la création d'un utilisateur avec `CREATE USER` *`nom_utilisateur`* `PASSWORD '`*`le_mot_de_passe`*`'`.

Les attributs d'un utilisateur peuvent être modifiés après sa création avec `ALTER USER`. Regardez les pages de références de ***CREATE USER*** et de ***ALTER USER*** pour plus de détails.

Un utilisateur peut aussi configurer ses options par défaut pour de nombreux paramètres de configuration décris dans la Section 16.4. Par exemple, si pour une raison ou une autre vous voulez désactiver les parcours d'index (conseil : ce n'est pas une bonne idée) à chaque fois que vous vous connectez, vous pouvez utiliser

```
ALTER USER nom_utilisateur SET enable_indexscan TO off;
```

Cela sauve les paramètres (mais ne les applique pas immédiatement) Dans les connexions ultérieures de cet utilisateur, c'est comme si `SET enable_indexscan TO off;` avait été appelé juste avant le démarrage de la session. Vous pouvez toujours modifier les paramètres durant la session. Pour défaire un des paramètres, utilisez `ALTER USER` *`nom_utilisateur`* `RESET` *`nom_variable`*`;`.

## 17.3. Groupes

Comme dans Unix, les groupes sont une manière logique de grouper les utilisateurs pour faciliter la gestion des privilèges : les droits peuvent être accordés ou révoqués à un groupe entier. Pour créer un groupe, utilisez la commande SQL *CREATE GROUP* :

```
CREATE GROUP nom_groupe;
```

Pour rajouter ou supprimer des utilisateurs d'un groupe, utilisez*ALTER GROUP* :

```
ALTER GROUP nom_group ADD USER nom_utilisateur_1, ... ;
ALTER GROUP nom_group DROP USER nom_utilisateur_1, ... ;
```

Pour détruire un groupe, utilisez *DROP GROUP* :

```
DROP GROUP name;
```

Ceci supprime uniquement le groupe, pas les utilisateurs membres du groupe.

Pour déterminer l'ensemble des groupes existants, examinez le catalogue système `pg_group`, par exemple :

```
SELECT groname FROM pg_group;
```

La méta−commande `\dg` du programme psql est aussi utile pour afficher les groupes existants.

## 17.4. Droits

Quand un objet est créé, il est affecté à un propriétaire. Ce dernier est habituellement l'utilisateur qui a exécuté l'instruction de création. Pour la plupart des objets, l'état initial est tel que seul le propriétaire (ou un superutilisateur) peut faire quelque chose avec cet objet. Afin de laisser les autres utilisateurs utiliser l'objet, des *droits* doivent être accordés. Il existe différents droits : `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `RULE`, `REFERENCES`, `TRIGGER`, `CREATE`, `TEMPORARY`, `EXECUTE` et `USAGE`. Pour plus d'informations sur le support des différents types de droits par PostgreSQL, regardez la page de référence *GRANT*.

Pour affecter des droits, la commande `GRANT` est utilisée. Ainsi, si `joe` est un utilisateur existant et `comptes` est une table existante, le droit pour mettre à jour la table peut être accordé avec

```
GRANT UPDATE ON comptes TO joe;
```

Pour accorder un droit à un groupe, utilisez

```
GRANT SELECT ON comptes TO GROUP staff;
```

Le nom spécial `PUBLIC` peut être utilisé pour accorder un privilège à chaque utilisateur du système. Écrire `ALL` à la place d'un droit spécifique signifie que tous les droits s'appliquant à l'objet seront accordés.

Pour révoquer un privilège, utilisez la commande nommée `REVOKE` :

```
REVOKE ALL ON accounts FROM PUBLIC;
```

Les droits spéciaux d'un propriétaire d'un objet (par exemple, le droit de modifier ou détruire un objet) sont toujours implicites et ne peuvent pas être accordés ou révoqués. Mais le propriétaire peut choisir de révoquer ses propres droits ordinaires, par exemple mettre une table en lecture seule pour lui−même aussi bien que pour les autres.

Un objet peut être affecté à un nouveau propriétaire avec une commande `ALTER` du genre approprié pour l'objet. Seuls les superutilisateurs peuvent faire ceci.

## 17.5. Fonctions et déclencheurs (triggers)

Les fonctions et les déclencheurs autorisent à l'intérieur du serveur les utilisateurs à insérer du code que d'autres utilisateurs ne connaissent pas mais peuvent exécuter. Par conséquent, les deux mécanismes permettent aux utilisateurs d'utiliser un << cheval de Troie >> contre d'autres avec une relative facilité. La seule protection réelle est d'effectuer un fort contrôle sur ceux qui peuvent définir des fonctions.

Les fonctions sont exécutées à l'intérieur du processus serveur avec les droits au niveau système d'exploitation du démon serveur de la base de données. Si le langage de programmation utilisé par la fonction autorise les accès mémoire non contrôlés, il est possible de modifier les structures de données internes du serveur. Du coup, parmi d'autres choses, de telles fonctions peuvent dépasser les contrôles d'accès au système. Les langages de fonctions qui permettent un tel accès sont considérées << sans confiance >> et PostgreSQL autorise uniquement les superutilisateurs à écrire des fonctions dans ces langages.

# Chapitre 18. Administration des bases de données

Chaque instance d'un serveur PostgreSQL gère une ou plusieurs bases de données. Les bases de données sont donc le niveau hiérarchique le plus élevé pour organiser des objets SQL (<< objets de base de données >>). Ce chapitre décrit les propriétés des bases de données et comment les créer, les administrer et les détruire.

## 18.1. Aperçu

Une base de données est un ensemble nommé d'objets SQL (<< objets de base de données >>). En général, chaque objet de base de données (table, fonction etc.) appartient à une et une seule base de données (mais certains catalogues système, par exemple `pg_database`, appartiennent à tout le groupe et sont accessibles depuis toutes les bases de données du groupe). Plus précisément, une base de données est une collection de schémas et les schémas contiennent les tables, fonctions, etc. Ainsi, la hiérarchie complète est : serveur, base de données, schéma, table (ou un autre type d'objet, comme une fonction).

Lors de la connexion au serveur de bases de données, une application cliente doit spécifier dans sa requête de connexion la base de données à laquelle elle veut se connecter. Il n'est pas possible d'accéder à plus d'une base de données via la même connexion (mais une application n'est pas limitée dans le nombre de connexions qu'elle établit avec une ou plusieurs bases de données). Les bases de données sont séparées physiquement et le contrôle d'accès est géré au niveau de la connexion. Si une instance de serveur PostgreSQL doit héberger des projets ou des utilisateurs censés rester séparés et sans interaction, il est recommandé de les répartir sur plusieurs bases de données. Si les projets ou les utilisateurs sont reliés et doivent pouvoir partager leurs ressources, alors ils devraient être placés dans la même base de données mais éventuellement dans des schémas différents. Les schémas sont une structure purement logique et qui peut accéder à ce qui est géré par le système des droits. Pour plus d'informations sur la manipulation des schémas, voir la Section 5.8.

Les bases de données sont créées avec la commande `CREATE DATABASE` (voir la Section 18.2) et détruites avec la commande `DROP DATABASE` (voir la Section 18.5). Pour déterminer l'ensemble des bases de données existantes, examinez le catalogue système `pg_database`, par exemple

```
SELECT datname FROM pg_database;
```

La méta−commande `\l` du programme psql et l'option en ligne de commande `-l` sont aussi utiles pour afficher les bases de données existantes.

> **Note :** Le standard SQL appelle les bases de données des << catalogues >> mais il n'y a aucune différence en pratique.

## 18.2. Création d'une base de données

Pour pouvoir créer une base de données, il faut que le serveur PostgreSQL soit lancé (voir la Section 16.3).

Les bases de données sont créées à l'aide de la commande SQL CREATE DATABASE:

```
CREATE DATABASE nom;
```

ou *nom* suit les règles habituelles pour les identifiants SQL. L'utilisateur actuel devient automatiquement le propriétaire de la nouvelle base de données. C'est au propriétaire de la base de données qu'il revient de la supprimer par la suite (ce qui supprime aussi tous les objets qu'elle contient, même s'ils ont un propriétaire différent).

La création de bases de données est une opération protégée. Voir la Section 17.2 sur la manière d'attribuer des droits.

Comme vous devez être connecté au serveur de base de données pour exécuter la commande `CREATE DATABASE`, reste à savoir comment créer la première base de données d'un site. La première base de données est toujours créée par la commande `initdb` quand l'aire de stockage des données est initialisée. (Voir la Section 16.2.) Cette base de données est appelée `template1`. Donc, pour créer la première base de données proprement dite, vous pouvez vous connecter à `template1`.

Le nom `template1` n'a pas été choisi au hasard : quand une nouvelle base de données est créée, la base de donnée modèle `template1` est en fait clonée. Cela signifie que tous les changements effectués sur `template1` sont propagés à toutes les bases de données créées ultérieurement. Cela implique que vous ne devriez pas utiliser la base de données modèle pour votre travail quotidien mais cette propriété, utilisée judicieusement, peut être utile. Pour plus de détails, voir la Section 18.3.

Pour plus de confort, il existe aussi un programme que vous pouvez exécuter à partir du shell pour créer de nouvelles bases de données, `createdb`.

```
createdb nom_base
```

`createdb` ne fait rien de magique. Il se connecte à la base de données `template1` et exécute la commande `CREATE DATABASE`, exactement comme ci−dessus. La page de référence sur createdb contient les détails de son invocation. Notez que `createdb` sans aucun argument crée une base de donnée portant le nom de l'utilisateur courant, ce qui n'est peut−être pas ce que vous voulez.

> **Note :** Le Chapitre 19 contient des informations sur la manière de restreindre l'accès à une base de données.

Parfois vous voulez créer une base de données pour quelqu'un d'autre. Cet utilisateur doit devenir le propriétaire de la nouvelle base de données, afin de pouvoir la configurer et l'administrer lui−même. Pour faire ceci, utilisez l'une des commandes suivantes :

```
CREATE DATABASE nom_base OWNER
nomutilisateur;
```

dans l'environment SQL ou

```
createdb -O nomutilisateur nom_base
```

dans le shell. Vous devez être super−utilisateur pour créer une base de données pour quelqu'un d'autre.

# 18.3. Bases de données modèles

En fait, CREATE DATABASE fonctionne en copiant une base de données préexistante. Par défaut, cette commande copie la base de données système standard template1. Ainsi, cette base de données est le << modèle >> à partir duquel de nouvelles bases de données sont créées. Si vous ajoutez des objets à template1, ces objets seront copiés dans les bases de données utilisateur créées ultérieurement. Ce comportement permet d'apporter des modifications locales au jeu standard d'objets des bases de données. Par exemple, si vous installez le langage de procédures PL/pgSQL dans template1, celui−ci sera automatiquement disponible dans les bases de données utilisateur sans qu'il soit nécessaire de faire quelque chose de spécial au moment où ces bases de données sont créées.

Il y a une seconde base de données système standard appelée template0. Cette base de données contient les mêmes données que le contenu initial de template1, c'est−à−dire seulement les objets standards prédéfinis dans votre version de PostgreSQL. template0 ne devrait jamais être modifiée après initdb. En indiquant à CREATE DATABASE de copier template0 au lieu de template1, vous pouvez créer une base de données utilisateur << vierge >> qui ne contient aucun des ajouts locaux à template1. Ceci est particulièrement pratique quand on restaure une sauvegarde réalisé avec pg_dump : le script de dump devrait être restauré dans une base de données vierge pour être sûr de recréer le contenu correct de la base de données sauvegardée, sans survenue de conflits avec des objets qui auraient été ajoutés à template1.

Pour créer une base de données à partir de template0, utilisez

```
CREATE DATABASE nom_base TEMPLATE template0;
```

dans l'environnement SQL ou

```
createdb −T template0 nom_base
```

dans le shell.

Il est possible de créer des bases de données modèles supplémentaires et, à vrai dire, on peut copier n'importe quelle base de données d'un cluster en la désignant comme modèle pour la commande CREATE DATABASE. Cependant, il importe de comprendre, que ceci n'est pas (encore) à prendre comme une commande << COPY DATABASE >> de portée générale. En particulier, il est essentiel que la base de données source soit inactive (pas de transactions en écriture en cours) pendant toute la durée de l'opération de copie. CREATE DATABASE vérifie qu'aucune autre session que la sienne n'est connectée à la base de données source au début de l'opération mais ceci ne garantie pas que des changements ne peuvent pas être effectués pendant le déroulement de la copie, ce qui aboutirait à une base de données copiée incohérente. C'est pourquoi nous recommandons que les bases de données utilisées comme modèles soient mises en lecture seule.

Deux drapeaux utiles existent dans pg_database pour chaque base de données : les colonnes datistemplate et datallowconn. datistemplate peut être positionné à vrai pour indiquer qu'une base de données a vocation à servir de modèle à CREATE DATABASE. Si ce drapeau est positionné à vrai, la base de données peut être clonée par tout utilisateur ayant le droit CREATEDB ; s'il est positionné à faux, seuls les super−utilisateurs et le propriétaire de la base de données peuvent la cloner. Si datallowconn est positionné à faux, alors aucune nouvelle connexion à cette base de données n'est autorisée (mais les sessions existantes ne sont pas tuées simplement en positionnant ce drapeau à faux). La base de données template0 est normalement marquée datallowconn = false pour empêcher qu'elle ne soit modifiée. Aussi bien template0 que template1 devraient toujours être marquées datistemplate = true.

Après avoir préparé ou modifé une base de données modèle, exécuter les commandes VACUUM FREEZE dans cette base de données est une bonne idée. Si cela se fait alors qu'il n'y a pas d'autre transaction ouverte dans la même base de données, alors il est garanti que toutes les lignes de la base de données seront << gelées >> et ne seront pas sujettes à des problèmes de réutilisation d'ID de transaction déjà attribués. C'est particulièrement important pour une base de données qui aura le drapeau datallowconn positionné à faux puisqu'il sera impossible d'effectuer les VACUUM de maintenance sur une telle base de données. Voir la Section 21.1.3 pour plus d'informations.

> **Note :** template1 et template0 n'ont pas de caractère particulier en dehors du fait que template1 est la base de données source par défaut pour la commande CREATE DATABASE et la base de données à laquelle se connectent par défaut divers programmes comme createdb. Par exemple, on pourrait supprimer template1 et la recréer à partir de template0 sans effet secondaire gênant. Ce procédé peut être utile lorsqu'on a encombré template1 d'objets inutiles.

# 18.4. Configuration d'une base de données

Comme il est dit dans le Section 16.4, le serveur PostgreSQL offre un grand nombre de variables de configuration à chaud. Vous pouvez spécifier des valeurs par défaut, valables pour une base de données particulière, pour nombre de ces variables.

Par exemple, si pour une raison quelconque vous voulez désactiver l'optimiseur GEQO pour une base de donnée particulière, vous n'avez pas besoin de le désactiver pour toutes les bases de données ou de faire en sorte que tout client se connectant exécute la commande SET geqo TO off;. Pour appliquer ce réglage par défaut à la base de données en question, vous pouvez exécuter la commande

```
ALTER DATABASE ma_base SET geqo TO off;
```

Cela sauvegarde le réglage (mais ne l'applique pas immédiatement). Lors des connexions ultérieures à cette base de données, tout se passe comme si la commande SET geqo TO off; est exécutée juste avant de commencer la session. Notez que les utilisateurs peuvent cependant modifier ce réglage pendant la session ; il s'agit seulement d'un réglage par défaut. Pour annuler un tel réglage par défaut, utilisez ALTER DATABASE *nom_base* RESET *nomvariable*;.

# 18.5. Détruire une base de données

Les bases de données sont détruites avec la commande DROP DATABASE :

```
DROP DATABASE nom;
```

Seul le propriétaire de la base de données (c'est–à–dire l'utilisateur qui l'a créé) ou un superutilisateur peut supprimer une base de données. Supprimer une base de données supprime tous les objets qui étaient contenus dans la base. La destruction d'une base de données ne peut pas être annulée.

Vous ne pouvez pas exécuter la commande DROP DATABASE en étant connecté à la base de données cible. Néanmoins, vous pouvez être connecté à une autre base de données, ceci incluant la base template1. template1 pourrait être la seule option pour supprimer la dernière base utilisateur d'un groupe donné.

Pour une certaine facilité, il existe un script shell qui supprime les bases de données, dropdb :

```
dropdb nom_base
```

(Contrairement à `createdb`, l'action par défaut n'est pas de supprimer la base possédant le nom de l'utilisateur en cours.)

# 18.6. Espaces logiques

Les espaces logiques dans PostgreSQL permettent aux administrateurs de bases de données de définir l'emplacement dans le système de fichiers où seront stockés les fichiers représentant les objets de la base de données. Une fois créé, un espace logique peut être référencé par son nom lors de la création d'objets.

En utilisant les espaces logiques, un administrateur peut contrôler les emplacements sur le disque d'une installation PostgreSQL. Ceci est utile dans au moins deux cas. Tout d'abord, si la partition ou le volume sur lequel le groupe a été initialisé arrive à court d'espace disque mais ne peut pas être étendu, un espace logique peut être créé sur une partition différente et utilisé jusqu'à ce que le système soit reconfiguré.

Deuxièmement, les espaces logiques permettent à un administrateur d'utiliser sa connaissance des objets de la base pour optimiser les performances. Par exemple, un index qui est très utilisé peut être placé sur un disque très rapide et disponible, comme un périphérique mémoire. En même temps, une table stockant des données archivées et peu utilisée ou dont les performances ne portent pas à conséquence pourra être stockée sur un disque système plus lent, moins cher.

Pour définir un espace logique, utilisez la commande CREATE TABLESPACE, par exemple :

```
CREATE TABLESPACE espace_rapide LOCATION '/mnt/sda1/postgresql/data';
```

L'emplacement doit être un répertoire existant, possédé par l'utilisateur système PostgreSQL. Tous les objets créés par la suite dans l'espace logique seront stockés dans des fichiers contenus dans ce répertoire.

> **Note :** Il n'y a généralement aucune raison de créer plus d'un espace logique sur un système de fichiers logique car vous ne pouvez pas contrôler l'emplacement des fichiers individuels à l'intérieur de ce système de fichiers logique. Néanmoins, PostgreSQL ne vous impose aucune limitation et, en fait, il n'est pas directement conscient des limites du système de fichiers sur votre système. Il stocke juste les fichiers dans les répertoires que vous lui indiquez.

La création d'un espace logique lui−même doit être fait en tant que superutilisateur de la base de données mais, après cela, vous pouvez autoriser des utilisateurs standards de la base de données à l'utiliser. Pour cela, donnez−leur le droit `CREATE` sur l'espace logique.

Les tables, index et des bases de données entières peuvent être affectés à des espaces logiques particuliers. Pour cela, un utilisateur disposant du droit `CREATE` sur un espace logique donné doit passer le nom de l'espace logique comme paramètre de la commande. Par exemple, ce qui suit crée une table dans l'espace logique `espace1` :

```
CREATE TABLE foo(i int) TABLESPACE espace1;
```

Autrement, utilisez le paramètre default_tablespace :

```
SET default_tablespace = espace1;
CREATE TABLE foo(i int);
```

Quand default_tablespace est configuré avec autre chose qu'une chaîne vide, il fournit une clause TABLESPACE implicite pour les commandes CREATE TABLE et CREATE INDEX qui n'en ont pas d'explicites.

L'espace logique associé avec une base de données est utilisé pour stocker les catalogues système de la base, ainsi que tous les fichier temporaires créés par les processus serveur utilisant cette base de données. De plus, il est l'espace par défaut pour les tables et index créés à l'intérieur de cette base de données si aucune clause TABLESPACE n'est fournie (soit explicitement soit via default_tablespace) lors de la création des objets. Si une base de données est créée sans spécifier d'espace logique pour elle, le serveur utilise le même espace logique que celui de la base modèle utilisée comme copie.

Deux espaces logiques sont automatiquement créés par initdb. L'espace logique pg_global est utilisé pour les catalogues système partagés. L'espace logique pg_default est l'espace logique par défaut des bases de données template1 et template0 (et, du coup, sera l'espace logique par défaut pour les autres bases de données sauf en cas de surcharge par une clause TABLESPACE dans CREATE DATABASE).

Une fois créé, un espace logique peut être utilisé à partir de toute base de données si l'utilisateur le souhaitant dispose du droit nécessaire. Ceci signifie qu'un espace logique ne peut pas supprimé tant que tous les objets de toutes les bases de données utilisant l'espace logique n'ont pas été supprimés.

Pour supprimer un espace logique vide, utilisez la commande DROP TABLESPACE.

Pour déterminer l'ensemble d'espaces logiques existants, examinez le catalogue système pg_tablespace, par exemple

```
SELECT spcname FROM pg_tablespace;
```

La métacommande \db du programme psql est aussi utile pour afficher les espaces logiques existants.

PostgreSQL fait une utilisation intensive des liens symboliques pour simplifier l'implémentation des espaces logiques. Ceci signifie que les espaces logiques peuvent être utilisés *seulement* sur les systèmes supportant les liens symboliques.

Le répertoire $PGDATA/pg_tblspc contient des liens symboliques qui pointent vers chacun des espaces logiques utilisateur dans le groupe. Bien que non recommandé, il est possible d'ajuster la configuration des espaces logiques à la main en redéfinissant ces liens. Deux avertissements : ne pas le faire alors que postmaster est en cours d'exécution, mettez à jour le catalogue pg_tablespace pour indiquer les nouveaux emplacements (si vous ne le faites pas, pg_dump continuera à afficher les anciens emplacements des espaces logiques).

# Chapitre 19. Authentification du client

Quand une application client se connecte au serveur de base de données, elle indique le nom de l'utilisateur PostgreSQL sous lequel elle désire se connecter, comme lorsqu'on se connecte sur un ordinateur Unix sous un nom d'utilisateur particulier. Au sein de l'environnement SQL, le nom d'utilisateur de la base de données active détermine les droits régissant l'accès aux objets de la base de données — voir le Chapitre 17 pour plus d'informations. Ainsi, il est essentiel de limiter le nombre des bases de données auxquelles les utilisateurs peuvent se connecter.

L'*authentification* est le processus par lequel le serveur de bases de données établit l'identité du client et, par extension, par lequel il détermine si l'application cliente (ou l'utilisateur sous le nom de laquelle elle tourne) est autorisée à se connecter sous le nom d'utilisateur demandé.

PostgreSQL offre quantité de méthodes d'authentification différentes. La méthode d'authentification d'une connection client particulière peut être sélectionnée d'après l'adresse, la base de données et l'utilisateur de l'hôte client.

Les noms d'utilisateurs PostgreSQL sont séparés de façon logique des noms d'utilisateurs du système d'exploitation sur lequel tourne le serveur. Si tous les utilisateurs d'un serveur donné ont aussi des comptes sur la machine serveur, il peut être pertinent d'attribuer des noms d'utilisateurs de la base de données qui correspondent aux noms d'utilisateurs du système d'exploitation. Cependant, un serveur qui accepte les connexions distantes peut avoir plusieurs utilisateurs de base de données dépourvus de compte correspondant sur le système d'exploitation, dans de tels cas il n'y a pas besoin de correspondance entre noms d'utilisateurs de bases de données et noms d'utilisateurs du système d'exploitation.

## 19.1. Le fichier `pg_hba.conf`

L'authentification du client est contrôlée par un fichier, traditionnellement nommé `pg_hba.conf` et situé dans le répertoire data du groupe de bases de données, par exemple `/usr/local/pgsql/data/pg_hba.conf` (HBA signifie << host−based authentication >> : authentification fondée sur l'hôte.) Un fichier `pg_hba.conf` par défaut est installé lorsque le répertoire data est initialisé par `initdb`. Néanmoins, il est possible de placer le fichier de configuration de l'authentification ailleurs ; voir le paramètre de configuration hba_file.

Le format général du fichier `pg_hba.conf` est un ensemble d'enregistrements, un par ligne. Les lignes vides sont ignorées tout comme n'importe quel texte placé après le caractère de commentaire `#`. Un enregistrement est constitué d'un certain nombre de champs séparés par des espace et/ou des tabulations. Les champs peuvent contenir des espaces si la valeur du champ est mise entre guillemets. Un enregistrement ne peut pas être continué sur plusieurs lignes.

Chaque enregistrement détermine un type de connexion, une plage d'adresses IP (si approprié au type de connexion), un nom de base de données, un nom d'utilisateur et la méthode d'authentification à utiliser pour les connexions correspondant à ces paramètres. Le premier enregistrement correspondant au type de connexion, à l'adresse client, à la base de données demandée et au nom d'utilisateur est utilisé pour effectuer l'authentification. Il n'y a pas de suite après erreur (<< fall−through >> ou << backup >>) : si un enregistrement est choisi et que l'authentification échoue, les enregistrements suivants ne sont pas considérés. Si aucun enregistrement ne correspond, l'accès est refusé.

Un enregistrement peut avoir l'un des formats suivants.

```
local      database  user  authentication-method  [authentication-option]
host       database  user  CIDR-address  authentication-method  [authentication-option]
hostssl    database  user  CIDR-address  authentication-method  [authentication-option]
hostnossl  database  user  CIDR-address  authentication-method  [authentication-option]
host       database  user  IP-address  IP-mask  authentication-method  [authentication-option]
hostssl    database  user  IP-address  IP-mask  authentication-method  [authentication-option]
hostnossl  database  user  IP-address  IP-mask  authentication-method  [authentication-option]
```

La signification des champs est la suivante :

`local`
: Cet enregistrement intercepte les tentatives de connexion utilisant les sockets du domaine Unix. Sans un enregistrement de ce type, les connections de sockets du domaine Unix ne sont pas permises.

`host`
: Cet enregistrement intercepte les tentatives de connexion utilisant les réseaux TCP/IP. Les lignes `host` enregistrent des tentatives de connexion soit SSL soit non SSL.

> **Note :** Supprimer les connexions TCP/IP ne sera pas possible si le serveur est lancé avec la valeur appropriée pour le paramètre de configuration listen_addresses car le comportement par défaut est d'écouter les connexions TCP/IP provenant seulement de l'adresse local `localhost`.

`hostssl`
: Cet enregistrement intercepte les tentatives de connexions utilisant TCP/IP mais seulement quand ces connexions utilisent le chiffrement SSL.

  Pour être en mesure de faire usage de cette fonction, le serveur doit être compilé avec le support de SSL. De plus, SSL doit être activé au lancement du serveur en positionnant le paramètre de configuration ssl (voir la Section 16.8 pour plus d'informations).

`hostnossl`
: Cet enregistrement a une logique opposée à `hostssl` : il n'intercepte que les tentatives de connexion n'utilisant pas SSL.

*`database`*
: Indique quelles bases de données l'enregistrement concerne. La valeur `all` indique qu'il concerne toutes les bases de données. La valeur `sameuser` spécifie que l'enregistrement n'intercepte que si la base de données demandée a le même nom que l'utilisateur demandé. La valeur `samegroup` spécifie que l'utilisateur demandé doit être membre du groupe portant le même nom que la base de données demandée. Sinon, c'est le nom d'une base de données PostgreSQL particulière. Des noms de bases de données multiples peuvent être fournis en les séparant par des virgules. Un fichier contenant des noms de bases de données peut être indiqué en faisant précéder le nom de fichier de `@`.

*`user`*
: Indique à quels utilisateurs PostgreSQL cet enregistrement correspond. La valeur `all` indique qu'il concerne tous les utilisateurs. Autrement, c'est le nom d'un utilisateur PostgreSQL particulier. Plusieurs noms d'utilisateurs peuvent être fournis en les séparant avec des virgules. Les noms de groupes peuvent être spécifiés en précédant le nom de groupe du signe +. Un fichier contenant des noms d'utilisateurs peut être indiqué en faisant précéder le nom de fichier du signe `@`.

*`CIDR-address`*
: Spécifie l'échelle d'adresses IP du client auquel correspond cet enregistrement. Il contient une adresse IP dans la notation décimale standard et une longueur de masque CIDR (les adresses IP peuvent seulement être spécifiées numériquement, mais pas en tant que nom d'hôte ou de domaine). La longueur du masque indique le nombre de bits pour lequel une correspondance doit être apportée avec

l'adresse IP du client. Les bits à droite doivent valoir zéro dans l'adresse IP indiquée. Il ne doit y avoir aucun espace blanc entre l'adresse IP, le `/` et la longueur du masque CIDR.

Une adresse CIDR (*`CIDR-address`*) est typiquement `172.20.143.89/32` pour un hôte seul ou `172.20.143.0/24` pour un réseau. Pour spécifier un seul hôte, utilisez un masque de 32 pour IPv4 ou 128 pour IPv6.

Une adresse IP au format IPv4 correspondra aux connexions IPv6 qui auront l'adresse correspondante. Par exemple, `127.0.0.1` correspondra à l'adresse IPv6 `::ffff:127.0.0.1`. Une entrée donnée au format IPv6 correspondra uniquement aux connexions IPv6 même si l'adresse représentée est dans le domaine IPv4−vers−IPv6. Notez que les adresses au format IPv6 seront rejetées si la bibliothèque système C ne supporte pas les adresses IPv6.

Ce champ ne concerne que les enregistrements `host`, `hostssl` et `hostnossl`.

*`IP-address`*
*`IP-mask`*

Ces champs pourraient être utilisés comme alternative à la notation *`CIDR-address`*. Au lieu de spécifier la longueur du masque, le masque actuel est spécifié comme une colonne séparée. Par exemple, `255.0.0.0` représente une longueur de masque CIDR IPv4 de 8, et `255.255.255.255` représente une longueur de masque de 32.

Ces champ ne concernent que les enregistrements `host`, `hostssl` et `hostnossl`.

*`authentication-method`*

Détermine la méthode d'authentification à utiliser lors d'une connexion via cet enregistrement. Les choix possibles sont résumés ici ; les détails se trouvent dans la Section 19.2.

`trust`
: Autorise la connexion sans conditions. Cette méthode permet à n'importe qui de se connecter au serveur de bases de données PostgreSQL, de s'enregistrer comme n'importe quel utilisateur PostgreSQL de son choix sans nécessiter de mot de passe. Voir la Section 19.2.1 pour les détails.

`reject`
: Rejette la connexion sans conditions. Ce cas est utile pour << filtrer >> certains hôtes d'un groupe.

`md5`
: Demande au client de fournir un mot de passe chiffré MD5 pour son authentification. Voir la Section 19.2.2 pour les détails.

`crypt`
: Requiert que le client fournisse un mot de passe crypté avec `crypt()` pour l'authentification. `md5` est préféré pour les clients 7.2 et suivants mais les versions antérieures ne supportent que `crypt`. Voir Section 19.2.2 pour les détails.

`password`
: Requiert que le client fournisse un mot de passe non crypté pour l'authentification. `md5` est préféré sur les clients 7.2 et ultérieures mais les versions antérieures ne supportent que `crypt`. Voir Section 19.2.2 pour les détails.

`krb4`
: Utilise Kerberos V4 pour authentifier l'utilisateur. Ceci n'est disponible que pour les connexions TCP/IP. Voir Section 19.2.3 pour les détails.

`krb5`
: Utilise Kerberos V5 pour authentifier l'utilisateur. Ceci n'est disponible que pour les connexions TCP/IP. Voir Section 19.2.3 pour les détails.

`ident`
: Récupère le nom de l'utilisateur du système d'exploitation du client (pour les connexions TCP/IP en contactant le serveur d'identification sur le client, pour les connexions locales, en l'obtenant du système d'exploitation) et vérifie si l'utilisateur est autorisé à se connecter en tant qu'utilisateur de la base de données demandé en consultant la correspondance indiquée après le mot clé `ident`. Voir la Section 19.2.4 ci−dessous pour les détails.

`pam`
: Authentifie en utilisant les Pluggable Authentification Modules (PAM) fournis par le système d'exploitation. Voir la Section 19.2.5 pour les détails.

*`authentication-option`*
: La signification de ce champ optionnel dépend de la méthode d'authentification choisie. Des détails sont disponibles ci−dessous.

Les fichiers inclus par les constructions `@` sont lus comme des listes de noms, qui peuvent être séparés par soit des espaces blancs soit des virgules. Les commentaires sont introduits par le caractère `#` comme dans `pg_hba.conf`, et les constructions `@` imbriquées sont autorisées. Sauf si le nom du fichier suivant `@` est un chemin absolu, il est supposé relatif au répertoire contenant le fichier le référençant.

Les enregistrements du fichier `pg_hba.conf` sont examinés séquentiellement pour chaque tentative de connexion, l'ordre des enregistrements est significatif. Généralement, les premiers enregistrements auront des paramètres d'interception de connexions plus stricts alors que les enregistrements suivants auront des paramètres plus larges et des méthodes d'authentification plus fortes. Par exemple, on pourrait souhaiter utiliser l'authentification `trust` pour les connexions TCP/IP locales mais demander un mot de passe pour les connexion TCP/IP distantes. Dans ce cas, un enregistrement spécifiant une authentification `trust` pour les connexions issues de 127.0.0.1 apparaîtrait avant un enregistrement spécifiant une authentifications par mot de passe pour une plage plus étendue d'adresses IP client autorisées.

Le fichier `pg_hba.conf` est lu au démarrage et quand le processus serveur principal (`postmaster`) reçoit un signal SIGHUP. Si vous éditez le fichier sur un système actif, vous aurez à signaler au `postmaster` (en utilisant `pg_ctl reload` ou `kill -HUP`) de relire le fichier.

Quelques exemples des entrées de `pg_hba.conf` sont décrit ci−dessous dans l'Exemple 19−1. Voir la section suivante pour les détails des méthodes d'authentification.

**Exemple 19−1. Exemple d'entrées de `pg_hba.conf`**

```
# Permet à n'importe quel utilisateur du système local de se connecter à la base
# de données sous n'importe quel nom d'utilisateur en utilisant les sockets du
# domaine Unix. (par défaut pour les connexions locales)
#
# TYPE  DATABASE    USER        CIDR-ADDRESS        METHOD
local   all         all                             trust

# Identique à ci-dessus mais utilise les connexions TCP/IP locales loopback.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS        METHOD
host    all         all         127.0.0.1/32        trust

# Identique à la dernière ligne mais en utilisant une colonne netmask séparée CDIR.
#
# TYPE  DATABASE    USER        IP-ADDRESS          IP-mask             METHOD
host    all         all         127.0.0.1           255.255.255.255     trust
```

```
# Permet à n'importe que utilisateur de n'importe quel hôte avec l'adresse IP
# 192.168.93.x de se connecter à la base de données "template1" sous le même nom
# d'utilisateur que l'identification le signale à la connexion (généralement le
# nom utilisateur Unix).
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    template1   all         192.168.93.0/24       ident sameuser

# Permet à un utilisateur de l'hôte 192.168.12.10 de se connecter à la base de
# données "template1" si le mot de passe de l'utilisateur est fourni sans
# erreur.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    template1   all         192.168.12.10/32      md5

# En l'absence de lignes "host" antérieures, ces deux lignes rejetteront toutes
# les connexions en provenance de 192.168.54.1 (puisque cette entrée déclenchera
# en premier), mais autorisera les connexions Kerberos 5 de n'importe où
# ailleurs sur l'Internet. Le masque zéro signifie qu'aucun bit sur l'ip de
# l'hôte n'est considéré, de sorte à correspondre à tous les hôtes.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    all         all         192.168.54.1/32       reject
host    all         all         0.0.0.0/0             krb5

# Permet à tous les utilisateurs de se connecter depuis 192.168.x.x à n'importe
# quelle base de données si ils passent la verification d'identification. Si,
# par exemple, l'identification indique que l'utilisateur est "bryanh" et qu'il
# demande à se connecter en tant qu'utilisateur PostgreSQL "guest1", la
# connexion n'est permise que s'il existe une entrée dans pg_ident.conf pour la
# correspondance "omicron" disant que "bryanh" est autorisé à se connecter en
# tant que "guest1".
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    all         all         192.168.0.0/16        ident omicron

# Si ce sont les trois seules lignes traitant les connexions locales, elles
# autoriseront les utilisateurs locaux à se connecter uniquement à leur propre
# base de données (bases de données ayant le même nom que leur nom
# d'utilisateur) exception faite pour les administrateurs et les membres du
# groupe "support" qui peuvent se connecter à toutes les bases de données.  Le
# fichier $PGDATA/admins contient une liste de noms d'utilisateurs. Un mot de
# passe est requis dans tous les cas.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
local   sameuser    all                               md5
local   all         @admins                           md5
local   all         +support                          md5

# Les deux dernières lignes ci-dessus peuvent être combinées en une seule ligne:
local   all         @admins,+support                                md5

# La colonne database peut aussi utiliser des listes et des noms de fichiers
# mais pas de groupes:
local   db1,db2,@demodbs  all                                       md5
```

# 19.2. Méthodes d'authentification

Les sous−sections suivantes décrivent les méthodes d'authentification plus en détail.

### 19.2.1. Authentification Trust

Quand l'authentification `trust` est spécifiée, PostgreSQL suppose que n'importe qui pouvant se connecter au serveur est autorisé à accéder à la base de données quel que soit le nom d'utilisateur de base de données qu'il fournisse (incluant le super−utilisateur de la base de données). Bien sûr, les restrictions apportées dans les colonnes `database` et `user` s'appliquent toujours. Cette méthode ne devrait être utilisée que s'il existe des protections au niveau système portant sur les connexions au serveur.

L'authentification `trust` est appropriée et très pratique lors de connexions locales sur une station de travail mono−utilisateur. Elle n'est généralement *pas* appropriée en soi sur une machine multi−utilisateur. Cependant, vous pouvez utiliser `trust` même sur une machine multi−utilisateur, si vous restreignez l'accès au fichier socket du domaine Unix en utilisant les permissions du système de fichiers. Pour ce faire, positionnez les paramètres de configuration `unix_socket_permissions` (et si besoin `unix_socket_group`) comme décrit dans la Section 16.4.2. Vous pouvez aussi positionner le paramètre de configuration `unix_socket_directory` de façon à placer le fichier de socket dans un répertoire à l'accès convenablement restreint.

Utiliser les droits du système de fichiers n'est utile que dans le cas de connexions utilisant des sockets Unix. Cela ne restreint pas les connexions TCP/IP locales ; ainsi, si vous voulez utiliser les droits du système de fichiers pour assurer la sécurité locale, supprimez la ligne `host ...127.0.0.1 ...` de `pg_hba.conf` ou changez−la − indiquer une méthode d'authentification différente de `trust`.

L'authentification `trust` n'est utile pour les connexions TCP/IP que si chaque utilisateur de chaque machine autorisée à se connecter au serveur par les lignes `pg_hba.conf` indiquant `trust` est digne de confiance. Il est rarement raisonnable d'utiliser `trust` pour une connexion autre que celles issues de localhost (127.0.0.1).

### 19.2.2. Authentification par mot de passe

Les méthodes basées sur une authentification par mot de passe sont `md5`, `crypt` et `password`. Ces méthodes fonctionnent de façon analogue, sauf pour le mode d'envoi du mot de passe au travers de la connexion. Néanmoins, `crypt` n'autorise pas le stockage des mots de passe cryptées dans `pg_shadow`.

Si vous êtes préoccupé par les attaques par << interception (sniffing) >> de mot de passe, alors `md5` est préférable, avec `crypt` en second choix si vous devez supporter les client pré 7.2. Le simple `password` devrait particulièrement être évité pour les connexion sur l'Internet ouvert (à moins d'utiliser SSL, SSH ou d'autres systèmes de sécurité par encapsulation de connexion).

Les mots de passe de bases de données PostgreSQL sont distincts des mots de passe du système d'exploitation. Le mot de passe de chaque utilisateur est enregistré dans la table catalogue système `pg_shadow`. Les mots de passes peuvent être gérés avec les commandes SQL *CREATE USER* et *ALTER USER*, par exemple **`CREATE USER foo WITH PASSWORD 'secret';`**. Par défaut, si aucun mot de passe n'a été fixé, le mot de passe enregistré sera nul et l'authentification par mot de passe échouera systématiquement pour cet utilisateur.

### 19.2.3. Authentification Kerberos

Kerberos est un système d'authentification sécurisé de standard industriel destiné à l'informatique distribuée sur un réseau public. Une description du système Kerberos est bien au−delà des objectifs de ce document

c'est généralement assez complexe (bien que puissant). La FAQ Kerberos ou le projet Athena du MIT peuvent être un bon point de départ pour une exploration. Il existe plusieurs sources de distribution Kerberos.

Bien que PostgreSQL supporte Kerberos 4 et 5, seul Kerberos 5 est recommandé. Kerberos 4 est considéré peu sûr et n'est plus recommandé pour un usage classique.

Pour utiliser Kerberos, son support doit être activé au moment de la compilation. Voir le Chapitre 14 pour plus d'informations. Kerberos 4 et 5 sont supportés mais une seule version peut être activée lors d'une compilation.

PostgreSQL fonctionne comme un service Kerberos normal. Le nom du service principal est *nom_service*/*nom_hote*@*domaine*, où *servicename* est `postgres` (à moins qu'un nom de service différent soit sélectionné lors de la configuration avec `./configure --with-krb-srvnam=quelquechose`). *nom_hote* est le nom de l'hôte pleinement qualifié (fully qualified host name) de la machine serveur. Le domaine principal du service est le domaine préféré du serveur.

Les principaux clients doivent avoir leur nom d'utilisateur PostgreSQL comme premier composant, par exemple `nomutilisateurpg/autreschoses@domaine`. Actuellement, le domaine du client n'est pas vérifié par PostgreSQL ; ainsi si vous avez activé l'authentification "cross−realm", chaque "principal" de chaque domaine qui peut communiquer avec le vôtre sera accepté.

Assurez−vous que le fichier de clés du serveur est en lecture (et de préférence en lecture seule) pour le compte serveur PostgreSQL (voir aussi la Section 16.1). L'emplacement du fichier de clés est indiqué grâce au paramètre de configuration krb_server_keyfile fourni à l'exécution. (Voir aussi Section 16.4.) Par défaut le fichier est `/etc/srvtab` si vous utilisez Kerberos 4 et `/usr/local/pgsql/etc/krb5.keytab` (ou le répertoire spécifié par `sysconfdir` à la compilation) avec Kerberos 5.

Pour générer le fichier keytab, utilisez par exemple (avec la version 5) :

```
kadmin% ank −randkey
postgres/server.my.domain.org kadmin% ktadd −k
krb5.keytab postgres/server.my.domain.org
```

Lisez la documentation Kerberos pour les détails.

Lors de la connexion à la base de données, assurez−vous que vous avez un ticket pour un "principal" correspondant au nom d'utilisateur de la base de données demandé. Exemple : pour le nom d'utilisateur de la base `fred`, les "principal" `fred@EXAMPLE.COM` et `fred/users.exemple.com@EXAMPLE.COM` peuvent être utilisés pour authentifier le serveur de bases de données.

Si vous utilisez mod_auth_kerb de http://modauthkerb.sf.net et mod_perl sur votre serveur web Apache, vous pouvez utiliser `AuthType KerberosV5SaveCredentials` avec un script mod_perl. Cela fournit un accès sûr aux bases de données, sans demander de mots de passe supplémentaires.

## 19.2.4. Authentification basée sur l'identification

La méthode d'authentification par identification fonctionne en obtenant les noms d'utilisateurs du système d'exploitation, puis en déterminant les noms d'utilisateurs de bases de données autorisés, en utilisant un fichier de correspondance qui liste les paires d'utilisateurs correspondants. Déterminer le nom d'utilisateur du client

est le point critique en matière de sécurité, et il fonctionne différemment selon le type de connexion.

### 19.2.4.1. Authentification par identification sur TCP/IP

Le << protocole d'identification >> est décrit dans la *RFC 1413*. Théoriquement, chaque système d'exploitation de type Unix contient un serveur d'identification qui écoute par défaut le port TCP 113. La fonctionnalité basique d'un serveur d'identification est la réponse aux questions telles que << Quel utilisateur a initié la connexion qui sort de votre port *X* et se connecte à mon port *Y*? >>. PostgreSQL connaissant *X* et *Y* quand une connexion physique est établie, il peut interroger le serveur d'identification de l'hôte du client qui se connecte et peut ainsi théoriquement déterminer quel est l'utilisateur du système d'exploitation pour n'importe quelle connexion.

Le défaut de cette procédure est qu'elle dépend de l'intégrité du client : si la machine client est douteuse ou compromise, un attaquant peut lancer n'importe quel programme sur le port 113 et renvoyer un nom d'utilisateur de son choix. Cette méthode d'authentification n'est par conséquent appropriée que dans le cas de réseaux fermés dans lesquels chaque machine client est soumise à un contrôle strict et dans lesquels les administrateurs du système et des bases de données opèrent en proche collaboration. En d'autres mots, vous devez pouvoir faire confiance à la machine hébergeant le serveur d'identification. Considérez cet avertissement:

> Le protocole d'identification n'a pas vocation à être un protocole d'autorisation ou de contrôle d'accès.
>
> --RFC 1413

Quelques serveurs ident ont une option non standard qui font que le nom de l'utilisateur est renvoyé crypté en utilisant une clé que seul l'administrateur de la machine d'origine connaît. Cette option *ne doit pas* être utilisée lors de l'utilisation du serveur ident avec PostgreSQL car PostgreSQL n'a aucun moyen de décrire la chaîne renvoyée pour déterminer le réel nom de l'utilisateur.

### 19.2.4.2. Authentification par l'identification sur sockets locaux

Sur les systèmes supportant les requêtes `SO_PEERCRED` pour les sockets du domaine Unix (actuellement Linux, FreeBSD, NetBSD, OpenBSD et BSD/OS), l'authentification par identification peut aussi être appliquée aux connexions locales. Dans ce cas, l'utilisation de l'authentification par identification n'ajoute aucun risque lié à la sécurité.

Sur les systèmes sans requêtes `SO_PEERCRED`, l'authentification par identification n'est disponible que pour les connexions TCP/IP. En complément, il est possible de préciser l'adresse localhost 127.0.0.1 et d'établir une connexion à cette adresse. Vous pouvez avoir confiance en cette méthode si vous avez confiance dans le serveur ident local.

### 19.2.4.3. Correspondance d'identité

Lorsque vous utilisez l'authentification basée sur l'identification, après avoir déterminé le nom de l'utilisateur du système d'exploitation qui a initié la connexion, PostgreSQL vérifie si cet utilisateur est autorisé à se connecter par le nom d'utilisateur de base de données qu'il demande. Ceci est contrôlé par l'argument ident map qui suit le mot clé `ident` dans le fichier `pg_hba.conf`. Il existe une correspondance d'identité prédéfinie, `sameuser`, qui permet à n'importe quel utilisateur du système d'exploitation de se connecter en tant qu'utilisateur de base de données du même nom (si ce dernier existe). Les autres correspondances doivent être créées manuellement.

Les correspondances d'identité autres que `sameuser` sont définies dans le fichier de correspondance d'identité, qui est nommé par défaut `pg_ident.conf` et qui est stocké par défaut dans le répertoire data (il est possible de placer le fichier de correspondance ailleurs ; voir le paramètre de configuration ident_file). Ce fichier contient des lignes de la forme suivante :

```
nom-correspondance nomutilisateur-ident base-donnee-utilisateur
```

Les commentaires et les espaces sont gérés de la façon habituelle dans le fichier `pg_hba.conf`. Le *`map-name`* est un nom arbitraire qui sera utilisé pour se référer à cette correspondance dans `pg_hba.conf`. Les deux autres champs spécifient quel utilisateur du système d'exploitation est autorisé à se connecter sous quel nom d'utilisateur de base de données. Le même *`nom-correspondance`* peut être répété pour spécifier plusieurs correspondances d'utilisateurs au sein d'une même table de correspondance. Il n'y a pas de restriction sur le nombre d'utilisateurs de bases de données auxquels un utilisateur de système d'exploitation donné peut correspondre et vice−versa.

Le fichier `pg_ident.conf` est lu au démarrage et quand le processus serveur principal (`postmaster`) reçoit un signal SIGHUP. Si vous éditez le fichier sur un système actif, vous aurez besoin de signaler au `postmaster` (en utilisant `pg_ctl reload` ou `kill -HUP`) qu'il doit relire le fichier.

L'Exemple 19−2 montre un fichier `pg_ident.conf` pouvant être utilisé conjointement avec le fichier `pg_hba.conf` de l'Exemple 19−1. Dans cette configuration d'exemple, n'importe qui connecté sur une machine du réseau 192.168 qui n'a pas de nom utilisateur Unix `bryanh`, `ann`, ou `robert` ne pourrait obtenir d'accès. L'utilisateur Unix `robert` ne serait autorisé à se connecter que lorsqu'il se connecte sous l'utilisateur PostgreSQL `bob` et non `robert` ni n'importe qui d'autre. `ann` ne serait autorisée à se connecter qu'en tant que `ann`. L'utilisateur `bryanh` ne serait autorisé à se connecter qu'en tant que `bryanh` lui−même ou comme `guest1`.

**Exemple 19−2. Un fichier d'exemple `pg_ident.conf`**

```
# CORRESPONDANCE     NOMUTILISATEUR-IDENT    NOMUTILISATEUR-PG

omicron              bryanh                  bryanh
omicron              ann                     ann
# bob a le nom d'utilisateur robert sur ces machines
omicron              robert                  bob
# bryanh peut aussi se connecter en tant que guest1
omicron              bryanh                  guest1
```

## 19.2.5. Authentification PAM

Cette méthode d'authentification fonctionne de façon similaire à `password` à ceci près qu'elle utilise PAM (Pluggable Authentication Modules) comme mécanisme d'authentification. Le nom du service PAM par défaut est `postgresql`. Vous pouvez éventuellement fournir votre nom de service grâce au mot clé `pam` du `pg_hba.conf`. PAM est seulement utilisé pour valider des paires nom utilisateur/mot de passe. Du coup, l'utilisateur doit déjà exister dans la base de données avant que PAM ne puisse être utilisé pour l'authentification. Pour plus d'informations sur PAM, merci de lire la page Linux−PAM et la page PAM Solaris.

# 19.3. Problèmes d'authentification

Les erreurs et problèmes d'authentification se manifestent généralement par des messages d'erreurs tels que ceux qui suivent.

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym", database "testdb"
```

C'est ce que vous risquez le plus d'obtenir si vous parvenez à contacter le serveur mais qu'il refuse de vous parler. Comme le suggère le message, le serveur a refusé la demande de connexion parce qu'il n'a trouvé aucune entrée l'y autorisant dans son fichier de configuration `pg_hba.conf`.

```
FATAL:  Password authentication failed for user "andym"
```

Les messages de ce type indiquent que vous avez contacté le serveur et qu'il veut vous parler mais pas avant que vous n'ayez franchi la méthode d'authentification spécifiée dans le fichier `pg_hba.conf`. Vérifiez le mot de passe que vous avez fourni ou vérifiez votre logiciel d'identification ou votre logiciel Kerberos si les plaintes mentionnent l'un de ces types d'authentification.

```
FATAL:  user "andym" does not exist
```

Le nom d'utilisateur indiqué n'a pas été trouvé.

```
FATAL:  database "testdb" does not exist
```

La base de données à laquelle vous essayez de vous connecter n'existe pas. Notez que si vous ne spécifiez pas un nom de base de données, le nom de la base par défaut est le nom de l'utilisateur de la base de données, ce qui peut être ou pas une bonne chose.

> **Astuce :** Les traces du serveur contiennent plus d'informations sur une erreur d'authentification que ce qui est rapporté au client. Si vous avez des doutes sur les raisons d'un échec, vérifiez les traces.

# Chapitre 20. Localisation

Ce chapitre décrit les fonctionnalités disponibles pour la localisation du point de vue de l'administrateur. PostgreSQL supporte la localisation en utilisant deux approches:

- Utiliser les fonctionnalités de locale du système d'exploitation pour donner un ordonnancement de tri, formatage de chiffre, des messages traduits et et autres aspects spécifiques à la locale.
- Donner un certain nombre de jeu de caractères différents définis dans le serveur PostgreSQL, y compris des jeux de caractères multi−bit, pour permettre de stocker du texte dans toutes sortes de langues, et offrir la traduction de jeu de caractère entre serveur et client.

## 20.1. Support de Locale

Le support de *Locale* fait référence à une application respectant les préférences culturelles en ce qui concerne les alphabets, le tri, le formatage des nombres, etc. PostgreSQL utilise les possibilités du standard ISO C et de la locale POSIX fourni par le système d'exploitation serveur. Pour plus d'information, consultez la documentation de votre serveur.

### 20.1.1. Aperçu

Le support de locale est automatiquement initialisé lorsque un cluster de base de données est créé avec `initdb`. `initdb` va initialiser le cluster avec la valeur de locale de son environnement d'exécution par défaut. Donc, si votre système est déjà paramétré pour utiliser la locale que vous voulez dans votre cluster, vous n'avez rien d'autre à faire. Si vous voulez utiliser une locale différente (ou si vous n'êtes pas sur de la locale qu'utilise votre système), vous pouvez dire à `initdb` exactement quel locale utiliser en spécifiant l'option `--locale`. Par exemple :

```
initdb --locale=sv_SE
```

Cette exemple met la locale à Suédois (`sv`) tel que parlé en Suède (`SE`). D'autres possibilités pourraient être `en_US` (l'anglais Américain) et `fr_CA` (français canadien). Si plus d'un jeu de caractères peuvent être utile pour une locale alors la spécification ressemble à ceci: `cs_CZ.ISO8859-2`. Quels locales sont disponibles sous quels noms dépend de l'éditeur de votre système d'exploitation et de ce qui est installé. (Sur la plupart des systèmes, la commande `locale -a` fournira une liste de locales disponibles.)

De façon occasionnelle, il est utile de mélanger les règles de plusieurs locales, i.e., utiliser les règles de tri anglais mais des messages en espagnol. Pour permettre ceci, des sous−catégories de locales existent qui ne contrôlent qu'un certain aspect des règles de localisation :

| | |
|---|---|
| `LC_COLLATE` | Ordre de tri des chaînes de caractères |
| `LC_CTYPE` | Classification de caractères(Qu'est ce qu'une lettre ? La majuscule équivalente ?) |
| `LC_MESSAGES` | Langage des messages |
| `LC_MONETARY` | Formatage des montants de monnaie |
| `LC_NUMERIC` | Formatage des nombres |
| `LC_TIME` | Formatage des dates et heures |

Les noms des catégories se traduisent en noms d'options `initdb` pour surcharger le choix de locale pour une catégorie donnée. Par exemple, pour mettre la locale en français canadien tout en utilisant les règles américaines pour le formatage de monnaie, utilisez `initdb --locale=fr_CA --lc-monetary=en_US`.

Si vous voulez que le système se comporte comme s'il n'avait pas de support de locale, utilisez les locales spéciales `C` ou `POSIX`.

La nature de ces catégories de locales est que leur valeur doit être fixées pour la durée de vie d'un cluster de base de données. C'est a dire qu'une fois `initdb` lancée, on ne peut plus les changer. `LC_COLLATE` et `LC_CTYPE` sont ces catégories. Ils affectent l'ordre de tri des index, donc ils doivent rester fixes, ou les index sur les colonnes de texte deviendront corrompus. PostgreSQL applique ceci en enregistrant les valeurs de `LC_COLLATE` et `LC_CTYPE` qui sont vus par `initdb`. Le serveur adopte automatiquement ces deux valeurs lorsqu'il est lancé.

Les autres catégories de locale peuvent être changés comme désiré lorsque le serveur est lancé en fixant les variables d'environnement qui ont le même nom que les catégories de locale (voir Section 16.4.8.2 pour plus de détails). Les valeurs par défaut choisies par `initdb` sont en fait seulement écrits dans le fichier de configuration `postgresql.conf` pour servir de valeur par défaut quand le serveur est lancé. Si vous effacez ces déclarations de `postgresql.conf`, alors le serveur héritera des paramètres de l'environnement d'exécution.

Notez que le comportement de locale du serveur est déterminé par les variables d'environnement vues par le serveur, pas l'environnement d'un client quelconque. Alors, faites attention de configurer les bons paramètres de locale avant de lancer le serveur. Une conséquence de ceci est que, si le client et le serveur ont des locales différents, les messages apparaîtront dans des langues différents suivant d'où ils viennent.

> **Note :** Lorsqu'on parle d'hériter la locale de l'environnement d'exécution, ceci veut dire la chose suivante sur la plupart des systèmes d'exploitation: Pour une catégorie de locale donnée, disons l'ordonnancement, les variables d'environnement suivantes sont consultées dans cet ordre jusqu'à ce qu'on en trouve une de fixée: `LC_ALL`, `LC_COLLATE` (la variable correspondante à la catégorie respective), `LANG`. Si aucune de ces variables n'est fixée, alors on utilise la locale par défaut de `C`.
>
> Certaines bibliothèques de localisation regardent aussi la variable d'environnement `LANGUAGE` qui surcharge tout autre paramètre pour le besoin de fixer la langue des messages. Si vous doutez, lisez la documentation de votre système d'exploitation, en particulier la documentation à propos de gettext, pour plus d'information.

Pour permettre la traduction des messages dans la langue préférée de l'utilisateur, NLS doit être activé pendant la compilation. Ce choix est indépendant d'autre support de locale.

## 20.1.2. Comportement

Le support de locale influence les fonctionnalités suivantes :

- Ordre de tri dans les requêtes utilisant `ORDER BY`
- La possibilité d'utiliser des index avec les clauses `LIKE`
- La famille de fonctions `to_char`

L'inconvénient d'utiliser le support de locale autres que C ou POSIX dans PostgreSQL est son impact sur les performances. Il ralentit la gestion des caractères et empêche l'utilisation des index ordinaires par LIKE. Pour cette raison, utilisez les locales seulement si vous en avez besoin.

### 20.1.3. Problèmes

Si le support de locale ne fonctionne pas malgré l'explication ci−dessus, vérifiez que le support de locale de votre système d'exploitation est correctement configuré. Pour vérifier quelles locales sont installées sur votre système, vous pouvez utiliser la commande `locale -a` si votre système d'exploitation le fournit.

Vérifiez que PostgreSQL utilise vraiment la locale que vous supposez. Les paramètres LC_COLLATE et LC_CTYPE sont déterminés lors de initdb et ne peuvent pas être modifiés sans répéter initdb. D'autres paramètres de locale, y compris LC_MESSAGES et LC_MONETARY sont déterminés initialement par l'environnement dans lequel le serveur est lancé mais peut être modifié en cours. Vous pouvez vérifier les paramétrages de la locale active en utilisant la commande SHOW.

Le répertoire src/test/locale dans la distribution du source contient une série de tests pour le support de locale dans PostgreSQL.

Les applications clientes qui gèrent les erreurs côté serveur en analysant le texte du message d'erreur auront, bien sur, des problèmes lorsque les messages sont dans une langue différente. Les auteurs de telles applications sont invités à utiliser le schéma de code d'erreur à la place.

Maintenir des catalogues de traductions de messages requiert les efforts permanents de beaucoup de volontaires qui souhaitent voir PostgreSQL bien interpréter leur langue préférée. Si des messages dans votre langue ne sont pas disponibles ou pas complètement traduits, votre aide serait très appréciée. Si vous voulez aider, consultez le Chapitre 44 ou écrivez à la liste de diffusion des développeurs.

## 20.2. Support des jeux de caractères

Le support des jeux de caractères dans PostgreSQL vous permet de rentrer du texte dans plusieurs jeux de caractères, dont des jeux de caractères à un octet tels que la série ISO 8859 et des jeux de caractères multi−octets tel que EUC (Extended Unix Code), Unicode et le code interne de Mule. Tous les jeux de caractères peuvent être utilisés de façon transparente au travers du serveur. (Si vous utilisez des fonctions d'extension d'autres sources, cela dépend de la qualité du code.) Le jeu de caractères par défaut est sélectionné pendant l'initialisation de votre cluster de base de données PostgreSQL avec initdb. Le choix peut être outrepassé lorsqu'on crée une base de données en utilisant createdb ou en utilisant la commande SQL CREATE DATABASE. Vous pouvez donc avoir de multiples bases chacune avec un jeux de caractères différents.

### 20.2.1. Jeux de caractères supportés

Le Tableau 20−1 montre les jeux de caractères disponibles à l'utilisation sur le serveur.

**Tableau 20−1. Jeux de Caractères Serveur**

| Nom | Description |
|---|---|

| | |
|---|---|
| SQL_ASCII | ASCII |
| EUC_JP | Japonais EUC |
| EUC_CN | Chinois EUC |
| EUC_KR | Coréen EUC |
| JOHAB | Coréen EUC (base Hangle) |
| EUC_TW | Taiwanais EUC |
| UNICODE | Unicode (UTF−8) |
| MULE_INTERNAL | Code interne de Mule |
| LATIN1 | ISO 8859−1/ECMA 94 (Alphabet latin no.1) |
| LATIN2 | ISO 8859−2/ECMA 94 (Alphabet latin no.2) |
| LATIN3 | ISO 8859−3/ECMA 94 (Alphabet latin no.3) |
| LATIN4 | ISO 8859−4/ECMA 94 (Alphabet latin no.4) |
| LATIN5 | ISO 8859−9/ECMA 128 (Alphabet latin no.5) |
| LATIN6 | ISO 8859−10/ECMA 144 (Alphabet latin no.6) |
| LATIN7 | ISO 8859−13 (Alphabet latin no.7) |
| LATIN8 | ISO 8859−14 (Alphabet latin no.8) |
| LATIN9 | ISO 8859−15 (Alphabet latin no.9) |
| LATIN10 | ISO 8859−16/ASRO SR 14111 (Alphabet latin no.10) |
| ISO_8859_5 | ISO 8859−5/ECMA 113 (Latin/Cyrillique) |
| ISO_8859_6 | ISO 8859−6/ECMA 114 (Latin/Arabe) |
| ISO_8859_7 | ISO 8859−7/ECMA 118 (Latin/Grec) |
| ISO_8859_8 | ISO 8859−8/ECMA 121 (Latin/Hébreu) |
| KOI8 | KOI8−R(U) |
| ALT | Windows CP866 |
| WIN874 | Windows CP874 (Thai) |
| WIN1250 | Windows CP1250 |
| WIN | Windows CP1251 |
| WIN1256 | Windows CP1256 (Arabe) |
| TCVN | TCVN−5712/Windows CP1258 (Vietnamien) |

**Important :** Avant PostgreSQL 7.2, LATIN5 voulait dire par erreur ISO 8859−5. A partir de 7.2, LATIN5 voulait dire ISO 8859−9. Si vous avez une base LATIN5 créée sur une version 7.1 ou antérieure et vous voulez migrer vers 7.2 ou une version plus récente, vous devriez faire attention à ce changement.

Tout les APIs ne supportent pas les jeux de caractères de la liste. Par exemple, le pilot JDBC de PostgreSQL ne supporte pas MULE_INTERNAL, LATIN6, LATIN8 et LATIN10.

## 20.2.2. Choisir le Jeu de Caractères

initdb définit le jeu de caractères par défaut pour un cluster PostgreSQL. Par exemple,

```
initdb -E EUC_JP
```

paramètre le jeu de caractères (codage) à EUC_JP (Extended Unix Code for Japanese). Vous pouvez utiliser l'option --encoding au lieu de -E si vous préférez saisir les noms d'options longs. Si aucune option -E ou --encoding n'est donnée, SQL_ASCII est utilisé.

Vous pouvez créer une base de données avec un jeu de caractère différent:

```
createdb -E EUC_KR korean
```

Ceci va créer une base de données appelée korean qui utilisera le jeu de caractère EUC_KR. Un autre moyen de réaliser ceci est d'utiliser cette commande SQL :

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR';
```

L'encodage pour une base de données est conservé dans le catalogue système pg_database. Vous pouvez voir ceci en utilisant l'option -l ou la commande \l de psql.

```
$ psql -l
            List of databases
   Database    |  Owner  |   Encoding
---------------+---------+---------------
 euc_cn        | t-ishii | EUC_CN
 euc_jp        | t-ishii | EUC_JP
 euc_kr        | t-ishii | EUC_KR
 euc_tw        | t-ishii | EUC_TW
 mule_internal | t-ishii | MULE_INTERNAL
 regression    | t-ishii | SQL_ASCII
 template1     | t-ishii | EUC_JP
 test          | t-ishii | EUC_JP
 unicode       | t-ishii | UNICODE
(9 rows)
```

> **Important :** Bien que vous pouvez spécifier tout codage que vous souhaitez pour une base de données, il est déconseillé de choisir un codage qui n'est pas attendu par la locale sélectionnée. Les paramètres LC_COLLATE et LC_CTYPE impliquent un codage particulier, et les opérations dépendantes de la locale (comme le tri) pourraient mal interprétées les données qui sont dans un codage incompatible.
>
> Comme ces paramètres de locale sont gelés par initdb, la flexibilité apparente pour utiliser différents codages dans les différentes bases de données d'un groupe est plus théorique que réel. Il est probable que ces mécanismes seront revus dans les prochaines versions de PostgreSQL.
>
> Une façon d'utiliser les codages multiples en toute sûreté est de configurer la locale à C ou POSIX lors d'initdb, désactivant toute connaissance réelle de la locale.

## 20.2.3. Conversion de Jeu de Caractères Automatique entre Serveur et Client

PostgreSQL supporte les conversions automatiques de jeu de caractères entre client et serveur pour certains jeux de caractères. Les informations de conversion sont conservés dans le catalogue système pg_conversion. Vous pouvez créer une nouvelle conversion en utilisant la commande SQL CREATE CONVERSION. PostgreSQL est livré avec certaines conversions prédéfinies. Ils sont listés dans Tableau 20–2.

**Tableau 20–2. Conversion de Jeux de Caractères Client/Serveur**

| Jeu de Caractères Serveur | Jeux de Caractères Clients Disponibles |
|---|---|
| SQL_ASCII | SQL_ASCII, UNICODE, MULE_INTERNAL |
| EUC_JP | EUC_JP, SJIS, UNICODE, MULE_INTERNAL |
| EUC_CN | EUC_CN, UNICODE, MULE_INTERNAL |
| EUC_KR | EUC_KR, UNICODE, MULE_INTERNAL |
| JOHAB | JOHAB, UNICODE |
| EUC_TW | EUC_TW, BIG5, UNICODE, MULE_INTERNAL |
| LATIN1 | LATIN1, UNICODE MULE_INTERNAL |
| LATIN2 | LATIN2, WIN1250, UNICODE, MULE_INTERNAL |
| LATIN3 | LATIN3, UNICODE, MULE_INTERNAL |
| LATIN4 | LATIN4, UNICODE, MULE_INTERNAL |
| LATIN5 | LATIN5, UNICODE |
| LATIN6 | LATIN6, UNICODE, MULE_INTERNAL |
| LATIN7 | LATIN7, UNICODE, MULE_INTERNAL |
| LATIN8 | LATIN8, UNICODE, MULE_INTERNAL |
| LATIN9 | LATIN9, UNICODE, MULE_INTERNAL |
| LATIN10 | LATIN10, UNICODE, MULE_INTERNAL |
| ISO_8859_5 | ISO_8859_5, UNICODE, MULE_INTERNAL, WIN, ALT, KOI8 |
| ISO_8859_6 | ISO_8859_6, UNICODE |
| ISO_8859_7 | ISO_8859_7, UNICODE |
| ISO_8859_8 | ISO_8859_8, UNICODE |
| UNICODE | EUC_JP, SJIS, EUC_KR, UHC, JOHAB, EUC_CN, GBK, EUC_TW, BIG5, LATIN1 to LATIN10, ISO_8859_5, ISO_8859_6, ISO_8859_7, ISO_8859_8, WIN, ALT, KOI8, WIN1256, TCVN, WIN874, GB18030, WIN1250 |
| MULE_INTERNAL | EUC_JP, SJIS, EUC_KR, EUC_CN, EUC_TW, BIG5, LATIN1 to LATIN5, WIN, ALT, WIN1250, BIG5, ISO_8859_5, KOI8 |
| KOI8 | ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL |
| ALT | ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL |
| WIN874 | WIN874, UNICODE |
| WIN1250 | LATIN2, WIN1250, UNICODE, MULE_INTERNAL |
| WIN | ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL |
| WIN1256 | WIN1256, UNICODE |
| TCVN | TCVN, UNICODE |

Pour activer la conversion automatique de jeux de caractères, vous devez dire à PostgreSQL quel jeu de caractères (encodage) vous voulez utiliser côté client. Il y'a plusieurs façons de faire cela:

- En utilisant la commande `\encoding` dans psql. `\encoding` vous permet de changer de jeu de caractères client à la volée. Par exemple, pour changer l'encodage en `SJIS`, tapez:

  ```
  \encoding SJIS
  ```
- En utilisant les fonctions libpq. `\encoding` appelle en fait `PQsetClientEncoding()` pour

faire son travail.

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

ou *conn* est une connexion au serveur, et *encoding* est l'encodage que vous voulez utiliser. Si la fonction fixe l'encodage, elle renvoie 0, sinon −1. L'encodage actuel pour cette connexion peut être déterminé en utilisant:

```
int PQclientEncoding(const PGconn *conn);
```

Notez que ceci renvoie l'ID d'encodage, pas une chaîne symbolique tel que EUC_JP. Pour convertir un ID d'encodage en NOM, vous pouvez utiliser:

```
char *pg_encoding_to_char(int encoding_id);
```

- En utilisant SET client_encoding TO. Fixer l'encodage client peut être fait avec cette commande SQL:

  ```
  SET CLIENT_ENCODING TO 'valeur';
  ```

  Vous pouvez aussi utiliser la syntaxe SQL plus standard SET NAMES pour ceci:

  ```
  SET NAMES 'valeur';
  ```

  Pour demander l'actuel encodage client:

  ```
  SHOW client_encoding;
  ```

  Pour revenir à l'encodage par défaut:

  ```
  RESET client_encoding;
  ```
- En utilisant PGCLIENTENCODING. Si la variable d'environnement PGCLIENTENCODING est définie dans l'environnement client, cet encodage client est automatiquement sélectionné lorsqu'une connexion au serveur est établie. (Ceci peut être surchargé avec n'importe quelle autre des méthodes ci−dessus.)
- En utilisant la variable de configuration client_encoding. Si la variable client_encoding est définie, cet encodage client est automatiquement sélectionné lorsqu'une connexion au serveur est établie. (Ceci peut être surchargé avec n'importe quelle autre des méthodes ci−dessus.)

Si la conversion d'un caractère particulier n'est pas possible — supposons que vous avez choisi EUC_JP pour le serveur et LATIN1 pour le client, alors certains caractères japonais ne pourront pas être convertis en LATIN1 — ils seront transformés en leur valeur hexadécimale entre parenthèses, i.e., (826C).

## 20.2.4. Plus de Lecture

Voici de bonnes sources pour commencer à maîtriser les différents jeux de caractères.

ftp://ftp.ora.com/pub/examples/nutshell/ujip/doc/cjk.inf
: Des explications détaillées de EUC_JP, EUC_CN, EUC_KR, EUC_TW apparaissent dans la section 3.2.

http://www.unicode.org/
: Le site web du Unicode Consortium

RFC 2044
: UTF−8 est défini ici.

# Chapitre 21. Planifier les tâches de maintenance

Pour bien fonctionner, un serveur PostgreSQL nécessite quelques opérations de maintenance régulières, décrites ci−après. Ces tâches sont par nature répétitives et peuvent facilement s'automatiser grâce aux outils standards d'UNIX, comme les scripts cron. La responsabilité de la mise en place de ces scripts et du contrôle de leur bon fonctionnement relève de l'administrateur de la base.

Une opération de maintenance évidente est la sauvegarde régulière des données. Sans une sauvegarde récente il est impossible de restaurer après un dommage grave (perte d'un disque, incendie, table supprimée par erreur, etc.). Les mécanismes de sauvegarde et restauration disponibles dans PostgreSQL sont détaillés dans le Chapitre 22.

L'autre tâche primordiale est de réaliser périodiquement un << vacuum >>, c'est à dire << faire le vide >> dans la base de données. Cette opération est détaillée dans la Section 21.1.

La gestion du fichier de traces mérite aussi une attention régulière. Cela est détaillé dans la Section 21.3.

PostgreSQL demande peu de maintenance par rapport à d'autres SGBD. Néanmoins, un suivi vigilant de ces tâches participera beaucoup à rendre le système productif et agréable à utiliser.

## 21.1. Nettoyages réguliers

La commande `VACUUM` de PostgreSQL doit être exécutée régulièrement pour plusieurs raisons :

1. pour récupérer l'espace disque occupé par les lignes supprimées ou mises à jour ;
2. pour mettre à jour les statistiques utilisées par l'optimiseur de PostgreSQL ;
3. pour prévenir la perte des données les plus anciennes à cause d'un *cycle de l'identifiant de transaction (XID)*.

La fréquence et le périmètre des exécutions de `VACUUM` variera pour chacune des raisons ci−dessus selon les besoins des sites. De plus, les administrateurs doivent appréhender chaque cas et développer une stratégie de maintenance appropriée. L'objectif de cette section est de décrire globalement les problèmes à résoudre ; pour la syntaxe et les autres détails, voir la page de référence de la commande VACUUM.

À partir de PostgreSQL 7.2, la forme standard de `VACUUM` peut être exécutée en parallèle des opérations classiques de manipulation des données (select, insert, update, delete, mais pas les modifications de définition). Les opérations de nettoyage par `VACUUM` sont largement moins pénalisantes qu'elles n'ont pu l'être par le passé et il n'est plus aussi impératif de les planifier pendant les plages d'utilisation peu intensives.

À partir de la version 8.0 de PostgreSQL, certains paramètres de configuration peuvent être ajustés pour réduire l'impact du vacuum en tâche de fond sur les performances. Voir Section 16.4.3.4.

### 21.1.1. Récupérer l'espace disque

Dans son fonctionnement normal, PostgreSQL ne supprime pas immédiatement les versions périmées des lignes après un `UPDATE` ou un `DELETE`. Cette approche est nécessaire pour la consistance des accès

concurrents (voir le Chapitre 12) : la version de la ligne ne doit pas être supprimée tant qu'elle est susceptible d'être lue par une autre transaction. Mais finalement, une ligne qui est plus vieille que toutes les transactions en cours n'est plus utile du tout. La place qu'elle utilise doit être rendu pour être réutilisée par d'autres lignes afin d'éviter un accroissement constant du volume occupé sur le disque. Cela est réalisé en exécutant `VACUUM`.

Évidemment, une table qui subit beaucoup de mises à jour et suppressions nécessitera des nettoyages plus fréquents que les tables rarement modifiées. Il peut être pertinent de programmer périodiquement par cron des tâches spécifiques qui nettoient uniquement les tables concernées (avec `VACUUM`) et ignorent les tables que l'on sait peu modifiées. Ceci ne sera vraiment utile que s'il y a à la fois des tables volumineuses intensément modifiées et des tables volumineuses peu modifiées. En effet, le coût supplémentaire lié au nettoyage d'une petite table ne mérite pas que l'on s'en préoccupe.

Il existe deux variantes de la commande `VACUUM`. La première forme, connu en tant que << vacuum fainéant >> ou plus simplement `VACUUM`, marque les données expirées dans les tables et les index pour une utilisation future ; il ne tente *pas* de récupérer immédiatement l'espace utilisée par cette donnée expirée. Du coup, le fichier de la table n'est pas plus petit et tout l'espace inutilisé dans le fichier n'est pas redonné au système d'exploitation. Cette variante de `VACUUM` peut être lancé en concurrence avec les autres opérations normales de la base de données.

La seconde forme est la commande `VACUUM FULL`. Elle utilise un algorithme plus agressif pour récupérer l'espace consommé par les versions expirées des lignes. Tout espace qui est libéré par `VACUUM FULL` est immédiatement rendu au système d'exploitation. Malheureusement, cette variante de la commande `VACUUM` acquiert un verrou exclusif sur chaque table avant que `VACUUM FULL` ne la traite. Du coup, utiliser fréquemment `VACUUM FULL` peut avoir un effet extrêmement négatif sur les performances des requêtes concurrentes sur la base de données.

La forme standard de `VACUUM` est mieux utilisé dans le but de maintenir une utilisation simple de l'espace disque. Donc, vous avez besoin de redonner de l'espace disque au système d'exploitation, vous pouvez utiliser `VACUUM FULL` — mais quel est l'intérêt de redonner de l'espace disque qui devra ensuite être de nouveau alloué ? Des `VACUUM` standard et d'une fréquence modérée sont une meilleure approche que des `VACUUM FULL`, même non fréquents, pour maintenir des tables mises à jour fréquemment.

La meilleure stratégie pour la plupart des sites est de planifier un `VACUUM` général sur toute la base une fois par jour, en dehors des horaires normaux de production, accompagné si nécessaire de nettoyages plus fréquents pour les tables subissant d'intenses modifications. (Quelques installations avec un taux extrêmement important de modifications de données lancent un `VACUUM` sur les tables très occupées jusqu'à une fois toutes les quelques minutes.) S'il y a plusieurs bases de données dans un cluster (groupe de bases de données), ne pas oublier de nettoyer chacune d'entre elles ; l'exécutable `vacuumdb` peut s'avérer utile.

> **Astuce :** Le programme `contrib/pg_autovacuum` peut être utile pour automatiser des opérations de `VACUUM` fréquentes.

`VACUUM FULL` est recommandé dans les cas où vous savez que vous avez supprimé la majorité des lignes dans une table, de façon à ce que la taille de la table soit réduit de façon conséquente avec l'approche plus plus agressive de `VACUUM FULL`. Utilisez le `VACUUM` standard, et non pas `VACUUM FULL`, pour les nettoyages standards.

Si vous avez une table dont le contenu est supprimé sur une base périodique, considérez de le faire avec `TRUNCATE` plutôt qu'avec `DELETE` suivi par un `VACUUM`. `TRUNCATE` supprime le contenu entier de la table immédiatement sans nécessiter un `VACUUM` ou `VACUUM FULL` pour réclamer l'espace disque maintenant

inutilisé.

## 21.1.2. Maintenir les statistiques du planificateur

L'optimiseur de requêtes de PostgreSQL s'appuie sur des informations statistiques sur le contenu des tables dans l'optique de générer des plans d'exécutions efficaces pour les requêtes. Ces statistiques sont collectées par la commande `ANALYZE`, qui peut être invoquée seule ou comme une option de `VACUUM`. Il est important d'avoir des statistiques relativement à jour sans quoi des mauvais choix dans les plans d'exécution pourraient pénaliser la performance de la base.

À l'instar du nettoyage pour récupérer l'espace, les statistiques doivent être plus souvent collectées pour les tables intensément modifiées que pour celles qui le sont moins. Mais même si la table est très modifiée, il se peut que ces collectes soient inutiles si la distribution probabiliste des données évolue peu. Une règle simple pour décider est de voir comment évoluent les valeurs minimum et maximum des données. Par exemple, une colonne de type `timestamp` qui contient la date de mise à jour de la ligne aura une valeur maximum en continuelle croissance au fur et à mesure des modifications ; une telle colonne nécessitera plus de collectes statistiques qu'une colonne qui contient par exemple les URL des pages accédées sur un site web. La colonne qui contient les URL peut très bien être aussi souvent modifiée mais la distribution probabiliste des données changera certainement moins rapidement.

Il est possible d'exécuter `ANALYZE` sur des tables spécifiques, voire des colonnes spécifiques ; il a donc toute flexibilité pour mettre à jour certaines statistiques plus souvent que les autres en fonction des besoins de l'application. Quoi qu'il en soit, dans la pratique, l'utilité de cette fonctionnalité est douteuse. En effet, depuis PostgreSQL 7.2, `ANALYZE` est une opération plutôt rapide, même pour les grosses tables, parce que la collecte se base sur un échantillon aléatoire de la table et non sur toutes les données. Il est donc probablement plus simple de l'utiliser systématiquement sur toute la base.

> **Astuce :** Même si il n'est pas très productif de règler précisément la fréquence de `ANALYZE` pour chaque colonne, il peut être intéressant d'ajuster le niveau de détail des statistiques collectées pour chaque colonne. Les colonnes très utilisées dans les clauses `WHERE` et dont la distribution n'est pas uniforme requièrent des histogrammes plus précis que les autres colonnes. Voir `ALTER TABLE SET STATISTICS`.

Pour la plupart des site, la meilleure stratégie est de programmer une collecte générale des statistiques sur toute la base, une fois par jour. Ceci peut être profitablement couplé avec un `VACUUM` (la nuit par exemple). Quoi qu'il en soit, les administrateurs des bases dont les statistiques changent peu pourront juger que cela est exagéré et que des exécutions moins fréquentes de `ANALYZE` sont bien suffisantes.

## 21.1.3. Éviter les cycles des identifiants de transactions

Le mécanisme de contrôle de concurrence multiversion (MVCC) de PostgreSQL s'appuie sur la possibilité de comparer des identifiants de transactions (XID) ; c'est un nombre croissant : la version d'une ligne dont le XID d'insertion est supérieur au XID de la transaction en cours est << dans le futur >> et ne doit pas être visible de la transaction courante. Comme les identifiants ont une taille limitée (32 bits à ce jour), un groupe qui est en activité depuis longtemps (plus de 4 milliards de transactions) connaîtra un cycle des identifiants de transaction : le XID reviendra à 0 et soudainement les transactions du passé sembleront appartenir au futur – ce qui signifie qu'elles deviennent invisibles. En bref, perte de données totale. (En réalité, les données sont toujours là mais c'est un piètre réconfort puisqu'elles resteront inaccessibles.)

Avant PostgreSQL 7.2, la seule parade contre ces cycles de XID était de ré−exécuter `initdb` au minimum tous les 4 milliards de transaction. Bien sûr, cela n'était pas satisfaisant pour les sites connaissant un trafic important, donc une nouvelle solution a été élaborée. La nouvelle approche permet à un cluster de fonctionner indéfiniment, sans `initdb` ni aucune sorte de réinitialisation. Le prix en est le suivant : *toute table dans la base doit être nettoyée au moins une fois tous les milliards de transactions.*

Dans la pratique, cette exigence n'est pas onéreuse mais comme son manquement aurait pour conséquence une perte totale des données (pas seulement de l'espace disque perdu ou des performances moindres), des dispositions ont été prises pour aider les administrateurs à surveiller le temps écoulé depuis le dernier `VACUUM`. La suite de cette section en explique les détails.

La nouvelle approche pour la comparaison des XID distingue deux XID spéciaux, numéros 1 et 2 (`BootstrapXID` et `FrozenXID`). Ces deux XID sont toujours considérés comme plus vieux que n'importe quel autre. Les XID normaux (ceux qui sont supérieurs à deux) sont comparés sur une base modulo−$2^{31}$. Cela signifie que pour chaque XID normal, il y en a deux milliards qui sont plus vieux et deux milliards qui sont plus récents. Une autre manière de le dire est que l'ensemble de définition des XID est circulaire et sans limite. De plus, une ligne créée avec un XID normal donné, la version de la ligne apparaîtra comme appartenant au passé pour les deux milliards de transactions qui suivront quelque soit le XID. Si la ligne existe encore après deux milliards de transactions, elle apparaîtra soudainement comme appartenant au futur. Pour éviter la disparition des données, les versions trop anciennes doivent se voir affecter le XID `FrozenXID` avant d'atteindre le seuil fatidique des deux milliards de transactions. Une fois qu'elles ont ce XID spécifique, elles appartiendront au passé pour toutes les transactions même en cas de cycle. Cette affectation est réalisée par `VACUUM`.

La politique normale de `VACUUM` est d'affecter `FrozenXID` à toute les lignes dont le XID se situe à plus de un milliard de transactions dans le passé. Elle préserve le XID original tant qu'il est utile. (En réalité, la plupart des lignes existeront et disparaîtront avant d'être << gelée >>. Avec cette méthode, l'intervalle de sécurité maximum entre les exécutions de `VACUUM` pour une table est d'exactement un milliard de transactions : en attendant plus longtemps, on s'expose à conserver des versions qui n'étaient pas assez vielles pour se voir affecter `FrozenXID` lors de la précédente exécution et qui apparaissent maintenant dans le futur du fait d'un cycle − c'est−à−dire que les données semblent perdues. (Bien sûr, elles réapparaîtront après deux nouveaux milliards de transactions mais cela n'a pas d'intérêt).

Puisque des exécutions périodiques de `VACUUM` sont nécessaires de toutes manières, pour les raisons évoquées ci−dessus, il est très peu probable qu'une table ne soit pas nettoyée du tout durant un milliard de transactions. Pour aider les administrateurs à assurer que cette exigence est remplie, `VACUUM` conserve des statistiques sur les XID dans la table système `pg_database`. Notamment, la table `pg_database` contient, pour chaque base, une colonne `datfrozenxid` qui est mise à jour après les `VACUUM` de la base (c'est−à−dire `VACUUM` qui ne spécifie aucune table particulière). La valeur qui est stockée est la limite en deçà de laquelle cette exécution de `VACUUM` a marqué la ligne comme << gelée >>. Tous les XID plus vieux que ce XID limite ont reçu le XID `FrozenXID` pour cette base. Pour obtenir cette information, il suffit d'exécuter la requête :

```
SELECT datname, age(datfrozenxid) FROM pg_database;
```

La colonne `age` calcule le nombre de transactions effectuées entre le XID limite et le XID courant.

Avec la méthode standard de gel du XID, La colonne `age` démarre à un milliard pour une base fraîchement nettoyée. Si l'`age` approche des deux milliards, la base doit de nouveau être nettoyée pour éviter les erreurs liées au cycle du XID. Il est recommandé d'exécuter un `VACUUM` une fois tous les demi milliard de transactions pour garder une marge de sécurité maximale. Pour aider à remplir cette exigence, chaque

VACUUM émet un message si n'importe lequel des enregistrements de pg_database indique un age de plus de 1,5 milliard de transactions, par exemple :

```
play=# VACUUM;
WARNING:  some databases have not been vacuumed in 1613770184 transactions
HINT:  Better vacuum them within 533713463 transactions, or you may have a wraparound failure.
VACUUM
```

Avec l'option FREEZE, la commande VACUUM a un comportement plus poussé : les versions des lignes sont gelées si elles sont suffisamment vieilles pour être visibles de toutes les transactions en cours. En particulier, sur une base en lecture seulement, VACUUM FREEZE aura pour résultat de geler toutes les lignes de la base. Donc, tant que la base n'est pas modifiée, aucun nettoyage supplémentaire n'est nécessaire pour éviter les problèmes de cycle du XID. Cette technique est notamment utilisée par initdb pour préparer la base template0. Cela pourrait également être utilisé pour préparer n'importe quelle base créée par l'administrateur avec datallowconn = false dans pg_database, puisqu'il n'y a pas moyen d'exécuter VACUUM sur une base à laquelle on ne peut pas se connecter. On notera que VACUUM n'émet aucun message d'avertissement pour les enregistrements de pg_database où datallowconn = false afin de ne pas induire d'erreur ; c'est donc à l'administrateur de s'assurer que ces bases sont correctement gelées.

| **Avertissement** |
| --- |
| Pour s'assurer contre les cycles de transaction (wraparound), il est nécessaire de lancer un VACUMM sur *chaque* table, ceci incluant les catalogues système pour chaque base de données au moins une fois tous les milliards de transactions. Nous avons vu des situations de pertes de données causées par des personnes décidant qu'ils avaient seulement besoin de lancer un VACUUM sur leurs tables actives plutôt que de lancer des commandes VACUUM sur toute la base de données. Il semblera que cela fonctionne bien... pendant un certain temps. |

## 21.2. Ré–indexation régulière

Dans certains cas, cela vaut la peine de reconstruire périodiquement les index par la commande REINDEX. (Il y a aussi contrib/reindexdb qui peut ré–indexer une base entière). Quoiqu'il en soit, PostgreSQL 7.4 a nettement réduit le besoin de cette maintenance en comparaison des versions précédentes.

## 21.3. Maintenance du fichier de traces

Sauvegarder les journaux de trace du serveur de bases de données dans un fichier plutôt que dans /dev/NULL est une bonne idée. Les journaux sont d'une utilité incomparable lorsqu'arrive le moment où des problèmes surviennent. Néanmoins, les journaux ont tendance à être volumineux (tout spécialement à des niveaux de débogage importants) et vous ne voulez pas les sauvegarder indéfiniment. Vous avez besoin de faire une << rotation >> des journaux pour que les nouveaux journaux sont commencés et que les anciens soient supprimés après une période de temps raisonnable.

Si vous redirigez simplement stderr du postmaster dans un fichier, vous aurez un journal des traces mais la seule façon de le tronquer sera d'arrêter et de relancer postmaster. Ceci peut convenir si vous utilisez PostgreSQL dans un environnement de développement mais peu de serveurs de production trouveraient ce comportement acceptable.

Une meilleure approche est d'envoyer la sortie stderr de `postmaster` dans un programme de rotation de journaux. Il existe un programme interne de rotation que vous pouvez utiliser en configurant le paramètre `redirect_stderr` à `true` dans `postgresql.conf`. Les paramètres de contrôle pour ce programme sont décrits dans Section 16.4.6.1.

Sinon, vous pourriez préférer utiliser un programme externe de rotation de journaux si vous en utilisez déjà un avec d'autres serveurs. Par exemple, l'outil rotatelogs inclus dans la distribution Apache peut être utilisé avec PostgreSQL. Pour cela, envoyez via un tube la sortie stderr de `postmaster` dans le programme désiré. Si vous lancez le serveur avec `pg_ctl`, alors stderr est déjà directement renvoyé dans stdout, donc vous avez juste besoin d'ajouter la commande via un tube, par exemple :

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

Une autre approche de production pour la gestion des journaux de trace est de les envoyer à syslog et de laisser syslog gérer la rotation des fichiers. Pour cela, initialisez le paramètre de configuration `log_destination` à `syslog` (pour tracer uniquement via syslog) dans `postgresql.conf`. Ensuite, vous pouvez envoyer un signal `SIGHUP` au démon syslog quand vous voulez le forcer à écrire dans un nouveau fichier. Si vous voulez automatiser la rotation des journaux, le programme logrotate peut être configuré pour fonctionner avec les journaux de traces provenant de syslog.

Néanmoins, sur beaucoup de systèmes, syslog n'est pas très fiable, particulièrement avec les messages très gros ; il pourrait tronquer ou supprimer des messages au moment où vous en aurez le plus besoin. De plus, sur Linux, syslog synchronisera tout message sur disque, amenant des performances assez pauvres. (Vous pouvez utiliser un − au début du nom de fichier dans le fichier de configuration syslog pour désactiver ce comportement.)

Notez que toutes les solutions décrites ci−dessus font attention à lancer de nouveaux journaux de traces à des intervalles configurables mais ils ne gèrent pas la suppression des vieux fichiers de traces, qui ne sont probablement plus très intéressants. Vous voudrez probablement configurer un script pour supprimer périodiquement les anciens journaux. Une autre possibilité est de configurer le programme de rotation pour que les anciens journaux de traces soient écrasés de façon cyclique.

# Chapitre 22. Sauvegardes et restaurations

Comme avec tout ce qui contient des données importantes, les bases de données PostgreSQL doivent être sauvegardées régulièrement. Bien que la procédure soit plutôt simple, il est important de comprendre les techniques sous−jacentes ainsi que les hypothèses prises.

Il y a trois approches fondamentalement différentes pour sauvegarder les données de PostgreSQL :

- La sauvegarde SQL ;
- La sauvegarde de niveau système de fichiers ;
- La sauvegarde à chaud (ou en ligne).

Chacune a ses avantages et inconvénients.

## 22.1. Sauvegarde SQL

Le principe est de générer un fichier texte de commandes SQL (appelé << fichier dump >>), qui, si on le renvoie au serveur, recrée une base de données identique à celle sauvegardée. PostgreSQL propose pour cela le programme utilitaire pg_dump. L'usage basique est :

```
pg_dump base_de_donnees > fichier_de_sortie
```

Comme vous le voyez, pg_dump écrit son résultat sur la sortie standard. Nous verrons plus loin que cela peut être pratique.

pg_dump est un programme client PostgreSQL classique (mais plutôt intelligent). Ceci veut dire que vous pouvez faire une sauvegarde depuis n'importe quel ordinateur ayant accès à la base. Mais rappelez−vous que pg_dump n'a pas de droits spéciaux. En particulier, il doit avoir accès en lecture à toutes les tables que vous voulez sauvegarder, si bien qu'il doit être lancé pratiquement toujours en tant que super−utilisateur de la base.

Pour préciser quel serveur de bases de données pg_dump doit contacter, utilisez les options de ligne de commande `-h serveur` et `-p port`. Le serveur par défaut est le serveur local, ou bien celui spécifié par la variable d'environnement `PGHOST`. De la même façon, le port par défaut est indiqué par la variable d'environnement `PGPORT` ou, en son absence, par la valeur par défaut précisée à la compilation. Heureusement, la serveur a normalement la même valeur par défaut à la compilation.

Comme tout programme client PostgreSQL, pg_dump se connecte par défaut avec l'utilisateur de base de données de même nom que l'utilisateur système courant. Pour passer outre, précisez l'option `-U` ou donnez une valeur à la variable d'environnement `PGUSER`. Rappelez−vous que les connexions de pg_dump sont soumises aux mécanismes normaux d'authentification des programmes clients (qui sont décrits dans le Chapitre 19).

Les sauvegardes créées par pg_dump sont cohérentes, ce qui veut dire que les modifications effectuées alors que pg_dump est en cours de fonctionnement ne sont pas dans le fichier de résultat. pg_dump ne bloque pas les autres opérations sur la base lorsqu'il fonctionne (sauf celles qui ont besoin d'un verrou exclusif, comme `VACUUM FULL`.)

**Important :** Lorsque votre base de données dépend des OID (par exemple en tant que clés

étrangères), vous devez indiquer à pg_dump de sauvegarder aussi les OID. Pour cela, utilisez l'option `-o` sur la ligne de commande. Les << gros objets >> ne sont pas non plus sauvegardés par défaut. Consultez la page de référence de la commande pg_dump si vous utilisez les << gros objets >>.

## 22.1.1. Restaurer la sauvegarde

Les fichiers texte créés par pg_dump sont prévus pour être lus par le programme psql. La syntaxe générale d'une commande de restauration est

```
psql base_de_donnees < fichier_d_entree
```

où *fichier_d_entree* est ce que vous avez précisé comme *fichier_de_sortie* à la commande pg_dump. La base de données *base_de_donnees* n'est pas créée par cette commande. Vous devez la créer vous−même à partir de `template0` avant d'exécuter psql (par exemple avec `createdb -T template0 base_de_donnees`). psql propose des options similaires à celles de pg_dump pour contrôler l'emplacement du serveur de bases de données et le nom d'utilisateur. Voyez la page de référence de psql pour plus d'informations.

La base de données cible doit déjà exister avant de lancer la restauration, ainsi que les utilisateurs possédant les objets dans la base de données sauvegardée ou ayant des droits sur ces objets. S'ils n'existent pas, la restauration échouera pour la création des objets dont ils sont propriétaires ou pour lesquels ils ont des droits (quelque fois, cela correspond à ce que vous souhaitez mais ce n'est pas le cas habituellement).

Une fois restauré, il est recommandé de lancer *ANALYZE* sur chacune des bases de données afin que l'optimiseur de requêtes dispose de statistiques utiles. Une façon aisée de le faire est de lancer la commande `vacuumdb -a -z` pour lancer `ANALYZE` sur toutes les bases de données ; ceci est équivalent à lancer `VACUUM ANALYZE` manuellement.

La capacité de pg_dump et psql à écrire et à lire dans des tubes permet de sauvegarder une base de données directement d'un serveur sur un autre. Par exemple :

```
pg_dump -h serveur1 base_de_donnees | psql -h serveur2 base_de_donnees
```

> **Important :** Les fichiers de sauvegarde produits par pg_dump sont relatifs à `template0`. Cela signifie que chaque langage, procédure, etc. ajoutés à `template1` seront aussi sauvegardés par pg_dump. En conséquence, si vous utilisez une base `template1` modifiée, vous devez créer la base vide à partir de `template0`, comme dans l'exemple précédent.

Pour des conseils sur le chargement efficace de grosses quantités de données dans PostgreSQL, référez−vous à la Section 13.4.

## 22.1.2. Utilisation de pg_dumpall

Le mécanisme précédent est peu pratique pour sauvegarder un serveur de bases de données complet. pg_dumpall est prévu pour cela. pg_dumpall sauvegarde toutes les bases de données d'un groupe de bases de données PostgreSQL (appelé cluster) et préserve les données communes au groupe de bases comme les utilisateurs et les groupes. L'utilisation basique de cette commande est :

```
pg_dumpall > fichier_de_sortie
```

Le fichier de sauvegarde résultant peut être restauré avec psql :

```
psql -f fichier_d_entree template1
```

(Vous pouvez utiliser n'importe quelle base de données pour vous connecter mais si vous êtes en train de recharger un serveur vide, `template1` est la seule base de données disponible.) Il faut obligatoirement être le super−utilisateur de la base pour restaurer une sauvegarde faite avec pg_dumpall, pour pouvoir restaurer les informations sur les utilisateurs et les groupes.

## 22.1.3. Gérer les grosses bases de données

Comme PostgreSQL permet que des tables soient plus grandes que la taille maximale d'un fichier sur votre système de fichiers, sauvegarder une telle table en fichier peut poser des problèmes. Comme pg_dump peut écrire sur la sortie standard, vous pouvez utiliser des outils standard d'Unix pour contourner ce problème éventuel.

**Compresser le fichier de sauvegarde.** Vous pouvez utiliser votre programme de compression habituel. Par exemple gzip.

```
pg_dump base_de_donnees | gzip > nom_fichier.gz
```

Pour restaurer :

```
createdb base_de_donnees
gunzip -c nom_fichier.gz | psql base_de_donnees
```

ou

```
cat nom_fichier.gz | gunzip | psql base_de_donnees
```

**Couper le fichier avec `split`.** La commande `split` vous permet de découper le fichier en morceaux d'une taille acceptable pour le système de fichiers sous−jacent. Par exemple, pour faire des morceaux de 1 Mo :

```
pg_dump base_de_donnees | split -b 1m - nom_fichier
```

Pour restaurer :

```
createdb base_de_donnees
cat nom_fichier* | psql base_de_donnees
```

**Utilisation du format de sauvegarde spécial.** Si PostgreSQL est installé sur un système où la bibliothèque de compression zlib est disponible, ce format de sauvegarde spécial peut être utilisé. Pour les grandes bases de données, cela produira un fichier de sauvegarde d'une taille comparable à celle de `gzip`, avec l'avantage supplémentaire de permettre de restaurer des tables sélectivement. La commande qui suit sauvegarde une base de données en utilisant le format de sauvegarde spécial :

```
pg_dump -Fc base_de_donnees > nom_fichier
```

Un format personnalisé de la sauvegarde n'est pas un script pour psql mais doit, à la place, être restauré avec pg_restore. Voir les pages de référence de pg_dump et pg_restore pour quelques détails.

### 22.1.4. Limitations

Pour des raisons de compatibilité avec les versions précédentes, pg_dump ne sauvegarde pas les gros objets par défaut. Pour les sauvegarder, vous devez utiliser soit le format de sauvegarde spécial soit le format TAR, et passer l'option `-b` à pg_dump. Voyez la page de référence de pg_dump pour plus de détails. Le répertoire `contrib/pg_dumplo` des fichiers sources de PostgreSQL contient aussi un programme qui peut sauvegarder les gros objets.

Merci de vous familiariser avec la page de référence de pg_dump.

## 22.2. Sauvegarde de niveau système de fichiers

Une autre stratégie de sauvegarde est de copier les fichiers utilisés par PostgreSQL pour enregistrer les données. Dans la Section 16.2, l'emplacement de ces fichiers sont donnés mais vous les avez probablement déjà trouvés si vous vous intéressez à cette méthode. Vous pouvez utiliser n'importe quelle méthode de sauvegarde, par exemple :

```
tar -cf sauvegarde.tar /usr/local/pgsql/data
```

Cependant, il y a deux restrictions qui rendent cette méthode inutilisable ou en tout cas inférieure à la méthode pg_dump.

1. Le serveur de base de données *doit* être arrêté pour obtenir une sauvegarde utilisable. Toutes les demi−mesures, comme supprimer toutes les connections, ne fonctionneront *pas* (principalement parce que `tar` et les outils similaires ne font pas une image atomique de l'état du système de fichiers à un moment spécifique). Vous trouverez des informations sur la façon d'arrêter le serveur PostgreSQL dans la Section 16.6.

   Il va sans dire que vous devez aussi éteindre le serveur avant de restaurer les données.
2. Si vous vous êtes aventurés dans les détails de l'organisation de la base de données, vous pouvez être tentés de ne sauvegarder et de ne restaurer que certaines tables ou bases de données particulières. Ceci ne fonctionnera *pas* parce que les informations contenues dans ces fichiers ne sont qu'à moitié vraies. L'autre moitié est dans les fichiers journaux de validation `pg_clog/*`, qui contiennent l'état de la validation de chaque transaction. Un fichier de table n'est utilisable qu'avec cette information. Bien entendu, il est impossible de ne restaurer qu'une table et les données `pg_clog` associées car cela rendrait toutes les autres tables du serveur inutilisables. Donc, les sauvegardes du système de fichiers fonctionnent seulement pour les restaurations complètes d'un groupe entier de bases de données.

Une autre approche à la sauvegarde du système de fichiers est de réaliser une << image cohérente >> du répertoire des données si le système de fichiers supporte cette fonctionnalité (et que vous avez confiance en sa bonne implémentation). La procédure typique est de faire une << image gelée >> du volume contenant la base de données, et enfin de copier le répertoire data complètement (pas seulement quelques parties, voir ci−dessus) de l'image sur un périphérique de sauvegarde, puis de libérer l'image gelé. Ceci fonctionnera même si le serveur de la base de données est en cours d'exécution. Néanmoins, une sauvegarde créée de cette façon sauvegarde les fichiers de la base de données dans un état où le serveur n'était pas correctement arrêté ; du coup, au lancement du serveur à partir des données sauvegardées, PostgreSQL pensera que le serveur s'est stoppé brutalement et rejouera les journaux WAL. Ceci n'est pas un problème, soyez−en juste conscient (et assurez−vous d'inclure les fichiers WAL dans votre sauvegarde).

Si votre base de données est répartie sur plusieurs systèmes de fichiers, il pourrait ne pas y avoir de moyens pour obtenir des images gelées exactement simultanément de tous les disques. Par exemple, si vos fichiers de données et vos journaux WAL sont sur des disques différents ou si les espaces logiques sont sur des systèmes de fichiers différents, il pourrait ne pas être possible d'utiliser une sauvegarde par image parce que ces dernières doivent être simultanées. Lisez la documentation de votre système de fichiers avec attention avant de faire confiance à la technique d'images cohérentes dans de telles situations. L'approche la plus sûre est d'arrêter le serveur de bases de données assez longtemps pour créer toutes les images gelées.

Une autre option est d'utiliser rsync pour réaliser une sauvegarde du système de fichiers. Ceci se fait tout d'abord en lançant rsync alors que le serveur de bases de données est en cours d'exécution, puis en arrêtant le serveur juste assez longtemps pour lancer rsync une deuxième fois. Le deuxième rsync sera beaucoup plus rapide que le premier car il aura relativement peu de données à transférer et le résultat final sera cohérent parce que le serveur était arrêté. Cette méthode permet de réaliser une sauvegarde du système de fichiers avec un arrêt minimal.

Notez aussi qu'une sauvegarde des fichiers de données ne sera pas forcément moins grosse qu'une sauvegarde SQL. Au contraire, elle sera très certainement plus grande (pg_dump ne sauvegarde pas le contenu des index, mais la commande pour les recréer).

## 22.3. Sauvegardes à chaud et récupération à un instant (PITR)

En permanence, PostgreSQL maintient des journaux WAL (*write ahead log*) dans le sous−répertoire `pg_xlog/` du répertoire des données du groupe. Ces journaux décrivent chaque modification effectuée sur les fichiers de données des bases. Ils existent principalement pour des raisons de sécurité suite à un arrêt brutal : si le système s'arrête brutalement, la base de données peut être restaurée pour avoir une cohérence des données en << rejouant >> les entrées des journaux enregistrées depuis le dernier point de vérification. Néanmoins, l'existence de ces journaux rend possible l'utilisation d'une troisième stratégie pour la sauvegarde des bases de données : nous pouvons combiner une sauvegarde au niveau système de fichiers avec la sauvegarde des fichiers WAL. Si la récupération est nécessaire, nous restaurons la sauvegarde, puis rejouons à partir des fichiers WAL sauvegardés pour amener la sauvegarde jusqu'à la date actuelle. Cette approche est plus complexe à administrer que toutes les autres approches mais elle apporte des bénéfices significatifs :

- Nous n'avons pas besoin de faire une sauvegarde parfaitement cohérente comme point de départ. Toute incohérence dans la sauvegarde sera corrigée par la ré−exécution des journaux (ceci n'est pas significativement différent de ce qu'il se passe lors d'une récupération après un arrêt brutal). Donc, nous n'avons pas besoin d'une fonctionnalité d'image système du système de fichiers, simplement tar ou un autre outil d'archivage.
- Comme nous pouvons assembler une longue séquence de fichiers à WAL pour les rejouer, la sauvegarde continue est possible en continuant simplement d'archiver les fichiers WAL. Ceci est particulièrement intéressant pour les grosses bases de données où il pourrait ne pas être facile de réaliser une sauvegarde complète fréquemment.
- Rien ne dit que nous devons rejouer les entrées WAL jusqu'à la fin. Nous pouvons stopper la ré−exécution à un certain point et avoir une image cohérente de la base de données à ce moment−là. Du coup, cette technique supporte la *récupération à un instant t* (PITR) : il est possible de restaurer la base de données à n'importe quel point dans le temps depuis la dernière sauvegarde de base.
- Si nous remplissons en continue la série de fichiers WAL dans une autre machine qui a été chargée avec le même fichier de sauvegarde de base, nous avons un système << à jour en permanence >> : à

tout moment, nous pouvons monter la deuxième machine et avoir une copie quasi complète de la base de données.

Comme avec la technique de sauvegarde standard du système de fichiers, cette méthode supporte la restauration d'un groupe de bases de données complet, pas un sous−ensemble. De plus, il requiert beaucoup d'espace d'archivage : la sauvegarde de base peut être légère mais un système très utilisé générera beaucoup de mégaoctets de trafic WAL qui seront à archiver. Malgré tout, c'est la technique de sauvegarde préférée dans beaucoup de situations où la haute fiabilité est nécessaire.

Pour récupérer avec succès suite à l'utilisation d'une sauvegarde à chaud, vous avez besoin d'une séquence continue de fichiers WAL archivés qui s'étendent au moins jusqu'au point de départ de votre sauvegarde. Pour commencer, vous devriez configurer et tester votre procédure d'archivage des journaux WAL *avant* de faire votre première sauvegarde de base. Il nous faut donc commencer par vous présenter les mécanismes d'archivage des fichiers WAL.

## 22.3.1. Configurer l'archivage WAL

Dans un sens abstrait, un système PostgreSQL fonctionnel produit une séquence indéfiniment longue d'enregistrements WAL. Le système divise physiquement cette séquence en *fichiers segment* WAL, qui font normalement 16 Mo chaque (bien que la taille puisse être modifiée lors de la construction de PostgreSQL). Les fichiers segment se voient donnés des noms numériques pour refléter leur position dans la séquence abstraite des WAL. Lorsque le système n'utilise pas l'archivage des WAL, il crée seulement quelques fichiers segment, puis les << recycle >> en renommant les fichiers segment devenus inutiles. Il est supposé qu'un fichier segment dont le contenu précède le dernier point de vérification n'a plus d'intérêt et peut être recyclé.

Lors de l'archivage des données WAL, nous voulons capturer le contenu de chaque fichier segment une fois qu'il est rempli et sauvegarder les données quelque part avant que le fichier segment ne soit recyclé pour être réutilisé. Suivant l'application et le matériel disponible, << sauvegarder les données quelque part >> peut se faire de plusieurs façons : nous pouvons copier les fichiers segment dans un répertoire NFS monté sur une autre machine, les écrire sur une cartouche (en vous assurant que vous avez un moyen de restaurer le fichier avec son nom d'origine) ou le grouper pour les graver sur un CD, ou encore autre chose. Pour fournir autant de flexibilité que possible à l'administrateur de la base de données, PostgreSQL essaie de ne faire aucune supposition sur la façon dont l'archivage est réalisé. À la place, PostgreSQL vous laisse spécifier une commande shell à exécuter pour copier le fichier segment rempli là où vous le souhaitez. La commande pourrait être aussi simple qu'un cp ou il pourrait impliquer un shell complexe — à vous de voir.

La commande shell à utiliser est spécifiée par le paramètre de configuration archive_command qui, en pratique, sera toujours placé dans le fichier `postgresql.conf`. Dans cette chaîne, tout `%p` est remplacé par le chemin absolu de l'archive alors que tout `%f` est remplacé seulement par le nom du fichier. Écrivez `%%` si vous avez besoin d'écrire le vrai caractère `%` dans la commande. La commande utile la plus simple est quelque chose comme

```
archive_command = 'cp -i %p /mnt/serveur/repertoire_archive/%f </dev/null'
```

qui copiera les segments WAL archivables dans le répertoire `/mnt/serveur/repertoire_archive`. (Ceci est un exemple, pas une recommandation, et pourrait ne pas fonctionner sur toutes les plateformes.)

La commande d'archivage sera exécutée en tant qu'utilisateur propriétaire du serveur PostgreSQL. Comme la série de fichiers WAL en cours d'archivage contient réellement tout ce qui se trouve dans votre base de données, vous vous assurerez que les données archivées sont protégées des autres utilisateurs ; par exemple, si

les archives sont stockées dans un répertoire où se trouvent des droits de lecture pour le groupe ou pour les autres.

Il est important que la commande d'archivage renvoie le code de sortie zéro si et seulement si l'exécution a réussi. En obtenant un résultat zéro, PostgreSQL supposera que le fichier segment WAL a été archivé avec succès et qu'il peut le supprimer ou le recycler. Néanmoins, un statut différent de zéro indique à PostgreSQL que le fichier n'a pas été archivé ; il essaiera périodiquement jusqu'à ce qu'il y arrive.

La commande d'archivage devrait être généralement conçue pour refuser d'écraser tout fichier archive déjà existant. Ceci est une fonctionnalité de sécurité importante pour préserver l'intégrité de votre archive dans le cas d'une erreur de l'administrateur (comme l'envoi de la sortie de deux serveurs différents dans le même répertoire d'archivage). Il est conseillé de tester votre commande d'archivage proposée pour vous assurer qu'en effet il n'écrase pas un fichier existant *et qu'il renvoie un statut différent de zéro dans ce cas*. Nous avons découvert que `cp -i` travaille correctement sur certaines plateformes, mais pas sur toutes. Si la commande choisie ne gère pas elle−même ce cas, vous pouvez ajouter une commande pour tester l'existence du fichier d'archivage. Par exemple, quelque chose comme

```
archive_command = 'test ! -f .../%f && cp %p .../%f'
```

fonctionne correctement sur la plupart des variantes Unix.

Lors de la conception de votre configuration d'archivage, considérez ce qui se passerait si la commande d'archivage échouait de façon répétée parce que certains aspects demanderaient une intervention de l'opérateur ou à cause d'un manque d'espace dans le répertoire d'archivage. Par exemple, ceci pourrait arriver si vous écrivez sur une cartouche sans changeur automatique ; quand la cartouche est remplie, rien ne peut être archivé tant que la cassette n'est pas changée. Vous devez vous assurer que toute erreur ou requête à un opérateur humain est rapportée de façon approprié pour que la situation puisse être résolue relativement rapidement. Le répertoire `pg_xlog/` continuera à se remplir de fichiers segment WAL jusqu'à la résolution de la situation.

La vitesse de la commande d'archivage n'est pas importante, tant qu'elle est au même rythme que la génération de données WAL du serveur. Les opérations normales continuent même si le processus d'archivage est un peu plus lent. Si l'archivage est significativement plus lent, alors la quantité de données qui pourrait être perdue va croître. Cela signifie aussi que le répertoire `pg_xlog/` contiendra un grand nombre de fichiers segment non archivés, qui finiront éventuellement par dépasser l'espace disque disponible. Il vous est conseillé de surveiller le processus d'archivage pour vous assurer que tout fonctionne normalement.

Si vous êtes encore inquiet sur votre capacité à récupérer tout à l'instant présent, vous devriez prendre quelques actions supplémentaires pour vous assurer que les fichiers segments WAL partiellement remplis sont aussi copiés quelque part. Ceci est particulièrement important si votre serveur génère peu de trafic WAL (ou que seules certaines périodes sont dans ce cas) car cela prendra beaucoup de temps avant qu'un fichier segment WAL soit complètement remplis et prêt à archiver. Une façon possible de gérer ceci est de configurer un job cron qui, périodiquement (une fois par minute, peut−être), identifie le fichier segment WAL en cours et le sauvegarde à un endroit sûr. Ensuite, la combinaison des segments WAL archivés et du segment courant sauvegardé seront suffisant pour vous assurer de pouvoir toujours restaurer à une minute près. Ce comportement n'est pas actuellement construit dans PostgreSQL parce que nous ne voulions pas compliquer la définition de archive_command en lui demandant de garder trace des différentes copies archivées successivement du même fichier WAL. archive_command est seulement appelée sur des segments WAL complets. Sauf dans le cas d'un échec, il sera appelé seulement une fois pour un nom de fichier donné.

En écrivant votre commande d'archivage, vous devez vous assurer que les noms de fichier à archiver auront 64 caractères maximum et peuvent contenir toute combinaison de lettres ASCII, de chiffres et de points. Il n'est pas nécessaire de rappeler le chemin complet original (`%p`) mais il est nécessaire de rappeler le nom du fichier (`%f`).

Notez que, bien que l'archivage WAL vous autorisera à restaurer toute modification réalisée sur les données de votre base PostgreSQL, il ne restaurera pas les modifications effectuées sur les fichiers de configuration (c'est−à−dire `postgresql.conf`, `pg_hba.conf` et `pg_ident.conf`) car ceux−ci sont édités manuellement plutôt qu'au travers d'opérations SQL. Vous pourriez souhaiter conserver les fichiers de configuration à un endroit où ils seront sauvegardés avec vos procédures standards de sauvegarde du système de fichiers. Voir la Section 16.4.1 pour savoir comment modifier l'emplacement des fichiers de configuration.

## 22.3.2. Réaliser une sauvegarde de base

La procédure pour réaliser une sauvegarde de base est relativement simple :

1. Assurez−vous que l'archivage WAL est activée et fonctionnelle.
2. Connectez−vous à la base de données en tant que superutilisateur et lancez la commande

   ```
   SELECT pg_start_backup('label');
   ```

   où `label` est toute chaîne que vous voulez utiliser pour identifier de façon unique l'opération de sauvegarde (une bonne pratique est d'utiliser le chemin complet où vous avez l'intention de placer le fichier de sauvegarde). `pg_start_backup` crée un fichier *label de sauvegarde* nommé `backup_label` dans le répertoire du groupe avec des informations sur votre sauvegarde.

   Peu importe la base de données à laquelle vous vous connectez pour lancer cette commande. Vous pouvez ignorer le résultat renvoyé par la fonction mais, si elle rapporte une erreur, gérez−la avant de continuer.
3. Lancez la sauvegarde en utilisant n'importe quel outil de sauvegarde du système de fichiers comme tar ou cpio. Il n'est ni nécessaire ni désirable de stopper les opérations normales de la base de données lorsque vous faites cela.
4. Connectez−vous de nouveau sur la base de données en tant que superutilisateur et lancez la commande

   ```
   SELECT pg_stop_backup();
   ```

   Elle devrait réussir.
5. Une fois que les fichiers des segments WAL utilisées lors de la sauvegarde sont archivées de la même façon qu'une partie de l'activité normale de sauvegarde de la base de données, vous avez terminé.

Il n'est pas nécessaire d'être très concerné sur le temps passé entre `pg_start_backup` et le début réel de la sauvegarde, pas plus qu'entre la fin de la sauvegarde et `pg_stop_backup` ; un délai de quelques minutes ne posera pas de problème. Vous devez néanmoins vous assurer que ces opérations sont effectuées dans la bonne séquence et ne se recouvrent pas.

Assurez−vous que votre sauvegarde inclut tous les fichiers du répertoire du groupe de bases de données (c'est−à−dire `/usr/local/pgsql/data`). Si vous utilisez des espaces logiques qui ne se trouvent pas dans ce répertoire, faites attention à bien les inclure (et assurez−vous que votre sauvegarde archive les liens symboliques comme des liens, sinon la restauration posera problème pour les espaces logiques).

Néanmoins, vous devriez omettre de sauvegarder les fichiers du sous−répertoire `pg_xlog/` contenu dans le répertoire du groupe. Cette petite complication est intéressante parce qu'elle réduit le risque d'erreurs lors de la restauration. Ceci est facile à arranger si `pg_xlog/` est un lien symbolique pointant quelque part à l'extérieur du répertoire du groupe, ce qui est une configuration commune pour des raisons de performance.

Pour utiliser cette sauvegarde, vous aurez besoin de conserver les fichiers segments WAL générés pendant ou après le lancement de la sauvegarde. Pour vous aider dans ce travail, la fonction `pg_stop_backup` crée un *fichier historique de la sauvegarde* qui est immédiatement stocké dans l'aire des archives WAL. Ce fichier est nommé d'après le nom du premier fichier segment WAL dont vous avez besoin pour utiliser la sauvegarde. Par exemple, si le fichier WAL du début est `000000010000123400000055CD`, le fichier historique sera nommé quelque chose comme `000000010000123400000055CD.007C9330.backup` (le deuxième nombre dans le nom de ce fichier contient la position exacte à l'intérieur du fichier WAL et peut être habituellement ignorée). Une fois que vous avez archivé de façon sûre la sauvegarde du système de fichiers et les segments WAL utilisés pendant la sauvegarde (comme spécifié dans le fichier d'historique des sauvegardes), tous les segments WAL archivés avec des noms numériquement plus petits ne sont plus nécessaires pour la récupération de la sauvegarde du système de fichiers et pourraient être supprimés. Néanmoins, vous devriez penser à conserver plusieurs ensembles de sauvegarde pour être absolument certain de pouvoir récupérer vos données. Gardez en tête que seuls les fichiers WAL complets sont archivés, donc il y aura un délai entre l'exécution de `pg_stop_backup` et l'archivage de tous les fichiers de segment WAL pour rendre la sauvegarde du système de fichiers cohérente.

Le fichier d'historique de la sauvegarde est un simple fichier texte. Il contient le label que vous avez donné à `pg_start_backup`, ainsi que l'heure de début et de fin de la sauvegarde. Si vous utilisez le label pour identifier où le fichier de sauvegarde associé est conservé, alors le fichier historique archivé est suffisant pour vous dire quel fichier de sauvegarde restaurer, si vous en avez besoin.

Comme vous devez conserver tous les fichiers WAL archivés depuis votre dernière sauvegarde de base, l'intervalle entre les sauvegardes de base devrait habituellement être choisie suivant la quantité de stockage que vous voulez consommer en fichiers archives WAL. Vous devriez aussi considérer combien de temps vous êtes prêt à dépenser pendant la récupération, si celle−ci est nécessaire — le système devra rejouer tous les segments WAL et ceci peut prendre beaucoup de temps si la dernière sauvegarde de base est ancienne.

Il est aussi important de noter que la fonction `pg_start_backup` crée un fichier nommé `backup_label` dans le répertoire du groupe de bases de données qui est ensuite supprimé de nouveau par `pg_stop_backup`. Ce fichier sera bien sûr archivé comme faisant parti du fichier de sauvegarde. Le fichier label de la sauvegarde inclut la chaîne label que vous avez donné à `pg_start_backup`, ainsi que l'heure à laquelle `pg_start_backup` a été exécuté et le nom du fichier WAL du début. En cas de confusion, il sera du coup possible de regarder dans le fichier sauvegarde et de déterminer exactement de quelle session de sauvegarde provient ce fichier.

Il est aussi possible de faire une sauvegarde alors que le postmaster est arrêté. Dans ce cas, vous ne pouvez évidemment pas utiliser `pg_start_backup` ou `pg_stop_backup` et vous serez donc contraint à garder trace vous−même des fichiers de sauvegarde et de jusqu'où vont les fichiers WAL associés. Il est généralement mieux de suivre la procédure de sauvegarde à chaud ci−dessus.

## 22.3.3. Récupérer à partir d'une sauvegarde à chaud

OK, le pire est arrivé et vous avez besoin de récupérer votre sauvegarde. Voici la procédure :

1. Arrêtez le postmaster s'il est en cours d'exécution.

2. Si vous avez de la place pour le faire, copiez le répertoire entier des données du groupe et tout espace logique dans un emplacement temporaire au cas où vous en auriez besoin plus tard. Notez que cette précaution demandera que vous ayez assez de place libre sur votre système pour contenir deux copies de votre base de données existante. Si vous n'avez pas assez de place, vous devez au moins copier le contenu du sous−répertoire `pg_xlog` du répertoire des données car il pourrait contenir des journaux qui n'ont pas été archivés avant l'arrêt du serveur.
3. Effacez tous les fichier et sous−répertoires existants sous le répertoire des données du groupe et sous les répertoires racines des espaces logiques que vous utilisez.
4. Restaurez les fichiers de la base de données à partir de votre sauvegarde. Faites attention à ce qu'ils soient restaurés avec le bon propriétaire (l'utilisateur système de la base de données, et non pas root !) et avec les bons droits. Si vous utilisez les espaces logiques, vous vérifirez que les liens symboliques dans `pg_tblspc/` ont été correctement restaurés.
5. Supprimez tout fichier présent dans `pg_xlog/` ; ils proviennent de la sauvegarde et sont du coup probablement obsolètes. Si vous n'avez pas archivé `pg_xlog/` du tout, alors re−créez ce répertoire ainsi que le sous−répertoire `pg_xlog/archive_status/`.
6. Si vous aviez des fichiers segments WAL non archivés que vous avez sauvegardé dans l'étape 2, copiez−les dans `pg_xlog/` (il est mieux de les copier, pas de les déplacer, car vous aurez toujours les fichiers non modifiés si un problème survient et que vous devez recommencer).
7. Créez un fichier de commandes de récupération `recovery.conf` dans le répertoire des données du groupe (voir Configuration de la récupération). De plus, vous pourriez vouloir modifier temporairement `pg_hba.conf` pour empêcher les utilisateurs ordinaires de se connecter tant que vous n'êtes pas certain que la récupération a réussi.
8. Lancez le postmaster. Le postmaster se trouvera en mode récupération et commencera la lecture des fichiers WAL archivés dont il a besoin. À la fin du processus de récupération, le postmaster renommera `recovery.conf` en `recovery.done` (pour empêcher de retourner accidentellement en mode de récupération dans le cas d'un arrêt brutal un peu plus tard), puis commencera les opérations normales de la base de données.
9. Inspectez le contenu de la base de données pour vous assurer que vous avez récupéré ce que vous vouliez. Sinon, retournez à l'étape 1. Si tout va bien, laissez vos utilisateurs venir en restaurant le fichier `pg_hba.conf` à son état normal.

La partie clé de tout ceci est de configurer le fichier de commandes de récupération. Il décrit comment vous voulez récupérer et à quel point la récupération doit fonctionner. Vous pouvez utiliser `recovery.conf.sample` (normalement présent dans le répertoire d'installation `share/`) comme prototype. La seule chose que vous devez absolument spécifier dans `recovery.conf` est `restore_command` indiquant à PostgreSQL comment récupérer les fichiers segments WAL archivés. Comme `archive_command`, ceci est une chaîne contenant le nom de la commande. Elle pourrait contenir `%f`, qui est remplacé par le nom du journal souhaité, et `%p`, qui est remplacé par le chemin absolu où copier le journal. Écrivez `%%` si vous avez besoin d'embarquer un vrai caractère `%` dans la commande. La commande utile la plus simple est quelque chose comme

```
restore_command = 'cp /mnt/serveur/répertoire_archive/%f %p'
```

qui copiera les segments WAL précédemment archivés à partir du répertoire `/mnt/serveur/répertoire_archive`. Vous pourriez bien sûr utiliser quelque chose de plus compliqué, peut−être même un script shell qui demandera à l'utilisateur de monter la cassette appropriée.

Il est important que la commande renvoie un code de sortie différent de zéro en cas d'échec. La commande *se verra demander* les journaux absents de l'archive ; il doit renvoyer autre chose que zéro dans ce cas. Ceci n'est pas une condition d'erreur. Soyez conscient que le nom de base du chemin `%p` sera différent de `%f` ; ne vous attendez pas à ce qu'ils soient interchangeables.

Les segments WAL qui n'ont pas pu être trouvés dans l'archive seront recherchés dans pg_xlog/ ; cela autorise l'utilisation des segments non archivés. Néanmoins, les segments disponibles à partir de l'archive seront utilisés de préférence par rapport aux fichiers dans pg_xlog/. Le système ne surchargera pas le contenu existant de pg_xlog/ lors de la récupération des fichiers archivés.

Normalement, la récupération traitera tous les segments WAL disponibles, restaurant du coup la base de données à cet instant (ou aussi proche que nous le pouvons suivant les segments WAL disponibles). Mais, si vous voulez récupérer à un instant particulier (disons, juste avant que l'administrateur junior ait supprimé votre table principale de transaction), spécifiez simplement le point d'arrêt requis dans recovery.conf. Vous pouvez spécifier le point d'arrêt, connu sous le nom de << cible de récupération >>, soit par la date/heure soit par l'ID de la dernière transaction. Au moment où nous écrivons ceci, seule l'option date/heure est tout à fait utilisable car il n'existe pas d'outils pour vous aider à identifier avec une certaine précision l'ID de transaction à utiliser.

> **Note :** Le point d'arrêt doit se situer après l'heure de fin de la sauvegarde de base (le moment de pg_stop_backup). Vous ne pouvez pas utiliser une sauvegarde de base pour récupérer à un tel instant où la sauvegarde était encore en cours (pour récupérer jusqu'à cet instant, vous devez récupérer votre sauvegarde de base précédente et recommencer à partir de là).

---

#### 22.3.3.1. Configuration de la récupération

Ces configurations peuvent seulement être placées dans le fichier recovery.conf et s'appliquent seulement pour la durée de la récupération. Ils doivent être réinitialisés pour toute récupération ultérieure que vous souhaitez réaliser. Ils ne peuvent pas être modifiés une fois que la récupération a commencé.

restore_command (string)
: La commande shell à exécuter pour récupérer un segment archivé de la série de fichiers WAL. Ce paramètre est requis. Tout %f dans la chaîne est remplacé par le nom du fichier à récupérer à partir de l'archive et tout %p est remplacé par le chemin absolu pour le copier sur le serveur. Écrivez %% pour embarquer un vrai caractère % dans la commande

  Il est important que la commande renvoie un code de sortie zéro si et seulement si elle a réussi. La commande *se verra demander* les noms des fichiers absents dans l'archive ; elle doit renvoyer une valeur différente de zéro dans ce cas. Exemples :

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy /mnt/server/archivedir/%f "%p"'  # Windows
```

recovery_target_time (timestamp)
: Ce paramètre spécifie le temps à partir duquel le serveur doit arrêter la récupération. Au plus un entre recovery_target_time et recovery_target_xid peut être spécifié. Par défaut, la récupération se passe jusqu'à la fin du journal WAL. Le point d'arrêt précis est aussi influencé par recovery_target_inclusive.

recovery_target_xid (string)
: Ce paramètre spécifie l'ID de transaction où arrêter la récupération. Gardez en tête qu'alors que les ID de transactions sont affectés séquentiellement au début de la transaction, les transactions peuvent se terminer dans un ordre numérique différent. Les transactions qui seront récupérées sont celles qui ont été validées avant celle spécifiée (et quelque fois en l'incluant). Au plus un entre recovery_target_xid et recovery_target_time peut être spécifié. La valeur par défaut est de récupérer jusqu'à la fin du journal WAL. Le point d'arrêt précis est aussi influencé par recovery_target_inclusive.

recovery_target_inclusive (boolean)

Spécifie si nous stoppons tout de suite après la cible de récupération spécifiée (`true`) ou tout juste avant (`false`). S'applique à recovery_target_time et recovery_target_xid, quelque soit celui qui est spécifié pour cette récupération. Ceci indique si les transactions ayant exactement l'instant de validation cible ou l'ID, respectivement, seront inclus dans la récupération. La valeur par défaut est `true`.

`recovery_target_timeline` (`string`)

Spécifie la récupération à un timeline particulier. La valeur par défaut est de récupérer suivant le timeline en cours au moment où la sauvegarde de base a été effectuée. Vous aurez seulement besoin de configurer ce paramètre dans les situations complexes de récupération où le besoin de retourner à un état qui a été atteint après une récupération à un temps donné. Voir la Section 22.3.4 pour des informations.

## 22.3.4. Timelines

La possibilité de restaurer la base de données à un instant précédent crée une complexité qui sont la base des histoires de science−fiction sur le voyage dans le temps et les univers parallèles. Dans l'historique original de la base de données, vous avez peut−être supprimé une table critique à 17h15 mardi soir. Imperturbable, vous récupérez votre sauvegarde, la restaurez jusqu'à 17h14 mardi soir et êtes de nouveau fonctionnel. Dans *cette* histoire de l'univers de la base de données, vous n'avez jamais supprimé la table. Mais, supposez que vous réalisez plus tard qu'après tout, ce n'était pas une si grande idée et que vous voudriez revenir à un point plus lointain dans l'historique original. Vous ne serez plus capable de le faire si, une fois que votre base de donnée était de nouveau fonctionnelle, elle a écrit par dessus certaines des séquences de fichiers segments WAL qui vous aurait permis de récupérer à cet instant. Donc, vous voulez réellement distinguer les séries d'enregistrements WAL générées après la récupération à un instant donné de celles générées dans l'historique originale de la base de données.

Pour gérer ces problèmes, PostgreSQL comprends la notion de *timelines*. Chaque fois que vous récupérez à un certain instant précédant la fin de la séquence WAL, un nouveau timeline est créé pour identifier les séries d'enregistrements WAL générées après la récupération (néanmoins, si la récupération continue jusqu'à la fin des WAL, nous ne commençons pas une nouvelle timeline : nous étendons celle qui existe). Le numéro d'identifiant de la timeline fait partie des noms des fichiers segment WAL et, du coup, une nouvelle timeline ne réécrit pas sur les données générées par des timelines précédentes. En fait, il est possible d'archiver plusieurs timelines différentes. Bien que cela semble être une fonctionnalité inutile, parfois, cela vous sauve la vie. Considérez la situation où vous n'êtes plus sûr de l'instant jusqu'où récupérer. Du coup, vous devez tester des récupérations à différents instants jusqu'à trouver le meilleur moment dans l'ancien historique. Sans les timelines, ce processus génèrerait un incroyable bazar. Avec les timelines, vous pouvez récupérer à *n'importe quel* état précédent, ceci incluant les états dans les branches de timelines que vous abandonnerez plus tard.

Chaque fois qu'une nouvelle timeline est créée, PostgreSQL crée un fichier d'<< historique des timeline >> qui montre les timelines, leur branchement et le moment auquel c'est arrivé. Ces fichiers historiques sont requis pour permettre au système de récupérer les bons fichiers segment WAL lors de la récupération à partir d'une archive contenant plusieurs timelines. Du coup, elles sont archivées dans l'aire des WAL comme tous les fichiers segment WAL. Les fichiers historiques sont de simples fichiers texte, donc il est peu coûteux et approprié de les conserver indéfiniment (contrairement aux fichiers segments qui sont gros). Vous pouvez, si vous le souhaitez, ajouter des commentaires au fichier historique pour ajouter vos propres notes sur comment et pourquoi cette timeline est particulièrement intéressante. De tels commentaires seront particulièrement utiles quand vous avez un ticket des différentes timelines comme résultat de l'expérimentation.

Le comportement par défaut de la récupération est de récupérer parmi la timeline en cours au moment où la sauvegarde de base a été effectuée. Si vous voulez récupérer à une timeline enfant (c'est−à−dire si vous voulez retourner à un état qui a été enregistré après la tentative de récupération), vous avez besoin de spécifier l'ID cible de la timeline dans `recovery.conf`. Vous ne pouvez pas récupérer des timelines qui ont effectué leur branchement plus tôt que le moment où a été effectuée la sauvegarde de base.

### 22.3.5. Avertissements

Au moment où nous écrivons ces lignes, il existe plusieurs limitations sur la technique de sauvegarde à chaud. Elles seront probablement corrigées dans une prochaine version :

- Les opérations sur des index autres que les B−tree (hash, R−treeet GiST) ne sont pas tracées actuellement dans les WAL, donc ces types d'index ne seront pas mis à jour. Le contournement recommandé est de `REINDEX`er manuellement chacun de ces index après avoir terminé une opération de récupération.
- Si une commande `CREATE DATABASE` est exécutée alors qu'une sauvegarde de la base est en cours et qu'ensuite la base de données modèle que `CREATE DATABASE` a copié est modifiée alors que la sauvegarde de base est toujours en cours, il est possible que la récupération causera la propagation de ces modifications dans la base de données créée. Bien sûr, ceci n'est pas désirable. Pour éviter ce risque, il est mieux de ne pas modifier les bases de données modèle lors d'une sauvegarde de base.
- Les commandes `CREATE TABLESPACE` sont tracées dans WAL avec le chemin absolu littéral et seront donc rejouées en tant que créations d'espaces logiques avec le même chemin absolu. Ceci pourrait être indésirable si la trace est rejouée sur la même machine mais dans un nouveau répertoire de données : la ré−exécution surchargera toujours le contenu de l'espace logique original. Pour éviter des problèmes potentiels de cette sorte, la meilleure pratique est de prendre une nouvelle sauvegarde de base après la création ou la suppression d'espaces logiques.

De plus, il devrait être noté que le format actuel des WAL est extrêmement difficile à gérer car il inclut de nombreuses images des pages disques. Ceci est approprié dans le cas des récupérations après un arrêt brutal car nous pourrions avoir besoin de corriger des pages disque partiellement écrites. Néanmoins, il n'est pas nécessaire de stocker autant de copies de page pour les opérations PITR. Un développement possible serait de compresser les données des WAL archivées en supprimant les copies inutiles de pages. Entre temps, les administrateurs pourraient souhaiter réduire le nombre d'images de pages inclus dans WAL en augmentant autant que possible les paramètres d'intervalle des points de vérification.

## 22.4. Migration entre les différentes versions

Cette section discute de la façon de migrer vos données de la base à partir d'une version de PostgreSQL vers une autre, plus récente. La procédure d'installation du logiciel *per se* n'est pas le sujet de cette section ; ces détails sont dans le Chapitre 14.

En règle générale, le format interne des données est modifié entre les différentes versions de PostgreSQL (quand le nombre après le deuxième point change). Ceci ne s'applique pas entre les différentes sorties mineures ayant le même numéro de version majeur (quand le nombre après le deuxième point change) ; elles ont toujours un format de stockage compatible. Par exemple, les versions 7.0.1, 7.1.2 et 7.2 ne sont pas compatibles, alors que les versions 7.1.1 et 7.1.2 le sont. Lorsque vous mettez à jour entre des versions compatibles, vous pouvez simplement remplacer les exécutables et ré−utiliser le répertoire des données sur le disque. Sinon, vous avez besoin de sauvegarder vos données et de les restaurer sur le nouveau serveur. Ceci doit se faire en utilisant pg_dump ; les méthodes de sauvegarde au niveau système de fichiers ne

fonctionneront évidemment pas. Il existe des vérifications en place pour vous empêcher d'utiliser un répertoire de données d'une version incompatible version de PostgreSQL, donc aucun mal ne sera fait si vous essayez de lancer un serveur d'une mauvaise version dans un répertoire de données.

Il est recommandé d'utiliser les programmes pg_dump et pg_dumpall à partir de la nouvelle version de PostgreSQL, pour utiliser les avantages de toutes améliorations effectuées sur ces programmes. Les versions actuelles des programmes de sauvegarde peuvent lire des données à partir des serveurs d'anciennes versions, jusqu'à la 7.0.

Vous minimiserez la durée d'indisponibilité en installant le nouveau serveur dans un répertoire différent et en lançant l'ancien et le nouveau serveur en parallèle sur des ports différents, puis en utilisant des commandes comme

```
pg_dumpall -p 5432 | psql -d template1 -p 6543
```

pour transférer les données. Ou utilisez un fichier intermédiaire si vous voulez. Vous pouvez alors éteindre le nouveau serveur et démarrer le nouveau sur le port que l'ancien utilisait. Vous devez vous assurer que l'ancienne base de données n'est pas modifiée après que vous ayez lancé pg_dumpall, sans quoi ces modifications seraient évidemment perdues. Référez vous au Chapitre 19 pour savoir comment interdire l'accès.

En pratique, vous voudrez certainement tester votre application sur le nouveau serveur avant de basculer définitivement. C'est une autre raison pour configurer des installations concurrentes avec l'ancienne et la nouvelle version.

Si vous ne pouvez pas ou ne voulez pas lancer les deux serveurs en parallèle, vous pouvez faire l'étape de sauvegarde avant d'installer la nouvelle version, éteindre le serveur, déplacer l'ancienne version à un autre endroit, installer la nouvelle, la démarrer et enfin restaurer les données. Par exemple :

```
pg_dumpall > sauvegarde.sql
pg_ctl stop
mv /usr/local/pgsql /usr/local/pgsql.old
cd ~/postgresql-8.0.5
gmake install
initdb -D /usr/local/pgsql/data
postmaster -D /usr/local/pgsql/data
psql -f sauvegarde.sql template1
```

Vous trouverez les méthodes pour arrêter et démarrer les serveurs, ainsi que d'autres détails dans le Chapitre 16. Les instructions d'installation vous donneront des conseils sur les endroits stratégiques pour réaliser ces opérations.

> **Note :** Quand vous << déplacez l'ancienne version à un autre endroit >>, l'ancienne installation pourrait ne plus être tout à fait utilisable. Certains des exécutables contiennent les chemins absolus vers les différents programmes et fichiers de données installés. Ceci n'est habituellement pas un gros problème mais si vous planifiez d'utiliser deux installations en parallèle pendant un moment, vous devez leur affecter des répertoires d'installation différents au moment de la construction. (Ce problème est rectifié pour PostgreSQL 8.0 et ultérieurs mais vous devez faire bien attention à déplacer les anciennes installations.)

# Chapitre 23. Surveiller l'activité de la base de données

Un administrateur de bases de données se demande fréquemment : << Que fait le système en ce moment ? >> Ce chapitre discute de la façon de le savoir.

Plusieurs outils sont disponibles pour surveiller l'activité de la base de données et pour analyser les performances. Une grande partie de ce chapitre concerne la description du récupérateur de statistiques de PostgreSQL mais personne ne devrait négliger les programmes de surveillance Unix standards tels que `ps`, `top`, `iostat` et `vmstat`. De plus, une fois qu'une requête peu performante a été identifiée, des investigations supplémentaires pourraient être nécessaires en utilisant la commande ***EXPLAIN*** de PostgreSQL. La Section 13.1 discute de `EXPLAIN` et des autres méthodes pour comprendre le comportement d'une seule requête.

## 23.1. Outils Unix standard

Sur la plupart des plateformes, PostgreSQL modifie son titre de commande reporté par `ps` de façon à ce que les processus serveur individuels puissent être rapidement identifiés. Voici un affichage d'exemple :

```
$ ps auxww | grep ^postgres
postgres   960  0.0  1.1  6104 1480 pts/1    SN   13:17   0:00 postmaster -i
postgres   963  0.0  1.1  7084 1472 pts/1    SN   13:17   0:00 postgres: stats buffer process
postgres   965  0.0  1.1  6152 1512 pts/1    SN   13:17   0:00 postgres: stats collector process
postgres   998  0.0  2.3  6532 2992 pts/1    SN   13:18   0:00 postgres: tgl runbogue 127.0.0.1 i
postgres  1003  0.0  2.4  6532 3128 pts/1    SN   13:19   0:00 postgres: tgl regression [local] S
postgres  1016  0.1  2.4  6532 3080 pts/1    SN   13:19   0:00 postgres: tgl regression [local] i
```

(L'appel approprié de `ps` varie suivant les différentes plateformes, de même que les détails affichés. Cet exemple est tiré d'un système Linux récent.) Le premier processus affiché ici est le postmaster, le processus serveur maître. Les arguments affichés pour cette commande sont les mêmes qu'à son lancement. Les deux processus suivant implémentent la récupération de statistiques qui sera décrite en détail dans la section suivante (elles ne seront pas présentes si vous avez configuré le système pour qu'il ne lance pas le récupérateur de statistiques). Chacun des autres processus est un processus serveur gérant une connexion cliente. Tous ces processus restant initialisent l'affichage de la ligne de commande de la forme

`postgres:` *`utilisateur base_de_données hôte activité`*

L'utilisateur, la base de données et les éléments de l'hôte de connexion restent identiques pendant toute la vie de connexion du client mais l'indicateur d'activité change. L'activité pourrait être `idle` (c'est–à–dire en attente d'une commande du client), `idle in transaction` (en attente du client à l'intérieur d'un bloc de `BEGIN`/`COMMIT`) ou un nom de commande du type `SELECT`. De plus, `waiting` est attaché si le processus serveur est en attente d'un verrou détenu par un autre processus serveur. Dans l'exemple ci–dessus, nous pouvons supposer que le processus 1003 attend que le processus 1016 ait terminé sa transaction et, du coup, libère un verrou.

> **Astuce :** Solaris requiert une gestion particulière. Vous devez utiliser `/usr/ucb/ps` plutôt que `/bin/ps`. Vous devez aussi utiliser deux options `w` et non pas seulement une. En plus, votre appel original de la commande `postmaster` doit avoir un affichage de statut dans `ps`

plus petit que celui fourni par les autres processus serveur. Si vous échouez dans les trois, l'affichage de `ps` pour chaque processus serveur sera la ligne de commande originale de `postmaster`.

# 23.2. Le récupérateur de statistiques

Le *récupérateur de statistiques* de PostgreSQL est un sous−système qui supporte la récupération et les rapports d'informations sur l'activité du serveur. Actuellement, le récupérateur peut compter les accès aux tables et index à la fois en terme de blocs disque et de lignes individuelles. Il supporte aussi la détermination de la commande exacte en cours d'exécution par les autres processus serveur.

## 23.2.1. Configuration de la récupération de statistiques

Comme la récupération de statistiques ajoute un temps supplémentaire à l'exécution de la requête, le système peut être configuré pour récupérer ou non des informations. Ceci est contrôlé par les paramètres de configuration qui sont normalement initialisés dans `postgresql.conf` (voir Section 16.4 pour plus de détails sur leur initialisation).

Le paramètre stats_start_collector doit valoir `true` pour que le récupérateur de statistiques soit seulement lancé. C'est la valeur par défaut et la configuration recommandée mais elle peut être désactivée si vous n'êtes pas intéressé par les statistiques et que vous souhaitez supprimer toute activité en trop (néanmoins, ce que vous sauverez sera assez restreint). Notez que cette option ne peut pas être changée alors que le serveur est en cours d'exécution.

Les paramètres stats_command_string, stats_block_level et stats_row_level contrôlent la quantité d'informations réellement envoyée au récupérateur et détermine du coup le temps supplémentaire réclamé. Ils déterminent respectivement si un processus serveur envoie sa chaîne de commande actuelle, les statistiques d'accès au niveau du bloc disque et celles au niveau de la ligne vers le récupérateur. Normalement, ces paramètres sont configurés dans `postgresql.conf` de façon à ce qu'ils s'appliquent à tous les processus serveur mais il est possible de les activer/désactiver sur des sessions individuelles en utilisant la commande *SET* (pour empêcher les utilisateurs ordinaires de cacher leur activité à l'administrateur, seuls les superutilisateurs sont autorisés à modifier ces paramètres avec `SET`).

> **Note :** Comme les paramètres `stats_command_string`, `stats_block_level` et `stats_row_level` valent par défaut `false`, très peu de statistiques sont récupérées dans la configuration par défaut. Activer une ou plus des variables de configuration améliorera significativement le nombre de données utiles produit par le récupérateur de statistiques au prix d'une surcharge supplémentaire à l'exécution.

## 23.2.2. Visualiser les statistiques récupérées

Plusieurs vues prédéfinies, listées dans le Tableau 23−1, sont disponibles pour afficher les résultats de la récupération de statistiques. Autrement, vous pouvez construire des vues personnalisées en utilisant les fonctions statistiques existantes.

En utilisant les statistiques pour surveiller l'activité en cours, il est important de réaliser que l'information n'est pas mise à jour instantanément. Chaque processus serveur individuel transmet le nouveau nombre d'accès au niveau des blocs et des lignes au récupérateur juste avant l'attente d'une nouvelle commande du client ; donc

une requête toujours en cours n'affecte pas les totaux affichés. De plus, le récupérateur lui−même émet un nouveau rapport une fois par `pgstat_stat_interval` millisecondes (500 par défaut). Donc, les totaux affichés sont bien derrière l'activité réelle. L'information sur la requête en cours est rapportée immédiatement par le récupérateur mais est toujours sujet au délai `pgstat_stat_interval` avant de devenir visible.

Un autre point important est que, lorsqu'un processus serveur se voit demander d'afficher une des statistiques, il récupère tout d'abord le rapport le plus récent émis pas le processus de récupération, puis continue d'utiliser cette image de toutes les vues et fonctions statistiques jusqu'à la fin de sa transaction en cours. Donc, les statistiques ne sembleront pas changer tant que vous restez dans la même transaction. Ceci est une fonctionnalité, et non pas un bogue, car il vous permet de traiter plusieurs requêtes sur les statistiques et de corréler les résultats sans vous inquiéter que les nombres aient pu changer. Mais si vous voulez voir les nouveaux résultats pour chaque requête, assurez−vous de lancer les requêtes en dehors de tout bloc de transaction.

**Tableau 23−1. Vues statistiques standards**

| Nom de la vue | Description |
|---|---|
| `pg_stat_activity` | Une ligne par processus serveur, affichant l'ID du processus, la base de données, l'utilisateur, la requête en cours et le temps où a commencé l'exécution de la requête. Les colonnes renvoyant des données sur la requête en cours sont seulement disponibles si le paramètre `stats_command_string` a été activé. De plus, ces colonnes se lisent comme NULL sauf si l'utilisateur examinant cette vue est un superutilisateur ou est le propriétaire du processus en cours de rapport. (Notez qu'à cause des délais du récupérateur, la requête en cours ne sera mise à jour que pour les requêtes de longue durée.) |
| `pg_stat_database` | Une ligne par base de données, affichant le nombre de processus serveur actifs, le nombre total de transactions validées et le nombre de celles qui ont été annulées, le nombre total de blocs disque lus et le nombre total de succès du tampon (c'est−à−dire le nombre de lectures de blocs évitées en trouvant déjà le bloc dans le cache du tampon). |
| `pg_stat_all_tables` | Pour chaque table dans la base de données en cours, les nombres totaux de parcours séquentiels et indexés, le nombre total de lignes renvoyés par chaque type de parcours et le nombre total d'insertions, de mises à jour et de suppressions de lignes. |
| `pg_stat_sys_tables` | Identique à `pg_stat_all_tables`, sauf que seules les tables systèmes sont affichées. |
| `pg_stat_user_tables` | Identique à `pg_stat_all_tables`, sauf que seules les tables utilisateurs sont affichées. |
| `pg_stat_all_indexes` | Pour chaque index dans la base de données en cours, le nombre total de parcours d'index qui ont utilisé cet index, le nombre de lignes d'index lus et le nombre de lignes d'en−tête récupérées avec succès. (Ceci pourrait être moindre quand les entrées d'index pointent vers des lignes supprimées.) |
| `pg_stat_sys_indexes` | Identique à `pg_stat_all_indexes`, sauf que seules les tables systèmes sont affichées. |
| `pg_stat_user_indexes` | Identique à `pg_stat_all_indexes`, sauf que seules les tables utilisateurs sont affichées. |

| | |
|---|---|
| pg_statio_all_tables | Pour chaque table de la base de données en cours, le nombre total de lecture de blocs disques à partir de cette table, le nombre de récupérations valides par les tampons, le nombre de lectures de blocs disque et le nombre de récupérations valides par les tampons dans tous les index de cette table, le nombre de lectures de blocs disque et le nombre de récupérations valides par les tampons dans tous les index à partir de la table auxiliaire TOAST (si elle existe) pour cette table, le nombre de lectures de blocs disque et le nombre de récupérations valides par les tampons pour l'index de la table TOAST. |
| pg_statio_sys_tables | Identique à pg_statio_all_tables, sauf que seules les tables systèmes sont affichées. |
| pg_statio_user_tables | Identique à pg_statio_all_tables, sauf que seules les tables systèmes sont affichées. |
| pg_statio_all_indexes | Pour chaque index de la base de données en cours, les nombres de lectures de blocs disque et de récupérations valides dans cet index. |
| pg_statio_sys_indexes | Identique à pg_statio_all_indexes, sauf que seuls les index systèmes sont affichés. |
| pg_statio_user_indexes | Identique à pg_statio_all_indexes, sauf que seuls les index systèmes sont affichés. |
| pg_statio_all_sequences | Pour chaque objet séquence de la base de données en cours, le nombres de lectures de blocs disque et de récupérations valides de tampons dans cette séquence. |
| pg_statio_sys_sequences | Identique à pg_statio_all_sequences, sauf que seules les séquences système sont affichées (actuellement, aucune séquence système n'est définie, donc cette vue est toujours vide) |
| pg_statio_user_sequences | Identique à pg_statio_all_sequences, sauf que seules les séquences utilisateur sont affichées. |

Les statistiques par index sont particulièrement utiles pour déterminer les index utilisés et leur efficacité.

Les vues pg_statio_ sont principalement utiles pour déterminer l'efficacité du cache tampon. Quand le nombre de lectures disques réelles est plus petit que le nombre de récupérations valides par le tampon, alors le cache satisfait la plupart des demandes de lecture sans faire appel au noyau. Néanmoins, ces statistiques ne nous donnent pas l'histoire complète : à cause de la façon dont PostgreSQL gère les entrées/sorties disque, les données qui ne sont pas dans le tampon de PostgreSQL pourraient toujours résider dans le tampon d'entrées/sorties du noyau et pourraient, du coup, être toujours récupérées sans nécessiter une lecture physique. Les utilisateurs intéressés pour obtenir des informations plus détaillées sur le comportement des entrées/sorties dans PostgreSQL sont invités à utiliser le récupérateur de statistiques de PostgreSQL avec les outils du système d'exploitation permettant une vue de la gestion des entrées/sorties par le noyau.

Il existe d'autres façons de regarder les statistiques. Cela se fait en écrivant des requêtes qui utilisent les mêmes fonctions d'accès aux statistiques que les vues standards. Ces fonctions sont listées dans le Tableau 23–2. Les fonctions d'accès par base de données prennent un OID de la base de données comme argument pour identifier la base de données du rapport. Les fonctions par table et par index prennent l'OID de la table ou de l'index (notez que seuls les tables et les index de la base de données en cours peuvent être vus par ces fonctions). Les fonctions d'accès au processus prennent le numéro d'identifiant du processus.

**Tableau 23–2. Fonctions d'accès aux statistiques**

| Fonction | Code de retour | Description |
|---|---|---|
| pg_stat_get_db_numbackends(oid) | integer | Nombre de processus actifs pour la base de données |
| pg_stat_get_db_xact_commit(oid) | bigint | Transactions validées dans la base de données |
| pg_stat_get_db_xact_rollback(oid) | bigint | Transactions annulées dans la base de données |
| pg_stat_get_db_blocks_fetched(oid) | bigint | Nombre de demandes de récupérations de blocs disque pour la base de données |
| pg_stat_get_db_blocks_hit(oid) | bigint | Nombre de demandes de récupérations de blocs disque trouvés dans le tampon pour la base de données |
| pg_stat_get_numscans(oid) | bigint | Nombre de parcours séquentiels réalisés lorsque l'argument est une table, ou nombre de parcours d'index lorsque l'argument est un index |
| pg_stat_get_tuples_returned(oid) | bigint | Nombre de lignes lues par les parcours séquentiels lorsque l'argument est une table, ou nombre de lignes d'index lues lorsque l'argument est un index |
| pg_stat_get_tuples_fetched(oid) | bigint | Nombre de lignes récupérées valides (non expirées) récupérées par des parcours séquentiels quand l'argument est une table, ou récupérées par des parcours d'index en utilisant cet index lorsque l'argument est un index |
| pg_stat_get_tuples_inserted(oid) | bigint | Nombre de lignes insérées dans la table |
| pg_stat_get_tuples_updated(oid) | bigint | Nombre de lignes mises à jour dans la table |
| pg_stat_get_tuples_deleted(oid) | bigint | Nombre de lignes supprimées dans la table |
| pg_stat_get_blocks_fetched(oid) | bigint | Nombre de demandes de récupération de blocs disques pour la table ou l'index |
| pg_stat_get_blocks_hit(oid) | bigint | Nombre de demandes de blocs disque récupérés dans le tampon pour la table ou l'index |
| pg_stat_get_backend_idset() | set of integer | Ensemble d'identifiants de processus actifs (de 1 au nombre de processus actifs). Voir l'exemple d'utilisation dans le |

| | | |
|---|---|---|
| | | texte. |
| pg_backend_pid() | integer | ID du processus pour le processus serveur attaché à la session en cours |
| pg_stat_get_backend_pid(integer) | integer | ID du processus pour le processus serveur donné |
| pg_stat_get_backend_dbid(integer) | oid | ID de la base de données pour le processus serveur en cours |
| pg_stat_get_backend_userid(integer) | oid | ID de l'utilisateur pour le processus serveur en cours |
| pg_stat_get_backend_activity(integer) | text | Commande active du processus serveur en cours (NULL si l'utilisateur courant n'est pas un superutilisateur ni le même utilisateur que celui de la session en cours de requêtage, ou si stats_command_string n'est pas activée) |
| pg_stat_get_backend_activity_start(integer) | timestamp with time zone | Le moment où la requête en cours de traitement a été lancée (NULL si l'utilisateur courant n'est pas un superutilisateur ou s'il ne s'agit pas du même utilisateur que celui de la session qui a lancé la requête ou si stats_command_string n'est pas actif) |
| pg_stat_reset() | boolean | Réinitialise toutes les statistiques en cours |

**Note :** pg_stat_get_db_blocks_fetched moins pg_stat_get_db_blocks_hit donne le nombre d'appels lancés pour la table, l'index ou la base de données ; mais le nombre réel de lectures physiques est habituellement moindre à cause des tampons du noyau.

La fonction pg_stat_get_backend_idset fournit un moyen agréable de générer une ligne pour chaque processus serveur actif. Par exemple, pour afficher les PID et les requêtes en cours pour tous les processus serveur :

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

# 23.3. Visualiser les verrous

Un autre outil utile pour surveiller l'activité des bases de données est la table système pg_locks. Elle permet à l'administrateur système de visualiser des informations sur les verrous restant dans le gestionnaire des verrous. Par exemple, cette fonctionnalité peut être utilisée pour :

- Visualiser tous les verrous en cours, tous les verrous sur les relations d'une base de données particulière ou tous les verrous détenus par une session PostgreSQL particulière.
- Déterminer la relation de la base de données disposant de la plupart des verrous non autorisés (et qui, du coup, pourraient être une source de contention parmi les clients de la base de données).
- Déterminer l'effet de la contention des verrous sur les performances générales des bases de données, ainsi que l'échelle dans laquelle varie la contention sur le trafic de la base de données.

Les détails sur la vue `pg_locks` apparaissent dans la Section 41.33. Pour plus d'informations sur les verrous et la gestion des concurrences avec PostgreSQL, référez−vous au Chapitre 12.

# Chapitre 24. Surveillance de l'utilisation de l'espace disque

Ce chapitre traite de la manière d'effectuer une surveillance sur l'utilisation de l'espace disque par un système de bases de données PostgreSQL.

## 24.1. Déterminer l'utilisation de l'espace disque

Chaque table possède un fichier d'en−tête principal dans lequel la plupart des données sont stockées. Si la table comprend des colonnes avec des valeurs potentiellement larges, il existe aussi un fichier TOAST associé à la table, qui est utilisé pour stocker les valeurs trop larges pour tenir confortablement dans la table principale (voir Section 49.2). Il y aura un index sur la table TOAST, si elle existe. Il pourrait aussi y avoir des index associés à la table de base. Chaque table et index sont stockés dans un fichier séparé — parfois plusieurs fichiers si ce fichier devait dépasser la taille d'1 Go. Les conventions de nommage de ces fichiers sont décrites dans Section 49.1.

Vous pouvez surveiller l'espace disque en utilisant trois méthodes : depuis psql en utilisant les résultats de la commande `VACUUM`, depuis psql en utilisant l'outil se trouvant dans `contrib/dbsize`, et depuis la ligne de commande en utilisant l'outil contenu dans `contrib/oid2name`. En utilisant psql sur une base de données sur laquelle ont été lancés `VACUUM` ou `ANALYZE`, vous pouvez lancer des requêtes vous permettant d'obtenir des informations sur la place occupée par n'importe quelle table :

```
SELECT relfilenode, relpages FROM pg_class WHERE relname = 'customer';

 relfilenode | relpages
-------------+----------
       16806 |       60
(1 ligne)
```

Chaque page utilise en général 8 Ko de d'espace disque (ne pas oublier que `relpage` n'est mis à jour que par l'utilisation des commandes `VACUUM` et `ANALYZE` et par quelques commandes DDL comme `CREATE INDEX`.) La valeur de `relfilenode` est d'un grand intérêt si vous voulez examiner le fichier de la table directement.

Afin de connaître l'espace disque utilisé par les tables TOAST, lancez une requête similaire à la suivante :

```
SELECT relname, relpages
    FROM pg_class,
         (SELECT reltoastrelid FROM pg_class
          WHERE relname = 'customer') ss
    WHERE oid = ss.reltoastrelid
       OR oid = (SELECT reltoastidxid FROM pg_class
                 WHERE oid = ss.reltoastrelid)
    ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

Vous pouvez afficher également la taille des index :

```
SELECT c2.relname, c2.relpages
    FROM pg_class c, pg_class c2, pg_index i
    WHERE c.relname = 'customer'
        AND c.oid = i.indrelid
        AND c2.oid = i.indexrelid
        ORDER BY c2.relname;

       relname        | relpages
----------------------+----------
 customer_id_indexdex |       26
```

Vous pouvez trouver les tables les plus grosses en utilisant la requête suivante :

```
SELECT relname, relpages FROM pg_class ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

`contrib/dbsize` crée dans votre base de données des fonctions permettant de trouver la taille des tables ou de la base de données à partir de psql sans avoir besoin d'exécuter `VACUUM` ou `ANALYZE` préalablement.

Vous pouvez aussi utiliser l'utilitaire `contrib/oid2name` afin d'afficher l'utilisation du disque. Lisez les exemples contenus dans le fichier `README.oid2name` dans ce même répertoire. Il inclut un script permettant d'afficher la quantité de disque utilisée par chaque base de données.

## 24.2. Échec sur disque plein

Concernant la surveillance de l'espace disque, la tâche la plus importante d'un administrateur de base de données est de s'assurer que le disque ne soit pas complètement rempli. Un disque de données rempli ne corrompra pas les données mais il pourrait empêcher bien des activités utiles. S'il s'agit du disque contenant les fichier WAL, le serveur de la base de données fera un PANIC et un arrêt pourrait survenir.

Si vous ne pouvez pas libérer un espace supplémentaire sur le disque en supprimant d'autres choses, vous pouvez déplacer quelques fichiers de la base de données vers d'autres systèmes de fichiers en utilisant les espaces logiques. Voir Section 18.6 pour plus d'informations là–dessus.

> **Astuce :** Certains systèmes de fichiers réagissent mals lorsqu'ils sont presque pleins. À toutes fins utiles, n'attendez pas que le disque soit complètement plein pour agir.

Si votre système supporte les quotas disques par utilisateur, alors la base de données sera naturellement sujette au quota placé sur l'utilisateur qui exécute le serveur de base de données. Dépasser le quota aura les mêmes mauvais effets que d'avoir un disque complètement rempli.

# Chapitre 25. Write−Ahead Logging (WAL)

*Write−Ahead Logging* (WAL) est une approche conventionnelle pour l'écriture d'un journal de transactions. Sa description détaillée peut être trouvée dans la plupart (si ce n'est tous) des livres sur le traitement transactionnel. Brièvement, le concept central des WAL est d'effectuer les changements des fichiers de données (où résident les tables et les index) uniquement après que ces changements ont été écrits dans un journal, c'est−à−dire quand l'enregistrement du journal décrivant les changements a été écrit vers le stockage permanent. Si nous suivons cette procédure, nous n'avons pas besoin d'écrire les pages de données vers le disque à chaque validation de transaction car nous savons que, dans l'éventualité d'une défaillance, nous serons capables de récupérer la base de données en utilisant le journal : chaque changement qui n'a pas été appliqué aux pages de données peut être ré−exécuté depuis les enregistrements du journal (ceci est une récupération roll−forward, aussi connue sous le nom de REDO).

## 25.1. Avantages des WAL

Le premier avantage majeur en utilisant les WAL est la réduction significative du nombre d'écritures sur le disque puisque seul le journal des transactions a besoin d'être écrit sur le disque au moment où la transaction est validée plutôt que d'écrire dans chaque fichier de données modifié par la transaction. Dans un environnement multi−utilisateurs, la validation de nombreuses transactions peut être accomplie avec un seul `fsync()` du journal. De plus, ce dernier est écrit séquentiellement et donc, le coût de synchronisation du journal est largement moindre que le coût d'écriture des pages de données. Ceci est spécialement vrai pour les serveurs gérant beaucoup de petites transactions touchant différentes parties du stockage de données.

Le second avantage est la cohérence des pages de données. La vérité est qu'avant WAL, PostgreSQL n'était pas capable de garantir la cohérence en cas de défaillance (crash). Avant WAL, une défaillance quelconque durant l'écriture pouvait être le résultat :

1. de lignes d'un index pointant vers des lignes inexistantes d'une table ;
2. de lignes perdues d'un index dans des opérations de séparation ;
3. de tables ou de contenu de pages d'index totalement corrompues à cause d'une écriture partielle des pages de données

Les problèmes avec les index (les problèmes 1 et 2) auraient pû être corrigés avec des appels supplémentaires à `fsync` mais il n'est pas évident de traiter le dernier cas sans WAL. WAL sauve le contenu entier de la page de données dans le journal si cela est requis pour assurer la cohérence de la page et permettre ainsi la récupération après une défaillance.

Enfin, les WAL rendent possible le support de sauvegarde en ligne et de récupération à un moment, comme décrit dans Section 22.3. En archivant les données WAL, nous pouvons supporter le retour à tout instant couvert par les données disponibles dans les WAL : nous installons simplement une ancienne sauvegarde physique de la base de données et nous rejouons les journaux WAL jusqu'au moment désiré. Qui plus est, la sauvegarde physique n'a pas besoin d'être une image instantanée de l'état de la base de données — si elle a été faire pendant une période de temps, alors rejouer les journaux WAL pour cette période corrigera toute inconsistance interne.

# 25.2. Configuration de WAL

Il y a plusieurs paramètres de configuration associés à WAL qui affectent les performances de la base de de données. Cette section explique leur utilisation. Consultez le Section 16.4 pour des détails sur la mise en place de ces paramètres de configuration.

Dans la séquence des transactions, les *points de contrôles* (checkpoints) sont des points qui garantissent que les fichiers de données ont été mis à jour avec toutes les informations enregistrées dans le journal avant le point de contrôle. Au moment du point de contrôle, toutes les pages de données non propres sont écrites sur le disque et une entrée spéciale, pour le point de contrôle, est écrite dans le journal. Cela a pour résultat qu'en cas de défaillance, la procédure de récupération sait à partir de quelle entrée du journal (connu sous le nom d'entrée de récupération) il doit démarrer l'opération de de récupération (REDO) puisque tous les changements faits sur les fichiers de données avant cette entrée sont déjà sur le disque. Après avoir effectué un point de contrôle, tous les segments de journaux écrits avant l'entrée de récupération ne sont plus nécessaires et peuvent être recyclés ou supprimés. (Quand la sauvegarde des WAL est en cours, les segments de journaux doivent être archivés avant d'être recyclés ou supprimés).

Le processus d'écriture en tâche de fond lancera automatiquement un point de contrôle de temps en temps. Un point de contrôle est créé tous les checkpoint_segments segments de journaux ou dès que checkpoint_timeout secondes se sont écoulées. Les paramètres par défaut sont respectivement 3 segments et 300 secondes. Il est également possible de forcer la création d'un point de contrôle en utilisant la commande SQL `CHECKPOINT`.

Réduire `checkpoint_segments` et/ou `checkpoint_timeout` a pour conséquence de faire des points de contrôle plus fréquent. Ceci permet une récupération plus rapide après une défaillance (puisque moins de travail a besoin d'être récupéré). Cependant, il faut équilibrer cela avec l'augmentation du coût d'écriture des pages de données non propres. De plus, après chaque point de contrôle, pour être certain de l'intégrité des pages de données, la première modification d'une page de données a pour conséquence d'écrire dans le journal le contenu entier de la page. Donc, un intervalle trop petit de points de contrôles augmente le volume écrit dans le journal WAL, ce qui annule, en partie, les avantages d'utiliser un petit intervalle. Dans tous les cas, cela causera plus d'accès en entrées/sorties (I/O) du disque.

Les points de contrôle sont assez coûteux, tout d'abord parce qu'ils écrivent tous les tampons utilisés, et ensuite parce que cela suscite un trafic WAL supplémentaire comme indiqué ci–dessus. Du coup, il est conseillé de configurer les paramètres en relation assez haut pour que ces points de contrôle n'arrivent pas trop fréquemment. En tant que simple vérification de santé de vos paramètres, vous pouvez configurer le paramètre checkpoint_warning. Si les points de contrôle arrivent plus rapidement que `checkpoint_warning` secondes, un message sera affiché dans les journaux du serveur, recommandant d'accroître `checkpoint_segments`. Une apparition occasionnelle d'un message ne doit pas vous alarmer mais, s'il apparaît souvent, alors les paramètres de contrôle devraient être augmentés.

Il y aura au moins un fichier segment WAL et normalement pas plus de 2 * `checkpoint_segments` + 1 fichiers. Chaque fichier de segment fait normalement 16 Mo (bien que cette taille puisse être modifiée lors de la compilation du serveur). Vous pouvez utiliser cela pour estimer l'espace disque nécessaire pour WAL. D'habitude, quand les vieux fichiers segment de journaux ne sont plus nécessaires, ils sont recyclés (renommés pour devenir les prochains segments dans une séquence numérotée). S'il y a plus de 2 * `checkpoint_segments` + 1 fichiers segments à cause d'un pic temporaire du taux d'écriture des journaux, ceux inutilisés seront effacés au lieu d'être recyclés jusqu'à ce que le système soit en–dessous de cette limite.

Il y a deux fonctions WAL couramment utilisées : `LogInsert` et `LogFlush`. `LogInsert` est utilisée pour placer une nouvelle entrée à l'intérieur des tampons WAL en mémoire partagée. S'il n'y a plus d'espace pour

une nouvelle entrée, LogInsert devra écrire (bouger dans le cache du noyau) quelques tampons WAL remplis. Ceci n'est pas désirable parce que LogInsert est utilisée lors de chaque modification bas niveau de la base (par exemple, insertion d'une ligne) quand un verrou exclusif est posé sur des pages de données affectées, donc l'opération nécessite d'être aussi rapide que possible. Pire encore, écrire des tampons WAL peut aussi forcer la création d'un nouveau segment de journal ce qui peut prendre beaucoup plus de temps. Normalement, les tampons WAL devraient être écrits et vidés par une requête de LogFlush qui est faite, la plupart du temps, au moment de la validation d'une transaction pour assurer que les entrées de la transaction sont écrites vers un stockage permanent. Sur les systèmes avec une importante écriture de journaux, les requêtes de LogFlush peuvent ne pas arriver assez souvent pour empêcher LogInsert d'avoir à écrire. Sur de tel système, on devrait augmenter le nombre de tampons WAL en modifiant le paramètre de configuration wal_buffers. Par défaut, le nombre de tampons est de 8. Augmenter cette valeur augmentera considérablement l'utilisation de la mémoire partagée. (Il devrait être noté qu'il existe actuellement peu de preuves pour suggérer que l'augmentation de wal_buffers au−delà de la valeur par défaut est intéressant.)

Le paramètre commit_delay définit combien de micro−secondes le processus serveur dormira après l'écriture d'une entrée de validation dans le journal avec LogInsert avant d'exécuter un LogFlush. Ce délai permet aux autres processus du serveur d'ajouter leurs entrées de validation dans le fichier de journal afin de tout écrire vers le disque avec une seule synchronisation du journal. Aucune mise en sommeil n'aura lieu si fsync n'est pas disponible ou si moins de commit_siblings autres sessions sont, à ce moment, dans des transactions actives ; cela évite de dormir quand il est improbable qu'une autre session fasse bientôt une validation. Notez que dans la plupart des plate−formes, la résolution d'une requête de sommeil est de 10 millisecondes, donc un commit_delay différent de zéro et configuré entre 1 et 10000 micro−secondes aura le même effet. Les bonnes valeurs pour ce paramètre ne sont pas encore claires ; les essais sont encouragés.

Le paramètre wal_sync_method détermine comment PostgreSQL demandera au noyau de forcer les mises à jour WAL sur le disque. Toutes les options devraient être les mêmes dans la mesure où la fiabilité ne disparaît pas, mais c'est avec des options spécifiques à la plate−forme que ça sera le plus rapide. Notez que ce paramètre est ignoré si fsync a été désactivé.

Configurer le paramètre wal_debug avec une valeur différente de zéro aura pour résultat d'enregistrer dans les journaux du serveur l'appel WAL à chaque LogInsert et LogFlush. En ce moment, il n'est fait aucune différence entre les valeurs supérieures à zéro. Cette option pourra être remplacée par un mécanisme plus général dans le futur.

Activer le paramètre de configuration wal_debug (à supposer que PostgreSQL ait été compilé avec le support de ce paramètre) résultera dans l'enregistrement de chaque appel WAL à LogInsert et LogFlush dans les journaux du serveur. Cette option pourrait être remplacée par un mécanisme plus général dans le futur.

## 25.3. Internes

WAL est automatiquement disponible ; aucune action n'est requise de la part de l'administrateur excepté de s'assurer que l'espace disque requis par les journaux WAL soit présent et que tous les réglages soient faits (regardez la Section 25.2).

Les journaux WAL sont stockés dans le répertoire pg_xlog sous le répertoire de données, comme un ensemble de fichiers segments, chacun d'une taille de 16 Mo généralement. Chaque segment est divisé en pages de généralement 8 Ko. Les en−têtes de l'entrée du journal sont décrites dans access/xlog.h ; le contenu de l'entrée dépend du type de l'événement qui est enregistré. Les fichiers segments sont nommés avec un chiffre qui est toujours incrémenté et qui commence à 000000010000000000000000. Les nombres ne bouclent pas actuellement, mais cela devrait prendre beaucoup de temps pour épuiser le stock de nombres

disponibles.

Les tampons WAL et la structure de contrôle sont situés dans la mémoire partagée et sont manipulés par les processus enfants du serveur. Ils sont protégés par des verrous légers. La demande en mémoire partagée est dépendante du nombre de tampons. La taille par défaut des tampons WAL est de 8 tampons de 8 Ko chacun, soit 64 Ko au total.

Il est avantageux que le journal soit situé sur un autre disque que celui des fichiers principaux de la base de données. Cela peut se réaliser en déplaçant le répertoire `pg_xlog` vers un autre emplacement (alors que le serveur est arrêté, naturellement) et en créant un lien symbolique de l'endroit d'origine dans le répertoire principal de données au nouvel emplacement.

Le but de WAL, s'assurer que le journal est écrit avant l'altération des entrées dans la base, peut être mis en échec par les lecteurs des disques qui faussement rapportent une écriture réussie au noyau quand, en fait, ils ont seulement mis en cache les données et ne les ont pas encore stockées sur le disque. Une coupure de courant dans ce genre de situation peut toujours mener à la corruption irrécupérable des données. Les administrateurs devraient s'assurer que les disques contenant les journaux WAL de PostgreSQL ne produisent pas ce genre de faux rapports.

Après qu'un point de contrôle ait été fait et que le journal ait été écrit, la position du point de contrôle est sauvegardée dans le fichier `pg_control`. Donc, quand la restauration doit être faite, le serveur lit en premier `pg_control` et ensuite l'entrée du point de contrôle ; ensuite, il exécute l'opération REDO en parcourant vers l'avant à partir de la position du journal indiquée dans l'entrée du point de contrôle. Parce que l'ensemble du contenu des pages de données est sauvegardé dans le journal à la première modification de page après un point de contrôle, toutes les pages changées depuis le point de contrôle seront restaurées dans un état cohérent.

Pour gérer le cas où `pg_control` est corrompu, nous devons supporter la possibilité de parcourir des segments de journaux existants en ordre inverse — de plus récent au plus ancien — pour trouver le dernier point de vérification. Ceci n'a pas encore été implémenté. `pg_control` est assez petit (moins d'une page disque) pour ne pas être sujet aux problèmes d'écriture partielle et, au moment où ceci est écrit, il n'y a eu aucun rapport d'échecs de la base de données uniquement à cause de son incapacité à lire `pg_control`. Donc, bien que cela soit théoriquement un point faible, `pg_control` ne semble pas être un problème en pratique.

# Chapitre 26. Tests de régression

Les tests de régression composent un ensemble exhaustif de tests pour l'implémentation SQL dans PostgreSQL. Ils testent les opérations SQL standards ainsi que les fonctionnalités étendues de PostgreSQL.

## 26.1. Lancer les tests

Les tests de régression peuvent être lancés sur un serveur déjà installé et fonctionnel ou en utilisant une installation temporaire à l'intérieur du répertoire de construction. De plus, ils peuvent être lancés en mode << parallèle >> ou en mode << séquentiel >>. Le mode séquentiel lance les scripts de test en série, tandis que le mode parallèle lance plusieurs processus serveurs pour parallèliser l'exécution des groupes de tests. Les tests parallèles permettent de s'assurer du bon fonctionnement des communications interprocessus et du verrouillage. Pour des raisons historiques, les tests séquentiels sont habituellement lancés sur une installation existante et la méthode parallèle préférentiellement sur une installation temporaire, mais il n'y a aucune raison technique à cela.

Pour lancer les tests de régression après la construction mais avant l'installation, il suffit de saisir

```
gmake check
```

dans le répertoire de premier niveau (on peut aussi se placer dans le répertoire `src/test/regress` et y lancer la commande). En premier lieu seront construits différents fichiers auxiliaires, tels des exemples de fonctions de déclencheurs utilisateur, puis le script de pilotage des tests sera exécuté. Au final, la sortie devrait ressembler à quelque chose comme

```
======================
 All 96 tests passed.
======================
```

ou une note indiquant l'échec des tests. Voir la Section 26.2 avant de supposer qu'un << échec >> représente un problème sérieux.

Comme cette méthode de tests fonctionne sur un serveur temporaire, elle ne fonctionnera pas en tant qu'utilisateur root (car le serveur refusera de se lancer en tant qu'utilisateur root) Si vous avez lancé la construction en tant que root, vous n'avez pas besoin de tout recommencer. À la place, rendez le répertoire des tests de régression modifiable par un autre utilisateur, devenez cet utilisateur et relancez les tests. Par exemple

```
root# chmod -R a+w src/test/regress
root# chmod -R a+w contrib/spi
root# su - joeuser
joeuser$ cd répertoire
construction haut niveau
joeuser$ gmake check
```

(le seul << risque en terme de sécurité >> est que les autres utilisateurs pourraient modifier les résultats des tests de régression dans votre dos. Utilisez le bon sens pour gérer les droits des utilisateurs.)

Autrement, lancez les tests après l'installation.

Si vous avez configuré PostgreSQL pour qu'il s'installe dans un emplacement où existe déjà une ancienne installation de PostgreSQL et que vous lancez `gmake check` avant d'installer la nouvelle version, vous pourriez trouver que les tests échouent parce que les nouveaux programmes essaient d'utiliser les bibliothèques partagées déjà installées (les symptômes typiques sont des plaintes concernant des symboles non définis). Si vous souhaitez lancer les tests avant d'écraser l'ancienne installation, vous devrez construire avec `configure --disable-rpath`. Néanmoins, il n'est pas recommandé d'utiliser cette option pour l'installation finale.

Les tests de régression en parallèle lancent quelques processus avec votre utilisateur. Actuellement, le nombre maximum est de vingt scripts de tests en parallèle, ce qui signifie 60 processus : il existe un processus serveur, un psql et habituellement un processus parent pour le psql de chaque script de tests. Si votre système force une limite par utilisateur sur le nombre de processus, assurez−vous que cette limite est d'au moins 65, sinon vous pourriez obtenir des échecs hasardeux dans les tests en parallèle. Si vous ne pouvez pas augmenter cette limite, vous pouvez diminuer le degré de parallélisme en initialisant le paramètre `MAX_CONNECTIONS`. Par exemple,

```
gmake MAX_CONNECTIONS=10 check
```

ne lance pas plus de dix tests en même temps.

Sur certains systèmes, le shell par défaut compatible Bourne (`/bin/sh`) a du mal à gérer autant de processus fils en parallèle. Cela pourrait causer des blocages ou des échecs dans les tests en parallèle. Dans de tels cas, spécifiez un shell compatible Bourne différent sur la liste de commande, par exemple :

```
gmake SHELL=/bin/ksh check
```

Si aucun shell ne le permet, vous pouvez contourner le problème en diminuant le nombre de connexions comme indiqué ci−dessus.

Pour lancer les tests après l'installation (voir le Chapitre 14), initialisez un espace de données et lancez le serveur comme expliqué dans le Chapitre 16, puis lancez

```
gmake installcheck
```

ou pour un test parallèle

```
gmake installcheck-parallel
```

Les tests s'attendront à contacter le serveur sur l'hôte local et avec le numéro de port par défaut, sauf en cas d'indication contraire avec les variables d'environnement `PGHOST` et `PGPORT`.

## 26.2. Évaluation des tests

Il arrive parfois qu'une instance de PostgreSQL, correctement installée et entièrement fonctionnelle, puisse << échouer >> à certains tests de régression, du fait de spécificités de la plateforme. Ces spécificités peuvent, par exemple, être la représentation des nombres à virgules flottantes ou le support des fuseaux horaires. Les tests sont, à ce jour, évalués en utilisant une simple comparaison `diff` avec les sorties engendrées sur un système de référence. Les résultats sont donc sensibles à de petites différences système. Lorsqu'un test est annoncé << échoué >>, il faudra examiner les différences entre les résultats attendus et obtenus ; il est probable que les différences ne soient pas significatives. Néanmoins, tout est fait pour maintenir des fichiers de références précises et actualisées pour toutes les plateformes supportées avec un soucis constant de réussite

des tests.

Les sorties actuelles des tests de régression sont stockées dans les fichiers du répertoire `src/test/regress/results`. Chaque fichier de sortie est comparé (`diff`) aux sorties de référence stockées dans le répertoire `src/test/regress/expected`. Les différences sont stockées dans le fichier `src/test/regress/regression.diffs`. La `diff` peut également être lancée par l'utilisateur.

## 26.2.1. Différences dans les messages d'erreurs

Certains des tests de régression impliquent des valeurs en entrée intentionnellement invalides. Les messages d'erreur peuvent provenir soit du code de PostgreSQL soit des routines système de la plateforme hôte. Dans ce dernier cas, les messages pourraient varier entre plateformes mais devraient toujours refléter des informations similaires. Ces différences dans les messages résulteront dans un test de régression << échoué >> qui pourrait être validé après vérification.

## 26.2.2. Différences au niveau des locales

Si vous lancez des tests sur un serveur déjà installé mais initialisé avec une locale autre que C, alors il pourrait y avoir des différences dans les ordres de tris. La suite de tests de régression est initialisée pour gérer ce problème en fournissant des fichiers de résultats alternatifs qui ensemble gèrent un grand nombre de locales. Par exemple, pour le test `char`, le fichier `char.out` attendu gère les locales `C` et `POSIX`, et le fichier `char_1.out` gère beaucoup d'autres locales. Le pilote des tests de régression choisira automatiquement le meilleur fichier lors de la vérification et pour la calcul des différences d'échecs. (Ceci signifie que les tests de régression ne peuvent pas détecter si les résultats sont appropriés pour la locale configurée. Ils vont simplement récupérer le fichier résultat qui fonctionne le mieux.)

Si, pour une quelconque raison, les fichiers attendus ne couvrent pas certaines locales, vous pouvez ajouter un nouveau fichier. Le schéma de nommage est *`nom_test_chiffre`*`.out`. Le chiffre n'a en fait pas de signification. Rappelez−vous que le pilote des tests de régression considérera tous les fichiers comme étant des résultats de tests valides. Si les résultats de tests sont spécifiques à une plateforme, la technique décrite dans Section 26.3 devrait être utilisée à la place.

## 26.2.3. Différences au niveau des dates/heures

Quelques requêtes des tests d'`horologie` échoueront si vous lancez les tests un jour de changement de heure ou le lendemain de ce jour. Ces requêtes s'attendent à ce que les intervalles entre minuit hier, minuit aujourd'hui et minuit demain soient exactement 24 heures −− ce qui est faux lorsqu'un changement d'heure intervient.

> **Note :** Comme les règles de changement d'heure des USA sont utilisées, ce problème arrive toujours le premier dimanche d'avril, le dernier dimanche d'octobre et les lundis suivants quelque soit le changement d'heure effectif où vous vivez. Notez aussi que le problème apparaît ou disparaît à minuit, UTC−7 ou UTC−8, pas à un minuit chez vous. Donc, l'échec pourrait arriver plus tard le dimanche ou persister jusqu'au mardi suivant l'endroit où vous vivez.

La plupart des résultats date/heure sont dépendants de l'environnement de zone horaire. Les fichiers de référence sont générés pour la zone horaire `PST8PDT` (Berkeley, Californie), et il y aura des échecs apparents si les tests ne sont pas lancés avec ce paramétrage de fuseau horaire. Le pilote des tests de régression initialise

la variable d'environnement PGTZ à PST8PDT ce qui nous assure normalement de bons résultats.

## 26.2.4. Différences sur les nombres à virgules flottantes

Quelques tests impliquent des calculs sur des nombres flottants à 64 bits (double precision) à partir de colonnes de tables. Des différences dans les résultats appliquant des fonctions mathématiques à des colonnes double precision ont été observées. Les tests de float8 et geometry sont particulièrement sensibles aux différences entre plateformes, voire aux différentes options d'optimisation des compilateurs. L'œil humain est nécessaire pour déterminer la véritable signification de ces différences, habituellement situées après la dixième décimale.

Certains systèmes affichent moins zéro comme -0 alors que d'autres affichent seulement 0.

Certains systèmes signalent les erreurs des fonctions pow() et exp() différemment du mécanisme attendu par le code de PostgreSQL.

## 26.2.5. Différences dans le tri des lignes

Des différences peuvent apparaître entre l'ordre d'affichage des lignes et celui du fichier de référence. Dans la plupart des cas, il ne s'agit pas vraiment d'un bogue. La plupart des scripts de tests de régression ne sont pas assez stricts pour utiliser un ORDER BY sur chaque SELECT. De ce fait, l'ordre des lignes n'est pas forcément défini suivant la spécification SQL exacte. En pratique, les mêmes requêtes étant exécutées sur les mêmes données avec le même logiciel, le même tri des résultats est généralement obtenu sur toutes les plateformes et le manque d'ORDER BY ne pose pas de problème. Il se peut toutefois que quelques requêtes affichent des différences de tri entre plateformes. (Ces différences peuvent aussi être la conséquence d'une locale différente de C.)

Ainsi, il n'y a pas lieu de s'inquiéter d'une différence de tri, sauf si la requête possède un ORDER BY que le résultat ne respecte pas. Il est alors intéressant de faire remonter cette information, afin qu'un ORDER BY soit ajouté à cette requête pour éliminer les faux << échecs >> dans les versions suivantes.

Si toutes les requêtes ne sont pas ordonnées, c'est simplement parce que cela rendrait les tests de régression moins utiles. En effet, ils tendraient à exercer des plans de requêtes produisant des résultats ordonnés excluant ainsi les autres types de plans.

## 26.2.6. Test << random >>

Le script de tests random a pour but de produire des résultats aléatoires. Dans de rares cas, ceci fait échouer random aux tests de régression. Saisir

```
diff results/random.out expected/random.out
```

ne devrait produire au plus que quelques lignes différentes. Cela est normal et ne devient préoccupant que si les tests random échouent en permanence lors de tests répétés

# 26.3. Fichiers de comparaison spécifiques à la plateforme

Comme certains des tests produisent de façon inhérente des résultats spécifiques à la plateforme, nous avons fourni un moyen de fournir des fichiers de comparaison de résultats spécifiques à la plateforme. Fréquemment, la même variation s'applique entre plusieurs plateformes ; plutôt que de fournir un fichier de comparaison séparé pour chaque plateforme, il existe un fichier de correspondance définissant les fichiers de comparaison à utiliser. Donc, pour éliminer les << échecs >> dûs à des tests bogués pour une plateforme particulière, vous devez choisir ou créer un fichier de résultat variant puis ajouter une ligne dans le fichier de correspondance, à savoir `src/test/regress/resultmap`.

Chaque ligne du fichier de correspondance est de la forme

```
nomtest/modeleplateform=fichiercomparaison
```

Le nom de tests est juste le nom du module de tests de régression particulier. Le modèle de plateforme est un modèle dans le style des outils Unix `expr` (c'est–à–dire une expression rationnelle avec une ancre implicite `^` au début). Il est testé avec le nom de plateforme affiche par `config.guess` suivi par `:gcc` ou `:cc`, suivant que vous utilisez un compilateur GNU ou le compilateur de base de votre système (sur les systèmes où il y a une différence). Le nom du fichier de comparaison est le nom du fichier de comparaison substitué.

Par exemple : certains systèmes interprètent les très petites valeurs en virgule flottante comme zéro, plutôt que de rapporter une erreur. Ceci fait quelques petites différences dans le test de régression `float8`. Du coup, nous fournissons un fichier de comparaison variable, `float8-small-is-zero.out`, qui inclut les résultats attendus sur ces systèmes. Pour faire taire les messages d'<< échec >> erronés sur les plateformes OpenBSD, `resultmap` inclut

```
float8/i.86-.*-openbsd=float8-small-is-zero
```

qui se déclenche sur toute machine où la sortie de `config.guess` correspond à `i.86-.*-openbsd`. D'autres lignes dans `resultmap` sélectionnent le fichier de comparaison variable pour les autres plateformes si c'est approprié.

# IV. Les interfaces clientes

Cette partie décrit les interfaces de programmation clients qui sont distribuées avec PostgreSQL. Chacun de ces chapitres peut être lu indépendamment. Notez qu'il y a beaucoup d'autres interfaces de programmation pour les programmes clients qui sont distribuées séparément et qui contiennent leur propre documentation. Les lecteurs de cette partie devraient être familiers avec l'utilisation des requêtes SQL pour manipuler et interroger une base (voir la Partie II) et bien sûr avec le langage de programmation que l'interface utilise.

**Table des matières**

# Chapitre 27. libpq – Bibliothèque C

libpq est l'interface de programmation pour les applications C avec PostgreSQL. libpq est un ensemble de fonctions permettant aux programmes clients d'envoyer des requêtes au serveur PostgreSQL et de recevoir les résultats de ces requêtes.

libpq est aussi le moteur sous-jacent de plusieurs autres interfaces de programmation de PostgreSQL, incluant ceux écrits pour C++, Perl, Python, Tcl et ECPG. Donc, certains aspects du comportement de libpq seront importants pour vous si vous utilisez un de ces paquetages. En particulier, Section 27.11, Section 27.12 et Section 27.13 décrivent le comportement qui verra l'utilisateur de toute application utilisant libpq.

Quelques petits programmes sont inclus à la fin de ce chapitre (Section 27.16) pour montrer comment écrire des programmes utilisant libpq. Il existe aussi quelques exemples complets d'applications libpq dans le répertoire `src/test/examples` venant avec la distribution des sources.

Les programmes clients utilisant libpq doivent inclure le fichier d'en-tête `libpq-fe.h` et doivent être lié avec la bibliothèque libpq.

## 27.1. Fonctions de contrôle de connexion à la base de données

Les fonctions suivantes concernent la réalisation d'une connexion avec un serveur PostgreSQL. Un programme peut avoir plusieurs connexions ouvertes sur des serveurs à un même moment. (Une raison de la faire est d'accéder à plusieurs bases de données.) Chaque connexion est représentée par un objet `PGconn`, obtenu avec la fonction `PQconnectdb` ou `PQsetdbLogin`. Notez que ces fonctions renverront toujours un pointeur d'objet non nul, sauf peut-être dans un cas de manque de mémoire pour l'allocation de l'objet `PGconn`. La fonction `PQstatus` doit être appelée pour vérifier si la connexion s'est bien effectuée avant de lancer des requêtes via l'objet de connexion.

`PQconnectdb`

Crée une nouvelle connexion au serveur de bases de données.

```
PGconn *PQconnectdb(const char *conninfo);
```

Cette fonction ouvre une nouvelle connexion à la base de données en utilisant les paramètres à partir de la chaîne `conninfo`. Contrairement à `PQsetdbLogin` ci-dessous, l'ensemble de paramètres peut être étendu sans changer la signature de la fonction, donc utiliser cette fonction (ou son analogue non bloquant, `PQconnectStart` et `PQconnectPoll`) est préféré pour la programmation de nouvelles applications.

La chaîne passée peut être vide pour utiliser tous les paramètres par défaut ou elle peut contenir un ou plusieurs paramétrages séparés par des espaces blancs. Chaque paramètre est de la forme `motclé = valeur`. Les espaces autour du signe égal sont optionnels. Pour écrire une valeur vide ou une valeur contenant des espaces, entourez-les de guillemets simples, c'est-à-dire `motclé = 'une valeur'`. Des guillemets simples et des antislashs à l'intérieur de la valeur peuvent être échappés avec un antislash, par exemple `\'` et `\\`.

Les mots clés actuellement reconnus sont :

`host`
: Nom de l'hôte sur lequel se connecter. S'il commence avec un slash, il spécifie une communication par domaine Unix plutôt qu'une communication TCP/IP ; la valeur est le nom du répertoire où le fichier socket est stocké. Par défaut, quand `hôte` n'est pas spécifié, il s'agit d'une communication par socket de domaine Unix dans `/tmp` (ou tout autre répertoire de socket spécifié lors de la construction de PostgreSQL). Sur les machines sans sockets de domaine Unix, la valeur par défaut est de se connecter à `localhost`.

`hostaddr`
: Adresse IP numérique de l'hôte de connexion. Elle devrait être au format d'adresse standard IPv4, c'est–à–dire `172.28.40.9`. Si votre machine supporte IPv6, vous pouvez aussi utiliser ces adresses. La communication TCP/IP est toujours utilisée lorsqu'une chaîne non vide est spécifiée pour ce paramètre.

  Utiliser `hostaddr` au lieu de `host` permet à l'application d'éviter une recherche de nom d'hôte, qui pourrait être importante pour les applications ayant des contraintes de temps. Néanmoins, l'authentification Kerberos requiert un nom d'hôte. Du coup, ce qui suit s'applique : si `host` est spécifié sans `hostaddr`, Une recherche de nom d'hôte a lieur. Si `hostaddr` est spécifié sans `host`, la valeur de `hostaddr` donne l'adresse distante. Lorsque Kerberos est utilisée, une recherche de nom inverse est effectuée pour obtenir le nom d'hôte pour Kerberos. Si à la fois `host` et `hostaddr` sont spécifiés, la valeur de `hostaddr` donne l'adresse distante ; la valeur de `host` est ignorée sauf si Kerberos est utilisé, auquel cas il s'agit de la valeur utilisé pour l'authentification Kerberos. (Notez que l'authentification a des chances d'échouer si libpq se voit donner un nom qui n'est pas le nom de la machine sur `hostaddr`.) De même, `host` plutôt que `hostaddr` est utilisé pour identifier la connexion dans `~/.pgpass` (voir Section 27.12).

  Sans un nom ou une adresse d'hôte, libpq se connectera en utilisant un socket local de domaine Unix. Sur des machines sans sockets de domaine Unix, il tentera une connexion sur `localhost`.

`port`
: Numéro de port pour la connexion au serveur ou extension du nom de fichier pour des connexions de domaine Unix.

`dbname`
: Le nom de la base de données. Par défaut, la même que le nom utilisateur.

`user`
: Nom de l'utilisateur PostgreSQL qui se connecte. Par défaut, il s'agit du nom de l'utilisateur ayant lancé l'application.

`password`
: Mot de passe à utiliser si le serveur demande une authentification par mot de passe.

`connect_timeout`
: Attente maximum pour une connexion, en secondes (saisie comme une chaîne d'entier décimaux). Zéro ou non spécifié signifie une attente indéfinie. Utiliser un décompte de moins de deux secondes n'est pas recommandé.

`options`
: Options en ligne de commande à envoyer au serveur.

`tty`
: Ignoré (auparavant, ceci indiquait où envoyer les traces de débogage du serveur).

`sslmode`
: Cette option détermine si ou avec quelle priorité une connexion SSL sera négocié avec le serveur. Il existe quatre modes : `disable` essaiera uniquement une connexion non cryptée SSL ; `allow` négociera, en essayant tout d'abord une connexion sans SSL puis, si cela

échoue, une connexion SSL ; prefer (la valeur par défaut) négociera en essayant d'abord une connexion SSL puis, en cas d'échec, une connexion non SSL ; require essaiera uniquement une connexion SSL.

Si PostgreSQL est compilé sans le support de SSL, l'utilisation de l'option require causera une erreur alors que les options allow et prefer seront acceptées mais libpq ne sera pas capable de négocier une connexion SSL.

requiressl
Cette option est obsolète et remplacée par l'option sslmode.

Si initialisée à 1, une connexion SSL au serveur est requise (ce qui est équivalent à un sslmode require). libpq refusera alors de se connecter si le serveur n'accepte pas une connexion SSL. Si initialisée à 0 (la valeur par défaut), libpq négociera le type de connexion avec le serveur (équivalent à un sslmode prefer). Cette option est seulement disponible si PostgreSQL est compilé avec le support SSL.

service
Nom du service à utiliser pour des paramètres supplémentaires. Il spécifie un nom de service dans pg_service.conf contenant des paramètres de connexion supplémentaires. Ceci permet aux applications de spécifier uniquement un nom de service donc les paramètres de connexion peuvent être maintenus de façon centrale. Voir share/pg_service.conf.sample dans le répertoire d'installation pour plus d'informations sur la configuration de ce fichier.

Si un paramètre manque, alors la variable d'environnement correspondante (voir Section 27.11) est vérifiée. Si elle n'est pas disponible, alors la valeur par défaut indiquée est utilisée.

PQsetdbLogin
Crée une nouvelle connexion sur le serveur de bases de données.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

C'est le prédécesseur de PQconnectdb avec un ensemble de paramètres fixe. Cette fonction a les mêmes fonctionnalités sauf que les paramètres manquants seront toujours initialisées avec leur valeurs par défaut. Écrire NULL ou un chaîne vide pour un de ces paramètres fixes dont vous souhaitez utiliser la valeur par défaut.

PQsetdb
Crée une nouvelle connexion sur le serveur de bases de données.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

C'est une macro faisant appel à PQsetdbLogin avec des pointeurs nuls pour les paramètres login et pwd. Elle est fournie pour une compatibilité ascendante des très vieux programmes.

PQconnectStart
PQconnectPoll
Crée une connexion au serveur de bases de données d'une façon non bloquante.

```
PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

Ces deux fonctions sont utilisées pour ouvrir une connexion au serveur de bases de données d'une façon telle que le thread de votre application n'est pas bloqué sur les entrées/sorties distantes en demandant la connexion. Le but de cette approche est que l'attente de la fin des entrées/sorties peut se faire dans la boucle principale de l'application plutôt qu'à l'intérieur de `PQconnectdb`, et donc l'application peut gérer des opérations en parallèle à d'autres activités.

La connexion à la base de données est faite en utilisant les paramètres pris dans la chaîne `conninfo`, passée à `PQconnectStart`. Cette chaîne est du même format que celle décrite pour `PQconnectdb`.

Ni `PQconnectStart` ni `PQconnectPoll` ne bloqueront, aussi longtemps qu'un nombre de restrictions sont respectées :

- ◊ Les paramètres `hostaddr` et `host` sont utilisés de façon appropriée pour vous assurer que la requête de nom et la requête inverse ne soient pas lancées. Voir la documentation de ces paramètres avec `PQconnectdb` ci−dessus pour les détails.
- ◊ Si vous appelez `PQtrace`, assurez−vous que l'objet de flux dans lequel vous enregistrez les traces ne bloquera pas.
- ◊ Assurez−vous que le socket soit dans l'état approprié avant d'appeler `PQconnectPoll`, comme décrit ci−dessous.

Pour commencer une demande de connexion non bloquante, appelez `conn = PQconnectStart("`*`connection_info_string`*`")`. Si `conn` est nul, alors libpq a été incapable d'allouer une nouvelle structure `PGconn`. Sinon, un pointeur valide vers une structure `PGconn` est renvoyé (bien qu'il ne représente pas encore une connexion valide vers la base de données). Au retour de `PQconnectStart`, appelez `status = PQstatus(conn)`. Si `status` vaut `CONNECTION_BAD`, `PQconnectStart` a échoué.

Si `PQconnectStart` réussit, la prochaine étape est d'appeler souvent libpq de façon à ce qu'il continue la séquence de connexion. Utilisez `PQsocket(conn)` pour obtenir le descripteur de socket sous la connexion à la base de données. Du coup, une boucle : si le dernier retour de `PQconnectPoll(conn)` est `PGRES_POLLING_READING`, attendez que la socket soit prête pour lire (comme indiqué par `select()`, `poll()` ou une fonction système similaire). Puis, appelez de nouveau `PQconnectPoll(conn)`. Par contre, si le dernier retour de `PQconnectPoll(conn)` est `PGRES_POLLING_WRITING`, attendez que la socket soit prête pour écrire, puis appelez de nouveau `PQconnectPoll(conn)`. Si vous devez encore appeler `PQconnectPoll`, c'est−à−dire juste après l'appel de `PQconnectStart`, continuez comme si il avait renvoyé `PGRES_POLLING_WRITING`. Continuez cette boucle jusqu'à ce que `PQconnectPoll(conn)` renvoie `PGRES_POLLING_FAILED`, indiquant que la procédure de connexion a échoué ou, ou `PGRES_POLLING_OK`, indiquant le succès de la procédure de connexion.

À tout moment pendant la connexion, le statut de cette connexion pourrait être vérifié en appelant `PQstatus`. Si le résultat est `CONNECTION_BAD`, alors la procédure de connexion a échoué ; si, au contraire, elle renvoie `CONNECTION_OK`, alors la connexion est prête. Ces deux états sont détectables à partir de la valeur de retour de `PQconnectPoll`, décrite ci−dessus. D'autres états pourraient survenir lors (et seulement dans ce cas) d'une procédure de connexion asynchrone. Ils indiquent l'état actuel de la procédure de connexion et pourraient être utile pour fournir un retour à

l'utilisateur. Ces statuts sont :

`CONNECTION_STARTED`
: Attente de la connexion à réaliser.

`CONNECTION_MADE`
: Connexion OK ; attente d'un envoi.

`CONNECTION_AWAITING_RESPONSE`
: Attente d'une réponse du serveur.

`CONNECTION_AUTH_OK`
: Authentification reçue ; attente de la fin du lancement du moteur.

`CONNECTION_SSL_STARTUP`
: Négociation du cryptage SSL.

`CONNECTION_SETENV`
: Négociation des paramétrages de l'environnement.

Notez que, bien que ces constantes resteront (pour maintenir une compatibilité), une application ne devrait jamais se baser sur un ordre pour celles−ci ou sur tout ou sur le fait que le statut fait partie de ces valeurs documentés. Une application pourrait faire quelque chose comme ça :

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Connexion en cours...";
        break;

    case CONNECTION_MADE:
        feedback = "Connecté au serveur...";
        break;
.
.
.
    default:
        feedback = "Connexion...";
}
```

Le paramètre de connexion `connect_timeout` est ignoré lors de l'utilisation `PQconnectPoll` ; c'est de la responsabilité de l'application de décider quand une période de temps excessive s'est écoulée. Sinon, `PQconnectStart` suivi par une boucle `PQconnectPoll` est équivalent à `PQconnectdb`.

Notez que si `PQconnectStart` renvoie un pointeur non nul, vous devez appeler `PQfinish` lorsque vous en avez terminé avec lui, pour supprimer la structure et tous les blocs mémoires qui lui sont associés. Ceci doit être fait même si la tentative de connexion échoue ou est abandonnée.

`PQconndefaults`
: Renvoie les options de connexion par défaut.

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
    char   *keyword;   /* Le mot clé de l'option */
    char   *envvar;    /* Nom de la variable d'environnement équivalente */
    char   *compiled;  /* Valeur par défaut interne */
    char   *val;       /* Valeur actuelle de l'option ou NULL */
    char   *label;     /* Label du champ pour la dialogue de connexion */
    char   *dispchar;  /* Caractère à afficher pour ce champ
                          dans un dialogue de connexion. Les valeurs sont :
```

```
                                  ""        Affiche la valeur entrée sans modification
                                  "*"       Champ de mot de passe - cache la valeur
                                  "D"       Option de débogage - non affiché par défaut
      */
          int     dispsize;  /* Taille du champ en caractère pour le dialogue */
      } PQconninfoOption;
```

Renvoie un tableau d'options de connexion. Ceci pourrait être utiliser pour déterminer toutes les options possibles de `PQconnectdb` et leur valeurs par défaut. La valeur de retour pointe vers un tableau de structures `PQconninfoOption` qui se termine avec une entrée utilisant un pointeur nul pour `keyword`. Notez que les valeurs par défaut actuelles (champs `val`) dépendront des variables d'environnement et d'autres contextes. Les demandeurs doivent traiter les données des options de connexion en lecture seule.

Après le traitement du tableau d'options, libérez–le en le passant à la fonction `PQconninfoFree`. Si cela n'est pas fait, un petit groupe de mémoire est perdu à chaque appel de `PQconndefaults`.

`PQfinish`

Ferme la connexion au serveur. Libère aussi la mémoire utilisée par l'objet `PGconn`.

```
void PQfinish(PGconn *conn);
```

Notez que même si la connexion au serveur a échoué (d'après l'indication de `PQstatus`), l' application devrait appeler `PQfinish` pour libérer la mémoire utilisée par l'objet `PGconn`. Le pointeur `PGconn` ne doit pas être encore utilisé après l'appel à `PQfinish`.

`PQreset`

Réinitialise la canal de communication avec le serveur.

```
void PQreset(PGconn *conn);
```

Cette fonction fermera la connexion au serveur et tentera le rétablissement d'une nouvelle connexion au même serveur en utilisant tous les paramètres utilisés précédemment. Ceci pourrait être utile en cas de récupération après une erreur sur la connexion est perdue.

`PQresetStart`
`PQresetPoll`

Réinitialise la canal de communication avec le serveur d'une façon non bloquante.

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

Ces fonctions fermeront la connexion au serveur et tenteront de rétablir une nouvelle connexion au même serveur en utilisant les mêmes paramètres que précédemment. Ceci pourrait être utile en cas de récupération après une erreur si une connexion est perdue. Elles diffèrent de `PQreset` (ci–dessus) car elles agissent d'une façon non bloquante. Ces fonctions souffrent des mêmes restrictions que `PQconnectStart` et `PQconnectPoll`.

Pour initier une réinitialisation de la connexion, appelez `PQresetStart`. S'il renvoie 0, la réinitialisation a échoué. S'il renvoie 1, activez la réinitialisation en utilisant `PQresetPoll` exactement de la même façon que vous créeriez la connexion en utilisant `PQconnectPoll`.

# 27.2. Fonctions de statut de connexion

Ces fonctions sont utilisées pour interroger le statut d'un objet de connexion existant.

> **Astuce :** Les développeurs d'application libpq devraient être attentif au maintien de leur abstraction `PGconn`. Utilisez les fonctions d'accès décrites ci–dessous pour obetnir le contenu de `PGconn`. Évitez de référencer directement les champs de la structure `PGconn` parce qu'ils sont sujet à modification dans le futur. (À partir de PostgreSQL version 6.4, la définition de la structure (`struct`) derrière `PGconn` n'est même plus fournie dans `libpq-fe.h`. Si vous avez un vieux code qui accède directemet aux champs de `PGconn`, vous pouvez le conserver en incluant en plus `libpq-int.h` mais vous êtes encouragé à corriger le code rapidement.)

Les fonctions suivantes renvoient les valeurs des paramètres utilisées pour la connexion. Ces valeurs sont fixes pour la durée de vie de l'objet `PGconn`.

`PQdb`
: Renvoie le nom de la base de données de la connexion.

```
char *PQdb(const PGconn *conn);
```

`PQuser`
: Renvoie le nom d'utilisateur utilisé pour la connexion.

```
char *PQuser(const PGconn *conn);
```

`PQpass`
: Renvoie le mot de passe utilisé pour la connexion.

```
char *PQpass(const PGconn *conn);
```

`PQhost`
: Renvoie le nom d'hôte du serveur utilisé pour la connexion.

```
char *PQhost(const PGconn *conn);
```

`PQport`
: Renvoie le numéro de port utilisé pour la connexion.

```
char *PQport(const PGconn *conn);
```

`PQtty`
: Renvoie le TTY de débogage pour la connexion. (Ceci est obsolète car le serveur ne fait plus attention au paramétrage du TTY mais les fonctions restent pour des raisons de compatibilité ascendante.)

```
char *PQtty(const PGconn *conn);
```

`PQoptions`
: Renvoie les options en ligne de commande passées lors de la demande de connexion.

```
char *PQoptions(const PGconn *conn);
```

Les fonctions suivantes renvoient le statut car il peut changer suite à l'exécution d'opérations sur l'objet `PGconn`.

`PQstatus`
: Renvoie l'état de la connexion.

```
ConnStatusType PQstatus(const PGconn *conn);
```

Le statut peut faire partie d'un certain nombre de valeurs. Néanmoins, seules deux ne concernent pas les procédures de connexion asynchrone : `CONNECTION_OK` et `CONNECTION_BAD`. Une bonne connexion de la base de données a l'état `CONNECTION_OK`. Une tentative échouée de connexion est signalée par le statut `CONNECTION_BAD`. D'habitude, un état OK restera ainsi jusqu'à `PQfinish` mais un échec de communications pourrait résulter dans le statut changeant prématurément `CONNECTION_BAD`. Dans ce cas, l'application pourrait essayer de récupérer en appelant `PQreset`.

Voir l'entrée de `PQconnectStart` et de `PQconnectPoll` en regard aux autres codes de statut, qui pourraient être vus.

`PQtransactionStatus`

Renvoie l'état actuel de la transaction du serveur.

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

Le statut peut être `PQTRANS_IDLE` (actuellement inactif), `PQTRANS_ACTIVE` (une commande est en cours), `PQTRANS_INTRANS` (inactif, dans un bloc valide de transaction) ou `PQTRANS_INERROR` (inactif, dans un bloc de transaction échoué). `PQTRANS_UNKNOWN` est reporté si la connexion est mauvaise. `PQTRANS_ACTIVE` est reporté seulement quand une requête a été envoyé au serveur mais qu'elle n'est pas terminée.

| **Attention** |
| --- |
| `PQtransactionStatus` donnera des résultats incorrects lors de l'utilisation d'un serveur PostgreSQL 7.3 qui a désactivé le paramètre `autocommit`. La fonctionnalité autocommit, côté serveur, est obsolète et n'existe pas dans les versions serveur ultérieures. |

`PQparameterStatus`

Recherche un paramétrage actuel du serveur.

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

Certaines valeurs de paramètres sont reportées par le serveur automatiquement ou lorsque leur valeurs changent. `PQparameterStatus` peut être utilisé pour interroger ces paramétrages. Il renvoie la valeur actuelle d'un paramètre s'il est connu et `NULL` si le paramètre est inconnu.

Les paramètres reportés pour la version actuelle incluent `server_version`, `server_encoding`, `client_encoding`, `is_superuser`, `session_authorization`, `DateStyle`, `TimeZone` et `integer_datetimes`. (`server_encoding`, `TimeZone` et `integer_datetimes` n'étaient pas rapportés dans les versions antérieures à la 8.0.) Notez que `server_version`, `server_encoding` et `integer_datetimes` ne peuvent pas changer après le lancement du serveur.

Les serveurs utilisant un protocole antérieur à la 3.0 ne reportent pas la configuration des paramètres mais libpq inclut la logique pour obtenir des valeurs pour `server_version` et `client_encoding`. Les applications sont encouragées à utiliser `PQparameterStatus` plutôt qu'un code *ad−hoc* modifiant ces valeurs. (Néanmoins, attention, les connexions pré−3.0, changeant `client_encoding` via `SET` après le lancement de la connexion ne seront pas reflétées par `PQparameterStatus`.) Pour `server_version`, voir aussi `PQserverVersion`, qui renvoie l'information dans un format numérique qui est plus facile à comparer.

Bien que le pointeur renvoyé est déclaré `const`, il pointe en fait vers un stockage mutable associé avec la structure `PGconn`. Il est déconseillé de supposer que le pointeur restera valide pour toutes les requêtes.

`PQprotocolVersion`

Interroge le protocole interface/moteur lors de son utilisation.

```
int PQprotocolVersion(const PGconn *conn);
```

Les applications souhaitent utiliser ceci pour déterminer si certaines fonctionnalités sont supportées. Actuellement, les seules valeurs possible sont 2 (protocole 2.0), 3 (protocole 3.0) ou zéro (mauvaise connexion). Ceci ne changera pas après la fin du lancement de la connexion mais cela pourrait être changé théoriquement avec une réinitialisation de la connexion. Le protocole 3.0 sera normalement utilisé lors de la communication avec les serveurs PostgreSQL 7.4 ou ultérieures ; les serveurs antérieurs à la 7.4 supportent uniquement le protocole 2.0. (Le protocole 1.0 est obsolète et non supporté par libpq.)

`PQserverVersion`

Renvoie un entier représentant la version du moteur.

```
int PQserverVersion(const PGconn *conn);
```

Les applications pourraient utiliser ceci pour déterminer la version du serveur de la base de données auquel ils sont connectés. Le numéro est formé en convertissant les nombres majeur, mineur et de révision en un nombre à deux chiffres décimaux et en leur assemblant. Par exemple, la version 7.4.2 sera renvoyée en tant que 70402 et la version 8.1 sera renvoyée en tant que 80100 (les zéros au début ne sont pas affichés). Zéro est renvoyée si la connexion est mauvaise.

`PQerrorMessage`

Renvoie le dernier message d'erreur généré par une opération sur la connexion.

```
char *PQerrorMessage(const PGconn* conn);
```

Pratiquement toutes les fonctions libpq initialiseront un message pour `PQerrorMessage` en cas d'échec. Notez que par la convention libpq, un résultat non vide de `PQerrorMessage` inclura un retour chariot à la fin. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand la poignée `PGconn` associée est passée à `PQfinish`. Vous ne devriez pas supposer que la chaîne résultante reste identique suite à toutes les opérations sur la structure `PGconn`.

`PQsocket`

Obtient le descripteur de fichier du socket de la connexion au serveur. Un descripteur valide sera plus grand ou égal à 0 ; un résultat de −1 indique qu'aucune connexion au serveur n'est actuellement ouverte. (Ceci ne changera pas lors de l'opération normale mais pourra changer lors d'une configuration de l'initialisation ou lors d'une réinitialisation.)

```
int PQsocket(const PGconn *conn);
```

`PQbackendPID`

Renvoie l'identifiant du processus (PID) du serveur gérant cette connexion.

```
int PQbackendPID(const PGconn *conn);
```

Le PID du moteur est utile pour des raisons de débogage et pour la comparaison avec les messages `NOTIFY` (qui incluent le PID du processus serveur lançant la notification). Notez que le PID appartient à un processus exécuté sur l'hôte du serveur de bases de données et non pas sur l'hôte local !

`PQgetssl`
: Retourne la structure SSL utilisée dans la connexion ou NULL si SSL n'est pas utilisé.

```
SSL *PQgetssl(const PGconn *conn);
```

Cette structure peut être utilisée pour vérifier les niveaux de cryptage, pour vérifier les certificats du serveur, et plus. Référez−vous à la documentation d'OpenSSL pour plus d'informations sur cette structure.

Vous pouvez définir `USE_SSL` pour obtenir le bon prototype de cette fonction. Faire cela inclura automatiquement `ssl.h` à partir de OpenSSL.

# 27.3. Fonctions de commandes d'exécution

Une fois une connexion au serveur de la base de données a été établie avec succès, les fonctions décrites ici sont utilisées pour exécuter les requêtes SQL et les commandes.

## 27.3.1. Fonctions principales

`PQexec`
: Soumet une commande au serveur et attend le résultat.

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Renvoie un pointeur `PGresult` ou peut−être un pointeur NULL. Un pointeur non NULL sera généralement renvoyé sauf dans des conditions de manque de mémoire ou d'erreurs sérieuses telles que l'incapacité à envoyer la commande au serveur. Si un pointeur NULL est renvoyé, il devrait être traité comme un résultat `PGRES_FATAL_ERROR`. Utilisez `PQerrorMessage` pour obtenir plus d'informations sur l'erreur.

Il est autorisé d'inclure plusieurs commandes SQL (séparées par des points virgules) dans la chaîne de commande. Les requêtes multiples envoyées dans un simple appel à `PQexec` sont exécutées dans une seule transaction sauf s'il y a des commandes explicites `BEGIN`/`COMMIT` incluses dans la chaîne de requête pour la diviser dans de nombreuses transactions. Notez, néanmoins que la structure `PGresult` renvoyée, décrit seulement le résultat de la dernière commande exécutée à partir de la chaîne. Si une des commandes doit échouer, l'exécution de la chaîne s'arrête et le `PGresult` renvoyé décrit la condition d'erreur.

`PQexecParams`
: Soumet une commande au serveur et attend le résultat, avec la possibilité de passer des paramètres séparément du texte de la commande SQL.

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

`PQexecParams` est identique à `PQexec` mais offre des fonctionnalités supplémentaires : des

valeurs de paramètres peuvent être spécifiées séparément de la chaîne de commande et les résultats de la requête peuvent être demandés soit au format texte soit au format binaire. PQexecParams est supporté seulement dans les connexion avec le protocole 3.0 et ultérieurs ; elle échouera lors de l'utilisation du protocole 2.0.

Si les paramètres sont utilisés, ils sont référencés dans la chaîne de commande avec $1, $2, etc. nParams correspond au nombre de paramètres fournis ; il a la longueur des tableaux paramTypes[], paramValues[], paramLengths[] et paramFormats[]. (Les pointeurs de tableaux pourraient être NULL lorsque nParams vaut zéro.) paramTypes[] spécifie, par OID, les types de données d'être affectés aux symboles de paramètre. Si paramTypes est NULL ou si tout autre élément du tableau vaut zéro, le serveur assigne un type de données au symbole de paramètre de la même façon qu'il le ferait pour une chaîne littérale non typée. paramValues[] spécifie les valeurs réelles des paramètres. Un pointeur nul dans ce tableau signifie que le paramètre correspondant est nul ; sinon, le pointeur pointe vers une chaîne textuelle terminée avec un zéro (pour le format texte) ou des données binaires dans le format attendu par le serveur (pour le format binaire). paramLengths[] spécifie les longueurs réelles des données des paramètres au format binaire. Il est ignoré pour les paramètres NULL et les paramètres au format texte. Le pointeur de tableau pourrait être NULL lorsqu'il n'y a pas de paramètres binaires. paramFormats[] spécifie si les paramètres sont de type texte (place un zéro dans le tableau ) ou binaire (place un un dans le tableau ). Si le pointeur de tableau est NULL, alors tous les paramètres sont supposés être du type texte. resultFormat vaut zéro pour obtenir les résultats au format texte ou un pour obtenir des résultats au format binaire. (Il n'existe actuellement aucune provision pour obtenir des colonnes de résultats différents dans des formats différents bien que cela soit possible dans le protocole sous−jacent.)

Le principal avantage de PQexecParams sur PQexec est que les valeurs de paramètres pourraient être séparés à partir de la chaîne de commande, évitant ainsi le besoin de guillemets et d'échappements. Contrairement à PQexec, PQexecParams autorise au plus une commande SQL dans une chaîne donnée. (Il peut y avoir des points virgules mais pas plus d'une commande non vide.) C'est une limitation du protocole sous−jacent mais cela a quelque utilité comme défense supplémentaire contre les attaques d'injection de SQL.

PQprepare

Soumet une requête pour créer une instruction préparée avec les paramètres donnés et attends la fin de son exécution.

```
PGresult *PQprepare(PGconn *conn,
const char *stmtName,
const char *query,
int nParams,
const Oid *paramTypes);
```

PQprepare crée une instruction préparée pour une exécution ultérieure avec PQexecPrepared. Cette fonction autorise les commandes qui seront utilisées de façon répété à être analysées et planifiées qu'une seule fois, plutôt qu'à chaque exécution. PQprepare est uniquement supporté par les connexions utilisant le protocole 3.0 et ultérieurs ; elle échouera avec le protocole 2.0.

La fonction crée une instruction préparée nommée stmtName à partir de la chaîne query, devant contenir une seule commande SQL. stmtName pourrait être "" pour créer une instruction non nommée, auquel cas toute instruction non nommée déjà existante est automatiquement remplacée par cette dernière  sinon, il y aura une erreur si le nom de l'instruction est déjà définie dans la session en cours. Si des paramètres sont utilisés, ils sont référencés dans la requête avec $1, $2, etc. nParams est le nombre de paramètres pour lesquels des types sont prédéfinis dans le tableau paramTypes[].

(Le pointeur du tableau pourrait être NULL quand nParams vaut zéro.) paramTypes[] spécifie, par OID, les types de données à affecter aux symboles de paramètres. Si paramTypes est NULL ou si un élément particulier du tableau est zéro, le serveur affecte un type de données au symbole du paramètre de la même façon qu'il le ferait pour une chaîne littérale non typée. De plus, la requête pourrait utiliser des symboles de paramètre avec des nombres plus importants que nParams ; les types de données seront aussi inférés pour ces symboles.

Comme avec PQexec, le résultat est normalement un objet PGresult dont le contenu indique le succès ou l'échec côté serveur. Un résultat NULL indique un manque de mémoire ou une incapacité à envoyer la commande. Utilisez PQerrorMessage pour obtenir plus d'informations sur de telles erreurs.

Actuellement, il n'existe aucun moyen de déterminer le type de données réel inféré pour tous les paramètres dont les types ne sont pas spécifiés dans paramTypes[]. C'est un oubli de libpq qui sera probablement rectifié dans une version future.

Les instructions préparées avec PQexecPrepared peuvent aussi être créées en exécutant les instructions SQL PREPARE. (Mais PQprepare est plus flexible car il ne requiert pas de spécification des types de paramètres.) De plus, bien qu'il n'y ait aucune fonction libpq pour supprimer une instruction préparée, l'instruction SQL DEALLOCATE peut être utilisée dans ce but.

PQexecPrepared
Envoie une requête pour exécuter une instruction séparée avec les paramètres donnés, et attend le résultat.

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

PQexecPrepared est identique à PQexecParams, mais la commande à exécuter est spécifiée en nommant l'instruction préparée précédemment au lieu de donner une chaîne de requête. Cette fonctionnalité permet aux commandes qui seront utilisées de façon répétée pour être analysées et planifiées seulement une fois plutôt que chaque fois qu'ils sont exécutés. L'instruction doit avoir été préparée précédemment dans la session en cours. PQexecPrepared est supporté seulement dans les connexions du protocole 3.0 et ultérieur ; il échouera lors de l'utilisation du protocole 2.0.

Les paramètres sont identiques à PQexecParams, sauf que le nom d'une instruction préparée est donné au lieu d'une chaîne de requête et le paramètre paramTypes[] n'est pas présente (il n'est pas nécessaire car les types des paramètres de l'instruction préparée ont été déterminés à la création).

La structure PGresult encapsule le résultat renvoyé par le serveur. Les développeurs d'applications libpq devraient faire attention au maintien de l'abstraction de PGresult. Utilisez les fonctions d'accès ci−dessous pour obtenir le contenu de PGresult. Évitez directement la référence aux champs de la structure PGresult car ils sont sujets à des changements dans le futur.

PQresultStatus
Renvoie l'état du résultat d'une commande.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

`PQresultStatus` peut renvoyer une des valeurs suivantes :

`PGRES_EMPTY_QUERY`
: La chaîne envoyée au serveur était vide.

`PGRES_COMMAND_OK`
: Fin avec succès d'une commande ne renvoyant aucune donnée.

`PGRES_TUPLES_OK`
: Fin avec succès d'une commande renvoyant des données (telle que `SELECT` ou `SHOW`).

`PGRES_COPY_OUT`
: Début de l'envoi (à partir du serveur) d'un flux de données.

`PGRES_COPY_IN`
: Début de la réception (sur le serveur) d'un flux de données.

`PGRES_BAD_RESPONSE`
: La réponse du serveur n'a pas été comprise.

`PGRES_NONFATAL_ERROR`
: Une erreur non fatale (une note ou un avertissement) est survenue.

`PGRES_FATAL_ERROR`
: Une erreur fatale est survenue.

Si le statut du résultat est `PGRES_TUPLES_OK`, alors les fonctions décrites ci–dessous peuvent être utilisées pour récupérer les lignes renvoyées pas la requête. Notez qu'une commande `SELECT` qui arrive à récupérer zéro lignes affichera toujours `PGRES_TUPLES_OK`. `PGRES_COMMAND_OK` est pour les commandes qui ne peuvent jamais renvoyer de lignes (`INSERT`, `UPDATE`, etc.). Une réponse `PGRES_EMPTY_QUERY` pourrait indiquer un bogue dans le logiciel client.

Un résultat de statut `PGRES_NONFATAL_ERROR` ne sera jamais renvoyé directement par `PQexec` ou d'autres fonctions d'exécution de requêtes ; les résultats de ce type sont passés à l'exécuteur de notifications (voir Section 27.10).

`PQresStatus`
: Convertit le type énuméré renvoyé par `PQresultStatus` en une constante de type chaîne décrivant le code d'état. L'appelant ne devrait pas libérer le résultat.

```
char *PQresStatus(ExecStatusType status);
```

`PQresultErrorMessage`
: Renvoie le message d'erreur associé avec la commande ou une chaîne vide s'il n'y a pas eu d'erreurs.

```
char *PQresultErrorMessage(const PGresult *res);
```

S'il y a eu une erreur, la chaîne renvoyée inclura un retour chariot en fin. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand la poignée `PGresult` associée est passée à `PQclear`.

Suivant immédiatement un appel à `PQexec` ou `PQgetResult`, `PQerrorMessage` (sur la connexion) renverra la même chaîne que `PQresultErrorMessage` (sur le résultat). Néanmoins, un `PGresult` conservera son message d'erreur jusqu'à destruction alors que le message d'erreur de la connexion changera lorsque des opérations suivantes seront réalisées. Utiliser `PQresultErrorMessage` quand vous voulez connaître le statut associé avec un `PGresult` particulier ; utilisez `PQerrorMessage` lorsque vous souhaitez connaître le statut à partir de la dernière opération sur la connexion.

`PQresultErrorField`

Renvoie un champ individuel d'un rapport d'erreur.

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

`fieldcode` est un identifiant de champ d'erreur ; voir les symboles listés ci−dessous. `NULL` est renvoyé si `PGresult` n'est pas une résultat d'erreur ou d'avertissement, ou n'inclut pas le champ spécifié. Les valeurs de champ n'incluront normalement pas un retour chariot en fin. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand la poignée `PGresult` associée est passée à `PQclear`.

Les codes de champs suivants sont disponibles :

`PG_DIAG_SEVERITY`
: La sévérité ; le contenu du champ peut être `ERROR`, `FATAL` ou `PANIC` (dans un message d'erreur), ou `WARNING`, `NOTICE`, `DEBUG`, `INFO`ou `LOG` (dans un message de notification), ou une traduction localisée de ceux−ci. Toujours présent.

`PG_DIAG_SQLSTATE`
: Le code SQLSTATE de l'erreur. Ce code identifie le type d'erreur qui a survenu ; il peut être utilisé par des interfaces qui réalisent les opérations spécifiques (telles que la gestion des erreurs) en réponse à une erreur particulière de la base de données. Pour une liste des codes SQLSTATE possibles, voir Annexe A. Ce champ n'est pas localisable et est toujours présent.

`PG_DIAG_MESSAGE_PRIMARY`
: Le principal message d'erreur, compréhensible par un humain (typiquement sur une ligne). Toujours présent.

`PG_DIAG_MESSAGE_DETAIL`
: Détail : un message d'erreur secondaire et optionnel proposant plus d'informations sur le problème. Pourrait être composé de plusieurs lignes.

`PG_DIAG_MESSAGE_HINT`
: Astuce : une suggestion supplémentaire sur ce qu'il vaut faire suite à ce problème. Elle a pour but de différer du détail dans le fait qu'elle offre un conseil (potentiellement inapproprié) plutôt que des faits établis. Pourrait être composé de plusieurs lignes.

`PG_DIAG_STATEMENT_POSITION`
: Une chaîne contenant un entier décimal indiquant le position du curseur d'erreur comme index dans la chaîne d'instruction originale. Le premier caractère se trouve à l'index 1 et les positions sont mesurées en caractères, et non pas en octets.

`PG_DIAG_INTERNAL_POSITION`
: Ceci est défini de la même façon que le champ `PG_DIAG_STATEMENT_POSITION` mais c'est utilisé quand la position du curseur fait référence à une commande générée en interne plutôt qu'une soumise par le client. Le champ `PG_DIAG_INTERNAL_QUERY` apparaîtra toujours quand ce champ apparaît.

`PG_DIAG_INTERNAL_QUERY`
: Le texte d'une commande échouée, générée en interne. Ceci pourrait être, par exemple, une requête SQL lancée par une fonction PL/pgSQL.

`PG_DIAG_CONTEXT`
: Une indication du contexte dans lequel l'erreur est apparue. Actuellement, cela inclut une trace de la pile d'appels des fonctions actives de langages de procédures et de requêtes générées en interne. La trace a une entrée par ligne, la plus récente se trouvant au début.

`PG_DIAG_SOURCE_FILE`
: Le nom du fichier contenant le code source où l'erreur a été rapportée.

`PG_DIAG_SOURCE_LINE`
: Le numéro de ligne dans le code source où l'erreur a été rapportée.

PG_DIAG_SOURCE_FUNCTION
Le nom de la fonction dans le code source, où l'erreur a été rapportée.

Le client est responsable du formatage des informations affichées pour correspondre à ses besoins ; en particulier, il doit supprimer les longues lignes si nécessaires. Les caractères de retour chariot apparaissant dans les champs de message d'erreur devraient être traités comme des changements de paragraphes, pas comme des changements de lignes.

Les erreurs générées en interne par libpq auront une sévérité et un message principal mais aucun autre champ. Les erreurs renvoyées par un serveur utilisant un protocole antérieure à la 3.0 incluront la sévérité et le message principale, et quelques fois un message détaillé, mais aucun autre champ.

Notez que les champs d'erreurs sont seulement disponibles pour les objets PGresult, et non pas pour les objets PGconn ; il n'existe pas de fonction PQerrorField.

PQclear
Libère le stockage associé avec un PGresult. Chaque résultat de commande devrait être libéré via PQclear lorsqu'il n'est plus nécessaire.

```
void PQclear(PGresult *res);
```

Vous pouvez conserver un objet PGresult aussi longtemps que vous en avez besoin ; il ne part pas lorsque vous lancez une nouvelle commande, même pas si vous fermez la connexion. Pour vous en débarrasser, vous devez appeler PQclear. En cas d'oubli, ceci résultera en des pertes mémoires pour votre application.

PQmakeEmptyPGresult
Construit un objet PGresult vide avec le statut donné.

```
PGresult* PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

Il s'agit d'une fonction interne de libpq pour allouer et initialiser un objet PGresult vide. Il est exporté car certaines applications la trouvent utiles pour générer eux−même des objets résultats (particulièrement des objets avec des statuts d'erreur). Si conn n'est pas NULL et que status indique une erreur, le message d'erreur actuel pour la connexion spécifiée est copié dans PGresult. Notez que PQclear devrait éventuellement être appelé sur l'objet, comme avec un PGresult renvoyé par libpq elle−même.

## 27.3.2. Récupérer l'information provenant des résultats des requêtes

Ces fonctions sont utilisées pour extraire des informations provenant d'un objet PGresult représentant un résultat valide pour une requête (il a le statut PGRES_TUPLES_OK). Pour les objets ayant d'autres valeurs de statut, elles agiront comme si le résultat n'avait aucune ligne et aucune colonne.

PQntuples
Renvoie le nombre de lignes (tuples) du résultat de la requête.

```
int PQntuples(const PGresult *res);
```

PQnfields
Renvoie le nombre de colonnes (champs) de chaque ligne du résultat de la requête.

```
int PQnfields(const PGresult *res);
```

PQfname

Renvoie le nom de la colonne associé avec le numéro de colonne donnée. Les numéros de colonnes commencent à zéro. L'appelant ne devrait pas libérer directement le numéro. Il sera libéré quand la poignée `PGresult` associée est passée à `PQclear`.

```
char *PQfname(const PGresult *res,
              int column_number);
```

`NULL` est renvoyé si le numéro de colonne est en dehors de la plage.

`PQfnumber`

Renvoie le numéro de colonne associé au nom de la colonne donné.

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

−1 est renvoyé si le nom donné ne correspond à aucune colonne.

Le nom donné est traité comme un identifiant dans une commande SQL, c'est−à−dire qu'il est mis en minuscule sauf s'il est entre des guillemets doubles. Par exemple, pour le résultat de la requête suivante

```
select 1 as FOO, 2 as "BAR";
```

nous devons obtenir les résultats suivants :

```
PQfname(res, 0)              foo
PQfname(res, 1)              BAR
PQfnumber(res, "FOO")        0
PQfnumber(res, "foo")        0
PQfnumber(res, "BAR")        -1
PQfnumber(res, "\"BAR\"")    1
```

`PQftable`

Renvoie l'OID de la table à partir de laquelle la colonne donnée a été récupérée. Les numéros de colonnes commencent à zéro mais les colonnes des tables ont des numéros différents de zéro.

```
Oid PQftable(const PGresult *res,
             int column_number);
```

`InvalidOid` est renvoyé si le numéro de colonne est en dehors de la plage ou si la colonne spécifiée n'est pas une simple référence à une colonne de table, ou lors de l'utilisation d'un protocole antérieur à la version 3.0. Vous pouvez lancer des requêtes vers la table système `pg_class` pour déterminer exactement quelle table est référencée.

Le type `Oid` et la constante `InvalidOid` sera définie lorsque vous incluerez le fichier d'en−tête libpq. Ils auront le même type entier.

`PQftablecol`

Renvoie le numéro de colonne (à l'intérieur de la table) de la colonne correspondant à la colonne spécifiée de résultat de la requête. Les numéros de la colonne résultante commencent à 0.

```
int PQftablecol(const PGresult *res,
                int column_number);
```

Zéro est renvoyé si le numéro de colonne est en dehors de la plage, ou si la colonne spécifiée n'est pas une simple référence à une colonne de table, ou lors de l'utilisation d'un protocole antérieur à la

version 3.0.

`PQfformat`

Renvoie le code de format indiquant le format de la colonne donné. Les numéros de colonnes commencent à zéro.

```
int PQfformat(const PGresult *res,
              int column_number);
```

Le code de format zéro indique une représentation textuelle des données alors qu'un code de format un indique une représentation binaire. (Les autres codes sont réservés pour des définitions futures.)

`PQftype`

Renvoie le type de données associé avec le numéro de colonne donné. L'entier renvoyé est le numéro OID interne du type. Les numéros de colonnes commencent à zéro.

```
Oid PQftype(const PGresult *res,
            int column_number);
```

Vous pouvez lancer des requêtes sur la table système `pg_type` pour obtenir les noms et propriétés des différents types de données. Les OID des types de données intégrés sont définis dans le fichier `src/include/catalog/pg_type.h` de la distribution des sources.

`PQfmod`

Renvoie le modifieur de type de la colonne associée avec le numéro de colonne donné. Les numéros de colonnes commencent à zéro.

```
int PQfmod(const PGresult *res,
           int column_number);
```

L'interprétation des valeurs du modificateur est spécifique au type ; elles indiquent la précision ou les limites de taille. La valeur −1 est utilisée pour indiquer qu'<< aucune information n'est disponible >>. La plupart des types de données n'utilisent pas les modificateurs, auquel cas la valeur est toujours −1.

`PQfsize`

Renvoie la taille en octets de la colonne associée avec le numéro de colonne donné. Les numéros de colonnes commencent à zéro.

```
int PQfsize(const PGresult *res,
            int column_number);
```

`PQfsize` renvoie l'espace alloué pour cette colonne dans une ligne de la base de données, en d'autres termes la taille de la représentation interne du serveur du type de données. (De façon cohérente, ce n'est pas réellement utile pour les clients.) Une valeur négative indique que les types de données ont une longueur variable.

`PQbinaryTuples`

Renvoie 1 si `PGresult` contient des données binaires et 0 s'il contient des données texte.

```
int PQbinaryTuples(const PGresult *res);
```

Cette fonction est obsolète (sauf dans le cas d'une utilisation en relation avec `COPY`) car un seul `PGresult` peut contenir du texte dans certaines colonnes et des données binaires dans d'autres. `PQfformat` est la fonction préférée. `PQbinaryTuples` renvoie 1 seulement si toutes les colonnes du résultat sont dans un format binaire (format 1).

`PQgetvalue`

Renvoie la valeur d'un seul champ d'une seule ligne d'un `PGresult`. Les numéros de lignes et de

colonnes commencent à zéro. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand la poignée `PGresult` associée est passée à `PQclear`.

```
char* PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

Pour les données au format texte, la valeur renvoyée par `PQgetvalue` est une représentation au format chaîne de caractères terminée par un octet nul de la valeur du champ. Pour les données au format binaire, la valeur est dans la représentation binaire déterminée par le type de la donnée, fonctions `typsend` et `typreceive`. (La valeur est en fait suivie d'un octet zéro dans ce cas aussi mais ce n'est pas réellement utile car la valeur a des chances de contenir d'autres valeurs NULL embarquées.)

Une chaîne vide est renvoyée si la valeur du champ est NULL. Voir `PQgetisnull` pour distinguer les valeurs NULL des valeurs de chaîne vide.

Le pointeur renvoyé par `PQgetvalue` pointe vers le stockage qui fait partie de la structure `PGresult`. Personne ne devrait modifier les données vers lesquelles il pointe et tout le monde devrait copier explicitement les données dans un autre stockage s'il n'est pas utilisé après la durée de vie de la struture `PGresult`.

`PQgetisnull`

Teste un champ pour savoir s'il est nul. Les numéros de lignes et de colonnes commencent à zéro.

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

Cette fonction renvoie 1 si le champ est nul et 0 s'il contient une valeur non NULL. (Notez que `PQgetvalue` renverra une chaîne vide, et non pas un pointeur nul, pour un champ nul.)

`PQgetlength`

Renvoie la longueur réelle de la valeur d'un champ en octet. Les numéros de lignes et de colonnes commencent à zéro.

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

C'est la longueur réelle des données pour la valeur particulière des données, c'est–à–dire la taille de l'objet pointé par `PQgetvalue`. Pour le format textuel, c'est identique à `strlen()`. Pour le format binaire, c'est une information essentielle. Notez que *personne* ne devrait se fier à `PQfsize` pour obtenir la taille réelle des données.

`PQprint`

Affiche toutes les lignes et, optionnellement, les noms des colonnes dans le flux de sortie spécifié.

```
void PQprint(FILE* fout,      /* flux de sortie */
             const PGresult *res,
             const PQprintOpt *po);

typedef struct {
    pqbool  header;      /* affiche le en-têtes des champs et le nombre de
                            lignes */
    pqbool  align;       /* aligne les champs */
    pqbool  standard;    /* vieux format (mort) */
```

```
    pqbool  html3;       /* affiche les tables en HTML */
    pqbool  expanded;    /* étend les tables */
    pqbool  pager;       /* utilise le paginateur pour la sortie si nécessaire
                            */
    char    *fieldSep;   /* séparateur de champ */
    char    *tableOpt;   /* attributs des éléments de table HTML */
    char    *caption;    /* titre de la table HTML */
    char    **fieldName; /* Tableau terminé par un NULL des noms de remplacement
                            des champs */
} PQprintOpt;
```

Cette fonction était auparavant utilisée par psql pour afficher les résultats des requêtes mais ce n'est plus le cas. Notez qu'elle assume que les données sont dans un format textuel.

## 27.3.3. Récupérer les informations de résultats pour les autres commandes

Ces fonctions sont utilisées pour extraire des informations des objets `PGresult` qui ne sont pas les résultats d'instructions `SELECT`.

`PQcmdStatus`
: Renvoie l'état de la commande depuis l'instruction SQL qui a généré le `PGresult`. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand la poignée `PGresult` associée est passée à `PQclear`.

```
char * PQcmdStatus(PGresult *res);
```

D'habitude, c'est juste le nom de la commande mais elle pourrait inclure des données supplémentaires comme le nombre de lignes traitées.

`PQcmdTuples`
: Renvoie le nombre de lignes affectées par la commande SQL.

```
char * PQcmdTuples(PGresult *res);
```

Cette fonction renvoie une chaîne contenant le nombre de lignes affectées par l'instruction SQL qui a généré `PGresult`. Cette fonction peut seulement être utilisée après l'exécution d'une instruction `INSERT`, `UPDATE`, `DELETE`, `MOVE` ou `FETCH`, ou `EXECUTE` avec une instruction préparée contenant une instruction `INSERT`, `UPDATE` ou `DELETE`. Si la commande qui a généré `PGresult` était autre chose, `PQcmdTuples` renverrait directement la chaîne. L'appelant ne devrait pas libérer la valeur de retour directement. Elle sera libérée quand la poignée `PGresult` associée est passée à `PQclear`.

`PQoidValue`
: Renvoie l'OID de la ligne insérée, si la commande SQL était une instruction `INSERT` qui a inséré exactement une ligne dans une table comprenant des OID ou un `EXECUTE` d'une requête préparée contenant une instruction `INSERT` convenable. Sinon, cette fonction renvoie `InvalidOid`. Cette fonction renverra aussi `InvalidOid` si la table touchée par l'instruction `INSERT` ne contient pas d'OID.

```
Oid PQoidValue(const PGresult *res);
```

`PQoidStatus`
: Renvoie une chaîne avec l'OID de la ligne insérée si la commande SQL était une instruction `INSERT` qui insère exactement une ligne, ou un `EXECUTE` d'une instruction préparée consistant en une

commande INSERT. (La chaîne sera 0 si l'instruction INSERT n'a inséré qu'une seule ligne ou si la table cible n'a pas d'OID.) Si la commande n'était pas un INSERT, renvoie une chaîne vide.

```
char * PQoidStatus(const PGresult *res);
```

Cette fonction est obsolète et remplacée par PQoidValue. Elle n'est pas compatible avec les threads.

## 27.3.4. Chaîne d'échappement à inclure dans les commandes SQL

PQescapeString échappe une chaîne à utiliser dans une commande SQL. Ceci est utile lors de l'insertion de valeurs comme constantes littérales. Certains caractères (tels que les guillemets et les antislashs) doivent être échappés pour les empêcher d'être interprétés spécialement par l'analyseur SQL. PQescapeString réalise cette opération.

> **Astuce :** Il est tout particulièrement important de faire cet échappement proprement lors de la gestion de chaînes reçus d'une source non sûre. Sinon, il existe un risque de sécurité : vous êtes vulnérable à une attaque par << injection de SQL >> où des commandes SQL non souhaitées remplissent votre base de données.

Notez que ceci n'est ni nécessaire ni correct de faire un échappement lorsque une valeur est passée comme paramètre séparé dans PQexecParams ou ses routines similaires.

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

Le paramètre from pointe le premier caractère d'une chaîne à protéger et le paramètre length donne le nombre de caractères dans cette chaîne. Un octet de terminaison, zéro, n'est pas requis et ne devrait pas être compté dans length. (Si un octet de terminaison est trouvé avant que length octets ne soient traités, PQescapeString s'arrête au zéro ; le comportement ressemble donc à strncpy.) to devrait pointer vers un tampon capable de contenir au moins un caractère de plus que le double de la valeur de length, sinon le comportement est indéfini. Un appel à PQescapeString écrit une version échappée de la chaîne from dans le tampon to, remplaçant les caractères spéciaux de façon à ce qu'ils ne puissent causer aucun mal et en ajoutant un octet de terminaison zéro. Les guillemets simples qui doivent entourer les littéraux de chaîne PostgreSQL ne sont pas inclus dans la chaîne résultante ; ils devront être fournis dans la commande SQL dans laquelle le résultat sera inséré.

PQescapeString renvoie le nombre de caractères écrits dans to, sans inclure l'octet de terminaison.

Le comportement est indéfini si les chaînes to et from se surchargent.

## 27.3.5. Échapper des chaînes binaires pour une inclusion dans des commandes SQL

PQescapeBytea
: Échappe des données binaires à utiliser à l'intérieur d'une commande SQL avec le type bytea. Comme avec PQescapeString, c'est seulement utilisé pour insérer des données directement dans une chaîne de commande SQL.

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

Certaines valeurs d'octets *doivent* être échappées (mais toutes les valeurs d'octets *pourraient* être échappées) lorsqu'elles sont partie d'un littéral `bytea` dans une instruction SQL. En général, pour échapper un octet, il est converti dans le nombre à trois chiffres correspondant à sa valeur octale et précédé par deux antislashs. Le guillemet simple et caractère antislash ont des séquences d'échappements alternatives. Voir Section 8.4 pour plus d'informations. `PQescapeBytea` réalise cette opération en échappant seulement les octets requis.

Le paramètre `from` pointe sur le premier octet de la chaîne à échapper et le paramètre `from_length` donne le nombre d'octets de cette chaîne binaire. (Un octet zéro de terminaison n'est ni nécessaire ni compté.) Le paramètre `to_length` pointe vers une variable qui contiendra la longueur de la chaîne échappée résultante. Cette longueur inclut l'octet zéro de terminaison.

`PQescapeBytea` renvoie une version échappée du paramètre `from` dans la mémoire allouée avec `malloc()`. Cette mémoire doit être libérée avec `PQfreemem` lorsque le résultat n'est plus nécessaire. Tous les caractères spéciaux de la chaîne de retour sont remplacés de façon à ce qu'ils puissent être traités proprement par l'analyseur de chaînes littérales de PostgreSQL et par l'entrée `bytea` de la fonction. Un octet zéro de terminaison est aussi ajouté. Les guillemets simples qui englobent les chaînes littérales de PostgreSQL ne font pas partie de la chaîne résultante.

`PQunescapeBytea`

Convertit une représentation de la chaîne échappée en donnés binaires –– l'inverse de `PQescapeBytea`. Ceci est nécessaire lors de la récupération de données `bytea` en format texte, mais pas lors de sa récupération au format binaire.

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

Le paramètre `from` pointe vers une chaîne échappée de telle façon qu'elle pourrait provenir de `PQgetvalue` lorsque la colonne est de type `bytea`. `PQunescapeBytea` convertit cette représentation de la chaîne en sa représentation binaire. Elle renvoie une pointeur vers le tampon alloué avec `malloc()`, ou NULL en cas d'erreur, et place la taille du tampon dans `to_length`. Le résultat doit être libéré en utilisant `PQfreemem` lorsque celui–ci n'est plus nécessaire.

`PQfreemem`

Libère la mémoire allouée par libpq.

```
void PQfreemem(void *ptr);
```

Libère la mémoire allouée par libpq, particulièrement `PQescapeBytea`, `PQunescapeBytea`, et `PQnotifies`. C'est nécessaire pour Microsoft Windows, qui ne peut pas libérer la mémoire des DLL, sauf dans le cas où des DLL multithreadés (`/MD` dans VC6) sont utilisées. Pour les autres plateformes, cette fonction est identique à la fonction `free()` de la bibliothèque standard.

## 27.4. Traitement des commandes asynchrones

La fonction `PQexec` est adéquate pour soumettre des commandes aux applications standards, synchrones. Néanmoins, il a quelques déficiences pouvant être d'importance à certains utilisateurs :

- PQexec attend que la commande se termine. L'application pourrait avoir d'autres travaux à réaliser (comme le rafraichissement de l'interface utilisateur), auquel cas il ne voudra pas être bloqué en attente de la réponse.
- Comme l'exécution de l'application cliente est suspendu en attendant le résultat, il est difficile pour l'application de décider qu'elle voudrait annuler la commande en cours. (C'est possible avec un gestionnaire de signaux mais pas autrement.)
- PQexec peut renvoyer qu'une structure PGresult. Si la chaîne de commande soumise contient plusieurs commandes SQL, toutes les structures PGresult sont annulées par PQexec, sauf la dernière.

Les applications qui n'apprécient pas ces limitations peuvent utiliser à la place les fonctions sous−jacentes à partir desquelles PQexec est construit : PQsendQuery et PQgetResult. Il existe aussi PQsendQueryParams, PQsendPrepare et PQsendQueryPrepared, pouvant être utilisé avec PQgetResult pour dupliquer les fonctionnalités de respectivement PQexecParams, PQprepare et PQexecPrepared.

PQsendQuery
: Soumet une commande au serveur sans attendre le(s) résultat(s). 1 est renvoyé si la commande a été correctement envoyée et 0 sinon (auquel cas, utilisez la fonction PQerrorMessage pour obtenir plus d'informations sur l'échec).

```
int PQsendQuery(PGconn *conn, const char *command);
```

Après un appel réussi à PQsendQuery, appelez PQgetResult une ou plusieurs fois pour obtenir les résultats. PQsendQuery pourrait être appelé de nouveau (sur la même connexion) jusqu'à ce que PQgetResult renvoie un pointeur nul, indiquant que la commande a terminé.

PQsendQueryParams
: Soumet une commande et des paramètres séparés au serveur sans attendre le(s) résultat(s).

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

Ceci est équivalent à PQsendQuery sauf que les paramètres de requêtes peuvent être spécifiés à partir de la chaîne de requête. Les paramètres de la fonction sont gérés de façon identique à PQexecParams. Comme PQexecParams, cela ne fonctionnera pas pour les connexions utilisant le protocole 2.0 et cela ne permettra qu'une seule commande dans la chaîne de requête.

PQsendPrepare
: Envoie une requête pour créer une instruction préparée avec les paramètres donnés et redonne la main sans attendre la fin de son exécution.

```
int PQsendPrepare(PGconn *conn,
const char *stmtName,
const char *query,
int nParams,
const Oid *paramTypes);
```

Ceci est la version asynchrone de PQprepare : elle renvoie 1 si elle a été capable d'envoyer la requête, 0 sinon. Après un appel terminé avec succès, appelez PQgetResult pour déterminer si le serveur a créé avec succès l'instruction préparée. Les paramètres de la fonction sont gérés de façon identique à PQprepare. Comme PQprepare, cela ne fonctionnera pas sur les connexions utilisant le protocole 2.0.

PQsendQueryPrepared

Envoie une requête pour exécuter une instruction préparée avec des paramètres donnés sans attendre le(s) résultat(s).

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

Ceci est similaire à PQsendQueryParams mais la commande à exécuter est spécifiée en nommant une instruction précédemment préparée au lieu de donner une chaîne contenant la requête. Les paramètres de la fonction sont gérés de façon identique à PQexecPrepared. Comme PQexecPrepared, cela ne fonctionnera pas pour les connexions utilisant le protocole 2.0.

PQgetResult

Attend le prochain résultat d'un appel précédent à PQsendQuery, PQsendQueryParams, PQsendPrepare ou PQsendQueryPrepared, et le renvoie. Un pointeur nul est renvoyé quand la commande est terminée et qu'il n'y aura plus de résultats.

```
PGresult *PQgetResult(PGconn *conn);
```

PQgetResult doit être appelé de façon répété jusqu'à ce qu'il retourne un pointeur nul indiquant que la commande s'est terminée. (Si appelé à un moment où aucune commande n'est active, PQgetResult renverra seulement un pointeur nul à la fois.) Chaque résultat non nul provenant de PQgetResult devrait être traité en utilisant les mêmes fonctions d'accès à PGresult que celles précédemment décrites. N'oubliez pas de libérer chaque objet résultat avec PQclear une fois que vous en avez terminé. Notez que PQgetResult bloquera seulement si la commande est active et que les données nécessaires en réponse n'ont pas encore été lues par PQconsumeInput.

Utiliser PQsendQuery et PQgetResult résout un des problèmes de PQexec : si une chaîne de commande contient plusieurs commandes SQL, les résultats de ces commandes peuvent être obtenus individuellement. (Ceci permet une simple forme de traitement en parallèle : le client peut gérer les résultats d'une commande alors que le serveur travaille sur d'autres requêtes de la même chaîne de commandes.) Néanmoins, appeler PQgetResult causera toujours un blocage du client jusqu'à la fin de la prochaine commande SQL. Ceci est évitable en utilisant proprement deux fonctions supplémentaires :

PQconsumeInput

Si l'entrée est disponible à partir du serveur, consommez–la.

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput renvoie normalement 1 indiquant << aucune erreur >>, mais renvoie zéro s'il y a eu une erreur (auquel cas PQerrorMessage peut être consulté). Notez que le résultat ne dit pas si des données ont été récupérées en entrées. Après avoir appelé PQconsumeInput, l'application devrait vérifier PQisBusy et/ou PQnotifies pour voir si leur état a changé.

PQconsumeInput pourrait être appelé même si l'application n'est pas encore préparé à gérer un résultat ou une notification. La fonction lira les données disponibles et les sauvegardera dans un tampon, indiquant ainsi qu'une lecture d'un select() est possible. L'application peut donc utiliser PQconsumeInput pour effacer la condition select() immédiatement, puis pour examiner les résultats autant que possible.

PQisBusy

Renvoie 1 si une commande est occupée, c'est–à–dire que PQgetResult bloquerait en attendant une entrée. Un zéro indiquerait que PQgetResult peut être appelé avec l'assurance de ne pas être bloqué.

```
int PQisBusy(PGconn *conn);
```

PQisBusy ne tentera pas lui–même de lire les données à partir du serveur ; du coup, PQconsumeInput doit être appelé d'abord ou l'état occupé ne s'arrêtera jamais.

Une application typique de l'utilisation de ces fonctions aura une boucle principale utilisant select() ou poll() pour attendre toutes les conditions auxquelles il doit répondre. Une des conditions sera la disponibilité des données à partir du serveur, ce qui signifie pour select() des données lisibles sur le descripteur de fichier identifié par PQsocket. Lorsque la boucle principale détecte la disponibilité de données, il devrait appeler PQconsumeInput pour lire l'entére. Il peut ensuite appeler PQisBusy suivi par PQgetResult si PQisBusy renvoie false (0). Il peut aussi appeler PQnotifies pour détecter les messages NOTIFY (voir Section 27.7).

Un client qui utilise PQsendQuery/PQgetResult peut aussi tenter d'annuler une commande en cours de traitement par le serveur ; voir Section 27.5. Mais quelque soit la valeur renvoyée par PQcancel, l'application doit continuez avec la séquence normale de lecture du résultat en utilisant PQgetResult. Une annulation réussie causera simplement une fin plus rapide de la commande.

En utilisant les fonctions décrites ci–dessus, il est possible d'éviter le blocage pendant l'attente de données du serveur. Néanmoins, il est toujours possible que l'application se bloque en attendant l'envoi vers le serveur. C'est relativement peu fréquent mais cela peut arriver si de très longues commandes SQL ou données sont envoyées. (C'est bien plus probable si l'application envoie des données via COPY IN.) Pour empêcher cette possibilité et réussir des opérations de bases de données totalement non bloquantes, les fonctions supplémentaires suivantes pourraient être utilisées.

PQsetnonblocking

Initialise le statut non bloquant de la connexion.

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Initialise l'état de la connexion à non bloquant si arg vaut 1 et à bloquant si arg vaut 0. Renvoie 0 si OK, –1 en cas d'erreur.

Dans l'état non bloquant, les appels à PQsendQuery, PQputline, PQputnbytes, et PQendcopy ne bloqueront pas mais renverront à la place une erreur s'ils ont besoin d'être de nouveau appelés.

Notez que PQexec n'honore pas le mode non bloquant ; s'il est appelé, il agira d'une façon bloquante malgré tout.

PQisnonblocking

Renvoie le statut bloquant de la connexion à la base de données.

```
int PQisnonblocking(const PGconn *conn);
```

Renvoie 1 si la connexion est en mode non bloquant, 1 dans le cas contraire.

`PQflush`

Tente de vider les données des queues de sortie du serveur. Renvoie 0 en cas de succès (ou si la queue d'envoi est vide), −1 en cas d'échec quelque soit la raison ou 1 s'il a été incapable d'envoyer encore toutes les données dans la queue d'envoi (ce cas arrive seulement si la connexion est non bloquante).

```
int PQflush(PGconn *conn);
```

Après avoir envoyé une commande ou des données dans une connexion non bloquante, appelez `PQflush`. S'il renvoie 1, attendez que la socket soit disponible en écriture et appelez−la de nouveau ; répétez cela jusqu'à ce qu'il renvoie 0. Une fois que `PQflush` renvoie 0, attendez que la socket soit disponible en lecture puis lisez la réponse comme décrit ci−dessus.

## 27.5. Annuler des requêtes en cours d'exécution

Une application client peut demander l'annulation d'une commande qui est toujours en cours d'exécution par le serveur en utilisant les fonctions décrites dans cette section.

`PQgetCancel`

Crée une structure de données contenant les informations nécessaires à l'annulation d'une commande lancée sur une connexion particulière à la base de données.

```
PGcancel *PQgetCancel(PGconn *conn);
```

`PQgetCancel` crée un objet fonction `PGcancel` avec un objet connexion `PGconn`. Il renverra NULL si le paramètre `conn` donné est NULL ou une connexion invalide. L'objet `PGcancel` est une structure opaque qui n'a pas pour but d'être accédé directement par l'application ; elle peut seulement être passée à `PQcancel` ou `PQfreeCancel`.

`PQfreeCancel`

Libère une structure de données créée par `PQgetCancel`.

```
void PQfreeCancel(PGcancel *cancel);
```

`PQfreeCancel` libère un objet donnée par `PQgetCancel`.

`PQcancel`

Demande que le serveur abandonne l'exécution de la commande en cours.

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

La valeur renvoyée est 1 si la demande d'annulation a été correctement envoyée et 0 sinon. Si non, `errbuf` contient un message d'erreur expliquant pourquoi. `errbuf` doit être un tableau de caractères d'une taille de `errbufsize` octets (la taille recommandée est de 256 octets).

Un envoi réussi ne garantie pas que la demande aura un quelconque effet. Si l'annulation est réelle, la commande en cours terminera plus tôt et renverra une erreur. Si l'annulation échoue (disons, parce que le serveur a déjà exécuté la commande), alors il n'y aura aucun résultat visible.

`PQcancel` peut être invoqué de façon sûr par le gestionnaire de signaux si `errbuf` est une variable locale dans le gestionnaire de signaux. L'objet `PGcancel` object est en lecture seule pour ce qui concerne `PQcancel`, pour qu'il puisse aussi être appelé à partir d'un thread séparé de celui manipulant l'objet `PGconn`.

`PQrequestCancel`
Demande que le serveur abandonne le traitement de la commande en cours.

```
int PQrequestCancel(PGconn *conn);
```

`PQrequestCancel` est une variante obsolète de `PQcancel`. Elle opère directement sur l'objet `PGconn` et, en cas d'échec, stocke le message d'erreur dans l'objet `PGconn` (d'où il peut être récupéré avec `PQerrorMessage`). Bien qu'il s'agisse de la même fonctionnalité, cette approche est hasardeuse en cas de programmes compatibles avec les threads ainsi que pour les gestionnaires de signaux car il est possible que la surcharge du message d'erreur de `PGconn` gênera l'opération en cours sur la connexion.

## 27.6. Interface à chemin rapide

PostgreSQL fournit une interface rapide pour envoyer des appels de fonctions simples au serveur.

**Astuce :** Cette interface est quelque peu obsolète car vous pourriez réaliser avec des performances similaires et plus de fonctionnalités en initialisant une instruction préparée pour définir l'appel de fonction. Puis, exécuter l'instruction avec une transmission binaire des paramètres et des substitutions de résultats pour une appel de fonction à chemin rapide.

La fonction `PQfn` demande l'exécution d'une fonction du serveur via l'interface de chemin rapide :

```
PGresult* PQfn(PGconn* conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct {
    int len;
    int isint;
    union {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

L'argument `fnid` est l'OID de la fonction à exécuter. `args` et `nargs` définissent les paramètres à passer à la fonction ; ils doivent correspondre à la liste d'argument déclaré de la fonction. Quand le champ `isint` d'une structure est vrai, la valeur de `u.integer` est envoyée au serveur en tant qu'entier de la longueur indiquée (qui doit être 1, 2 ou 4 octets) ; les bons échanges d'octets se passent. Quand `isint` est faux, le nombre d'octets indiqué sur `*u.ptr` est envoyé au traitement ; les données doivent être dans le format attendu par le serveur pour la transmission binaire du type de données de l'argument de la fonction. `result_buf` est le tampon dans lequel placer le code de retour. L'appelant doit avoir alloué suffisamment d'espace pour stocker

le code de retour. (Il n'y a pas de vérification !) La longueur actuelle du résultat sera renvoyé dans l'entier pointé par `result_len`. Si un résultat sur un entier de 1, 2 ou 4 octets est attendu, initialisez `result_is_int` à 1, sinon initialisez–le à 0. Initialiser `result_is_int` à 1 fait que libpq échange les octets de la valeur si nécessaire, de façon à ce que la bonne valeur `int` soit délivrée pour la machine cliente. Quand `result_is_int` vaut 0, la chaîne d'octets au format binaire envoyée par le serveur est renvoyée non modifiée.

`PQfn` renvoie toujours un pointeur `PGresult` valide. L'état du résultat devrait être vérifié avant que le résultat ne soit utilisé. Le demandeur est responsable de la libération de la structure `PGresult` avec `PQclear` lorsque celle–ci n'est plus nécessaire.

Notez qu'il n'est pas possible de gérer les arguments nuls, les résultats nuls et les résultats d'ensembles nuls en utilisant cette interface.

## 27.7. Notification asynchrone

PostgreSQL propose des notifications asynchrone via les commandes `LISTEN` and `NOTIFY`. Une session cliente enregistre son intérêt dans une notification particulière avec la commande `LISTEN` (et peut arrêter son écoute avec la commande `UNLISTEN`). Toutes les sessions écoutant une condition particulière seront notifiées de façon asynchrone lorsqu'une commande `NOTIFY` avec ce nom de condition sera exécutée par une session. Aucune autre information n'est passée du notifieur au notifié. Du coup, typiquement, toute donnée qui a besoin d'être communiqué est transféré via une table de la base. D'habitude, le nom de la condition est identique à la table associée mais il n'est pas nécessaire d'avoir une table associée.

Les applications libpq soumettent les commandes `LISTEN` et `UNLISTEN` comme des commandes SQL ordinaires. L'arrivée des messages `NOTIFY` peut être détectée ensuite en appelant `PQnotifies`.

La fonction `PQnotifies` renvoie la prochaine notification à partir d'une liste de messages de notification non gérés reçus à partir du serveur. Il renvoie un pointeur nul s'il n'existe pas de notifications en attente. Une fois qu'une notification est renvoyée à partir de `PQnotifies`, elle est considérée comme étant gérée et sera supprimée de la liste des notifications.

```
PGnotify* PQnotifies(PGconn *conn);

typedef struct pgNotify {
    char *relname;              /* nom de la condition de la notification */
    int  be_pid;                /* ID du processus serveur */
    char *extra;                /* paramètre de notification */
} PGnotify;
```

Après avoir traité un objet `PGnotify` renvoyé par `PQnotifies`, assurez–vous de libérer le pointeur `PQfreemem`. Il est suffisant de libérer le pointeur `PGnotify` ; les champs `relname` et `extra` ne représentent pas des allocations séparées. (Actuellement, le champ `extra` est inutilisé et pointera en permanence vers une chaîne vide.)

> **Note :** Avec PostgreSQL 6.4 et ultérieur, le champ `be_pid` est celui du processus serveur ayant lancé la notification alors que, pour les versions précédentes il s'agissait toujours du PID de votre propre processus serveur.

Exemple 27–2 donne un programme d'exemple illustrant l'utilisation d'une notification asynchrone.

PQnotifies ne lit pas réellement les données à partir du serveur ; il renvoie simplement les messages précédemment absorbés par une autre fonction de libpq. Dans les précédentes versions de libpq, la seule façon de s'assurer une réception à temps des messages NOTIFY consistait à soumettre constamment des commandes de soumission, même vides, puis de vérifier PQnotifies après chaque PQexec. Bien que ceci fonctionnait, cela a été abandonné à cause de la perte de puissance.

Une meilleure façon de vérifier les messages NOTIFY lorsque vous n'avez pas de commandes utiles à exécuter est d'appeler PQconsumeInput puis de vérifier PQnotifies. Vous pouvez utiliser select() pour attendre l'arrivée des données à partir du serveur, donc en utilisant aucune puissance du CPU sauf lorsqu'il y a quelque chose à faire. (Voir PQsocket pour obtenir le numéro du descripteur de fichiers à utiliser avec select().) Notez que ceci fonctionnera bien que vous soumettiez les commandes avec PQsendQuery/PQgetResult ou que vous utilisez simplement PQexec. Néanmoins, vous devriez vous rappeler de vérifier PQnotifies après chaque PQgetResult ou PQexec, pour savoir si des notifications sont arrivées lors du traitement de la commande.

## 27.8. Fonctions associées avec la commande COPY

La commande COPY dans PostgreSQL a des options pour lire ou écrire à partir de la connexion réseau utilisée par libpq. Les fonctions décrites dans cette section autorisent les applications à prendre avantage de cette capacité en apportant ou en consommant les données copiés.

Le traitement complet est que l'application lance tout d'abord la commande SQL COPY via PQexec ou une des fonctions équivalents. La réponse à ceci (s'il n'y a pas d'erreur dans la commande) sera une objet PGresult avec un code de retour PGRES_COPY_OUT ou PGRES_COPY_IN (suivant la direction spécifiée pour la copie). L'application devrait alors utiliser les fonctions de cette section pour recevoir ou transmettre des lignes de données. Quand le transfert de données est terminé, un autre objet PGresult est renvoyé pour indiquer le succès ou l'échec du transfert. Son statut sera PGRES_COMMAND_OK en cas de succès et PGRES_FATAL_ERROR si un problème a été rencontré. À ce point, toute autre commande SQL pourrait être lancée via PQexec. (Il n'est pas possible d'exécuter d'autres commandes SQL en utilisant la même connexion tant que l'opération COPY est en cours.)

Si une commande COPY est lancée via PQexec dans une chaîne qui pourrait contenir d'autres commandes supplémentaires, l'application doit continuer à récupérer les résultats via PQgetResult après avoir terminé la séquence COPY. C'est seulement quand PQgetResult renvoie NULL que vous pouvez être certain que la chaîne de commandes PQexec est terminée et qu'il est possible de lancer d'autres commandes.

Les fonctions de cette section devraient seulement être exécutées pour obtenir un statut de résultat PGRES_COPY_OUT ou PGRES_COPY_IN à partir de PQexec ou PQgetResult.

Un objet PGresult gérant un de ces statuts comporte quelques données supplémentaires sur l'opération COPY qui commence. La données supplémentaire est disponible en utilisant les fonctions qui sont aussi utilisées en relation avec les résultats de requêtes :

PQnfields
: Renvoie le nombre de colonnes (champs) à copier.

PQbinaryTuples
: 0 indique que le format de copie complet est textuel (lignes séparées par des retours chariots, colonnes séparées par des caractères de séparation, etc). 1 indique que le format de copie complet est binaire. Voir *COPY* pour plus d'informations.

`PQfformat`
: Renvoie le code de format (0 pour le texte, 1 pour le binaire) associé avec chaque colonne de l'opération de copie. Les codes de format par colonne sera toujours zéro si le format de copie complet est textuel mais le format binaire supporte à la fois des colonnes textuelles et des colonnes binaires. (Néanmoins, avec l'implémentation actuelle de `COPY`, seules les colonnes binaires apparaissent dans une copie binaire  donc les formats par colonnes correspondent toujours au format complet actuellement.)

**Note :** Ces valeurs de données supplémentaires sont seulement disponibles en utilisant le protocole 3.0. Lors de l'utilisation du protocole 2.0, toutes ces fonctions renvoient 0.

## 27.8.1. Fonctions d'envoi de données pour `COPY`

Ces fonctions sont utilisées pour envoyer des données lors d'un `COPY FROM STDIN`. Elles échoueront si elles sont appelées alors que la connexion ne se trouve pas dans l'état `COPY_IN`.

`PQputCopyData`
: Envoie des données au serveur pendant un état `COPY_IN`.

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

Transmet les données de `COPY` dans le `buffer` spécifié, sur `nbytes`octets, au serveur. Le résultat vaut 1 si les données ont été envoyées, zéro si elles n'ont pas été envoyées car la tentative pourrait bloquer (ce cas n'est possible que dans le cas d'une connexion en mode non bloquant) ou −1 si une erreur s'est produite. (Utilisez `PQerrorMessage` pour récupérer des détails si la valeur de retour vaut −1. Si la valeur vaut zéro, attendez qu'il soit prêt en écriture et ré−essayez.)

L'application pourrait diviser le flux de données de `COPY` dans des chargements de tampon de taille convenable. Les limites n'ont pas de signification sémantique lors de l'envoi. Le contenu du flux de données doit correspondre au format de données attendu par la commande `COPY` ; voir ***COPY*** pour des détails.

`PQputCopyEnd`
: Envoie une indication de fin de transfert au serveur lors de l'état `COPY_IN`.

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

Termine l'opération `COPY_IN` avec succès si `errormsg` est `NULL`. Si `errormsg` n'est pas `NULL` alors `COPY` échoue, la chaîne pointée par `errormsg` étant utilisée comme message d'erreur. (néanmoins, vous ne devriez pas supposer que ce message d'erreur précis reviendra du serveur car le serveur pourrait avoir déjà échouée sur la commande `COPY` pour des raisons qui lui sont propres). Notez aussi que l'option forçant l'échec ne fonctionnera pas lors de l'utilisation de connexions avec un protocole pre−3.0.

Le résultat est 1 si la donnée de fin a été envoyée, zéro si elle ne l'a pas é té car cette tentative serait bloquante (ce cas est uniquement possible si la connexion est dans un mode non bloquant) ou −1 si une erreur est survenue. (Utilisez `PQerrorMessage` pour récupérer les détails si le code de retour est −1. Si la valeur vaut zéro, attendez que le serveur soit prêt en écriture et ré−essayez de nouveau.)

Après un appel réussi à PQputCopyEnd, appelez PQgetResult pour obtenir le statut de résultat final de la commande COPY. Vous pourriez attendre que le résultat soit disponible de la même façon. Puis, retournez aux opérations normales.

## 27.8.2. Fonctions pour recevoir des données de COPY

Ces fonctions sont utilisées pour recevoir des données lors d'un COPY TO STDOUT. Elles échoueront si elles sont appelées alors que la connexion n'est pas dans l'état COPY_OUT

PQgetCopyData
: Reçoit des données à partir du serveur lors d'un état COPY_OUT.

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

Tente d'obtenir une autre ligne de données du serveur lors d'une opération COPY. Les données ne sont renvoyées qu'une ligne à la fois ; si seulement une ligne partielle est disponible, elle n'est pas renvoyée. Le retour d'une ligne avec succès implique l'allocation d'une portion de mémoire pour contenir les données. Le paramètre buffer ne doit pas être NULL. *buffer est initialisé pour pointer vers la mémoire allouée ou vers NULL au cas où aucun tampon n'est renvoyé. Un tampon résultat non NULL devra être libéré en utilisant PQfreemem lorsqu'il ne sera plus utile.

Lorsqu'une ligne est renvoyée avec succès, le code de retour est le nombre d'octets de la données dans la ligne (et sera donc supérieur à zéro). La chaîne renvoyée est toujours terminée par un octet nul bien que ce ne soit utile que pour les COPY textuels. Un résultat zéro indique que la commande COPY est toujours en cours mais qu'aucune ligne n'est encore disponible (ceci est seulement possible lorsque async est vrai). Un résultat de −1 indique que COPY a terminé. Un résultat de −2 indique qu'une erreur est survenue (consultez PQerrorMessage pour en connaître la raison).

Lorsque async est vraie (différent de zéro), PQgetCopyData ne bloquera pas en attente d'entrée ; il renverra zéro si COPY est toujours en cours mais qu'aucune ligne n'est encore disponible. (Dans ce cas, attendez qu'il soit prêt en lecture puis appelez PQconsumeInput avant d'appeler de nouveau PQgetCopyData) Quand async est faux (zéro), PQgetCopyData bloquera tant que les données ne seront pas disponibles ou tant que l'opération n'aura pas terminée.

Après que PQgetCopyData ait renvoyé −1, appelez PQgetResult pour obtenir le statut de résultat final de la commande COPY. Vous pourriez attendre la disponibilité de ce résultat comme d'habitude. Puis, retournez aux opérations habituelles.

## 27.8.3. Fonctions obsolètes pour COPY

Ces fonctions représentent d'anciennes méthodes de gestion de COPY. Bien qu'elles fonctionnent toujours, elles sont obsolètes à cause de leur pauvre gestion des erreurs, des méthodes non convenables de détection d'une fin de transmission, et du manque de support des transferts binaires et des transferts non bloquants.

PQgetline
: Lit une ligne de caractères terminée par un retour chariot (transmis par le serveur) dans un tampon de taille length.

```
int PQgetline(PGconn *conn,
```

```
                char *buffer,
                int length);
```

Cette fonction copie jusqu'à `length`−1 caractères dans le tampin et convertit le retour chariot en un octet nul. `PQgetline` renvoie `EOF` à la fin de l'entrée, 0 si la ligne entière a été lu et 1 si le tampon est complet mais que le retour chariot à la fin n'a pas encore été lu.

Notez que l'application doit vérifier si un retour chariot est constitué de deux caractères `\.`, ce qui indique que le serveur a terminé l'envoi des résultats de la commande `COPY`. Si l'application peut recevoir des lignes de plus de `length`−1 caractères, une attention toute particulière est nécessaire pour s'assurer qu'elle reconnaisse la ligne `\.` correctement (et ne la confond pas, par exemple, avec la fin d'une longue ligne de données).

`PQgetlineAsync`

Lit une ligne de données `COPY` (transmise par le serveur) dans un tampon sans blocage.

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

Cette fonction est similaire à `PQgetline` mais elle peut être utilisée par des applications qui doivent lire les données de `COPY` de façon asynchrone, c'est−à−dire sans blocage. Après avoir lancé la commande `COPY` et obtenu une réponse `PGRES_COPY_OUT`, l'application devrait appeler `PQconsumeInput` et `PQgetlineAsync` jusqu'à ce que le signal de fin de données ne soit détecté.

Contrairement à `PQgetline`, cette fonction prend la responsabilité de détecter la fin de données.

À chaque appel, `PQgetlineAsync` renverra des données si une ligne de données complète est disponible dans le tampon d'entrée de libpq. Sinon, aucune ligne n'est renvoyée jusqu'à l'arrivée du reste de la ligne. La fonction renvoie −1 si le marqueur de fin de copie des données a été reconnu ou 0 si aucune données n'est disponible ou un nombre positif indiquant le nombre d'octets renvoyés. Si −1 est renvoyé, l'appelant doit ensuite appeler `PQendcopy` puis retourner aux traitements habituels.

Les données renvoyées ne seront pas étendues au delà de la limite de la ligne. Si possible, une ligne complète sera retournée en une fois. Mais si le tampon offert par l'appelant est trop petit pour contenir une ligne envoyée par le serveur, alors une ligne de données partielle sera renvoyée. Avec des données textuelles, ceci peut être détecté en testant si le dernier octet renvoyé est `\n` ou non. (Dans un `COPY` binaire, l'analyse réelle du format de données `COPY` sera nécessaire pour faire la détermination équivalente.) La chaîne renvoyée n'est pas terminée par un octet nul. (Si vous voulez ajouter un octet nul de terminaison, assurez−vous de passer un `bufsize` inférieur de 1 par rapport à l'espace réellement disponible.)

`PQputline`

Envoie une chaîne terminée par un octet nul au serveur. Renvoie 0 si tout va bien et `EOF` s'il est incapable d'envoyer la chaîne.

```
int PQputline(PGconn *conn,
              const char *string);
```

Le flux de données de `COPY` envoyé par une série d'appels à `PQputline` a le même format que celui renvoyé par `PQgetlineAsync`, sauf que les applications ne sont pas obligées d'envoyer exactement une ligne de données par appel à `PQputline` ; il est correct d'envoyer une ligne partielle ou plusieurs lignes par appel.

**Note :** Avant le protocole 3.0 de PostgreSQL, il était nécessaire pour l'application d'envoyer explicitement les deux caractères `\.` comme ligne finale pour indiquer qu'il a terminé l'envoi des données du `COPY` data. Bien que ceci fonctionne toujours, cette méthode est abandonnée et la signification spéciale de `\.` pourrait être supprimée dans une prochaine version. Il est suffisant d'appeler `PQendcopy` après avoir envoyé les vraies données.

`PQputnbytes`

Envoie une chaîne non terminée par un octet nul au serveur. Renvoie 0 si tout va bien et `EOF` s'il n'a pas été capable d'envoyer la chaîne.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

C'est exactement comme `PQputline` sauf que le tampon de donnée n'a pas besoin d'être terminé avec un octet nul car le nombre d'octets envoyés est spécifié directement. Utilisez cette procédure pour envoyer des données binaires.

`PQendcopy`

Se synchronise avec le serveur.

```
int PQendcopy(PGconn *conn);
```

Cette fonction attend que le serveur ait terminé la copie. Il devrait soit indiquer quand la dernière chaîne a été envoyée au serveur en utilisant `PQputline` soit le moment où la dernière chaîne a été reçue du serveur en utilisant `PGgetline`. Cela doit être fait ou le serveur renverra un << out of sync >> (perte de synchronisation) au client. Suivant le retour de cette fonction, le serveur est prêt pour recevoir la prochaine commande SQL. Le code de retour 0 indique un succès complet et est différent de zéro dans le cas contraire. (Utilisez `PQerrorMessage` pour récupérer des détails sur l'échec.)

Lors de l'utilisation de `PQgetResult`, l'application devrait répondre à un résultat `PGRES_COPY_OUT` en exécutant `PQgetline` de façon répétée, suivi par un `PQendcopy` une fois la ligne de terminaison aperçue. Il devrait ensuite retourner à la boucle `PQgetResult` jusqu'à ce que `PQgetResult` renvoie un pointeur nul. De façon similaire un résultat `PGRES_COPY_IN` est traité par une série d'appels à `PQputline` suivis par un `PQendcopy`, ensuite retour à la boucle `PQgetResult`. Cet arrangement vous assurera qu'une commande `COPY` intégrée dans une série de commandes SQL sera exécutée correctement.

Les anciennes applications soumettent un `COPY` via `PQexec` et assument que la transaction est faite après un `PQendcopy`. Ceci fonctionnera correctement seulement si `COPY` est la seule commande SQL dans la chaîne de commandes.

## 27.9. Fonctions de contrôle

Ces fonctions contrôlent divers détails du comportement de libpq.

`PQsetErrorVerbosity`

Détermine la verbosité des messages renvoyés par `PQerrorMessage` et `PQresultErrorMessage`.

```
typedef enum {
```

```
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

`PQsetErrorVerbosity` initialise le mode de verbosité, renvoyant le paramétrage précédent de cette connexion. Dans le mode *terse*, les messages renvoyés incluent seulement la sévérité, le texte principal et la position ; ceci tiendra normalement sur une seule ligne. Le mode par défaut produit des messages qui inclut ces champs ainsi que les champs détail, astuce ou contexte (ils pourraient être sur plusieurs lignes). Le mode *VERBOSE* inclut tous les champs disponibles. Modifier la verbosité n'affecte pas les messages disponibles à partir d'objets `PGresult` déjà existants, seulement ceux créés après.

`PQtrace`

Active les traces de communication entre client et serveur dans un flux fichier de débogage.

```
void PQtrace(PGconn *conn, FILE *stream);
```

`PQuntrace`

Désactive les traces commencées avec `PQtrace`.

```
void PQuntrace(PGconn *conn);
```

# 27.10. Traitement des messages

Les messages de note et d'avertissement générés par le serveur ne sont pas renvoyés par les fonctions d'exécution des requêtes car elles n'impliquent pas d'échec dans la requête. À la place, elles sont passées à la fonction de gestion des messages et l'exécution continue normalement après le retour du gestionnaire. La fonction par défaut de gestion des messages affiche le message sur `stderr` mais l'application peut surcharger ce comportement en proposant sa propre fonction de gestion.

Pour des raisons historiques, il existe deux niveaux de gestion de messages, appelés la réception des messages et le traitement. Pour la réception, le comportement par défaut est de formater le message et de passer une chaîne au traitement pour affichage. Néanmoins, une application qui choisit de fournir son propre receveur de messages ignorera typiquement la couche d'envoi de messages et fera tout travail au niveau du receveur.

La fonction `PQsetNoticeReceiver` initialise ou examine le receveur actuel de messages pour un objet de connexion. De la même façon, `PQsetNoticeProcessor` initialise ou examine l'émetteur actuel de messages.

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                    PQnoticeReceiver proc,
                    void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

Chacune de ces fonctions reçoit le pointeur de fonction du précédent receveur ou émetteur de messages et configure la nouvelle valeur. Si vous fournissez un pointeur de fonction nul, aucune action n'est réalisée mais le pointeur actuel est renvoyé.

Quand un message de note ou d'avertissement est reçu du serveur ou généré de façon interne par libpq, la fonction de réception du message est appelée. Le message lui est passé sous la forme d'un `PGresult PGRES_NONFATAL_ERROR`. (Ceci permet au receveur d'extraire les champs individuels en utilisant `PQresultErrorField` ou le message complet préformaté en utilisant `PQresultErrorMessage`.) Le même pointeur void passé à `PQsetNoticeReceiver` est aussi passé. (Ce pointeur peut être utilisé pour accéder à un état spécifique de l'application si nécessaire.)

Le receveur de messages par défaut extrait simplement le message (en utilisant `PQresultErrorMessage`) et le passe au système de traitement du message.

Ce dernier est responsable de la gestion du message de note ou d'avertissement donné au format texte. La chaîne texte du message est passée avec un retour chariot supplémentaire, plus un pointeur sur void identique à celui passé à `PQsetNoticeProcessor`. (Ce pointeur peut être utilisé pour accéder à un état spécifique de l'application si nécessaire.)

Le traitement des messages par défaut est simplement

```
static void
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

Une fois que vous avez initialisé un receveur ou une fonction de traitement des messages, vous devez vous attendre à ce que la fonction soit appelée aussi longtemps que l'objet `PGconn` ou qu'un objet `PGresult` réalisé à partir de celle−ci existent. À la création d'un `PGresult`, les pointeurs de gestion actuels de `PGconn` sont copiés dans `PGresult` pour une utilisation possible par des fonctions comme `PQgetvalue`.

## 27.11. Variables d'environnement

Les variables d'environnement suivantes peuvent être utilisées pour sélectionner des valeurs par défaut de paramètres de connexion, qui seront utilisées par `PQconnectdb`, `PQsetdbLogin` et `PQsetdb` si aucune valeur n'est directement spécifiée par le code d'appel. Elles sont utiles pour éviter de coder en dur les informations de connexion à la base de données dans les applications clients, par exemple.

- `PGHOST` initialise le nom du serveur de la base de données. S'il commence avec une barre oblique, il spécifie une communication de domaine Unix plutôt qu'une communication TCP/IP ; la valeur est le nom du répertoire où le fichier socket est stocké (dans une installation par défaut, cela serait `/tmp`).
- `PGHOSTADDR` spécifie l'adresse IP numérique du serveur de la base de données. Elle peut être initialisée avec `PGHOST` pour éviter la surcharge des recherches DNS. Voir la documentation de ces paramètres, dans `PQconnectdb` ci−dessus, pour des détails sur leur interaction.

  Quand ni `PGHOST` ni `PGHOSTADDR` n'est initialisé, le comportement par défaut est de se connecter en utilisant un socket de domaine Unix ; ou sur les machines sans sockets de domaine Unix, libpq essaiera de se connecter à `localhost`.

- PGPORT initialise le numéro de port TCP ou l'extension du fichier socket domaine Unix pour la communication avec le serveur PostgreSQL.
- PGDATABASE initialise le nom de la base de données sous PostgreSQL.
- PGUSER initialise le nom de l'utilisateur se connectant à la base de données.
- PGPASSWORD initialise le mot de passe utilisé si le serveur demande une authentification par mot de passe. L'utilisation de cette variable d'environnement n'est pas recommandée pour des raisons de sécurité (certains systèmes d'exploitation autorisent les utilisateurs autres que root de voir les variables d'environnement du processus via ps) ; à la place, considérez l'utilisation du fichier ~/.pgpass (voir Section 27.12).
- PGSERVICE initialise le nom du service à recherche dans pg_service.conf. Cela offre un raccourci pour la configuration de tous les paramètres.
- PGREALM initialise la royaume Kerberos à utiliser avec PostgreSQL s'il est différent du royaume local. Si PGREALM est initialisé, les applications libpq tenteront une authentification avec les serveurs de ce royaume et utiliseront les fichiers tickets séparés pour éviter les conflits avec les fichiers tickets locaux. Cette variable d'environnement est seulement utilisée si l'authentification Kerberos est sélectionnée par le serveur.
- PGOPTIONS initialise les options d'exécution supplémentaires pour le serveur PostgreSQL.
- PGSSLMODE détermine si et avec quelle priorité une connexion SSL sera négociée avec le serveur. Il existe quatre modes : disable tentera seulement une connexion non cryptée, donc sans SSL ; allow négociera en commençant par une connexion non SSL puis, s'il échoue, essaiera une connexion SSL ; prefer (la valeur par défaut) négociera en commençant par une connexion SSL puis, en cas d'échec, essaiera une connexion non SSL ; require essaiera seulement une connexion SSL. Si PostgreSQL est compilé sans le support de SSL, utiliser l'option require générera une erreur bien que les options allow et prefer seront acceptées mais, en fait, libpq ne tentera pas de connexion SSL.
- PGREQUIRESSL initialise une connexion via SSL. S'il est initialisé à << 1 >>, libpq refusera de se connecter si le serveur n'accepte pas de connexion SSL (équivalent à un sslmode valant prefer). Cette option est obsolète pour laisser la place au paramétrage sslmode et est seulement disponible si PostgreSQL est compilé avec le support de SSL.
- PGCONNECT_TIMEOUT initialise le nombre de secondes maximum que libpq attendra pour une connexion au serveur PostgreSQL. Si non initialisée ou si valant zéro, libpq attendra indéfiniment. Il n'est pas recommandé d'initialiser le délai à moins de deux secondes.

Les variables d'environnement par défaut peuvent être utilisées pour spécifier le comportement par défaut de chaque session PostgreSQL. (Voir aussi les commandes *ALTER USER* et *ALTER DATABASE* pour des moyens d'initialiser le comportement par défaut sur des bases par utilisateur ou par bases de données.)

- PGDATESTYLE initialise le style par défaut de la représentation de la date et de l'heure. (Équivalent à SET datestyle TO ....)
- PGTZ initialise le fuseau horaire par défaut (Équivalent à SET timezone TO ....)
- PGCLIENTENCODING initialise le codage de l'ensemble des caractères par défaut. (Équivalent à SET client_encoding TO ....)
- PGGEQO initialise le mode par défaut pour l'optimiseur générique de requêtes. (Équivalent à SET geqo TO ....)

Référez−vous à la commande SQL *SET* pour plus d'informations sur des valeurs correctes pour ces variables d'environnement.

Les variables d'environnement suivantes déterminent le comportement interne de libpq ; elles surchargent les valeurs internes par défaut.

- PGSYSCONFDIR configure le répertoire contenant le fichier pg_service.conf.
- PGLOCALEDIR configure le répertoire contenant les fichiers locale pour l'internationalisation des messages.

## 27.12. Fichier de mots de passe

Le fichier .pgpass, situé dans le répertoire personnel de l'utilisateur est un fichier contenant les mots de passe à utiliser si la connexion requiert un mot de passe (et si aucun mot de passe n'a été spécifié). Sur Microsoft Windows, le fichier est nommé %APPDATA%\postgresql\pgpass.conf (où %APPDATA% fait référence au sous−répertoire Application Data du profile de l'utilisateur).

Ce fichier devra être composé de lignes du format :

*nom_hote*:*port*:*database*:*nomutilisateur*:*motdepasse*

Chacun des quatre premiers champs pourraient être une valeur littérale ou *, qui correspond à tout. Le champ de mot de passe à partir de la première ligne qui correspond aux paramètres de connexion actuels sera utilisé. (Du coup, placez les entrées plus spécifiques en premier lorsque vous utilisez des jokers.) Si une entrée a besoin de contenir : ou \, échappez ce caractère avec \.

Les droits sur .pgpass doivent interdire l'accès aux autres et au groupe ; réalisez ceci avec la commande chmod 0600 ~/.pgpass. Si les droits sont aussi stricts que ça, le fichier sera ignoré. (Néanmoins, les droits du fichier ne sont actuellement pas vérifiés sur Microsoft Windows.)

## 27.13. Support de SSL

PostgreSQL dispose d'un support natif des connexions SSL pour crypter les connexions client/serveur et améliorer ainsi la sécurité. Voir Section 16.8 pour des détails sur la fonctionnalité SSL côté serveur.

Si le serveur demande un certificat au client, libpq enverra le certificat stocké dans le fichier ~/.postgresql/postgresql.crt se trouvant à l'intérieur du répertoire personnel de l'utilisateur. Un fichier de la clé privée correspondante, ~/.postgresql/postgresql.key, doit aussi être présent et ne doit pas être lisible par tout le monde. (Sur les systèmes Microsoft Windows, ces fichiers sont nommés %APPDATA%\postgresql\postgresql.crt et %APPDATA%\postgresql\postgresql.key.)

Si le fichier ~/.postgresql/root.crt est présent dans le répertoire personnel de l'utilisateur, libpq utilisera la liste de certificats stockée pour vérifier le certificat du serveur. (Sur les systèmes Microsoft Windows, le fichier est nommé %APPDATA%\postgresql\root.crt.) La connexion SSL échouera si le serveur ne présente pas de certificat ; du coup, pour utiliser cette fonctionnalité, le serveur doit aussi avoir un fichier root.crt.

## 27.14. Comportement des programmes threadés

libpq est réentrante et sûre avec les threads si l'option en ligne de commande de configure, --enable-thread-safety, a été utilisée lors de la construction de PostgreSQL. De plus, vous pourriez avoir besoin d'utiliser des options de compilation supplémentaires en ligne lorsque vous compiler le code de

votre application. Référez−vous aux documentations de votre système pour savoir comment construire des applications actives au niveau thread ou recherchez `PTHREAD_CFLAGS` et `PTHREAD_LIBS` dans `src/Makefile.global`.

Une restriction est qu'il ne doit pas y avoir deux tentatives de threads manipulant le même objet `PGconn` à la fois. En particulier, vous ne pouvez pas lancer des commandes concurrentes à partir de threads différents à travers le même objet de connexion. (Si vous avez besoin de lancer des commandes concurrentes, utilisez plusieurs connexions.)

Les objets `PGresult` sont en lecture seule après leur création et, du coup, ils peuvent être passés librement entre les threads.

Les fonctions obsolètes `PQrequestCancel`, `PQoidStatus` et `fe_setauthsvc` ne gèrent pas les threads et ne devraient pas être utilisées dans des programmes multithread. `PQrequestCancel` peut être remplacé par `PQcancel`. `PQoidStatus` peut être remplacé par `PQoidValue`. Il n'existe pas aucune bonne raison pour appeler `fe_setauthsvc`.

Les applications libpq qui utilisent la méthode d'authentification `crypt` sont liés à la fonction `crypt()` du système d'exploitation, qui est souvent non conforme avec les threads. Il est mieux d'utiliser la méthode `md5`, qui est compatible avec les threads sur toutes les plateformes.

Si vous expérimentez des problèmes avec les applications utilisant des threads, lancez le programme dans `src/tools/thread` pour voir si votre plateforme à des fonctions non compatibles avec les threads. Ce programme est lancé par `configure` mais, dans le cas des distributions binaires, votre bibliothèque pourrait ne pas correspondre à la bibliothèque utilisée pour construire les binaires.

## 27.15. Construire des applications avec libpq

Pour construire (c'est−à−dire compiler et lier) un programme utilisant libpq, vous avez besoin de faire tout ce qui suit :

- Incluez le fichier d'en−tête `libpq-fe.h` :

  ```
  #include <libpq-fe.h>
  ```

  Si vous ne le faites pas, alors vous obtiendrez normalement les messages d'erreurs similaires à ceci

  ```
  foo.c: In function `main':
  foo.c:34: `PGconn' undeclared (first use in this function)
  foo.c:35: `PGresult' undeclared (first use in this function)
  foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
  ```
- Pointez votre compilateur sur le répertoire où les fichiers d'en−tête de PostgreSQL ont été installés, en fournissant l'option `-I`*`répertoire`* à votre compilateur. (Dans certains cas, le compilateur cherchera dans le répertoire en question par défaut, donc vous pouvez omettre cette option.) Par exemple, votre ligne de commande de compilation devrait ressembler à ceci :

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

Si vous utilisez des makefiles, alors ajoutez l'option à la variable `CPPFLAGS` :

```
CPPFLAGS += -I/usr/local/pgsql/include
```

S'il existe une chance pour que votre programme soit compilé par d'autres utilisateurs, alors vous ne devriez pas coder en dur l'emplacement du répertoire. À la place, vous pouvez exécuter l'outil `pg_config` pour trouver où sont placés les fichiers d'en−tête sur le système local :

```
$ pg_config --includedir
/usr/local/include
```

Un échec sur la spécification de la bonne option au compilateur résultera en un message d'erreur tel que

```
testlibpq.c:8:22: libpq-fe.h: No such file or directory
```

- Lors de l'édition des liens du programme final, spécifiez l'option `-lpq` de façon à ce que les bibliothèques libpq soient intégrées, ainsi que l'option `-L`*répertoire* pour pointer le compilateur vers le répertoire où les bibliothèques libpq résident. (De nouveau, le compilateur cherchera certains répertoires par défaut.) Pour une portabilité maximale, placez l'option `-L` avant l'option `-lpq`. Par exemple :

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

  Vous pouvez aussi récupérer le répertoire des bibliothèques en utilisant `pg_config` :

  ```
  $ pg_config --libdir
  /usr/local/pgsql/lib
  ```

  Les messages d'erreurs, pointant vers des problèmes de ce style, pourraient ressembler à ce qui suit.

  ```
  testlibpq.o: In function `main':
  testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
  testlibpq.o(.text+0x71): undefined reference to `PQstatus'
  testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
  ```

  Ceci signifie que vous avez oublié `-lpq`.

  ```
  /usr/bin/ld: cannot find -lpq
  ```

  Ceci signifie que vous avez oublié l'option `-L` ou que vous n'avez pas indiqué le bon répertoire.

Si votre code référence le fichier d'en−tête `libpq-int.h` et que vous refusez de corriger votre code pour ne pas l'utiliser, à partir de PostgreSQL 7.2, ce fichier sera disponible dans *includedir*`/postgresql/internal/libpq-int.h`, donc vous aurez besoin d'ajouter l'option `-I` à votre ligne de commande pour le compilateur.

## 27.16. Exemples de programmes

Ces exemples et d'autres sont disponibles dans le répertoire `src/test/examples` de la distribution des sources.

**Exemple 27−1. Premier exemple de programme pour libpq**

```
/*
 * testlibpq.c
 *
 *              Test the C version of LIBPQ, the POSTGRES frontend library.
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
        PQfinish(conn);
        exit(1);
}

int
main(int argc, char **argv)
{
        const char *conninfo;
        PGconn     *conn;
        PGresult   *res;
        int                     nFields;
        int                     i,
                                j;

        /*
         * If the user supplies a parameter on the command line, use it as
         * the conninfo string; otherwise default to setting dbname=template1
         * and using environment variables or defaults for all other connection
         * parameters.
         */
        if (argc > 1)
                conninfo = argv[1];
        else
                conninfo = "dbname = template1";

        /* Make a connection to the database */
        conn = PQconnectdb(conninfo);

        /* Check to see that the backend connection was successfully made */
        if (PQstatus(conn) != CONNECTION_OK)
        {
                fprintf(stderr, "Connection to database failed: %s",
                        PQerrorMessage(conn));
                exit_nicely(conn);
        }

        /*
         * Our test case here involves using a cursor, for which we must be
         * inside a transaction block.  We could do the whole thing with a
         * single PQexec() of "select * from pg_database", but that's too
         * trivial to make a good example.
         */

        /* Start a transaction block */
        res = PQexec(conn, "BEGIN");
        if (PQresultStatus(res) != PGRES_COMMAND_OK)
        {
                fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
```

```
                PQclear(res);
                exit_nicely(conn);
        }

        /*
         * Should PQclear PGresult whenever it is no longer needed to avoid
         * memory leaks
         */
        PQclear(res);

        /*
         * Fetch rows from pg_database, the system catalog of databases
         */
        res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
        if (PQresultStatus(res) != PGRES_COMMAND_OK)
        {
                fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
                PQclear(res);
                exit_nicely(conn);
        }
        PQclear(res);

        res = PQexec(conn, "FETCH ALL in myportal");
        if (PQresultStatus(res) != PGRES_TUPLES_OK)
        {
                fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
                PQclear(res);
                exit_nicely(conn);
        }

        /* first, print out the attribute names */
        nFields = PQnfields(res);
        for (i = 0; i < nFields; i++)
                printf("%-15s", PQfname(res, i));
        printf("\n\n");

        /* next, print out the rows */
        for (i = 0; i < PQntuples(res); i++)
        {
                for (j = 0; j < nFields; j++)
                        printf("%-15s", PQgetvalue(res, i, j));
                printf("\n");
        }

        PQclear(res);

        /* close the portal ... we don't bother to check for errors ... */
        res = PQexec(conn, "CLOSE myportal");
        PQclear(res);

        /* end the transaction */
        res = PQexec(conn, "END");
        PQclear(res);

        /* close the connection to the database and cleanup */
        PQfinish(conn);

        return 0;
}
```

**Exemple 27–2. Deuxième exemple de programme pour libpq**

```
/*
 * testlibpq2.c
 *              Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 * Repeat four times to get this program to exit.
 *
 * Or, if you want to get fancy, try this:
 * populate a database with the following commands
 * (provided in src/test/examples/testlibpq2.sql):
 *
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * and do this four times:
 *
 *   INSERT INTO TBL1 VALUES (10);
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
        PQfinish(conn);
        exit(1);
}

int
main(int argc, char **argv)
{
        const char *conninfo;
        PGconn     *conn;
        PGresult   *res;
        PGnotify   *notify;
        int                     nnotifies;

        /*
         * If the user supplies a parameter on the command line, use it as
         * the conninfo string; otherwise default to setting dbname=template1
         * and using environment variables or defaults for all other connection
         * parameters.
         */
        if (argc > 1)
                conninfo = argv[1];
        else
                conninfo = "dbname = template1";

        /* Make a connection to the database */
        conn = PQconnectdb(conninfo);
```

```
/* Check to see that the backend connection was successfully made */
if (PQstatus(conn) != CONNECTION_OK)
{
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
}

/*
 * Issue LISTEN command to enable notifications from the rule's NOTIFY.
 */
res = PQexec(conn, "LISTEN TBL2");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
        fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
}

/*
 * should PQclear PGresult whenever it is no longer needed to avoid
 * memory leaks
 */
PQclear(res);

/* Quit after four notifies are received. */
nnotifies = 0;
while (nnotifies < 4)
{
        /*
         * Sleep until quelquechose happens on the connection.  We use select(2)
         * to wait for input, but you could also use poll() or similar
         * facilities.
         */
        int                     sock;
        fd_set          input_mask;

        sock = PQsocket(conn);

        if (sock < 0)
                break;                          /* shouldn't happen */

        FD_ZERO(&input_mask);
        FD_SET(sock, &input_mask);

        if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
        {
                fprintf(stderr, "select() failed: %s\n", strerror(errno));
                exit_nicely(conn);
        }

        /* Now check for input */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
                fprintf(stderr,
                                "ASYNC NOTIFY of '%s' received from backend pid %d\n",
                                notify->relname, notify->be_pid);
                PQfreemem(notify);
                nnotifies++;
        }
}
```

```
        fprintf(stderr, "Done.\n");

        /* close the connection to the database and cleanup */
        PQfinish(conn);

        return 0;
}
```

**Exemple 27–3. Troisième exemple de programme pour libpq**

```
/*
 * testlibpq3.c
 *              Test out-of-line parameters and binary I/O.
 *
 * Before running this, populate a database with the following commands
 * (provided in src/test/examples/testlibpq3.sql):
 *
 * CREATE TABLE test1 (i int4, t text, b bytea);
 *
 * INSERT INTO test1 values (1, 'joe''s place', '\\000\\001\\002\\003\\004');
 * INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\\000');
 *
 * The expected output is:
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(PGconn *conn)
{
        PQfinish(conn);
        exit(1);
}

int
main(int argc, char **argv)
{
        const char *conninfo;
        PGconn     *conn;
        PGresult   *res;
        const char *paramValues[1];
        int                     i,
                                j;
        int                     i_fnum,
                                t_fnum,
```

```
                        b_fnum;

/*
 * If the user supplies a parameter on the command line, use it as
 * the conninfo string; otherwise default to setting dbname=template1
 * and using environment variables or defaults for all other connection
 * parameters.
 */
if (argc > 1)
        conninfo = argv[1];
else
        conninfo = "dbname = template1";

/* Make a connection to the database */
conn = PQconnectdb(conninfo);

/* Check to see that the backend connection was successfully made */
if (PQstatus(conn) != CONNECTION_OK)
{
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
}

/*
 * The point of this program is to illustrate use of PQexecParams()
 * with out-of-line parameters, as well as binary transmission of
 * results.  By using out-of-line parameters we can avoid a lot of
 * tedious mucking about with quoting and escaping.  Notice how we
 * don't have to do anything special with the quote mark in the
 * parameter value.
 */

/* Here is our out-of-line parameter value */
paramValues[0] = "joe's place";

res = PQexecParams(conn,
                                    "SELECT * FROM test1 WHERE t = $1",
                                    1,          /* one param */
                                    NULL,       /* let the backend deduce param type */
                                    paramValues,
                                    NULL,       /* don't need param lengths since text */
                                    NULL,       /* default to all text params */
                                    1);         /* ask for binary results */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
}

/* Use PQfnumber to avoid assumptions about field order in result */
i_fnum = PQfnumber(res, "i");
t_fnum = PQfnumber(res, "t");
b_fnum = PQfnumber(res, "b");

for (i = 0; i < PQntuples(res); i++)
{
        char       *iptr;
        char       *tptr;
        char       *bptr;
```

```
                int                     blen;
                int                     ival;

                /* Get the field values (we ignore possibility they are NULL!) */
                iptr = PQgetvalue(res, i, i_fnum);
                tptr = PQgetvalue(res, i, t_fnum);
                bptr = PQgetvalue(res, i, b_fnum);

                /*
                 * The binary representation of INT4 is in network byte order,
                 * which we'd better coerce to the local byte order.
                 */
                ival = ntohl(*((uint32_t *) iptr));

                /*
                 * The binary representation of TEXT is, well, text, and since
                 * libpq was nice enough to append a zero byte to it, it'll work
                 * just fine as a C string.
                 *
                 * The binary representation of BYTEA is a bunch of bytes, which
                 * could include embedded NULLs so we have to pay attention to
                 * field length.
                 */
                blen = PQgetlength(res, i, b_fnum);

                printf("tuple %d: got\n", i);
                printf(" i = (%d bytes) %d\n",
                           PQgetlength(res, i, i_fnum), ival);
                printf(" t = (%d bytes) '%s'\n",
                           PQgetlength(res, i, t_fnum), tptr);
                printf(" b = (%d bytes) ", blen);
                for (j = 0; j < blen; j++)
                        printf("\\%03o", bptr[j]);
                printf("\n\n");
        }

        PQclear(res);

        /* close the connection to the database and cleanup */
        PQfinish(conn);

        return 0;
}
```

# Chapitre 28. Objets larges

PostgreSQL a des fonctionnalités pour les *objets larges*, fournissant un accès style flux aux données utilisateurs stockées dans une structure spéciale. L'accès en flux est utile pour travailler avec des valeurs de données trop larges pour être manipuler convenablement en entier.

Ce chapitre décrit l'implémentation, la programmation et les interfaces du langage de requêtes pour les données de type objet large de PostgreSQL. Nous utilisons la bibliothèque C libpq pour les exemples de ce chapitre mais la plupart des interfaces natives de programmation de PostgreSQL supportent des fonctionnalités équivalentes. D'autres interfaces pourraient utiliser l'interface des objets larges en interne pour fournir un support générique des valeurs larges. Ceci n'est pas décrit ici.

## 28.1. Historique

POSTGRES 4.2, le prédécesseur indirect de PostgreSQL, supportait trois implémentations standards des objets larges : par des fichiers externes au serveur POSTGRES, par des fichiers externes gérés par le serveur POSTGRES et par des données stockées dans la base de données POSTGRES. Ceci a causé une énorme confusion parmi les utilisateurs. Du coup, seul le support des objets larges en tant que donnée stockée dans la base de données a été conservé dans PostgreSQL. Bien qu'il soit plus lent en accès, il fournit une intégrité stricte des données. Pour des raisons historiques, le schéma de stockage est référencé comme *Inversion large objects*. (Vous apercevrez le terme inversion utilisé occasionnellement pour signifier aussi un objet large.) Depuis PostgreSQL 7.1, tous les objets larges sont placés dans une table système appelée `pg_largeobject`.

PostgreSQL 7.1 a introduit un mécanisme (nom de code << TOAST >>) qui permet à des données d'être bien plus larges que des pages de données individuelles. Ceci rend l'interface des objets larges partiellement obsolète. Un des avantages restant de l'interface des objets larges est qu'il permet des tailles importantes, avec un maximum de 2 Go alors que les champs TOAST ne peuvent gérer qu'un maximum de 1 Go. De plus, les objets larges peuvent être manipulés élément par élément bien plus facilement que des champs ordinaires de données, donc les limites pratiques sont considérablement différentes.

## 28.2. Fonctionnalités d'implémentation

L'implémentation des objets larges coupe les objets larges en << morceaux >> (*chunks*) et stocke les morceaux dans les lignes de la base de données. Un index B−tree garantit des recherches rapides pour le bon numéro du morceau lors d'accès aléatoires en lecture et écriture.

## 28.3. Interfaces client

Cette section décrit les possibilités que les bibliothèques d'interfaces client de PostgreSQL fournissent pour accéder aux objets larges. Toutes les manipulations d'objets larges utilisant ces fonctions *doivent* prendre place dans un bloc de transaction SQL. (Ce prérequis est forcé strictement avec PostgreSQL 6.5 bien que c'était un prérequis implicite dans les versions précédentes, ce qui a conduit à des mauvais comportements si celui−ci était ignoré.) L'interface des objets larges de PostgreSQL prend comme modèle l'interface des systèmes de fichiers Unix avec des fonctions analogues pour `open`, `read`, `write`, `lseek`, etc.

Les applications clients utilisant l'interface des objets larges dans libpq devraient inclure le fichier d'en–tête `libpq/libpq-fs.h` et établir un lien avec la bibliothèque libpq.

## 28.3.1. Créer un objet large

La fonction

```
Oid lo_creat(PGconn *conn, int mode);
```

crée un nouvel objet large. *`mode`* est un masque décrivant les différents attributs du nouvel objet. Les constantes symboliques utilisées ici sont définies dans le fichier d'en–tête `libpq/libpq-fs.h`. Le type d'accès (lecture, écriture ou les deux) est contrôlé par une combinaison OU des bits `INV_READ` et `INV_WRITE`. Les 16 bits de poids faible du masque ont été utilisés historiquement à Berkeley pour désigner le numéro du gestionnaire de stockage sur lequel l'objet large devrait résider. Ces bits devraient toujours être à zéro maintenant. (Le type d'accès ne fait réellement rien de plus mais un ou deux des bits doivent être configurés pour éviter une erreurs.) La valeur de retour est l'OID assigné au nouvel objet large ou InvalidOid (zéro) en cas d'erreur.

Un exemple :

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

## 28.3.2. Importer un objet large

Pour importer un fichier du système d'exploitation en tant qu'objet large, appelez

```
Oid lo_import(PGconn *conn, const char *filename);
```

*`filename`* spécifie le nom du fichier à importer comme objet large. Le code de retour est l'OID assigné au nouvel objet large ou InvalidOid (zero) en cas d'échec. Notez que le fichier est lu par la bibliothèque d'interface du client, pas par le serveur. Donc il doit exister dans le système de fichier du client et lisible par l'application du client.

## 28.3.3. Exporter un objet large

Pour exporter un objet large en tant que fichier du système d'exploitation, appelez

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

L'argument `lobjId` spécifie l'OID de l'objet large à exporter et l'argument `filename` spécifie le nom du fichier. Notez que le fichier est écrit par la bibliothèque d'interface du client, pas par le serveur. Renvoie 1 en cas de succès, –1 en cas d'échec.

## 28.3.4. Ouvrir un objet large existant

Pour ouvrir un objet large existant pour lire ou écrire, appelez

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

L'argument lobjId spécifie l'OID de l'objet large à ouvrir. Les bits mode contrôlent si l'objet est ouvert en lecture (INV_READ), écriture (INV_WRITE) ou les deux. Un objet large ne peut pas être ouvert avant d'avoir été créé. lo_open renvoie un descripteur (positif) d'objet large pour une utilisation future avec lo_read, lo_write, lo_lseek, lo_tell et lo_close. Le descripteur est uniquement valide pour la durée de la transaction en cours. En cas d'échec, −1 est renvoyé.

## 28.3.5. Écrire des données dans un objet large

La fonction

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

écrit len octets de buf dans le descripteur fd de l'objet large. L'argument fd doit avoir été renvoyé par un appel précédent à lo_open. Le nombre d'octets réellement écrits est renvoyé. Dans le cas d'une erreur, une valeur négative est renvoyée.

## 28.3.6. Lire des données à partir d'un objet large

La fonction

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

lit len octets du descripteur de l'objet large fd et les place dans buf. L'argument fd doit avoir été renvoyé par un appel précédent à lo_open. Le nombre d'octets réellement lus est renvoyé. Dans le cas d'une erreur, une valeur négative est renvoyée.

## 28.3.7. Recherche dans un objet large

Pour modifier l'emplacement courant de lecture ou écriture associé au descripteur d'un objet large, appelez

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

Cette fonction déplace le pointeur d'emplacement courant pour le descripteur de l'objet large identifié par fd au nouvel emplacement spécifié avec le décalage (offset). Les valeurs valides pour whence sont SEEK_SET (rechercher depuis le début de l'objet), SEEK_CUR (rechercher depuis la position courante) et SEEK_END (rechercher depuis la fin de l'objet). Le code de retour est le nouvel emplacement du pointeur ou −1 en cas d'erreur.

## 28.3.8. Obtenir la position de recherche d'un objet large

Pour obtenir la position actuelle de lecture ou écriture d'un descripteur d'objet large, appelez

```
int lo_tell(PGconn *conn, int fd);
```

S'il y a une erreur, le code de retour est négatif.

### 28.3.9. Fermer un descripteur d'objet large

Un descripteur d'objet large peut être fermé en appelant

```
int lo_close(PGconn *conn, int fd);
```

où `fd` est un descripteur d'objet large renvoyé par `lo_open`. En cas de succès, `lo_close` renvoie zéro. Renvoie 1 en cas de succès, −1 en cas d'échec.

Tous les descripteurs d'objets larges restant ouverts à la fin d'une transaction seront automatiquement fermés.

### 28.3.10. Supprimer un objet large

Pour supprimer un objet large de la base de données, appelez

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

L'argument `lobjId` spécifie l'OID de l'objet large à supprimer. Dans le cas d'une erreur, le code de retour est négatif.

## 28.4. Fonctions du côté serveur

Ce sont des fonctions côté serveur appelables à partir de SQL et correspondant à chaque fonction côté client décrite ci−dessus ; en fait, pour leur grande part, les fonctions côté client sont simplement des interfaces vers les fonctions équivalentes côté serveur. Celles qui sont réellement utiles à appeler via des commandes SQL sont `lo_creat`, `lo_unlink`, `lo_import` et `lo_export`. Voici des exemples de leur utilisation :

```
CREATE TABLE image (
    nom             text,
    donnees         oid
);

SELECT lo_creat(-1);       -- renvoie l'OID du nouvel objet large

SELECT lo_unlink(173454);  -- supprime l'objet large d'OID 173454

INSERT INTO image (nom, donnees)
    VALUES ('superbe image', lo_import('/etc/motd'));

SELECT lo_export(image.donnees, '/tmp/motd') FROM image
    WHERE nom = 'superbe image';
```

Les fonctions `lo_import` et `lo_export` côté serveur se comportent considérablement différemment de leurs analogues côté client. Ces deux fonctions lisent et écrivent des fichiers dans le système de fichiers du serveur, en utilisant les droits du propriétaire du serveur de base de données. Du coup, leur utilisation est restreinte aux superutilisateurs. Au contraire des fonctions côté serveur, les fonctions d'import et d'export côté client lisent et écrivent des fichiers dans le système de fichiers du client en utilisant les droits du programme client. Les fonctions côté client peuvent être utilisées par tout utilisateur PostgreSQL.

# 28.5. Programme d'exemple

Exemple 28−1 est un programme d'exemple qui montre une utilisation de l'interface des objets larges avec libpq. Des parties de ce programme disposent de commentaires au bénéfice de l'utilisateur. Ce programme est aussi disponible dans la distribution des sources (`src/test/examples/testlo.c`).

**Exemple 28−1. Exemple de programme sur les objets larges avec libpq**

```
/*--------------------------------------------------------------
 *
 * testlo.c--
 *    test utilisant des objets larges avec libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *--------------------------------------------------------------
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile
 *    importe le fichier "in_filename" dans la base de données
 *    en tant qu'objet "lobjOid"
 *
 */
Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * ouvre le fichier à lire
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n", filename);
    }

    /*
     * crée l'objet large
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "can't create large object\n");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * lit le fichier Unix écrit dans le fichier inversion
```

```
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading large object\n");
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = ' ';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
```

```
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile *    exporte l'objet large "lobjOid" dans le fichier
 * "out_filename"
 *
 */
void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * crée un "objet" inversion
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    /*
     * ouvre le fichier à écrire
     */
    fd = open(filename, O_CREAT | O_WRONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n",
                filename);
    }

    /*
     * lit à partir du fichier Unix et écrit dans le fichier inversion
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing %s\n",
                    filename);
        }
```

```
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * initialise la connexion
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", database);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);

    printf("importing file %s\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
/*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X's\n");
```

```
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file %s\n", out_filename);
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

# Chapitre 29. ECPG – SQL embarqué dans du C

Ce chapitre décrit l'interface SQL embarqué pour PostgreSQL. Il a été écrit par Linus Tolke (<linus@epact.se>) et Michael Meskes (<meskes@postgresql.org>). Originellement, il a été écrit pour fonctionner avec le langage C. Il fonctionne aussi avec le C++ mais il ne reconnaît pas encore toutes les constructions C++.

Cette documentation est assez incomplète. Mais du fait de la standardisation de cette interface, des informations complémentaires sont disponibles à travers de nombreuses ressources traitant du SQL.

## 29.1. Concept

Un programme SQL embarqué consiste en du code écrit dans un langage de programmation ordinaire, dans le cas présent, le C, mélangé à des commandes SQL incluses dans des sections spécialement marquées. Pour construire le programme, le code source est d'abord passé au préprocesseur SQL embarqué qui le convertit en un programme C ordinaire. Il peut alors être traité par un compilateur C.

Le SQL embarqué a des avantages par rapport aux autres méthodes de gestion de commandes SQL dans du code C. Premièrement, il gère le passage laborieux des informations de et vers les variables du programme C. Deuxièmement, le code SQL du programme est vérifié syntaxiquement au moment de la construction. Troisièmement, le SQL embarqué en C est spécifié dans le standard SQL et supporté par de nombreux systèmes de bases de données SQL. L'implémentation PostgreSQL est conçue pour correspondre au mieux à ce standard. Il est de ce fait assez facile de porter les programmes SQL embarqués écrits pour d'autres bases de données SQL vers PostgreSQL.

Comme indiqué précédemment, les programmes écrits pour l'interface SQL embarqué sont des programmes C normaux contenant un code spécial inséré pour réaliser les actions en relation avec la base de données. Ce code spécial a toujours la forme

```
EXEC SQL ...;
```

Ces instructions prennent syntaxiquement la place d'une instruction C. Suivant l'instruction particulière, elles peuvent apparaître dans le contexte global ou à l'intérieur d'une fonction. Les instructions SQL embarquées suivent les règles de sensibilité à la casse d'un code SQL normal, et non pas ceux du C.

Les sections suivantes expliquent toutes les instructions SQL embarquées.

## 29.2. Se connecter au serveur de bases de données

La connexion à une base de données se fait en utilisant l'instruction suivante :

```
EXEC SQL CONNECT TO cible [AS
nom-connexion] [USER
nom-utilisateur];
```

La *cible* peut être spécifiée d'une des façons suivantes :

- *nom_base*[@*nomhôte* ][:*port*]

- `tcp:postgresql://`*`nomhôte`* `[:`*`port`* `] [/`*`nom_base`*`][? `*`options`*`]`
- `unix:postgresql://`*`nomhôte`*`[: `*`port`*`][/`*`nom_base`* `][? `*`options`*`]`
- une chaîne SQL littérale contenant une des formes précédentes
- une référence à une variable contenant une des formes précédentes (voir les exemples)
- `DEFAULT`

Si la cible de connexion est spécifiée littéralement (c'est–à–dire non pas via une variable de référence) et la valeur n'est pas mise entre guillemets, les règles d'insensibilité à la casse du SQL standard sont appliquées. Dans ce cas, il est possible, si cela s'avérait nécessaire, d'encadrer séparément les paramètres individuels de guillemets doubles. En pratique, l'utilisation d'une chaîne littérale (entre guillemets simples) ou d'une variable de référence engendre moins d'erreurs. La cible de connexion `DEFAULT` initie une connexion à la base de données standard avec l'utilisateur standard. Aucun nom d'utilisateur ou de connexion ne peut être spécifié isolément dans ce cas.

Il existe également différentes façons de préciser le nom de l'utilisateur :

- *`nomutilisateur`*
- *`nomutilisateur/ motdepasse`*
- *`nomutilisateur`* `IDENTIFIED BY` *`motdepasse`*
- *`nomutilisateur`* `USING` *`motdepasse`*

Comme indiqué ci–dessus, les paramètres *`nomutilisateur`* et *`motdepasse`* peuvent être un identificateur SQL, une chaîne SQL littérale ou une référence à une variable de type caractère.

*`nom-connexion`* est utilisé pour gérer plusieurs connexions dans un même programme. Il peut être omis si un programme n'utilise qu'une seule connexion. La connexion la plus récemment ouverte devient la connexion courante, utilisée par défaut lorsqu'une instruction SQL est à exécuter (voir plus loin dans ce chapitre).

Voici quelques exemples d'instructions `CONNECT` :

```
EXEC SQL CONNECT TO ma_base@sql.mondomaine.com;

EXEC SQL CONNECT TO 'unix:postgresql://sql.mondomaine.com/ma_base' AS
maconnexion USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *cible = "ma_base@sql.mondomaine.com";
const char *utilisateur = "john";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :cible USER :utilisateur;
```

La dernière forme utilise la variante dite de la variable de référence, à laquelle il est fait allusion ci–dessus. Nous verrons dans les prochaines sections comment utiliser des variables C dans des instructions SQL en les préfixant par un caractère double–points.

Il est à noter que le format de la cible de connexion n'est pas spécifié dans le standard SQL. Ainsi, lorsque l'on souhaite développer des applications portables, il est préférable d'utiliser une syntaxe basée sur le dernier exemple ci–dessus pour encapsuler la chaîne de la cible de connexion.

## 29.3. Fermer une connexion

Pour fermer une connexion, l'instruction suivante est utilisée :

```
EXEC SQL DISCONNECT [connexion];
```

*connexion* peut être spécifiée de différentes façons :

- *nom-connexion*
- DEFAULT
- CURRENT
- ALL

Si aucun nom de connexion n'est spécifié, la connexion en cours est fermée.

Il est toujours préférable qu'une application ferme explicitement chaque connexion qu'elle a ouverte.

## 29.4. Exécuter des commandes SQL

Toute commande SQL peut être exécutée à l'intérieur d'une application SQL embarquée. Ci-dessous se trouvent quelques exemples de façons de procéder.

Création d'une table :

```
EXEC SQL CREATE TABLE foo (nombre integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(nombre);
EXEC SQL COMMIT;
```

Insertion de lignes :

```
EXEC SQL INSERT INTO foo (nombre, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Suppression de lignes :

```
EXEC SQL DELETE FROM foo WHERE nombre = 9999;
EXEC SQL COMMIT;
```

Sélection d'une ligne :

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Sélection utilisant des curseurs :

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT nombre, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

Mises à jour :

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE nombre = 9999;
EXEC SQL COMMIT;
```

Les marques de la forme `:quelquechose` sont des *variables hôtes*, c'est–à–dire qu'elles font référence à des variables dans le programme C. Elles sont expliquées dans Section 29.6.

Dans le mode par défaut, les instructions ne sont validées que lorsque `EXEC SQL COMMIT` est exécuté. L'interface SQL embarquée supporte aussi la validation automatique des transactions (similaire au comportement de libpq) via l'option `-t` en ligne de commande pour `ecpg` (voir ci–dessous) ou via l'instruction `EXEC SQL SET AUTOCOMMIT TO ON`. En mode de validation automatique, chaque commande est automatiquement validée sauf si elle est à l'intérieur d'un bloc de transaction explicite. Ce mode peut être explicitement désactivé en utilisant `EXEC SQL SET AUTOCOMMIT TO OFF`.

## 29.5. Choisir une connexion

Les instructions SQL affichées dans la section précédente sont exécutées à partir de la connexion courante, c'est–à–dire la dernière à avoir été ouverte. Il y a deux façons de gérer l'utilisation de plusieurs connexions dans une application.

La première option est de choisir explicitement une connexion pour chaque instruction SQL, par exemple

```
EXEC SQL AT nom-connexion SELECT ...;
```

Cette option est particulièrement adaptée si l'application a besoin d'utiliser plusieurs connexions en ordre divers.

Si votre application utilise plusieurs threads d'exécution, ils ne peuvent pas partager de connexion. Vous devez soit contrôler explicitement l'accès à la connexion (en utilisant des mutex) soit utiliser une connexion pour chaque thread. Si chaque thread utilise sa propre connexion, vous aurez besoin d'utiliser la clause AT pour spécifier quelle connexion le thread utilisera.

La seconde option est d'exécuter une instruction pour basculer la connexion courante. L'instruction est :

```
EXEC SQL SET CONNECTION connection-name;
```

Cette option est particulièrement intéressante si un grand nombre d'instructions doivent être exécutées à partir de la même connexion. Elle ne tient pas compte des threads.

## 29.6. Utiliser des variables hôtes

Dans la section Section 29.4, nous avons vu comment exécuter des instructions SQL à partir d'un programme SQL embarqué. Quelques–unes de ces instructions n'utilisent que des valeurs fixes. Elles n'offrent pas la possibilités d'insérer des valeurs fournies par l'utilisateur dans les instructions. Elles ne permettent pas non plus au programme de traiter les valeurs renvoyées par la requête. Ces types d'instructions ne sont pas vraiment utiles dans les applications réelles. Cette section explique en détail comment échanger des données entre votre programme C et les instructions SQL embarquées en utilisant un mécanisme simple appelé

*variables hôtes*.

## 29.6.1. Aperçu

Echanger des données entre le programme C et les instructions SQL est particulièrement simple en SQL embarqué. Plutôt que de laisser le programme copier les données dans l'instruction, ce qui implique un certain nombre de complications, dont la bonne mise entre guillemets de la valeur, il est plus simple d'écrire le nom de la variable C dans l'instruction SQL en la préfixant par un caractère deux−points. Par exemple:

```
EXEC SQL INSERT INTO unetable VALUES (:v1, 'foo', :v2);
```

Cette instruction fait référence à deux variables C nommées `v1` et `v2` et utilise également une chaîne littérale SQL pour illustrer l'absence de restriction à l'utilisation d'un type de données ou d'un autre.

Ce style d'insertions de variables C dans des instructions SQL fonctionne dans tous les cas où l'on attend une expression de valeur dans une instruction SQL. Dans l'environnement SQL, nous appelons les références à des variables C des *variables hôtes*.

## 29.6.2. Sections de déclaration

Pour passer des données du programme à la base de données, comme paramètre d'une requête par exemple, ou pour passer des données de la base au programme, les variables C supposées contenir ces données doivent être déclarées dans des sections spécialement marquées pour que le préprocesseur du SQL embarqué soit averti de leur présence.

Cette section commence avec

```
EXEC SQL BEGIN DECLARE SECTION;
```

et se termine avec

```
EXEC SQL END DECLARE SECTION;
```

Entre ces lignes, on trouvera des déclarations normales de variables C, comme

```
int   x;
char  foo[16], bar[16];
```

Il peut y avoir autant de sections de déclarations dans un programme que souhaité.

Les declarations sont aussi placées dans le fichier de sortie comme des variables C normales, du coup, il n'est plus besoin de les déclarer à nouveau. Les variables qui n'ont pas pour but d'être utilisées dans des commandes SQL peuvent être normalement déclarées en dehors des sections spéciales.

La définition d'une structure ou union doit aussi être saisie dans une section `DECLARE`. Sinon, le préprocesseur, ne connaissant pas leur définition, ne pourra pas gérer ces types.

Le type spécial `VARCHAR` est converti dans une `struct` nommée pour chaque variable. Une déclaration telle que

```
VARCHAR var[180];
```

est convertie en

```
struct varchar_var { int len; char arr[180]; } var;
```

Cette structure est utilisable pour créer une interface des données SQL de type `varchar`.

## 29.6.3. `SELECT INTO` et `FETCH INTO`

Nous savons maintenant insérer des données engendrées par un programme dans une commande SQL. Mais comment récupérer les résultats d'une requête ? Dans ce but, le SQL embarqué fournit des variantes spéciales des commandes habituelles `SELECT` et `FETCH`. Ces commandes ont une clause `INTO` particulière qui spécifie les variables hôtes dans lesquelles seront stockées les valeurs récupérées.

Voici un exemple :

```
/*
 * Soit la table suivante :
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

La clause `INTO` apparaît donc entre les champs du `select` et la clause `FROM`. Le nombre d'éléments dans la liste du `select` et celui de la liste après `INTO` (aussi appelée liste cible) doivent être identiques.

Voici un exemple utilisant la commande `FETCH` :

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do {
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

Ici, la clause `INTO` apparaît après toutes les autres clauses.

Ces deux méthodes ne permettent de récupérer qu'une ligne à la fois. Pour traiter des ensembles de résultats contenant potentiellement plus d'une ligne, il faut utiliser un curseur, comme indiqué dans le second exemple.

### 29.6.4. Indicateurs

Les exemples ci−dessus ne gèrent pas les valeurs NULL. En fait, ces exemples de récupération afficheront une erreur s'ils récupèrent une valeur NULL à partir de la base de données. Pour être capable de passer des valeurs NULL à la base de données ou de récupérer des valeurs NULL de la base de données, il est nécessaire d'ajouter une deuxième spécification de variable hôte pour chaque variable hôte contenant des données. Cette seconde variable est appelée l'*indicateur* et contient un drapeau indiquant si la valeur est NULL, auquel cas la valeur de la variable hôte réelle est ignorée. Voici un exemple qui gère correctement la récupération de valeurs NULL :

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

La variable indicateur `val_ind` vaudra zéro si la valeur est non NULL et elle sera négative si la valeur est NULL.

L'indicateur a une autre fonction : si la valeur de l'indicateur est positive, cela signifie que la valeur est non NULL mais qu'elle a été tronquée lors de son stockage dans la variable hôte.

## 29.7. SQL dynamique

Dans de nombreux cas, les instructions SQL particulières qu'une application doit exécuter sont connues au moment de l'écriture de l'application. Néanmoins, dans certains cas, les instructions SQL sont composées à l'exécution ou fournies par une source externe. Dans ces cas, il n'est pas possible d'embarquer directement les instructions SQL dans le code source C. Pour ce faire, il existe une fonction permettant d'appeler des instructions SQL arbitraires fournies par l'intermédiaire d'une variable de type chaîne.

La façon la plus simple d'exécuter une instruction SQL arbitraire est d'utiliser la commande `EXECUTE IMMEDIATE`. Par exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

Les instructions de ce type ne peuvent pas être utilisées pour récupérer des données (c'est−à−dire un `SELECT`).

Une façon plus puissante d'exécuter des instructions SQL arbitraires est de les préparer une seule fois et de les exécuter ensuite aussi souvent que nécessaire. Il est également possible de préparer une version généralisée d'une instruction, puis d'exécuter les versions spécifiques en substituant les paramètres. Lors de la préparation de l'instruction, il suffit d'écrire des points d'interrogation aux endroits où des paramètres seront substitués par la suite. Par exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
```

```
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

Si l'instruction exécutée retourne des valeurs, il es nécessaire d'ajouter une clause `INTO` :

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3;
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO v1, v2, v3 USING 37;
```

Une commande `EXECUTE` peut avoir une clause `INTO`, une clause `USING`, les deux ou encore aucune.

Lorsqu'une instruction préparée n'est plus utile, il est préférable de la désallouer :

```
EXEC SQL DEALLOCATE PREPARE name;
```

## 29.8. Utiliser les zones des descripteurs SQL

Une zone de descripteur SQL est une méthode plus sophistiquée pour traiter le résultat d'un `SELECT` ou d'un `FETCH`. La zone du descripteur SQL groupe les données d'une ligne avec les éléments de métadonnées en une seule structure de données. Les métadonnées sont particulièrement utiles lors de l'exécution d'instructions SQL dynamiques, pour lesquelles la nature des colonnes de résultats n'est pas forcément connue à l'avance.

Une zone de descripteur SQL consiste en un en−tête, contenant des informations sur le descripteur complet, et un ou plusieurs éléments des zones du descripteur, décrivant basiquement une colonne de la ligne de résultat.

Avant d'utiliser une zone de descripteur SQL, il est nécessaire d'en allouer une :

```
EXEC SQL ALLOCATE DESCRIPTOR identifiant;
```

L'identifiant sert de << nom de variable >> à la zone du descripteur. Lorsque le descripteur n'est plus utilisé, il est recommandé de le désallouer :

```
EXEC SQL DEALLOCATE DESCRIPTOR identifiant;
```

Pour utiliser la zone d'un descripteur, il faut le spécifier comme cible de stockage dans une clause `INTO`, à la place de la liste des variables hôtes :

```
EXEC SQL FETCH NEXT FROM moncurseur INTO DESCRIPTOR mondesc;
```

Comment récupérer les données de la zone du descripteur ? Celle−ci peut être envisagée comme une structure contenant des champs nommés. Pour récupérer la valeur d'un champ à partir de l'en−tête et la stocker dans une variable hôte, on utilise la commande suivante :

```
EXEC SQL GET DESCRIPTOR nom
:varhote = champ;
```

Actuellement, il n'existe qu'un seul champ d'en−tête défini : *COUNT*, qui indique le nombre d'éléments dans la zone de descripteur (c'est−à−dire le nombre de colonnes contenues dans le résultat). La variable hôte nécessite d'être du type entier. Pour récupérer un champ à partir de l'élément de la zone du descripteur, on utilise la commande suivante :

```
EXEC SQL GET DESCRIPTOR nom VALUE
numero :varhote =
champ;
```

*numero* peut être un entier littéral ou une variable hôte contenant un entier. Les champs possibles sont :

`CARDINALITY` (integer)
: nombre de lignes dans l'ensemble du résultat

`DATA`
: élément de données en cours (de fait, le type de données de ce champ dépend de la requête)

`DATETIME_INTERVAL_CODE` (integer)
: ?

`DATETIME_INTERVAL_PRECISION` (integer)
: non implémenté

`INDICATOR` (integer)
: l'indicateur (indiquant une valeur NULL ou une troncature de la valeur)

`KEY_MEMBER` (integer)
: non implémenté

`LENGTH` (integer)
: longueur de la donnée en caractères

`NAME` (string)
: nom de la colonne

`NULLABLE` (integer)
: non implémenté

`OCTET_LENGTH` (integer)
: longueur en octets de la représentation en caractères de la donnée

`PRECISION` (integer)
: précision (pour le type `numeric`)

`RETURNED_LENGTH` (integer)
: longueur de la donnée en caractère

`RETURNED_OCTET_LENGTH` (integer)
: longueur en octets de la représentation en caractères de la donnée

`SCALE` (integer)
: échelle (pour le type `numeric`)

`TYPE` (integer)
: code numérique du type de données de la colonne

## 29.9. Gestion des erreurs

Cette section décrit la gestion des conditions exceptionnelles et des avertissements dans un programme SQL embarqué. Il existe plusieurs fonctions non exclusives pour cela.

## 29.9.1. Configurer des rappels

Une méthode simple de récupération des erreurs et des avertissements consiste à configurer une action spécifique à exécuter à chaque fois qu'une condition particulière survient. En général :

```
EXEC SQL WHENEVER condition action;
```

*condition* peut prendre une des valeurs suivantes :

SQLERROR
: L'action spécifiée est appelée lorsqu'une erreur survient pendant l'exécution d'une instruction SQL.

SQLWARNING
: L'action spécifiée est appelée lorsqu'un avertissement survient pendant l'exécution d'une instruction SQL.

NOT FOUND
: L'action spécifiée est appelée lorsqu'une instruction ne récupère ou n'affecte aucune ligne. (Cette condition n'est pas une erreur mais il peut être intéressant de la gérer de façon particulière.)

*action* peut avoir une des valeurs suivantes :

CONTINUE
: Signifie effectivement que la condition est ignorée. Ceci est la valeur par défaut.

GOTO *label*
GO TO *label*
: Saute au label spécifié (en utilisant une instruction C `goto`).

SQLPRINT
: Affiche un message sur la sortie standard. Ceci est utile pour des programmes simples ou lors d'un prototypage. Les détails du message ne peuvent pas être configurés.

STOP
: Appel de `exit(1)`, ce qui terminera le programme.

BREAK
: Exécute l'instruction C `break`. Ceci devrait être utilisé uniquement dans des boucles ou dans des instructions `switch`.

CALL *nom* (*args*)
DO *nom* (*args*)
: Appelle les fonctions C spécifiées avec les arguments spécifiés.

Le standard SQL ne définit que les actions `CONTINUE` et `GOTO` (et `GO TO`).

Voici un exemple utilisable dans un programme simple. Il affiche un message lorsqu'un avertissement survient et termine le programme quand une erreur se produit.

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

L'instruction `EXEC SQL WHENEVER` est une directive du préprocesseur SQL, pas une instruction C. Les actions sur les erreurs et avertissements qu'elle définie s'appliquent à toutes les instructions SQL embarquées qui apparaissent avant l'endroit où le gestionnaire est défini, à moins qu'une action différente n'ait été définie pour la même condition entre le premier `EXEC SQL WHENEVER` et l'instruction SQL qui a engendré la condition, quelque soit le flux de contrôle du programme C. De ce fait, aucun des deux extraits de programme C qui suivent n'aura le comportement désiré.

```
/*
 * MAUVAIS
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}

/*
 * MAUVAIS
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

## 29.9.2. sqlca

Pour une gestion plus puissante des erreurs, l'interface du SQL embarqué fournit une variable globale de nom `sqlca` qui a la structure suivante :

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[70];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

(Dans un programme multithreadé, chaque thread obtient automatiquement sa propre copie de `sqlca`. Ceci fonctionne de façon similaire à la gestion de la variable globale C standard `errno`.)

`sqlca` couvre à la fois les avertissements et les erreurs. Si plusieurs avertissements ou erreurs surviennent lors de l'exécution d'une instruction, alors `sqlca` ne contiendra que les informations relatives à la dernière.

Si aucune erreur ne survient dans la dernière instruction SQL, `sqlca.sqlcode` vaudra 0 et `sqlca.sqlstate` vaudra `"00000"`. Si un avertissement ou une erreur a eu lieu, alors `sqlca.sqlcode` sera négatif et `sqlca.sqlstate` sera différent de `"00000"`. Un `sqlca.sqlcode` positif indique une condition sans dommage, telle que << aucune ligne renvoyée par la dernière requête >>. `sqlcode` et `sqlstate` sont deux schémas de code d'erreur différents ; les détails apparaissent ci−dessous.

Si la dernière instruction SQL a réussi, alors `sqlca.sqlerrd[1]` contient l'OID de la ligne traitée, si applicable, et `sqlca.sqlerrd[2]` contient le nombre de lignes traitées ou renvoyées, si applicable à la commande.

Dans le cas d'une erreur ou d'un avertissement, `sqlca.sqlerrm.sqlerrmc` contiendra une chaîne décrivant l'erreur. Le champ `sqlca.sqlerrm.sqlerrml` contient la longueur du message d'erreur stocké dans `sqlca.sqlerrm.sqlerrmc` (le résultat de `strlen()`, sans réel intérêt pour un programmeur C). Notez que certains messages sont trop longs pour entrer dans le tableau `sqlerrmc` de taille fixe ; ils seront tronqués.

Dans le cas d'un avertissement, `sqlca.sqlwarn[2]` est positionné à `W`. (Dans tous les autres cas, il est positionné à quelque chose de différent de `W`.) Si `sqlca.sqlwarn[1]` est positionné à `W`, alors une valeur a été tronquée lors de son stockage dans une variable hôte. `sqlca.sqlwarn[0]` est positionné à `W` si un autre élément est positionné pour indiquer un avertissement.

Les champs `sqlcaid`, `sqlcabc`, `sqlerrp`, et les éléments restant de `sqlerrd` et `sqlwarn` ne contiennent actuellement aucune information utile.

La structure `sqlca` n'est pas définie dans le standard SQL mais elle est implémentée dans plusieurs autres systèmes de bases de données SQL. Leurs définitions sont similaires dans leur esprit, mais l'écriture d'applications portables nécessite une étude attentive des autres implémentations.

### 29.9.3. `SQLSTATE` contre `SQLCODE`

Les champs `sqlca.sqlstate` et `sqlca.sqlcode` sont deux schémas différents fournissant des codes d'erreur. Les deux sont spécifiés dans le standard SQL mais `SQLCODE` est indiqué comme obsolète dans l'édition de 1992 du standard et a été supprimé dans celle de 1999. Du coup, il est fortement encouragé d'utiliser `SQLSTATE` dans les nouvelles applications.

`SQLSTATE` est un tableau de cinq caractères. Ces cinq caractères contiennent des chiffres ou des lettres en majuscules représentant les codes de différentes conditions d'erreurs ou d'avertissements. `SQLSTATE` dispose d'un schéma hiérarchique : les deux premiers caractères indiquent la classe générale de la condition, les trois derniers caractères indiquent une sous−classe de la condition générale. Un état de succès est indiqué par le code `00000`. Les codes `SQLSTATE` sont pour la plupart définis dans le standard SQL. Le serveur PostgreSQL supporte nativement les codes d'erreurs `SQLSTATE` ; du coup, un haut degré de cohérence peut être atteint en utilisant ce schéma de code d'erreur au travers de toutes vos applications. Pour plus d'informations, voir Annexe A.

`SQLCODE`, le schéma obsolète de codes d'erreurs, est un simple entier. Une valeur 0 indique un succès, une valeur positive indique un succès avec des informations supplémentaires, une valeur négative indique une erreur. Le standard SQL définit seulement la valeur positive +100, qui indique que la dernière commande n'a renvoyé ou modifié aucune ligne, et aucune valeur négative. Du coup, ce schéma n'est que faiblement portable et n'a pas d'affectation de code hiérarchique. Historiquement, le processeur de SQL embarqué pour PostgreSQL a affecté quelques valeurs `SQLCODE` spécifiques pour sa propre utilisation, qui sont listées

ci−dessous avec leurs valeurs numériques et leurs noms symboliques. Il est important de garder à l'esprit qu'elles ne sont pas portables vers d'autres implémentations SQL. Pour simplifier le portage d'applications au schéma `SQLSTATE`, le code `SQLSTATE` correspondant est également affiché. Néanmoins, il n'y a pas de correspondance une−à−une ou une−à−plusieurs entre les deux schémas (en fait, c'est plutôt plusieurs−à−plusieurs). Il est préférable de consulter le schéma `SQLSTATE` global dans Annexe A pour chaque cas.

Voici les valeurs affectées à `SQLCODE` :

−12 (`ECPG_OUT_OF_MEMORY`)
: Indique que la mémoire virtuelle est épuisée. (SQLSTATE YE001)

−200 (`ECPG_UNSUPPORTED`)
: Indique que le préprocesseur a engendré quelque chose que la bibliothèque ne connaît pas. Il peut s'agir de l'exécution de versions incompatibles du préprocesseur et de la bibliothèque. (SQLSTATE YE002)

−201 (`ECPG_TOO_MANY_ARGUMENTS`)
: Ceci signifie que la commande spécifie plus de variables hôtes que n'en attend la commande. (SQLSTATE 07001 or 07002)

−202 (`ECPG_TOO_FEW_ARGUMENTS`)
: Ceci signifie que la commande spécifie moins de variables hôtes que n'en attend la commande. (SQLSTATE 07001 or 07002)

−203 (`ECPG_TOO_MANY_MATCHES`)
: Ceci signifie qu'une requête a retourné plusieurs lignes mais que l'instruction n'était préparée à ne stocker qu'une ligne de résultat (par exemple, parce que les variables spécifiées ne sont pas des tableaux). (SQLSTATE 21000)

−204 (`ECPG_INT_FORMAT`)
: La variable hôte est de type `int` et la donnée de la base de données est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un `int`. La bibliothèque utilise `strtol()` pour cette conversion. (SQLSTATE 42804)

−205 (`ECPG_UINT_FORMAT`)
: La variable hôte est de type `unsigned int` et la donnée de la base de données est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un `unsigned int`. La bibliothèque utilise `strtoul()` pour cette conversion. (SQLSTATE 42804)

−206 (`ECPG_FLOAT_FORMAT`)
: La variable hôte est de type `float` et la donnée de la base de données est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un `float`. La bibliothèque utilise `strtod()` pour cette conversion. (SQLSTATE 42804)

−207 (`ECPG_CONVERT_BOOL`)
: Ceci signifie que la variable hôte est de type `bool` et que la donnée de la base de données n'est ni `'t'` ni `'f'`. (SQLSTATE 42804)

−208 (`ECPG_EMPTY`)
: L'instruction envoyée au serveur PostgreSQL était vide. (Ceci ne peut normalement pas survenir dans un programme SQL embarqué, cela peut donc indiquer une erreur interne.) (SQLSTATE YE002)

−209 (`ECPG_MISSING_INDICATOR`)
: Une valeur NULL a été retournée et aucune variable d'indicateur nul n'a été fournie. (SQLSTATE 22002)

−210 (`ECPG_NO_ARRAY`)
: Une variable ordinaire a été utilisée à un endroit qui requiert un tableau. (SQLSTATE 42804)

−211 (`ECPG_DATA_NOT_ARRAY`)
: La base de données a retourné une variable ordinaire à un endroit qui requiert une valeur de tableau. (SQLSTATE 42804)

−220 (ECPG_NO_CONN)
: Le programme a tenté d'accéder à une connexion qui n'existe pas. (SQLSTATE 08003)

−221 (ECPG_NOT_CONN)
: Le programme a tenté d'accéder à une connexion qui existe mais n'est pas ouverte. (Ceci est une erreur interne.) (SQLSTATE YE002)

−230 (ECPG_INVALID_STMT)
: Une tentative d'utilisation d'une instruction qui n'a pas été préparée est survenue. (SQLSTATE 26000)

−240 (ECPG_UNKNOWN_DESCRIPTOR)
: Le descripteur spécifié n'a pas été trouvé. Une tentative d'utilisation d'une instruction qui n'a pas été préparée est survenue. (SQLSTATE 33000)

−241 (ECPG_INVALID_DESCRIPTOR_INDEX)
: L'index du descripteur spécifié est hors échelle. (SQLSTATE 07009)

−242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM)
: Un élément invalide du descripteur a été demandé. (Ceci est une erreur interne.) (SQLSTATE YE002)

−243 (ECPG_VAR_NOT_NUMERIC)
: Lors de l'exécution d'une instruction dynamique, la base de données a retourné une valeur numérique et la variable hôte n'était pas numérique. (SQLSTATE 07006)

−244 (ECPG_VAR_NOT_CHAR)
: Lors de l'exécution d'une instruction dynamique, la base de données a retourné une valeur non numérique et la variable hôte était numérique. (SQLSTATE 07006)

−400 (ECPG_PGSQL)
: Une erreur a été causée par le serveur PostgreSQL. Le message contient le message d'erreur du serveur PostgreSQL.

−401 (ECPG_TRANS)
: Le serveur PostgreSQL a signalé que la transaction ne peut être commencée, validée ou annulée. (SQLSTATE 08007)

−402 (ECPG_CONNECT)
: La tentative de connexion à la base de données a échoué. (SQLSTATE 08001)

100 (ECPG_NOT_FOUND)
: Ceci est une condition sans gravité indiquant que la dernière commande n'a récupéré ou traité aucune ligne, ou que la fin du curseur est atteinte. (SQLSTATE 02000)

## 29.10. Inclure des fichiers

Pour inclure un fichier externe dans un programme SQL embarqué, il suffit d'utiliser :

```
EXEC SQL INCLUDE nomfichier;
```

Le préprocesseur du SQL embarqué cherchera un fichier nommé *nomfichier*.h, le traitera et l'inclura dans la sortie C résultante. Du coup, les instructions C embarquées du fichier inclus sont gérées correctement.

Il est à noter que ceci n'est *pas* la même chose que

```
#include <nomfichier.h>
```

parce que ce fichier ne serait pas sujet au pré−traitement des commandes SQL. Naturellement, la directive C #include peut toujours être utilisée pour inclure d'autres fichiers d'en−tête.

> **Note :** Le nom du fichier inclus est sensible à la casse, même si le reste de la commande EXEC SQL INCLUDE suit les règles habituelles de sensibilité à la casse.

# 29.11. Traiter les programmes comportant du SQL embarqué

Après avoir vu comment former des programmes C incluant du SQL embarqué, il est tout aussi intéressant de savoir comment les compiler. Avant d'être compilé, le fichier est passé au travers du préprocesseur C pour le SQL embarqué, qui convertit les instructions SQL utilisées en appels de fonctions spéciales. Après compilation, le programme doit être lié avec une bibliothèque spéciale contenant les fonctions nécessaires. Ces fonctions récupèrent l'information provenant des arguments, exécutent la commande SQL en utilisant l'interface libpq et placent le résultat dans les arguments spécifiés pour la sortie.

Le préprocesseur est appelé `ecpg` et est inclus dans une installation standard de PostgreSQL. Les programmes SQL embarqués sont nommés typiquement avec une extension `.pgc`. Un fichier programme nommé `prog1.pgc` peut être passé au préprocesseur par la simple commande

```
ecpg prog1.pgc
```

Ceci créera un fichier nommé `prog1.c`. Si les fichiers en entrée ne suivent pas le modèle de nommage suggéré, il est possible de spécifier explicitement le fichier de sortie en utilisant l'option `-o`.

Le fichier traité par le préprocesseur peut être compilé normalement. Par exemple :

```
cc -c prog1.c
```

Les fichiers sources en C engendrés incluent les fichiers d'en–tête provenant de l'installation de PostgreSQL, donc si PostgreSQL a été installé à un emplacement qui n'est pas parcouru par défaut, il faudra ajouter une option comme `-I/usr/local/pgsql/include` sur la ligne de commande de la compilation.

Pour lier un programme SQL embarqué, il faudra inclure la bibliothèque `libecpg` de cette façon :

```
cc -o monprog prog1.o prog2.o ... -lecpg
```

Là encore, il pourrait être nécessaire d'ajouter une option comme `-L/usr/local/pgsql/lib` sur la ligne de commande.

Si le processus de construction d'un grand projet est géré par l'utilisation de make, il pourrait être plus agréable d'inclure la règle implicite suivante dans les fichiers makefile :

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

La syntaxe complète de la commande `ecpg` est détaillée dans ecpg.

La bibliothèque ecpg est compatible avec les threads si elle a été compilée en utilisant l'option en ligne de commande `--enable-thread-safety` de `configure`. (Il pourrait s'avérer nécessaire de préciser d'autres options de threading sur la ligne de commande pour compiler le code client.)

## 29.12. Fonctions de la bibliothèque

La bibliothèque `libecpg` contient principalement des fonctions << cachées >> utilisées pour implémenter les fonctionnalités exprimées par les commandes SQL embarquées. Mais il existe quelques fonctions qu'il peut être utile d'appeler directement. Il est à noter que ceci rendra le code non portable.

- `ECPGdebug(int` *`on`*`, FILE *`*`flux`*`)` active le débogage s'il est appelé avec une valeur différente de zéro pour le premier argument. Les traces de débogages sont envoyées sur le *`flux`*. Les traces contiennent toutes les instructions SQL avec toutes les variables en entrée et les résultats du serveur PostgreSQL. Ceci peut être très utile pour rechercher des erreurs dans les instructions SQL.
- `ECPGstatus(int` *`no_ligne`*`, const char*` *`nom_connexion`*`)` retourne vrai si vous êtes connecté à une base de données, faux sinon. *`nom_connexion`* peut être `NULL` si une seule connexion est en cours d'utilisation.

## 29.13. Internes

Cette section explique comment ECPG fonctionne en interne. Cette information peut quelque fois être utile pour aider les utilisateurs à comprendre l'utilisation d'ECPG.

Les quatre premières lignes écrites par `ecpg` sur la sortie sont des lignes figées. Deux sont des commentaires et deux sont des lignes d'inclusion de fichiers d'en−tête nécessaires pour l'interfaçage avec la bibliothèque. Ensuite, le préprocesseur lit le fichier et écrit la sortie. Normalement, il répète tout sur la sortie.

Lorsqu'ecpg lit une instruction `EXEC SQL`, il intervient et la modifie. La commande commence avec `EXEC SQL` et se termine avec `;`. Tout ce qui se trouve entre est traité comme une instruction SQL et est analysé pour la substitution de variable.

La substitution de variable intervient quand un symbole commence avec un caractère deux−points (`:`). La variable possédant ce nom est recherchée parmi toutes les variables précédemment déclarées dans une section `EXEC SQL DECLARE`.

La fonction la plus importante de la bibliothèque est `ECPGdo`, qui prend en charge l'exécution de la plupart des commandes. Elle prend un nombre variable d'arguments. Ceci permet d'aller jusqu'à quelques 50 arguments, et nous espérons que cela ne posera de problème sur aucune plateforme.

Les arguments sont :

Un numéro de ligne
: C'est le numéro de ligne de la ligne originale ; utilisé seulement dans les messages d'erreur.

Une chaîne
: C'est la commande SQL à exécuter. Elle est modifiée par les variables en entrée, c'est−à−dire les variables qui n'étaient pas connues au moment de la compilation mais qui doivent être précisées dans la commande. A l'emplacement des variables, la chaîne contient le signe `?`.

Variables en entrée
: Chaque variable en entrée entraîne la création de dix arguments. (Voir ci−dessous.)

`ECPGt_EOIT`
: Un `enum` indiquant qu'il n'y a plus de variables en entrée.

Variables en sortie
: Chaque variable en sortie entraîne la création de dix arguments. (Voir ci−dessous.) La valeur des

variables est fournie par la fonction.

ECPGt_EORT
: Un `enum` indiquant qu'il n'y a plus de variables.

Pour chaque variable faisant partie de la commande SQL, la fonction récupère dix arguments :

1. Le type comme symbole spécial.
2. Un pointeur vers la valeur ou un pointeur vers le pointeur.
3. La taille de la variable dans le cas d'un type `char` ou `varchar`.
4. Le nombre d'éléments du tableau (pour les parcours de tableaux).
5. Le décalage pour obtenir le prochain élément dans le tableau (pour les parcours de tableaux).
6. Le type de variable indicateur comme un symbole spécial.
7. Un pointeur vers la variable indicateur.
8. 0
9. Le nombre d'éléments dans le tableau indicateur (pour les parcours de tableaux).
10. Le décalage pour obtenir le prochain élément dans le tableau indicateur (pour les parcours de tableaux).

Il est à noter que toutes les commandes SQL ne sont pas traitées ainsi. Par exemple, une instruction d'ouverture de curseur comme

```
EXEC SQL OPEN curseur;
```

n'est pas copiée sur la sortie. À la place, la commande `DECLARE` de déclaration du curseur est utilisée à la position de la commande `OPEN` car, en fait, elle ouvre le curseur.

Voici un exemple complet décrivant la sortie du préprocesseur d'un fichier `foo.pgc` (les détails pourraient varier avec chaque version particulière du préprocesseur) :

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int resultat;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :resultat FROM matable WHERE index = :index;
```

est traduit en :

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int resultat;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM matable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(resultat),1L,1L,sizeof(int),
```

```
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(L'indentation a été ajoutée ici pour des raisons de lisibilité et n'est pas réalisée par le préprocesseur.)

# Chapitre 30. Schéma d'informations

Le schéma d'informations consiste en un ensemble de vues contenant des informations sur les objets définis dans la base de données actuelle. Le schéma d'informations est défini dans le standard SQL et, du coup, est supposé portable et stable — contrairement aux catalogues système, qui sont spécifiques à PostgreSQL et modelés suivant l'implémentation. Néanmoins, les vues du schéma d'information ne contiennent pas d'information sur les fonctionnalités spécifiques à PostgreSQL ; pour cela, vous devez travailler avec les catalogues système ou d'autres vues spécifiques à PostgreSQL.

## 30.1. Le schéma

Le schéma d'informations est lui−même un schéma nommé `information_schema`. Ce schéma existe automatiquement dans toutes les bases de données. Le propriétaire de ce schéma est le propriétaire initial de la base de données du cluster et cet utilisateur a naturellement tous les privilèges sur ce schéma, incluant la possibilité de le supprimer (mais l'espace gagné ainsi sera minuscule).

Par défaut, le schéma d'informations n'est pas dans le chemin de recherche des schémas, donc vous avez besoin d'accéder à tous les objets qu'il contient via des noms qualifiés. Comme les noms de certains des objets du schéma d'information sont des noms génériques pouvant survenir dans les applications utilisateur, vous devez faire attention si vous placez le schéma d'information dans le chemin.

## 30.2. Types de données

Les colonnes des vues du schéma d'informations utilisent des types de données spéciaux, définis dans le schéma d'informations. Ils sont définis comme des domaines simples sur des types intégrés. Vous ne devriez pas utiliser ces types en dehors du schéma d'informations mais vos applications doivent y être préparées si elles font des sélections à partir du schéma d'informations.

Ces types sont :

`cardinal_number`
: Un entier non négatif.

`character_data`
: Une chaîne de caractères (sans longueur maximum spécifiée).

`sql_identifier`
: Une chaîne de caractères. Elle est utilisée pour les identifieurs SQL, le type de données `character_data` est utilisé pour tout autre type de données texte.

`time_stamp`
: Un domaine au−dessus du type `timestamp`

Chaque colonne du schéma d'informations a un des ces quatre types.

Les données booléenne (true/false) sont représentées dans le schéma d'informations par une colonne de type `character_data` contenant soit `YES` soit `NO`. (Le schéma d'informations a été inventé par le standard SQL, du coup cette convention est nécessaire pour conserver la compatibilité du moteur du schéma d'informations.)

## 30.3. information_schema_catalog_name

information_schema_catalog_name est une table qui contient en permanence une ligne et une colonne contenant le nom de la base de données actuelle (catalogue courant dans la terminologie SQL).

**Tableau 30–1. Colonnes de information_schema_catalog_name**

| Nom | Type de données | Description |
|---|---|---|
| catalog_name | sql_identifier | Nom de la base de données contenant ce schéma d'informations |

## 30.4. applicable_roles

La vue applicable_roles identifie tous les groupes dont l'utilisateur est membre. (Un rôle est identique à un groupe.) Généralement, il est préférable d'utiliser la vue enabled_roles au lieu de celle–ci ; voir aussi ici.

**Tableau 30–2. Colonnes de applicable_roles**

| Nom | Type de données | Description |
|---|---|---|
| grantee | sql_identifier | Toujours le nom de l'utilisateur courant |
| role_name | sql_identifier | Nom d'un groupe |
| is_grantable | character_data | S'applique à une fonctionnalité non disponible avec PostgreSQL |

## 30.5. check_constraints

La vue check_constraints contient toutes les contraintes de vérification, définies soit sur une table soit sur un domaine, possédées par l'utilisateur courant. (Le propriétaire d'une table ou d'un domaine est le propriétaire de la contrainte.)

**Tableau 30–3. Colonnes de check_constraints**

| Nom | Type de données | Description |
|---|---|---|
| constraint_catalog | sql_identifier | Nom de la base de données contenant la contrainte (toujours la base de données courante) |
| constraint_schema | sql_identifier | Nom du schéma contenant la contrainte |
| constraint_name | sql_identifier | Nom de la contrainte |
| check_clause | character_data | L'expression de vérification de la contrainte |

## 30.6. column_domain_usage

La vue column_domain_usage identifie toutes les colonnes (d'une table ou d'une vue) utilisant un domaine défini dans la base de données courante et possédé par l'utilisateur courant.

**Tableau 30–4. Colonnes de `column_domain_usage`**

| Nom | Type de données | Description |
|---|---|---|
| domain_catalog | sql_identifier | Nom de la base de données contenant le domaine (toujours la base de données courante) |
| domain_schema | sql_identifier | Nom du schéma contenant le domaine |
| domain_name | sql_identifier | Nom du domaine |
| table_catalog | sql_identifier | Nom de la base de données contenant la table (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table |
| table_name | sql_identifier | Nom de la table |
| column_name | sql_identifier | Nom de la colonne |

# 30.7. `column_privileges`

La vue `column_privileges` identifie tous les privilèges octroyés sur les colonnes à l'utilisateur courant ou par l'utilisateur courant. Il existe une ligne pour chaque combinaison de colonne, de personne donnant des droits et de personne recevant des droits. Les privilèges donnés aux groupes sont identifiés dans la vue `role_column_grants`.

Dans PostgreSQL, vous pouvez seulement donner des privilèges sur des tables entières, pas sur des colonnes individuelles. Du coup, la vue contient les mêmes informations que `table_privileges`, avec une représentation d'une ligne pour chaque colonne dans chaque table appropriée mais cela couvre seulement les types privilèges où la granularité des colonnes est possible : `SELECT`, `INSERT`, `UPDATE`, `REFERENCES`. Si vous souhaitez que vos applications puissent remplir les développements futurs possibles, le bon choix est d'utiliser cette vue en tant que `table_privileges` si un des types privilège est concerné.

**Tableau 30–5. Colonnes `column_privileges`**

| Nom | Type de données | Description |
|---|---|---|
| grantor | sql_identifier | Nom de l'utilisateur ayant donné ce privilège |
| grantee | sql_identifier | Nom de l'utilisateur et du groupe auxquels les privilèges reviendront |
| table_catalog | sql_identifier | Nom de la base de données contenant la table ayant les colonnes (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table et les colonnes |
| table_name | sql_identifier | Nom de la table contenant la colonne |
| column_name | sql_identifier | Nom de la colonne |
| privilege_type | character_data | Type de privilège : `SELECT`, `INSERT`, `UPDATE` ou `REFERENCES` |
| is_grantable | character_data | `YES` si le privilège peut être accordé, `NO` sinon |

Notez que la colonne `grantee` ne fait aucune distinction entre utilisateurs et groupes. Si vous avez des utilisateurs et des groupes du même nom, il n'y a malheureusement aucun moyen de les distinguer. Une prochaine version de PostgreSQL pourrait empêcher d'avoir des utilisateurs et des groupes de même nom.

# 30.8. column_udt_usage

La vue column_udt_usage identifie toutes les colonnes utilisant les types de données dont l'utilisateur courant est propriétaire. Notez que dans PostgreSQL, les types de données internes se comportent comme des types définis par l'utilisateur, donc ils sont aussi inclus ici. Voir aussi Section 30.9 pour plus de détails.

**Tableau 30–6. Colonnes de column_udt_usage**

| Nom | Type de données | Description |
|---|---|---|
| udt_catalog | sql_identifier | Nom de la base de données où le type de donnée de la colonne (le type sous−jacent du domaine, si applicable) est défini (toujous la base de données courante) |
| udt_schema | sql_identifier | Nom du schéma où le type de donnée de la colonne (le type sous−jacent du domaine, si applicable) est défini |
| udt_name | sql_identifier | Nom du type de données de la colonne (le type sous−jacent du domaine, si applicable) |
| table_catalog | sql_identifier | Nom de la base de données contenant la table (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table |
| table_name | sql_identifier | Nom de la table |
| column_name | sql_identifier | Nom de la colonne |

# 30.9. columns

La vue columns contient des informations sur toutes les colonnes de table (et colonnes de vue) de la base de données. Les colonnes système (oid, etc.) ne sont pas inclus. Seules les colonnes, pour lesquelles l'utilisateur a accès, sont affichés (qu'il soit le propriétaire ou qu'il ait quelques droits).

**Tableau 30–7. Colonnes de columns**

| Nom | Type de données | Description |
|---|---|---|
| table_catalog | sql_identifier | Nom de la base de données contenant la table (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table |
| table_name | sql_identifier | Nom de la table |
| column_name | sql_identifier | Nom de la colonne |
| ordinal_position | cardinal_number | Position de la colonne dans la table (le comptage commençant à 1) |
| column_default | character_data | Expression par défaut de la colonne (NULL si l'utilisateur courant n'est pas le propriétaire de la table contenant la colonne) |
| is_nullable | character_data | |

| | | |
|---|---|---|
| | | YES si la colonne peut contenir des valeurs NULL, NO dans le cas contraire. Une contrainte not NULL est une façon pour obtenir qu'une colonne soit connue comme ne pouvant pas contenir de valeurs NULL mais il en existe d'autres. |
| data_type | character_data | Le type de données de la colonne, s'il s'agit d'un type intégré ou d'un tableau (ARRAY) (dans ce cas, voir la vue the view element_types) sinon défini par l'utilisateur (USER-DEFINED) (dans ce cas, le type est identifié dans udt_name et dispose de colonnes associés). Si la colonne est basée sur un domaine, cette colonne est une référence au type sous-jacent du domaine (et le domaine est identifié dans domain_name et dispose de colonnes associées). |
| character_maximum_length | cardinal_number | Si data_type identifie un caractère ou un type de chaîne de bits, la longueur maximum déclarée ; NULL pour tous les autres types de données ou si aucune longueur maximum n'a été déclarée. |
| character_octet_length | cardinal_number | Si data_type identifie un type caractère, la longueur maximum en octets (bytes) d'un datum (ceci ne devrait pas concerner les utilisateurs PostgreSQL) ; NULL pour les autres types de données. |
| numeric_precision | cardinal_number | Si data_type identifie un type numérique, cette colonne contient la précision (déclarée ou implicite) du type pour cette colonne. Cette précision indique le nombre de chiffres significatifs. Elle pourrait être exprimée en décimal (base 10) ou en binaire (base 2) comme spécifié dans la colonne numeric_precision_radix. Pour tous les autres types de données, la colonne est NULL. |
| numeric_precision_radix | cardinal_number | Si data_type identifie un type numérique, cette colonne indique dans quel base les valeurs des colonnes numeric_precision et numeric_scale sont exprimées. La |

| | | |
|---|---|---|
| | | valeur est soit 2 soit 10. pour tous les autres types de données, cette colonne est NULL. |
| numeric_scale | cardinal_number | Si data_type identifie un type numeric exactement, cette colonne contient l'échelle (déclarée ou implicite) du type de cette colonne. L'échelle indique le nombre de chiffres significatifs à la droite du point décimal. Elle pourrait être exprimée en décimal (base 10) ou en binaire (base 2), comme spécifié dans la colonne numeric_precision_radix. Pour tous les autres types de données, cette colonne est NULL. |
| datetime_precision | cardinal_number | Si data_type identifie une date, une heure ou un type interval, la précision déclarée ; NULL pour tous les autres types de données ou si aucune précision n'a été déclarée. |
| interval_type | character_data | Pas encore implémenté |
| interval_precision | character_data | Pas encore implémenté |
| character_set_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL |
| character_set_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| character_set_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| domain_catalog | sql_identifier | Si la colonne a un type domain, le nom de la base de données où le type est défini (toujours la base de données courante), sinon NULL. |
| domain_schema | sql_identifier | Si la colonne a un type domain, le nom du schéma où le domaine est défini, sinon NULL. |
| domain_name | sql_identifier | Si la colonne a un type de domaine, le nom du domaine, sinon NULL. |
| udt_catalog | sql_identifier | Nom de la base de données où le type de données de la colonne (le type sous−jacent du domaine, si applicable) est défini (toujours la base de données |

| | | |
|---|---|---|
| | | courante) |
| udt_schema | sql_identifier | Nom du schéma où le type de données de la colonne (le type sous−jacent du domaine, si applicable) est défini |
| udt_name | sql_identifier | Nom du type de données de la colonne (le type sous−jacent du domaine, si applicable) |
| scope_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| scope_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| scope_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| maximum_cardinality | cardinal_number | Toujours NULL, car les tableaux ont toujours une cardinalité maximum illimitée avec PostgreSQL |
| dtd_identifier | sql_identifier | Un identifiant du descripteur du type de données de la colonne, unique parmi les descripteurs de type de données contenu dans la table. Ceci est principalement utile pour joindre d'autres instances de ces identifieurs. (Le format spécifique de l'identifiant n'est ni défini ni assuré de rester identique dans les versions futures.) |
| is_self_referencing | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |

Comme les types de données peuvent être définis avec une grande variété de moyens en SQL et comme PostgreSQL contient des moyens supplémentaires pour définir des types de données, leurs représentations dans le schéma d'information peut être assez complexe. La colonne `data_type` est supposée identifier le type de données intégré sous−jacent de la colonne. Dans PostgreSQL, cela signifie que le type est défini dans le schéma du catalogue système `pg_catalog`. Cette colonne pourrait être utile si l'application peut correctement gérer les types intégrés (par exemple, formater les types numériques différemment ou utiliser les données dans les colonnes de précisions). Les colonnes `udt_name`, `udt_schema` et `udt_catalog` identifient toujours le type de données sous−jacent de la colonne même si la colonne est basée sur un domaine. (Comme PostgreSQL traite les types intégrés comme des types définis par l'utilisateur, les types intégrés apparaissant aussi ici. Ceci est une extension du standard SQL.) Ces colonnes devraient être utilisées si une application souhaite traiter les données différemment suivant le type car, dans ce cas, peu importe si la colonne est réellement basée sur un domaine. Si la colonne est basée sur un domaine, l'identité du domaine est stockée dans les colonnes `domain_name`, `domain_schema` et `domain_catalog`. Si vous souhaitez assembler les colonnes avec leur types de données associés et traiter les domaines comme des types séparés, vous pouvez écrire `coalesce(domain_name, udt_name)`, etc.

## 30.10. constraint_column_usage

La vue `constraint_column_usage` identifie toutes les colonnes de la base de données courante utilisées par des contraintes. Seules sont affichées les colonnes contenues dans une table possédée par l'utilisateur courant. Pour une contrainte de vérification, cette vue identifie les colonnes utilisées dans

l'expression de la vérification. Pour une contrainte de clé étrangère, cette vue identifie les colonnes que la clé étrangère référence. Pour une contrainte unique ou de clé primaire, cette vue identifie les colonnes contraintes.

**Tableau 30–8. Colonnes de `constraint_column_usage`**

| Nom | Type de données | Description |
|---|---|---|
| `table_catalog` | `sql_identifier` | Nom de la base de données contenant la table, contenant la colonne utilisée par certaines contraintes (toujours la base de données courante) |
| `table_schema` | `sql_identifier` | Nom du schéma contenant la table, contenant la colonne utilisée par certaines contraintes |
| `table_name` | `sql_identifier` | Nom de la table, contenant la colonne utilisée par certaines contraintes |
| `column_name` | `sql_identifier` | Nom de la colonne utilisée par certaines contraintes |
| `constraint_catalog` | `sql_identifier` | Nom de la base de données contenant la contrainte (toujours la base de données courante) |
| `constraint_schema` | `sql_identifier` | Nom du schéma contenant la contrainte |
| `constraint_name` | `sql_identifier` | Nom de la contrainte |

## 30.11. `constraint_table_usage`

La vue `constraint_table_usage` identifie toutes les tables de la base de données courante utilisées par des contraintes et possédées par l'utilisateur courant. (Ceci est différent de la vue `table_constraints` qui identifie toutes les contraintes avec la table de leur définition.) Pour une contrainte de clé étrangère, cette vue identifie la table que la clé étrangère référence. Pour une contrainte unique ou de clé primaire, cette vue identifie simplement la table à laquelle appartient la contrainte. Les contraintes de vérification et les contraintes non NULL ne sont pas inclus dans cette vue.

**Tableau 30–9. Colonnes de `constraint_table_usage`**

| Nom | Type de données | Description |
|---|---|---|
| `table_catalog` | `sql_identifier` | Nom de la base de données contenant la table utilisée par quelques contraintes (toujours la base de données courante) |
| `table_schema` | `sql_identifier` | Nom du schéma contenant la table utilisée par quelques contraintes |
| `table_name` | `sql_identifier` | Nom de la table utilisée par quelques contraintes |
| `constraint_catalog` | `sql_identifier` | Nom de la base de données contenant la contrainte (toujours la base de données courante) |
| `constraint_schema` | `sql_identifier` | Nom du schéma contenant la contrainte |
| `constraint_name` | `sql_identifier` | Nom de la contrainte |

## 30.12. data_type_privileges

La vue data_type_privileges identifie tous les descripteurs de type de données auxquels l'utilisateur a accès, parce qu'il en est le propriétaire ou parce qu'il dispose de quelques droits sur le descripteur. Un descripteur de type de données est généré quand un type de données est utilisé dans la définition d'une colonne de table, d'un domaine ou d'une fonction (en tant que paramètre ou que code de retour) et stocke quelques informations sur la façon dont est utilisé le type de données (par exemple la longueur maximum déclarée, si applicable). Chaque descripteur de type de données se voit affecter un identifiant unique parmi les descripteurs de type de données affectés à un objet (table, domaine, fonction). Cette vue n'est probablement pas utile pour les applications mais est utilisée pour définir d'autres vues dans le schéma d'informations.

**Tableau 30–10. Colonnes de data_type_privileges**

| Nom | Type de données | Description |
|---|---|---|
| object_catalog | sql_identifier | Nom de la base de données contenant l'objet décrit (toujours la base de données courante) |
| object_schema | sql_identifier | Nom du schéma contenant l'objet décrit |
| object_name | sql_identifier | Nom de l'objet décrit |
| object_type | character_data | Le type d'objet décrit : fait partie de TABLE (le descripteur de type de données concerne une colonne de cette table), DOMAIN (le descripteur concerne ce domaine), ROUTINE (le descripteur est lié à un type de paramètre ou de code de retour de cette fonction). |
| dtd_identifier | sql_identifier | Identifieur du descripteur de type de données, unique parmi les descripteurs de type de données pour le même objet. |

## 30.13. domain_constraints

La vue domain_constraints contient toutes les contraintes appartenant aux domaines dont l'utilisateur courant est le propriétaire.

**Tableau 30–11. Colonnes de domain_constraints**

| Nom | Type de données | Description |
|---|---|---|
| constraint_catalog | sql_identifier | Nom de la base de données contenant la contrainte (toujours la base de données courante) |
| constraint_schema | sql_identifier | Nom du schéma contenant la contrainte |
| constraint_name | sql_identifier | Nom de la contrainte |
| domain_catalog | sql_identifier | Nom de la base de données contenant le domaine (toujours la base de données courante) |
| domain_schema | sql_identifier | Nom du schéma contenant le domaine |
| domain_name | sql_identifier | Nom du domaine |
| is_deferrable | character_data | YES si la contrainte est déférable, NO sinon |
| initially_deferred | character_data | |

| | | |
|---|---|---|
| | | YES si la contrainte est déférable et initialement déférée, NO sinon |

## 30.14. domain_udt_usage

La vue domain_udt_usage identifie toutes les colonnes utilisant les types de données possédés par l'utilisateur courant. Notez que dans PostgreSQL, les types de données intégrés se comportent comme des types définis par l'utilisateur, donc ils sont aussi inclus ici.

**Tableau 30–12. Colonnes de domain_udt_usage**

| Nom | Type de données | Description |
|---|---|---|
| udt_catalog | sql_identifier | Nom de la base de données où le type de données domaine est défini (toujours la base de données courante) |
| udt_schema | sql_identifier | Nom du schéma où est défini le type de données domaine |
| udt_name | sql_identifier | Nom du type de données domaine |
| domain_catalog | sql_identifier | Nom de la base de données contenant le domaine (toujours la base de données courante) |
| domain_schema | sql_identifier | Nom du schéma contenant le domaine |
| domain_name | sql_identifier | Nom du domaine |

## 30.15. domains

La vue domains contient tous les domaines définis dans la base de données courante.

**Tableau 30–13. Colonnes de domains**

| Nom | Type de données | Description |
|---|---|---|
| domain_catalog | sql_identifier | Nom de la base de données contenant le domaine (toujours la base de données courante) |
| domain_schema | sql_identifier | Nom du schéma contenant le domaine |
| domain_name | sql_identifier | Nom du domaine |
| data_type | character_data | Type de données du domaine, s'il s'agit d'un type intégré, ou ARRAY s'il s'agit d'un tableau (dans ce cas, voir la vue element_types), sinon USER-DEFINED (dans ce cas, le type est identifié dans udt_name et a des colonnes associées). |
| character_maximum_length | cardinal_number | Si le domaine a un type caractère ou chaîne de bits, la longueur maximale déclarée ; NULL pour tous les autres types de données ou si aucune longueur |

| | | |
|---|---|---|
| | | maximale n'a été déclarée. |
| character_octet_length | cardinal_number | Si le domaine a un type caractère, la longueur maximale en octets (bytes) d'un datum (ceci ne devrait pas concerner les utilisateurs PostgreSQL) ; NULL pour tous les autres types. |
| character_set_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| character_set_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| character_set_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| numeric_precision | cardinal_number | Si le domaine a un type numérique, cette colonne contient la précision (déclarée ou implicite) du type de cette colonne. Cette précision indique le nombre de chiffres significatifs. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), comme spécifié dans la colonne numeric_precision_radix. Pour les autres types de données, cette colonne est NULL. |
| numeric_precision_radix | cardinal_number | Si le domaine a un type numérique, cette colonne indique la base des valeurs des colonnes numeric_precision et numeric_scale. La valeur est soit 2 soit 10. Pour tous les autres types de données, cette colonne est NULL. |
| numeric_scale | cardinal_number | Si le domaine a le type numeric, cette colonne contient l'échelle (déclarée ou implicite) du type pour cette colonne. L'échelle indique le nombre de chiffres significatifs à la droite du point décimal. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), comme spécifié dans la colonne numeric_precision_radix. Pour tous les autres types de données, cette colonne est NULL. |

| | | |
|---|---|---|
| datetime_precision | cardinal_number | Si le domaine a une date, heur ou un type intervalle, la précision déclarée ; NULL pour les autres types de données ou si la précision n'a pas été déclarée. |
| interval_type | character_data | Pas encore implémenté |
| interval_precision | character_data | Pas encore implémenté |
| domain_default | character_data | Expression par défaut du domaine |
| udt_catalog | sql_identifier | Nom de la base de données dans laquelle est défini le type de données domaine (toujours la base de données courante) |
| udt_schema | sql_identifier | Nom du schéma où le type de données domaine est défini |
| udt_name | sql_identifier | Nom du type de données domaine |
| scope_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| scope_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| scope_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| maximum_cardinality | cardinal_number | Toujours NULL, car les tableaux ont toujours une cardinalité maximale illimitée dans PostgreSQL |
| dtd_identifier | sql_identifier | Un identifiant du descripteur de type de données du domaine, unique parmi les descripteurs de type de données restant dans le domaine (ce qui est trivial car un domaine contient seulement un descripteur de type de données). Ceci est principalement utile pour joindre d'autres instances de tels identifieurs. (Le format spécifique de l'identifiant n'est pas défini et il n'est pas garanti qu'il restera identique dans les versions futures.) |

## 30.16. element_types

La vue element_types contient les descripteurs de type de données des éléments de tableaux. Lorsqu'une colonne de table, domaine, paramètre de fonction ou code de retour de fonction est définie comme un type tableau, la vue du schéma d'informations respective contient seulement ARRAY dans la colonne data_type. Pour obtenir des informations sur le type d'élément du tableau, vous pouvez joindre la vue respective avec cette vue. Par exemple, pour afficher les colonnes d'une table avec les types de données et les types d'élément de tableau, si applicable, vous pouvez faire

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE',
```

```
c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_type,
e.array_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

Cette vue inclut seulement les objets auxquels l'utilisateur courant a accès en étant le propriétaire ou en disposant de quelques droits.

**Tableau 30–14. Colonnes de `element_types`**

| Nom | Type de données | Description |
|---|---|---|
| `object_catalog` | `sql_identifier` | Nom de la base de données contenant l'objet qui utilise le tableau en cours de description (toujours la base de données courante) |
| `object_schema` | `sql_identifier` | Nom du schéma contenant l'objet utilisant le tableau en cours de description |
| `object_name` | `sql_identifier` | Nom de l'objet utilisant le tableau en cours de description |
| `object_type` | `character_data` | Le type de l'objet utilisant le tableau en cours de description : il fait partie de `TABLE` (le tableau est utilisé par une colonne de cette table), `DOMAIN` (le tableau est utilisé par ce domaine), `ROUTINE` (le tableau est utilisé par un paramètre ou le type du code de retour de cette fonction). |
| `array_type_identifier` | `sql_identifier` | L'identifiant du descripteur de type de données du tableau en cours de description. Utilisez ceci pour joindre avec les colonnes de `dtd_identifier`, ayres vues du schéma d'informations. |
| `data_type` | `character_data` | Le type de données des éléments du tableau, s'il s'agit d'un type intégré, sinon `USER-DEFINED` (dans ce cas, le type est identifié comme `udt_name` et dispose de colonnes associées). |
| `character_maximum_length` | `cardinal_number` | Toujours NULL, car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL |
| `character_octet_length` | `cardinal_number` | Toujours NULL, car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL |
| `character_set_catalog` | `sql_identifier` | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| `character_set_schema` | `sql_identifier` | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| `character_set_name` | `sql_identifier` | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| `collation_catalog` | `sql_identifier` | S'applique à une fonctionnalité non disponible dans PostgreSQL. |

| | | |
|---|---|---|
| collation_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| numeric_precision | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL |
| numeric_precision_radix | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL |
| numeric_scale | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL |
| datetime_precision | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL |
| interval_type | character_data | Toujours NULL, car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL |
| interval_precision | character_data | Toujours NULL, car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL |
| domain_default | character_data | Pas encore implémenté |
| udt_catalog | sql_identifier | Nom de la base de données pour lequel le type de données est défini (toujours dans la base de données courante) |
| udt_schema | sql_identifier | Nom du schéma dans lequel sont définis les types de données des éléments |
| udt_name | sql_identifier | Nom du type de données des éléments |
| scope_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| scope_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| scope_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| maximum_cardinality | cardinal_number | Toujours NULL, car les tableaux ont une cardinalité maximum illimitée dans PostgreSQL |
| dtd_identifier | sql_identifier | Un identifiant de données de l'élément. Ce n'est pas utile actuellement. |

## 30.17. enabled_roles

La vue enabled_roles identifie tous les groupes dont l'utilisateur courant est membre. (Un rôle est la même chose qu'un groupe.) La différence entre cette vue et applicable_roles est que dans le futur, cela pourrait être un mécanisme pour (dés)activer des groupes lors d'une session. Dans ce cas, cette vue identifie les groupes actuellement activés.

**Tableau 30–15. Colonnes de `enabled_roles`**

| Nom | Type de données | Description |
|---|---|---|
| role_name | sql_identifier | Nom d'un groupe |

# 30.18. key_column_usage

La vue key_column_usage identifie toutes les colonnes de la base de données courante restreintes par une contrainte unique, clé primaire ou clé étrangère. Les contraintes de vérification ne sont pas inclues dans cette vue. Seules sont affichées les colonnes contenues dans une table appartenant à l'utilisateur courant.

**Tableau 30–16. Colonnes de `key_column_usage`**

| Nom | Type de données | Description |
|---|---|---|
| constraint_catalog | sql_identifier | Nom de la base de données contenant la contrainte (toujours la base de données courante) |
| constraint_schema | sql_identifier | Nom du schéma contenant la contrainte |
| constraint_name | sql_identifier | Nom de la contrainte |
| table_catalog | sql_identifier | Nom de la base de données contenant la colonne restreinte par la contrainte (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table, qui contient la colonne restreinte par la contrainte |
| table_name | sql_identifier | Nom de la table contenant la colonne restreinte par la contrainte |
| column_name | sql_identifier | Nom de la colonne restreinte par une contrainte |
| ordinal_position | cardinal_number | Position ordinal de la colonne dans la clé de contrainte (le comptage commence à 1) |

# 30.19. parameters

La vue parameters contient des informations sur les paramètres (arguments) de toutes les fonctions de la base de données courante. Seules sont affichées les fonctions auxquelles l'utilisateur courant a accès (soit en étant le propriétaire soit en ayant quelques privilèges).

**Tableau 30–17. Colonnes de `parameters`**

| Nom | Type de données | Description |
|---|---|---|
| specific_catalog | sql_identifier | Nom de la base de données contenant la fonction (toujours la base de données courante) |
| specific_schema | sql_identifier | Nom du schéma contenant la fonction |
| specific_name | sql_identifier | |

| | | |
|---|---|---|
| | | Le << nom spécifique >> de la fonction. Voir Section 30.26 pour plus d'informations. |
| ordinal_position | cardinal_number | Position ordinale du paramètre dans la liste des arguments de la fonction (le comptage commence à 1) |
| parameter_mode | character_data | Toujours IN, signifiant un paramètre en entrée (Dans le futur, il pourrait y avoir d'autres modes de paramètres.) |
| is_result | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| as_locator | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| parameter_name | sql_identifier | Nom du paramètre ou NULL si le paramètre n'a pas de nom |
| data_type | character_data | Type de données du paramètre, s'il s'agit d'un type intégré, ou ARRAY s'il s'agit d'un tableau (dans ce cas, voir la vue element_types), sinon USER-DEFINED (dans ce cas, le type est identifié dans udt_name et dispose de colonnes associées). |
| character_maximum_length | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données dans PostgreSQL |
| character_octet_length | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données dans PostgreSQL |
| character_set_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| character_set_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| character_set_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| numeric_precision | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données dans PostgreSQL |
| numeric_precision_radix | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données dans PostgreSQL |

| | | |
|---|---|---|
| numeric_scale | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données dans PostgreSQL |
| datetime_precision | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données dans PostgreSQL |
| interval_type | character_data | Toujours NULL, car cette information n'est pas appliquée aux types de données dans PostgreSQL |
| interval_precision | character_data | Toujours NULL, car cette information n'est pas appliquée aux types de données dans PostgreSQL |
| udt_catalog | sql_identifier | Nom de la base de données où est défini le paramètre (toujours la base de données courante) |
| udt_schema | sql_identifier | Nom du schéma où est défini le type de données du paramètre |
| udt_name | sql_identifier | Nom du type de données du paramètres |
| scope_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| scope_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| scope_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| maximum_cardinality | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données dans PostgreSQL |
| dtd_identifier | sql_identifier | Un identifiant du descripteur de type de données du paramètre, unique parmi les descripteurs de type de données restant dans la fonction. Ceci est principalement utile pour réaliser une jointure avec les autres instances de tels identifieurs. (Le format spécifique de l'identifiant n'est pas défini et il n'est pas garanti qu'il reste identique dans les prochaines versions.) |

## 30.20. `referential_constraints`

La vue `referential_constraints` contient toutes les contraintes référentielles (clés étrangères) dans la base de données actuelles, appartenant à une table possédée par l'utilisateur courant.

**Tableau 30–18. Colonnes de `referential_constraints`**

| Nom | Type de données | Description |
|---|---|---|
| constraint_catalog | sql_identifier | Nom de la base de données contenant la contrainte (toujours la base de données courante) |
| constraint_schema | sql_identifier | Nom du schéma contenant la contrainte |
| constraint_name | sql_identifier | Nom de la contrainte |
| unique_constraint_catalog | sql_identifier | Nom de la base de données contenant la contrainte unique ou de clé primaire que la contrainte de clé étrangère référence (toujours la base de données courante) |
| unique_constraint_schema | sql_identifier | Nom du schéma contenant la contrainte unique ou de clé primaire que la contrainte de clé étrangère référence |
| unique_constraint_name | sql_identifier | Nom de la contrainte unique ou de clé primaire que la contrainte de clé étrangère référence |
| match_option | character_data | Correspond aux options de la contrainte de clé étrangère : FULL, PARTIAL ou NONE. |
| update_rule | character_data | Règle de mise à jour de la contrainte de clé étrangère : CASCADE, SET NULL, SET DEFAULT, RESTRICT ou NO ACTION. |
| delete_rule | character_data | Règle de suppression de la contrainte de clé étrangère : CASCADE, SET NULL, SET DEFAULT, RESTRICT, or NO ACTION. |

## 30.21. role_column_grants

La vue role_column_grants identifie tous les droits donnés sur les colonnes des groupes dont l'utilisateur courant est membre. Plus d'informations sont disponibles avec column_privileges.

**Tableau 30–19. Colonnes de role_column_grants**

| Nom | Type de données | Description |
|---|---|---|
| grantor | sql_identifier | Nom de l'utilisateur qui a donné le droit |
| grantee | sql_identifier | Nom du groupe qui a reçu le droit |
| table_catalog | sql_identifier | Nom de la base de données contenant la table, qui contient la colonne (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table qui contient la colonne |
| table_name | sql_identifier | Nom de la table contenant la colonne |
| column_name | sql_identifier | Nom de la colonne |

| | | |
|---|---|---|
| privilege_type | character_data | Type de droit : SELECT, INSERT, UPDATE ou REFERENCES |
| is_grantable | character_data | YES si le privilège peut être accordé, NO sinon |

## 30.22. role_routine_grants

La vue role_routine_grants identifie tous les droits donnés sur des fonctions d'un groupe dont l'utilisateur courant est membre. Plus d'informations sont disponibles dans routine_privileges.

**Tableau 30–20. Colonnes de role_routine_grants**

| Nom | Type de données | Description |
|---|---|---|
| grantor | sql_identifier | Nom de l'utilisateur qui a donné le droit |
| grantee | sql_identifier | Nom du groupe qui a reçu le droit |
| specific_catalog | sql_identifier | Nom de la base de données contenant la fonction (toujours la base de données courante) |
| specific_schema | sql_identifier | Nom du schéma contenant la fonction |
| specific_name | sql_identifier | Le << nom spécifique >> de la fonction. Voir Section 30.26 pour plus d'informations. |
| routine_catalog | sql_identifier | Nom de la base de données contenant la fonction (toujours la base de données courante) |
| routine_schema | sql_identifier | Nom du schéma contenant la fonction |
| routine_name | sql_identifier | Nom de la fonction (pourrait être dupliqué en cas de surchargement) |
| privilege_type | character_data | Toujours EXECUTE (le seul type de droit des fonctions) |
| is_grantable | character_data | YES si le privilège peut être accordé, NO sinon |

## 30.23. role_table_grants

La vue role_table_grants identifie tous les droits donnés sur des tables ou vues d'un groupe dont l'utilisateur courant est membre. Plus d'informations sont disponibles dans table_privileges.

**Tableau 30–21. Colonnes de role_table_grants**

| Nom | Type de données | Description |
|---|---|---|
| grantor | sql_identifier | Nom de l'utilisateur qui a donné le droit |
| grantee | sql_identifier | Nom du groupe qui a reçu le droit |
| table_catalog | sql_identifier | Nom de la base de données contenant la table (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table |
| table_name | sql_identifier | Nom de la table |
| privilege_type | character_data | Type de droit : SELECT, DELETE, INSERT, UPDATE, REFERENCES, RULE ou TRIGGER |

| | | |
|---|---|---|
| is_grantable | character_data | YES si le privilège peut être accordé, NO sinon |
| with_hierarchy | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |

## 30.24. role_usage_grants

La vue role_usage_grants est utilisée pour identifier les droits d'USAGE donnés à différents types d'objets d'un groupe dont l'utilisateur est membre. Avec PostgreSQL, ceci s'applique seulement aux domaines et, comme les domaines n'ont pas de vrais droits dans PostgreSQL, cette vue est vide. Plus d'informations sont disponibles dans usage_privileges. Dans le futur, cette vue pourrait contenir plus d'informations utiles.

**Tableau 30–22. Colonnes de role_usage_grants**

| Nom | Type de données | Description |
|---|---|---|
| grantor | sql_identifier | Dans le futur, le nom de l'utilisateur qui a donné le droit |
| grantee | sql_identifier | Dans le futur, le nom de l'utilisateur qui a reçu le droit |
| object_catalog | sql_identifier | Nom de la base de données contenant l'objet (toujours la base de données courante) |
| object_schema | sql_identifier | Nom du schéma contenant l'objet |
| object_name | sql_identifier | Nom de l'objet |
| object_type | character_data | Dans le futur, le type de l'objet |
| privilege_type | character_data | Toujours USAGE |
| is_grantable | character_data | YES si le privilège peut être accordé, NO sinon |

## 30.25. routine_privileges

La vue routine_privileges identifie tous les droits donnés aux fonctions de l'utilisateur courant ou par l'utilisateur courant. Il existe une ligne pour chaque combinaison de fonctions, de donneur de droits et de receveur de droits. Les droits donnés aux groupes sont identifiés dans la vue role_routine_grants.

**Tableau 30–23. Colonnes de routine_privileges**

| Nom | Type de données | Description |
|---|---|---|
| grantor | sql_identifier | Nom de l'utilisateur qui a donné le droit |
| grantee | sql_identifier | Nom de l'utilisateur ou du groupe à qui a été donné le droit |
| specific_catalog | sql_identifier | Nom de la base de données contenant la fonction (toujours la base de données courante) |
| specific_schema | sql_identifier | Nom du schéma contenant la fonction |
| specific_name | sql_identifier | Le << nom spécifique >> de la fonction. Voir Section 30.26 pour plus d'informations. |
| routine_catalog | sql_identifier | Nom de la base de données contenant la fonction (toujours la base de données courante) |
| routine_schema | sql_identifier | Nom du schéma contenant la fonction |
| routine_name | sql_identifier | |

| | | |
|---|---|---|
| | | Nom de la fonction (pourrait être dupliqué en cas de surchargement) |
| privilege_type | character_data | Toujours EXECUTE (le seul type de droit pour les fonctions) |
| is_grantable | character_data | YES si le privilège peut être accordé, NO sinon |

Notez que la colonne grantee ne fait pas de distinction entre les utilisateurs et les groupes. Si vous avez des utilisateurs et des groupes de même nom, il n'existe aucune façon de les distinguer. Une prochaine version de PostgreSQL pourrait empêcher d'avoir des utilisateurs et des groupes de même nom.

## 30.26. routines

La vue routines contient toutes les fonctions de la base de données courante. Seules sont affichées les fonctions auxquelles l'utilisateur courant a accès (soit parce qu'il en est le propriétaire, soit parce qu'il possède des privilèges dessus).

**Tableau 30–24. Colonnes de routines**

| Nom | Type de données | Description |
|---|---|---|
| specific_catalog | sql_identifier | Nom de la base de données contenant la fonction (toujours la base de données courante) |
| specific_schema | sql_identifier | Nom du schéma contenant la fonction |
| specific_name | sql_identifier | Le << nom spécifique >> de la fonction. Ce nom identifie de façon unique la fonction dans le schéma, même si le nom réel de la fonction est surchargé. Le format du nom spécifique n'est pas défini, il devrait seulement être utilisé pour le comparer à d'autres instances de noms de routines spécifiques. |
| routine_catalog | sql_identifier | Nom de la base de données contenant la fonction (toujours la base de données courante) |
| routine_schema | sql_identifier | Nom du schéma contenant la fonction |
| routine_name | sql_identifier | Nom de la fonction (pourrait être dupliqué en cas de surchargement) |
| routine_type | character_data | Toujours FUNCTION (dans le futur, il pourrait y avoir d'autres types de routines.) |
| module_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| module_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| module_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| udt_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| udt_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| udt_name | sql_identifier | |

| | | |
|---|---|---|
| | | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| data_type | character_data | Type de données de retour de la fonction, s'il est intégré, ARRAY s'il s'agit d'un tableau (dans ce cas, voir la vue element_types), sinon USER-DEFINED (dans ce cas, le type est identifié dans type_udt_name et dispose de colonnes associées). |
| character_maximum_length | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données renvoyées avec PostgreSQL |
| character_octet_length | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données renvoyées par PostgreSQL |
| character_set_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| character_set_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| character_set_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| collation_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| numeric_precision | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données renvoyées par PostgreSQL |
| numeric_precision_radix | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données renvoyées par PostgreSQL |
| numeric_scale | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux types de données renvoyées par PostgreSQL |
| datetime_precision | cardinal_number | Toujours NULL, car cette information n'est pas appliquée aux aux types de données renvoyées par PostgreSQL |
| interval_type | character_data | Toujours NULL, car cette information n'est pas appliquée aux types de données renvoyées par PostgreSQL |
| interval_precision | character_data | Toujours NULL, car cette information n'est pas appliquée aux types de données renvoyées par PostgreSQL |
| type_udt_catalog | sql_identifier | Nom de la base de données où est défini le type de données en retour de la fonction (toujours la base de données courante) |

| | | |
|---|---|---|
| type_udt_schema | sql_identifier | Nom du schéma où est défini le type de données en retour de la fonction |
| type_udt_name | sql_identifier | Nom du type de données en retour de la fonction |
| scope_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| scope_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| scope_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| maximum_cardinality | cardinal_number | Toujours NULL, car les tableaux ont une cardinalité maximum illimitée dans PostgreSQL |
| dtd_identifier | sql_identifier | Un identifiant du descripteur de type de données du type de données en retour, unique parmi les descripteurs de type de données restant dans la fonction. Ceci est principalement utile pour la jointure avec d'autres instances de tels identifieurs. (Le format spécifique de l'identifiant n'est pas défini et il n'est pas certain qu'il restera identique dans les versions futures.) |
| routine_body | character_data | Si la fonction est une fonction SQL, alors SQL, sinon EXTERNAL. |
| routine_definition | character_data | Le texte source de la fonction (NULL si l'utilisateur courant n'est pas le propriétaire de la fonction). (Suivant le standard SQL, cette colonne est seulement applicable si routine_body est SQL, mais avec PostgreSQL il contiendra tout texte source qui a été spécifié à la création de la fonction.) |
| external_name | character_data | Si la fonction est une fonction C, alors le nom externe (link symbol) de la fonction ; sinon NULL. (Ceci fonctionne de telle façon qu'il s'agit de la même valeur montrée dans routine_definition.) |
| external_language | character_data | Le langage dans lequel est écrit la fonction |
| parameter_style | character_data | Toujours GENERAL (Le standard SQL définit d'autres styles de paramètres qui ne sont pas disponibles avec PostgreSQL.) |
| is_deterministic | character_data | Si la fonction est déclarée immutable (appelée déterministe dans le standard SQL), alors YES, sinon NO. (Vous ne pouvez pas connaître les autres niveaux de volatilité disponible dans PostgreSQL via le schéma d'informations.) |
| sql_data_access | character_data | Toujours MODIFIES, signifiant que la fonction peut modifier les données SQL. Cette information n'est pas utile pour PostgreSQL. |
| is_null_call | character_data | Si la fonction renvoie automatiquement NULL si un de ces arguments est NULL, alors YES, sinon |

| | | |
|---|---|---|
| | | NO. |
| sql_path | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| schema_level_routine | character_data | Toujours YES (L'opposé serait une méthode d'un type défini par l'utilisateur, fonctionnalité non disponible dans PostgreSQL.) |
| max_dynamic_result_sets | cardinal_number | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| is_user_defined_cast | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| is_implicitly_invocable | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| security_type | character_data | Si la fonction est exécutée avec les droits de l'utilisateur courant, alors INVOKER, si la fonction est exécutée avec les droits de l'utilisateur l'ayant défini, alors DEFINER. |
| to_sql_specific_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| to_sql_specific_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| to_sql_specific_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| as_locator | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |

## 30.27. schemata

La vue schemata contient tous les schémas de la base de données courante dont l'utilisateur est propriétaire.

**Tableau 30–25. Colonnes de schemata**

| Nom | Type de données | Description |
|---|---|---|
| catalog_name | sql_identifier | Nom de la base de données dans lequel se trouve le schéma (toujours la base de données courante) |
| schema_name | sql_identifier | Nom du schéma |
| schema_owner | sql_identifier | Nom du propriétaire du schéma |
| default_character_set_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| default_character_set_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| default_character_set_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| sql_path | character_data | |

| | | |
|---|---|---|
| | | S'applique à une fonctionnalité non disponible dans PostgreSQL. |

## 30.28. sql_features

La table sql_features contient des informations sur les fonctionnalités formelles définies dans le standard SQL et supportées par PostgreSQL. Ce sont les mêmes informations que celles présentées dans Annexe D. Là, vous pouvez aussi trouver des informations de fond supplémentaires.

**Tableau 30–26. Colonnes de sql_features**

| Nom | Type de données | Description |
|---|---|---|
| feature_id | character_data | Chaîne identifiant de la fonctionnalité |
| feature_name | character_data | Nom descriptif de la fonctionnalité |
| sub_feature_id | character_data | Chaîne identifiant de la sous–fonctionnalité ou une chaîne de longueur NULL s'il ne s'agit pas d'une sous–fonctionnalité |
| sub_feature_name | character_data | Nom descriptif de la sous–fonctionnalité ou une chaîne de longueur NULL s'il ne s'agit pas d'une sous–fonctionnalité |
| is_supported | character_data | YES si la fonctionnalité est complètement supportée par la version actuelle de PostgreSQL, NO sinon |
| is_verified_by | character_data | Toujours NULL, car le groupe de développement PostgreSQL ne réalise pas de tests formels sur la conformité des fonctionnalités |
| comments | character_data | Un commentaire, si possible, sur le statut supporté de la fonctionnalité |

## 30.29. sql_implementation_info

La table sql_implementation_info contient des informations sur différents aspects qui sont laissés à la définition de l'implémentation par le standard SQL. Cette information a pour but principal d'être utilisé dans le contexte de l'interface ODBC ; les utilisateurs des autres interfaces trouveront certainement peu d'utilité à cette information. Pour cette raison, les éléments d'informations d'implémentation individuelle ne sont pas décrits ici ; vous les trouverez dans la description de l'interface ODBC.

**Tableau 30–27. Colonnes de sql_implementation_info**

| Nom | Type de données | Description |
|---|---|---|
| implementation_info_id | character_data | Chaîne identifiant de l'élément d'information d'implémentation |
| implementation_info_name | character_data | Nom descriptif de l'élément d'information d'implémentation |

| | | |
|---|---|---|
| integer_value | cardinal_number | Valeur de l'élément d'information d'implémentation, ou NULL si la valeur est contenue dans la colonne character_value |
| character_value | character_data | Valeur de l'élément d'information d'implémentation, ou NULL si la valeur est contenue dans la colonne integer_value |
| comments | character_data | Un possible commentaire restant dans l'élément d'information d'implémentation |

## 30.30. sql_languages

La table sql_languages contient une ligne pour chaque langage lié SQL supporté par PostgreSQL. PostgreSQL supporte directement le SQL et le SQL intégré dans le C ; c'est tout ce que vous apprendrez de cette table.

**Tableau 30–28. Colonnes de sql_languages**

| Nom | Type de données | Description |
|---|---|---|
| sql_language_source | character_data | Le nom de la source de la définition du langage ; toujours ISO 9075, c'est–à–dire le standard SQL |
| sql_language_year | character_data | L'année de l'approbation du standard dans sql_language_source ; actuellement 2003 |
| sql_language_comformance | character_data | Le niveau de conformité au standard pour le langage. Pour ISO 9075:2003 c'est toujours CORE. |
| sql_language_integrity | character_data | Toujours NULL (Cette valeur n'a d'intérêt que pour les versions précédentes du standard SQL.) |
| sql_language_implementation | character_data | Toujours NULL |
| sql_language_binding_style | character_data | Le style de lien du langage, soit DIRECT soit EMBEDDED |
| sql_language_programming_language | character_data | Le langage de programmation si le style de lien est EMBEDDED, sinon NULL. PostgreSQL supporte uniquement le langage C. |

## 30.31. sql_packages

La table sql_packages contient des informations sur les paquets de fonctionnalités définis dans le standard SQL et supportés par PostgreSQL. Référez−vous à Annexe D pour des informations d'arrière plan sur les paquets de fonctionnalités.

**Tableau 30−29. Colonnes de sql_packages**

| Nom | Type de données | Description |
|---|---|---|
| feature_id | character_data | Chaîne identifiant du paquet |
| feature_name | character_data | Nom descriptif du paquet |
| is_supported | character_data | YES si le paquet est complètement supporté par la version actuelle de, NO sinon |
| is_verified_by | character_data | Toujours NULL, car le groupe de développement de PostgreSQL ne réalise pas de tests formels pour la conformité des fonctionnalités |
| comments | character_data | Un possible commentaire sur l'état supporté par le paquet |

## 30.32. sql_sizing

La table sql_sizing contient des informations sur les différentes limites de tailles et valeurs maximale dans PostgreSQL. Cette information a pour but principal d'être utilisée dans le contexte de l'interface ODBC ; les utilisateurs des autres interfaces trouveront probablement peu d'utilité à cette information. Pour cette raison, les éléments de taille individuelle ne sont pas décrits ici ; vous les trouverez dans la description de l'interface ODBC.

**Tableau 30−30. Colonnes de sql_sizing**

| Nom | Type de données | Description |
|---|---|---|
| sizing_id | cardinal_number | Identifieur de l'élément de taille |
| sizing_name | character_data | Nom descriptif de l'élément de taille |
| supported_value | cardinal_number | Valeur de l'élément de taille, ou 0 si la taille est illimitée ou ne peut pas être déterminée, ou NULL si les fonctionnalités pour lesquelles l'élément de taille est applicable ne sont pas supportées |
| comments | character_data | Un possible commentaire restant dans l'élément de taille |

## 30.33. sql_sizing_profiles

La table sql_sizing_profiles contient des informations sur les valeurs sql_sizing requises par plusieurs profiles du standard SQL. PostgreSQL ne garde pas trace des profiles SQL, donc la table est vide.

**Tableau 30–31. Colonnes de `sql_sizing_profiles`**

| Nom | Type de données | Description |
|---|---|---|
| sizing_id | cardinal_number | Identifieur de l'élément de taille |
| sizing_name | character_data | Nom descriptif de l'élément de taille |
| profile_id | character_data | Chaîne identifiant d'un profile |
| required_value | cardinal_number | La valeur requise par le profile SQL pour l'élément de taille, ou 0 si le profile ne place aucune limite sur l'élément de taille, ou NULL si le profile ne requiert aucune fonctionnalité pour laquelle l'élément de style est applicable |
| comments | character_data | Un possible commentaire restant dans l'élément de taille à l'intérieur du profile |

## 30.34. `table_constraints`

La vue table_constraints contient toutes les contraintes appartenant aux tables possédées par l'utilisateur courant.

**Tableau 30–32. Colonnes de `table_constraints`**

| Nom | Type de données | Description |
|---|---|---|
| constraint_catalog | sql_identifier | Nom de la base de données contenant la contrainte (toujours la base de données courante) |
| constraint_schema | sql_identifier | Nom du schéma contenant la contrainte |
| constraint_name | sql_identifier | Nom de la contrainte |
| table_catalog | sql_identifier | Nom de la base de données contenant la table (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table |
| table_name | sql_identifier | Nom de la table |
| constraint_type | character_data | Type de contrainte : CHECK, FOREIGN KEY, PRIMARY KEY, or UNIQUE |
| is_deferrable | character_data | YES si la contrainte est déférable, NO sinon |
| initially_deferred | character_data | YES si la contrainte est déférable et initialement déférée, NO sinon |

## 30.35. `table_privileges`

La vue table_privileges identifie tous les droits donnés sur des tables ou vues de l'utilisateur courant ou par l'utilisateur courant. Il y a une ligne pour chaque combinaison de table, de donneur de droits et de receveur de droits. Les droits donnés aux groupes sont identifiés dans la vue role_table_grants.

**Tableau 30–33. Colonnes de `table_privileges`**

| Nom | Type de données | Description |
|---|---|---|

| | | |
|---|---|---|
| grantor | sql_identifier | Nom de l'utilisateur qui a donné le droit |
| grantee | sql_identifier | Nom de l'utilisateur ou groupe à qui le droit a été donné |
| table_catalog | sql_identifier | Nom de la base de données contenant la table (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table |
| table_name | sql_identifier | Nom de la table |
| privilege_type | character_data | Type de droit : SELECT, DELETE, INSERT, UPDATE, REFERENCES, RULE ou TRIGGER |
| is_grantable | character_data | YES si le privilège peut être accordé, NO sinon |
| with_hierarchy | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |

Notez que la colonne grantee ne fait aucune distinction entre les utilisateurs et groupes. Si vous avez des utilisateurs et groupes de même nom, il n'y a malheureusement aucun moyen de les distinguer. Une version future de PostgreSQL empêchera probablement d'avoir des utilisateurs et des groupes de même nom.

## 30.36. tables

La vue tables contient toutes les tables et vues définies dans la base de données courantes. Seules sont affichées les tables et vues auxquelles l'utilisateur courant a accès (soit parce qu'il en est le propriétaire soit parce qu'il possède certains droits dessus).

**Tableau 30–34. Colonnes de tables**

| Nom | Type de données | Description |
|---|---|---|
| table_catalog | sql_identifier | Nom de la base de données contenant la table (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table |
| table_name | sql_identifier | Nom de la table |
| table_type | character_data | Type de table : BASE TABLE pour une table de base persistante (le type de table normal), VIEW pour une vue ou LOCAL TEMPORARY pour une table temporaire |
| self_referencing_column_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| reference_generation | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| user_defined_type_catalog | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| user_defined_type_schema | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| user_defined_type_name | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |

# 30.37. triggers

La vue triggers contient tous les déclencheurs définis dans la base de données courante et possédés par l'utilisateur courant. (Le propriétaire de la table est le propriétaire du déclencheur.)

**Tableau 30–35. Colonnes de triggers**

| Nom | Type de données | Description |
|---|---|---|
| trigger_catalog | sql_identifier | Nom de la base de données contenant le déclencheur (toujours la base de données courante) |
| trigger_schema | sql_identifier | Nom du schéma contenant le déclencheur |
| trigger_name | sql_identifier | Nom du déclencheur |
| event_manipulation | character_data | Événement qui a lancé le déclencheur (INSERT, UPDATE ou DELETE) |
| event_object_catalog | sql_identifier | Nom de la base de données contenant la table où est défini le déclencheur (toujours la base de données courante) |
| event_object_schema | sql_identifier | Nom du schéma contenant la table où est défini le déclencheur |
| event_object_table | sql_identifier | Nom de la table où est défini le déclencheur |
| action_order | cardinal_number | Pas encore implémenté |
| action_condition | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| action_statement | character_data | Instruction exécutée par le déclencheur (actuellement toujours EXECUTE PROCEDURE *function*(...)) |
| action_orientation | character_data | Identifie si le déclencheur a été lancé une fois pour chaque ligne traitée ou une fois par instruction (ROW ou STATEMENT) |
| condition_timing | character_data | Moment de l'exécution de déclencheur (BEFORE ou AFTER) |
| condition_reference_old_table | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| condition_reference_new_table | sql_identifier | S'applique à une fonctionnalité non disponible dans PostgreSQL. |

Les déclencheurs dans PostgreSQL ont deux incompatibilités avec le standard SQL qui affectent la représentation dans le schéma d'information. Tout d'abord, les noms de déclencheurs sont locaux à la table avec PostgreSQL, plutôt qu'indépendant des objets schéma. Du coup, il pourrait y avoir des noms de déclencheurs dupliqués dans un schéma, s'ils s'occupent de tables différentes. (trigger_catalog et trigger_schema sont réellement les valeurs restant dans la table sur laquelle est défini le déclencheur.) Deuxièmement, les déclencheurs peuvent être définis pour s'exécuter sur plusieurs événements dans PostgreSQL (c'est–à–dire ON INSERT OR UPDATE) alors que le standard SQL n'en autorise qu'un. Si un déclencheur est défini pour s'exécuter sur plusieurs événements, il est représenté sur plusieurs lignes dans le schéma d'information, une pour chaque type d'événement. Comme conséquence de ces deux problèmes, la clé

primaire de la vue triggers est en fait (trigger_catalog, trigger_schema, trigger_name, event_object_table, event_manipulation) au lieu de (trigger_catalog, trigger_schema, trigger_name), ce que spécifie le standard SQL. Néanmoins, si vous définissez les déclencheurs d'une manière conforme au standard SQL (des noms de déclencheurs uniques dans le schéma et seulement un type d'événement par déclencheur), ceci ne vous concernera pas.

## 30.38. usage_privileges

La vue usage_privileges est utile pour identifier les droits USAGE donnés sur différents objets de l'utilisateur courant ou par l'utilisateur courant. Dans PostgreSQL, ceci s'applique uniquement aux domaines et, comme les domaines n'ont pas de vrais droits avec PostgreSQL, cette vue affiche les droits USAGE implicites données par PUBLIC pour tous les domaines. Dans le futur, cette vue pourrait contenir plus d'informations utiles.

**Tableau 30–36. Colonnes de usage_privileges**

| Nom | Type de données | Description |
|---|---|---|
| grantor | sql_identifier | Actuellement initialisé avec le nom du propriétaire de l'objet |
| grantee | sql_identifier | Actuellement toujours PUBLIC |
| object_catalog | sql_identifier | Nom de la base de données contenant l'objet (toujours la base de données courante) |
| object_schema | sql_identifier | Nom du schéma contenant l'objet |
| object_name | sql_identifier | Nom de l'objet |
| object_type | character_data | Actuellement toujours DOMAIN |
| privilege_type | character_data | Toujours USAGE |
| is_grantable | character_data | Actuellement toujours NO |

## 30.39. view_column_usage

La vue view_column_usage identifie toutes les colonnes utilisées dans l'expression de la requête d'une vue (l'instruction SELECT définissant la vue). Une colonne est seulement inclue sur l'utilisateur courant qui est le propriétaire de la table contenant la colonne.

> **Note :** Les colonnes des tables système ne sont pas inclues. Ceci devra être corrigé.

**Tableau 30–37. Colonnes de view_column_usage**

| Nom | Type de données | Description |
|---|---|---|
| view_catalog | sql_identifier | Nom de la base de données contenant la vue (toujours la base de données courante) |
| view_schema | sql_identifier | Nom du schéma contenant la vue |
| view_name | sql_identifier | Nom de la vue |

| | | |
|---|---|---|
| table_catalog | sql_identifier | Nom de la base de données contenant la table, qui contient la colonne utilisée par la vue (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table contenant la colonne utilisée par la vue |
| table_name | sql_identifier | Nom de la table contenant la colonne utilisée par la vue |
| column_name | sql_identifier | Nom de la colonne utilisée par la vue |

## 30.40. view_table_usage

La vue view_table_usage identifie toutes les tables utilisées dans l'expression de la requête d'une vue (l'instruction SELECT définissant la vue). Une table est inclue seulement si l'utilisateur courant est le propriétaire de cette table.

**Note :** Les tables système ne sont pas inclus. Ceci devra être corrigé.

**Tableau 30–38. Colonnes de view_table_usage**

| Nom | Type de données | Description |
|---|---|---|
| view_catalog | sql_identifier | Nom de la base de données contenant la vue (toujours la base de données courante) |
| view_schema | sql_identifier | Nom du schéma contenant la vue |
| view_name | sql_identifier | Nom de la vue |
| table_catalog | sql_identifier | Nom de la base de données contenant la table utilisée par la vue (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la table utilisée par la vue |
| table_name | sql_identifier | Nom de la table utilisée par la vue |

## 30.41. views

La vue views contient toutes les vues définies dans la base de données courantes. Seules sont affichées les vues auxquelles l'utilisateur a accès (parce qu'il en est le propriétaire ou parce qu'il possède quelques droits dessus).

**Tableau 30–39. Colonnes de views**

| Nom | Type de données | Description |
|---|---|---|
| table_catalog | sql_identifier | Nom de la base de données contenant la vue (toujours la base de données courante) |
| table_schema | sql_identifier | Nom du schéma contenant la vue |
| table_name | sql_identifier | Nom de la vue |
| view definition | character_data | Expression de la requête définissant la vue (NULL si l'utilisateur courant n'est pas le propriétaire de la vue) |

| | | |
|---|---|---|
| check_option | character_data | S'applique à une fonctionnalité non disponible dans PostgreSQL. |
| is_updatable | character_data | Pas encore implémenté |
| is_insertable_into | character_data | Pas encore implémenté |

# V. Programmation Serveur

Cette partie traite de l'extension des fonctionnalités du serveur avec les fonctions définies par l'utilisateur, les types de données, les déclencheurs (triggers), etc. Ce sont des sujets avancés qui devraient probablement être abordées uniquement après que toutes les autres documentations utilisateurs sur PostgreSQL aient été comprises. Les derniers chapitres de cette partie décrivent les langages de programmation côté serveur disponibles dans la distribution de PostgreSQL ainsi que les problèmes généraux qui les concernent. Il est essentiel de lire au moins les premières sections du Chapitre 31 (traitant des fonctions) avant de se plonger dans les langages de programmation du côté serveur.

**Table des matières**

# Chapitre 31. Extension de SQL

Dans cette section, nous discuterons de la façon d'étendre le langage de requêtes PostgreSQL SQL en ajoutant des :

- fonctions (commençant à la Section 31.3) ;
- agrégats (commençant à la Section 31.10) ;
- types de données (commençant à la Section 31.11) ;
- opérateurs (début à la Section 31.12) ;
- classes d'opérateurs pour les index (commençant à la Section 31.14).

## 31.1. Comment fonctionne l'extensibilité

PostgreSQL est extensible parce qu'il opère grâce à un catalogue. Si vous êtes familier avec les systèmes standards de bases de données, vous savez qu'ils enregistrent les informations concernant les bases de données, les tables, les colonnes, etc., dans ce qu'on nomme communément des catalogues systèmes (certains systèmes appellent cela le dictionnaire de données). Pour l'utilisateur, les catalogues apparaissent comme des tables ordinaires mais le SGBD y enregistre ses écritures internes. Une différence essentielle entre PostgreSQL et les systèmes de bases de données relationnelles est que PostgreSQL enregistre beaucoup plus d'informations dans ses catalogues : pas seulement l'information à propos des tables et des colonnes, mais aussi l'information au sujet des types de données, des fonctions, des méthodes d'accès, etc. Ces tables peuvent être modifiées par l'utilisateur et, puisque PostgreSQL fonde ses opérations sur ces tables, cela signifie que PostgreSQL peut être étendu par les utilisateurs. En comparaison, les systèmes de bases de données conventionnels peuvent seulement être étendus en modifiant les procédures dans le code source ou en installant des modules spécifiquement écrits par le vendeur de SGBD.

Encore mieux, le serveur PostgreSQL peut incorporer à lui−même du code écrit par l'utilisateur grâce au chargement dynamique. C'est−à−dire que l'utilisateur peut spécifier un fichier de code objet (par exemple une bibliothèque partagée) qui implémente un nouveau type ou une nouvelle fonction et que PostgreSQL chargera à la demande. Il est encore plus évident d'ajouter au serveur du code écrit en SQL. Cette possibilité de modifier son fonctionnement << à la volée >> fait de PostgreSQL un outil unique pour le prototypage rapide de nouvelles applications et de structures de stockage.

## 31.2. Système de typage de PostgreSQL

Les types de données de PostgreSQL sont répartis en types de base, types composite, types de domaine et pseudo−types.

Les types de base sont ceux qui, comme `int4`, sont implémentés sous le niveau du langage SQL (typiquement dans un langage de bas niveau comme le C). Ils correspondent généralement à ce qui est souvent connu comme des types de données abstraits. PostgreSQL peut opérer sur de tels types seulement au moyen de fonctions fournies par l'utilisateur et n'en comprend le fonctionnement que dans la mesure où l'utilisateur les décrit. Les types de base sont divisés en types scalaires et types tableaux. Pour chaque type scalaire, un type tableau est automatiquement créé. Ce type tableau peut contenir des tableaux à taille variable du type scalaire correspondant.

Les types composites, ou types lignes, sont créés chaque fois qu'un utilisateur crée une table ; il est aussi possible de définir un type composite autonome sans table associée. Un type composite est simplement une liste de types de base avec des noms de champs associés. Une valeur pour un type composite correspond à une ligne ou un enregistrement de valeurs de champ. L'utilisateur peut accéder à ces champs à partir de requêtes SQL.

Un type de domaine est basé sur un type de base particulier, et, à de multiples fins, il est interchangeable avec son type de base. Toutefois, un domaine peut avoir des contraintes qui restreignent ses valeurs valides à un sous−ensemble de ce qui est permis par le type de base sous−jacent. Les domaines peuvent être créés par de simples commandes SQL.

Enfin, il existe quelques << pseudo−types >> pour des besoins particuliers. Les pseudo−types ne peuvent pas apparaître comme des champs de table ou des types composites, mais ils peuvent être utilisés pour déclarer les types des arguments et des résultats de fonctions. Ils fournissent dans le système de typage un mécanisme pour identifier des classes spéciales de fonctions. La Tableau 8−20 donne la liste des pseudo−types existants.

## 31.2.1. Types et fonctions polymorphes

Deux pseudo−types particulièrement intéressants sont `anyelement` et `anyarray`, collectivement appelés *types polymorphes*. Toute fonction déclarée utilisant ces types est dite *fonction polymorphe*. Une fonction polymorphe peut opérer sur de nombreux et différents types de données, les types de données spécifiques étant déterminés par les types des données réellement passés lors d'un appel particulier de la fonction.

Les arguments et les résultats polymorphes sont liés les uns aux autres et sont résolus dans un type de données spécifique quand une requête faisant appel à une fonction polymorphe est analysée. Chaque occurrence déclarée comme `anyelement` (soit argument, soit valeur renvoyée par la fonction) peut prendre n'importe quel type réel de données, mais lors d'un appel de fonction donné, elles doivent toutes avoir le *même* type réel. Chaque occurrence déclarée comme `anyarray` peut prendre n'importe quel type de données tableau, mais de la même façon, elles doivent toutes être du *même* type. Si des occurrences sont déclarées comme `anyarray` et d'autres comme `anyelement`, le type de tableau réel des occurrences `anyarray` doit être un tableau dont les éléments sont du même type que ceux apparaissant dans les occurrences de type `anyelement`.

Ainsi, quand plus d'une occurrence d'argument est déclarée avec un type polymorphe, l'effet direct est que seulement certaines combinaisons de types réels d'argument sont autorisées. Par exemple, une fonction déclarée comme `foo(anyelement, anyelement)` prendra en argument n'importe quelles valeurs en argument, à condition qu'elles soient du même type de données.

Quand la valeur renvoyée par une fonction est déclarée de type polymorphe, il doit exister au moins une occurrence d'argument également polymorphe, et le type réel de donnée passé comme argument détermine le type réel de résultat renvoyé lors de cet appel à la fonction. Par exemple, s'il n'existait pas déjà un mécanisme d'extraction d'éléments (indexation) de tableau, on pourrait définir une fonction qui implémente ce mécanisme comme ceci : `indice(anyarray, integer) returns anyelement`. La déclaration de fonction contraint le premier argument réel à être de type tableau et permet à l'analyseur d'inférer le type correct de résultat à partir du type réel du premier argument.

## 31.2.2. Types de base

Les types de base sont ceux implémentés avant le langage SQL comme `int4` (typiquement dans un langage de base niveau comme le C). Ils correspondent généralement à ce qu'on nomme types de données abstraits. PostgreSQL peut opérer avec de tels types seulement au travers de fonctions apportées par l'utilisateur et comprend uniquement le comportement de tels types par la description qu'en a fait l'utilisateur. Les types de base sont ensuite sous−divisés en types scalaires et tableaux. Pour chaque type scalaire, un type de tableau correspondant est automatiquement créé, pouvant contenir des tableaux à la taille des variables de ce type scalaire.

## 31.2.3. Types composites

Les types composites, ou types lignes, sont crées lorsqu'un utilisateur crée une table ; il est aussi possible de définir un type composite se suffisant à lui−même (<< stand−alone >>) sans table associée. Un type composite est simplement une liste de types de base avec les noms des champs associés. Une valeur d'un type composite est une ligne ou un enregistrement de champs. L'utilisateur peut accéder aux champs du composant à partir de requêtes SQL.

## 31.2.4. Domaines

Un domaine est basé sur un type de base particulier et est interchangeable pour de nombreux buts avec son type de base. Néanmoins, un domaine pourrait avoir des contraintes restreignant ses valeurs valides à un sous−ensemble du type de base de départ.

Les domaines peuvent être créés avec les commandes SQL *CREATE DOMAIN*. Leur création et utilisation ne sont pas discutées dans ce chapitre.

## 31.2.5. Pseudo−Types

Il existe quelques << pseudo−types >> pour certains buts très spécifiques. Les pseudo−types ne peuvent pas apparaître comme colonnes de tables ou d'attributs de types composites, mais peuvent être utilisés pour déclarer les types d'argument ou de résultat de fonctions. Cela fournit un mécanisme à l'intérieur du système de types pour identifier les classes spéciales de fonctions. Tableau 8−20 liste les pseudos−types existants.

## 31.2.6. Types polymorphiques

Deux pseudo−types d'un grand intérêt sont `anyelement` et `anyarray`, collectivement appelés des *types polymorphiques*. Toute fonction déclarée utilisant ces types est dite être une *fonction polymorphique*. Une fonction polymorphique peut opérer sur différents types de données, les types de données spécifiques étant déterminés par les types de données réellement passés lors d'un appel particulier.

Les arguments et résultats polymorphiques sont liés les uns aux autres et sont modifiés en un type spécifique de données lorsqu'une requête appelant une fonction polymorphique est analysée. Chaque position (argument ou valeur de retour) déclarée en tant que `anyelement` est autorisée à avoir tout spécifique type de donnée, mais sur un seul appel, elles doivent toutes avoir le *même* type. Chaque position déclarée comme `anyarray` peut avoir n'importe quel type de données, mais similairement, elles doivent toutes avoir le même type. Si des positions sont déclarées `anyarray` et d'autres `anyelement`, le type de tableau réel dans les positions `anyarray` doit être un tableau dont les éléments sont du même type que ceux des positions `anyelement`.

Du coup, lorsque plus d'un argument est déclaré polymorphique, l'effet est que seules certaines combinaisons de types d'argument sont autorisées. Par exemple, une fonction déclarée de cette façon `equal(anyelement, anyelement)` prendra seulement deux valeurs en entrées, à condition qu'elles soient du même type.

Lorsque la valeur de retour d'une fonction est déclarée polymorphique, Il doit exister au moins un argument lui−aussi polymorphique et le type de données fourni en argument détermine le type de données du code de retour pour cet appel. Par exemple, s'il n'existait pas déjà un mécanisme d'abonnement d'un tableau, nous pourrions définir une fonction qui implémente cet abonnement avec `subscript(anyarray, integer) returns anyelement`. Cette déclaration contraint le premier argument à être de type tableau et permet à l'analyseur d'inférer le bon type de retour à partir du type du premier argument.

# 31.3. Fonctions définies par l'utilisateur

PostgreSQL propose quatre types de fonctions :

- fonctions en langage de requête (fonctions écrites en SQL) (Section 31.4)
- fonctions en langage de procédures (fonctions écrites, par exemple, en PL/pgSQL ou PL/Tcl) (Section 31.7)
- fonctions internes (Section 31.8)
- fonctions en langage C (Section 31.9)

Chaque sorte de fonction peut accepter comme arguments (paramètres) des types de base, des types composites ou une combinaison de ceux−ci. De plus, chaque sorte de fonction peut renvoyer un type de base ou un type composite. Les fonctions pourraient aussi être définies pour renvoyer des ensembles de valeurs de base ou de valeurs composites.

De nombreuses sortes de fonctions peuvent accepter ou renvoyer certains pseudo−types (comme les types polymorphes) mais avec des fonctionnalités variées. Consultez la description de chaque type de fonction pour plus de détails.

Il est plus facile de définir des fonctions SQL aussi allons−nous commencer par celles−ci. La plupart des concepts présentés pour les fonctions SQL seront aussi gérés par les autres types de fonctions.

Lors de la lecture de ce chapitre, il peut être utile de consulter la page de référence de la commande *CREATE FUNCTION* pour mieux comprendre les exemples. Quelques exemples extraits de ce chapitre peuvent être trouvés dans les fichiers `funcs.sql` et `funcs.c` du répertoire du tutoriel de la distribution source de PostgreSQL.

# 31.4. Fonctions en langage de requêtes (SQL)

Les fonctions SQL exécutent une liste arbitraire d'instructions SQL et renvoient le résultat de la dernière requête de cette liste. Dans le cas d'un résultat simple (pas d'ensemble), la première ligne du résultat de la dernière requête sera renvoyée (gardez à l'esprit que << la première ligne >> d'un résultat multiligne n'est pas bien définie à moins d'utiliser `ORDER BY` ). Si la dernière requête de la liste ne renvoie aucune ligne, la valeur NULL est renvoyée.

Une fonction SQL peut être déclarée de façon à renvoyer un ensemble (set) en spécifiant le type renvoyé par la fonction comme SETOF *untype*. Dans ce cas, toutes les lignes de la dernière requête sont renvoyées. Des détails supplémentaires sont donnés plus loin dans ce chapitre.

Le corps d'une fonction SQL doit être constitué d'une liste d'une ou de plusieurs instructions SQL séparées par des points−virgule. Un point−virgule après la dernière instruction est optionnel. Sauf si la fonction déclare renvoyer void, la dernière instruction doit être un SELECT.

Toute collection de commandes dans le langage SQL peut être assemblée et définie comme une fonction. En plus des requêtes SELECT, les commandes peuvent inclure des requêtes de modification des données (INSERT, UPDATE et DELETE), ainsi que d'autres commandes SQL. (La seule exception est que vous ne pouvez pas placer de commandes BEGIN, COMMIT, ROLLBACK ou SAVEPOINT dans une fonction SQL.) Néanmoins, la commande finale doit être un SELECT qui renvoie ce qui a été spécifié comme type de retour de la fonction. Autrement, si vous voulez définir une fonction SQL qui réalise des actions mais n'a pas de valeur utile à renvoyer, vous pouvez la définir comme renvoyant void. Dans ce cas, le corps de la fonction ne doit pas finir avec un SELECT. Par exemple, cette fonction supprime les lignes avec des salaires négatifs depuis la table emp :

```
CREATE FUNCTION nettoie_emp() RETURNS void AS '
DELETE FROM emp
WHERE salaire < 0;
' LANGUAGE SQL;

SELECT nettoie_emp();

nettoie_emp
-----------

(1 row)
```

La syntaxe de la commande CREATE FUNCTION requiert que le corps de la fonction soit écrit comme une constante de type chaîne. Il est habituellement plus agréable d'utiliser les guillemets dollar (voir Section 4.1.2.2) pour cette constante. Si vous choisissez d'utiliser la syntaxe habituelle avec des guillemets simples, vous devez échapper les marques de guillemet simple (') et les antislashs (\) utilisés dans le corps de la fonction, typiquement en les doublant (voir Section 4.1.2.1).

Les arguments de la fonction SQL sont référencés dans le corps de la fonction en utilisant la syntaxe suivante. $*n*:$1 se réfère au premier argument, $2 au second et ainsi de suite. Si un argument est de type composite, on utilisera la notation par point, par exemple $1.name, pour accéder aux attributs de l'argument. Les arguments peuvent seulement être utilisés comme valeurs des données, pas comme des identifieurs. Du coup, par exemple, ceci est raisonnable :

```
INSERT INTO matable VALUES ($1);
```

mais ceci ne fonctionnera pas :

```
INSERT INTO $1 VALUES (42);
```

## 31.4.1. Fonctions SQL sur les types de base

La fonction SQL la plus simple possible n'a pas d'argument et retourne un type de base tel que integer :

```
CREATE FUNCTION un() RETURNS integer AS $$
```

```
    SELECT 1 AS resultat;
$$ LANGUAGE SQL;

-- Autre syntaxe pour les chaînes littérales :
CREATE FUNCTION un() RETURNS integer AS '
    SELECT 1 AS resultat;
' LANGUAGE SQL;

SELECT un();

 un
----
  1
```

Notez que nous avons défini un alias de colonne avec le nom `resultat` dans le corps de la fonction pour se référer au résultat de la fonction mais cet alias n'est pas visible hors de la fonction. En effet, le résultat est nommé `un` au lieu de `resultat`.

Il est presque aussi facile de définir des fonctions SQL acceptant des types de base comme arguments. Dans l'exemple suivant, remarquez comment nous faisons référence aux arguments dans le corps de la fonction avec `$1` et `$2`.

```
CREATE FUNCTION ajoute(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT ajoute(1, 2) AS reponse;

 reponse
---------
       3
```

Voici une fonction plus utile, qui pourrait être utilisée pour débiter un compte bancaire :

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS integer AS $$
    UPDATE banque
        SET balance = balance - $2
        WHERE no_compte = $1;
    SELECT 1;
$$ LANGUAGE SQL;
```

Un utilisateur pourrait exécuter cette fonction pour débiter le compte 17 de 100 000 euros ainsi :

```
SELECT tf1(17, 100.000);
```

Dans la pratique, on préférera vraisemblablement un résultat plus utile que la constante 1. Une définition plus probable sera :

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
    UPDATE banque
        SET balance = balance - $2
        WHERE no_compte = $1;
    SELECT balance FROM banque WHERE no_compte = $1;
$$ LANGUAGE SQL;
```

qui ajuste le solde et renvoie sa nouvelle valeur.

## 31.4.2. Fonctions SQL sur les types composites

Quand nous écrivons une fonction avec des arguments de type composite, nous devons non seulement spécifier quel argument utiliser (comme nous l'avons fait précédemment avec `$1` et `$2`), mais aussi spécifier l'attribut désiré de cet argument (champ). Par exemple, supposons que `emp` soit le nom d'une table contenant des données sur les employés et donc également le nom du type composite correspondant à chaque ligne de la table. Voici une fonction `double_salaire` qui calcule ce que serait le salaire de quelqu'un s'il était doublé :

```
CREATE TABLE emp (
    nom         text,
    salaire     numeric,
    age         integer,
    cubicle     point
);

CREATE FUNCTION double_salaire(emp) RETURNS numeric AS $$
    SELECT $1.salaire * 2 AS salaire;
$$ LANGUAGE SQL;

SELECT nom, double_salaire(emp.*) AS reve
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | reve
------+-------
 Bill |  8400
```

Notez l'utilisation de la syntaxe `$1.salaire` pour sélectionner un champ dans la valeur de la ligne argument. Notez également comment la commande `SELECT` utilise `*` pour sélectionner la ligne courante entière de la table comme une valeur composite (`emp`). La ligne de la table peut aussi être référencée en utilisant seulement le nom de la table ainsi :

```
SELECT nom, double_salaire(emp) AS reve
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

mais cette utilisation est obsolète car elle est facilement obscure.

Quelque fois, il est pratique de construire une valeur d'argument composite en direct. Ceci peut se faire avec la construction `ROW`. Par exemple, nous pouvons ajuster les données passées à la fonction :

```
SELECT nom, double_salaire(ROW(nom, salaire*1.1, age, cubicle)) AS reve
        FROM emp;
```

Il est aussi possible de construire une fonction qui renvoie un type composite. Voici un exemple de fonction renvoyant une seule ligne de type `emp` :

```
CREATE FUNCTION nouvel_emp() RETURNS emp AS $$
    SELECT text 'Aucun' AS nom,
        1000.0 AS salaire,
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

Dans cet exemple, nous avons spécifié chacun des attributs avec une valeur constante, mais un quelconque calcul aurait pu être substitué à ces valeurs.

Notez deux aspects importants à propos de la définition de fonction :

- L'ordre de la liste du select doit être exactement le même que celui dans lequel les colonnes apparaissent dans la table associée au type composite (donner des noms aux colonnes dans le corps de la fonction, comme nous l'avons fait dans l'exemple, n'a aucune interaction avec le système).
- Vous devez transtyper les expressions pour concorder avec la définition du type composite ou bien vous aurez l'erreur suivante :

```
ERROR:  function declared to return emp returns varchar instead of text at
column 1
```

Un autre façon de définir la même fonction est :

```
CREATE FUNCTION nouveau_emp() RETURNS emp AS $$
SELECT ROW('Aucun', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Ici, nous écrivons un `SELECT` qui renvoie seulement une colonne du bon type composite. Ceci n'est pas vraiment meilleur dans cette situation mais c'est une alternative pratique dans certains cas — par exemple, si nous avons besoin de calculer le résultat en appelant une autre fonction qui renvoie la valeur composite désirée.

Nous pourrions appeler cette fonction directement de deux façons :

```
SELECT nouveau_emp();

nouveau_emp
--------------------------
(None,1000.0,25,"(2,2)")

SELECT * FROM nouveau_emp();

nom   | salaire | age | cubicle
------+---------+-----+---------
Aucun |  1000.0 |  25 | (2,2)
```

La deuxième façon est décrite plus complètement dans Section 31.4.3.

Quand vous utilisez une fonction qui renvoie un type composite, vous pourriez vouloir seulement un champ (attribut) depuis ce résultat. Vous pouvez le faire avec cette syntaxe :

```
SELECT (nouveau_emp()).nom;

 name
------
 None
```

Les parenthèses supplémentaires sont nécessaires pour éviter une erreur de l'analyseur. Si vous essayez de le faire sans, vous obtiendrez quelque chose comme ceci :

```
SELECT nouveau_emp().nom;
ERROR:  syntax error at or near "." at character 17
LINE 1: SELECT nouveau_emp().name;
                        ^
```

Une autre option est d'utiliser la notation fonctionnelle pour extraire un attribut. Une manière simple d'expliquer cela, est de dire que nous pouvons échanger les notations `attribut(table)` et `table.attribut`.

```
SELECT nom(nouveau_emp());

 name
------
 None

-- C'est la même chose que
-- SELECT emp.nom AS leplusjeune FROM emp WHERE emp.age < 30;

SELECT nom(emp) AS leplusjeune FROM emp WHERE age(emp) < 30;

 leplusjeune
-------------
 Sam
 Andy
```

> **Astuce :** L'équivalence entre la notation fonctionnelle et la notation d'attribut rend possible l'utilisation de fonctions sur des types composite pour émuler les << champs calculés >>. Par exemple, en utilisant la définition précédente pour `double_salaire(emp)`, nous pouvons écrire
>
> ```
> SELECT emp.nom, emp.double_salaire FROM emp;
> ```
>
> Une application utilisant ceci n'aurait pas besoin d'être consciente directement que `double_salaire` n'est pas une vraie colonne de la table. (Vous pouvez aussi émuler les champs calculés avec des vues.)

Une autre façon d'utiliser une fonction renvoyant un résultat ligne est de passer le résultat à une autre fonction qui accepte le bon type ligne en entrée :

```
CREATE FUNCTION recupnom(emp) RETURNS text AS $$
    SELECT $1.nom;
$$ LANGUAGE SQL;

SELECT recupnom(nouveau_emp());
 recupnom
----------
 Aucun
(1 row)
```

Une autre façon d'utiliser une fonction qui renvoie une type composite est de l'appeler comme une fonction de table (décrite ci−dessous).

## 31.4.3. Fonctions SQL comme sources de table

Toutes les fonctions SQL peuvent être utilisées dans la clause FROM d'une requête mais ceci est particulièrement utile pour les fonctions renvoyant des types composite. Si la fonction est définie pour renvoyer un type de base, la fonction table produit une table d'une seule colonne. Si la fonction est définie pour renvoyer un type composite, la fonction table produit une colonne pour chaque attribut du type composite.

Voici un exemple :

```
CREATE TABLE foo (fooid int, foosousid int, foonom text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION recupfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(2 rows)
```

Comme le montre cet exemple, nous pouvons travailler avec les colonnes du résultat de la fonction comme s'il s'agissait des colonnes d'une table normale.

Notez que nous n'obtenons qu'une ligne comme résultat de la fonction. Ceci parce que nous n'avons pas utilisé l'instructionSETOF. Cette instruction est décrite dans la prochaine section.

## 31.4.4. Fonctions SQL renvoyant un ensemble

Quand une fonction SQL est déclarée renvoyer un SETOF *untype*, la requête finale SELECT de la fonction est complètement exécutée et chaque ligne extraite est renvoyée en tant qu'élément de l'ensemble résultat.

Cette caractéristique est normalement utilisée lors de l'appel d'une fonction dans une clause FROM. Dans ce cas, chaque ligne renvoyée par la fonction devient une ligne de la table vue par la requête. Par exemple, supposons que la table foo ait le même contenu que précédemment et écrivons :

```
CREATE FUNCTION recupfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM recupfoo(1) AS t1;
```

Alors nous obtenons :

```
 fooid | foosousid | foonom
-------+-----------+--------
     1 |         1 | Joe
     1 |         2 | Ed
(2 rows)
```

Actuellement, les fonctions renvoyant des ensembles peuvent aussi être appelées dans la liste du select d'une requête. Pour chaque ligne générée par la requête, la fonction renvoyant un ensemble est appelée et une ligne est générée pour chaque élément de l'ensemble résultat. Notez cependant que cette fonctionnalité est déconseillée et pourra être supprimée dans des versions futures. Voici un exemple de fonction renvoyant un ensemble à partir de la liste d'un select :

```
CREATE FUNCTION listeenfant(text) RETURNS SETOF text AS $$
    SELECT nom FROM noeuds WHERE parent = $1
$$ LANGUAGE SQL;

SELECT * FROM noeuds;
   nom        | parent
--------------+--------
 Haut         |
 Enfant1      | Haut
 Enfant2      | Haut
 Enfant3      | Haut
 Sous-Enfant1 | Enfant1
 Sous-Enfant2 | Enfant1
(6 rows)

SELECT listeenfant('Haut');
 listeenfant
--------------
 Enfant1
 Enfant2
 Enfant3
(3 rows)

SELECT nom, listeenfant(nom) FROM noeuds;
  nom    | listeenfant
---------+--------------
 Haut    | Enfant1
 Haut    | Enfant2
 Haut    | Enfant3
 Enfant1 | Sous-Enfant1
 Enfant1 | Sous-Enfant2
(5 rows)
```

Notez, dans le dernier `SELECT`, qu'aucune ligne n'est renvoyée pour `Enfant2`, `Enfant3`, etc. C'est parce que la fonction `listeenfant` renvoie un ensemble vide pour ces arguments et ainsi aucune ligne n'est générée.

## 31.4.5. Fonctions SQL polymorphes

Les fonctions SQL peuvent être déclarées pour accepter et renvoyer les types << polymorphe >> `anyelement` et `anyarray`. Voyez la Section 31.2.1 pour une explication plus approfondie. Voici une fonction polymorphe `cree_tableau` qui construit un tableau à partir de deux éléments de type arbitraire :

```
CREATE FUNCTION cree_tableau(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
SELECT cree_tableau(1, 2) AS tableau_entier, cree_tableau('a'::text, 'b') AS
tableau_texte;

 tableau_entier | tableau_texte
----------------+---------------
 {1,2}          | {a,b}
```

```
(1 row)
```

Notez l'utilisation du transtypage `'a'::text` pour spécifier le type `text` de l'argument. Ceci est nécessaire si l'argument est une chaîne de caractères car, autrement, il serait traité comme un type `unknown`, et un tableau de type `unknown` n'est pas un type valide. Sans le transtypage, vous obtiendrez ce genre d'erreur :

```
ERROR:  could not determine ANYARRAY/ANYELEMENT type because input is UNKNOWN
```

Il est permis d'avoir des arguments polymorphes avec un type de renvoi fixe, mais non l'inverse. Par exemple :

```
CREATE FUNCTION est_plus_grand(anyelement, anyelement) RETURNS bool AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT est_plus_grand(1, 2);
 est_plus_grand
----------------
 f
(1 row)

CREATE FUNCTION fonction_invalide() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result datatype
DETAIL:  A function returning ANYARRAY or ANYELEMENT must have at least one
argument of either type.
```

# 31.5. Surcharge des fonctions

Plusieurs fonctions peuvent être définies avec le même nom SQL, à condition que les arguments soient différents. En d'autres termes, les noms de fonction peuvent être *surchargés*. Quand une requête est exécutée, le serveur déterminera la fonction à appeler à partir des types de données des arguments et du nombre d'arguments. La surcharge peut aussi être utilisée pour simuler des fonctions avec un nombre variable d'arguments, jusqu'à un nombre maximum fini.

Lors de la création d'une famille de fonctions surchargées, vous devriez être attentif à ne pas créer d'ambiguïtés. Par exemple, avec les fonctions

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

il n'est pas immédiatement clair de savoir quelle fonction sera appelée avec une entrée triviale comme `test(1, 1.5)`. Les règles de résolution actuellement implémentées sont décrites dans Chapitre 10, mais il est déconseillé de concevoir un système qui serait basé subtilement sur ce comportement.

Une fonction qui prend un seul argument d'un type composite devrait généralement ne pas avoir le même nom que tout attribut (champ) de ce type. Rappellez–vous que `attribut(table)` est considéré comme équivalent à `table.attribut`. Dans le cas où il existe une ambiguïté entre une fonction sur un type composite et sur un attribut d'un type composite, l'attribut sera toujours utilisé. Il est possible de contourner ce choix en qualifiant le nom de la fonction avec celui du schéma (c'est–à–dire `schema.fonction(table)`) mais il est préférable d'éviter le problème en ne choisissant aucun nom conflictuel.

Lors de la surcharge de fonctions en langage C, il existe une contrainte supplémentaire : le nom C de chaque fonction dans la famille des fonctions surchargées doit être différent des noms C de toutes les autres fonctions, soit internes soit chargées dynamiquement Si cette règle est violée, le comportement n'est pas portable. Vous pourriez obtenir une erreur de l'éditeur de lien ou une des fonctions sera appelée (habituellement l'interne). L'autre forme de clause AS pour la commande SQL CREATE FUNCTION découple le nom de la fonction SQL à partir du nom de la fonction dans le code source C. Par exemple,

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
LANGUAGE C;
```

Les noms des fonctions C reflètent ici une des nombreuses conventions possibles.

# 31.6. Catégories de volatibilité des fonctions

Chaque fonction a une classification *volatility* comprenant VOLATILE, STABLE ou IMMUTABLE. VOLATILE est la valeur par défaut si la commande CREATE FUNCTION ne spécifie pas de catégorie. La catégorie de volatibilité est une promesse à l'optimiseur sur le comportement de la fonction :

- Une fonction VOLATILE peut tout faire, y compris modifier la base de données. Elle peut renvoyer différents résultats sur des appels successifs avec les mêmes arguments. L'optimiseur ne fait aucune supposition sur le comportement de telles fonctions. Une requête utilisant une fonction volatile ré−évaluera la fonction à chaque ligne où sa valeur est nécessaire.
- Une fonction STABLE ne peut pas modifier la base de données et est garantie de renvoyer les mêmes résultats si elle est appelée avec les mêmes arguments pour tous les appels à l'intérieur d'une même requête. Cette catégorie permet à l'optimiseur d'optimiser plusieurs appels de la fonction dans une seule requête. En particulier, vous pouvez utiliser en toute sécurité une expression contenant une telle fonction dans une condition de parcours d'index. (Car un parcours d'index évaluera la valeur de la comparaison une seule fois, pas une fois pour chaque ligne, il n'est pas valide d'utiliser une fonction VOLATILE dans une condition de parcours d'index.)
- Une fonction IMMUTABLE ne peut pas modifier la base de données et est garantie de toujours renvoyer les mêmes résultats si elle est appelée avec les mêmes arguments. Cette catégorie permet à l'optimiseur de pré−évaluer la fonction quand une requête l'appelle avec des arguments constants. Par exemple, une requête comme SELECT ... WHERE x = 2 + 2 peut être simplifiée pour obtenir SELECT ... WHERE x = 4 car la fonction sous−jacente de l'opérateur d'addition est indiquée IMMUTABLE.

Pour une meilleure optimisation des résultats, vous devez mettre un label sur les fonctions avec la catégorie la plus volatile valide pour elles.

Toute fonction avec des effets de bord *doit* être indiquée comme VOLATILE, de façon à ce que les appels ne puissent pas être optimisés. Même une fonction sans effets de bord doit être indiquée comme VOLATILE si sa valeur peut changer à l'intérieur d'une seule requête ; quelques exemples sont random(), currval(), timeofday().

Il y a relativement peu de différences entre les catégories STABLE et IMMUTABLE en considérant les requêtes interactives qui sont planifiées et immédiatement exécutées : il importe peu que la fonction soit exécutée une

fois lors de la planification ou une fois au lancement de l'exécution de la requête mais cela fait une grosse différence si le plan est sauvegardé et utilisé plus tard. Placer un label `IMMUTABLE` sur une fonction quand elle ne l'est pas vraiment pourrait avoir comme conséquence de la considérer prématurément comme une constante lors de la planification, et résulterait en une valeur erronée lors d'une utilisation ultérieure de ce plan d'exécution. C'est un danger qui arrive lors de l'utilisattion d'instructions préparées ou avec l'utilisation de langages de fonctions mettant les plans d'exécutions en cache (comme PL/pgSQL).

À cause du comportement à base d'images de MVCC (voir Chapitre 12), une fonction contenant seulement des commandes `SELECT` peut être indiquée `STABLE` en toute sécurité même si il sélectionne des données à partir de tables qui pourraient avoir subi des modifications entre temps par des requêtes concurrentes. PostgreSQL exécutera une fonction `STABLE` en utilisant l'image établie par la requête appelant et n'aura qu'une vision figée de la base de données au cours de la requête. Notez aussi que la famille de fonctions `current_timestamp` est qualifiée de stable car leurs valeurs ne changent pas dans une transaction.

Ce même comportement d'images est utilisé pour les commandes `SELECT` à l'intérieur de fonctions `IMMUTABLE`. Il est généralement déconseillé de sélectionner des tables de la base de données à l'intérieur de fonctions `IMMUTABLE` car l'immutabilité sera rompue si le contenu de la table change. Néanmoins, PostgreSQL ne vous force pas à ne pas le faire.

Une erreur commune est de placer un label sur une fonction `IMMUTABLE` quand son résultat dépend d'un paramètre de configuration. Par exemple, une fonction qui manipule des types date/heure pourrait bien avoir des résultats dépendant du paramètre timezone. Pour être sécurisées, de telles fonctions devraient avoir le label `STABLE` à la place.

> **Note :** Avant PostgreSQL version 8.0, le prérequis que les fonctions `STABLE` et `IMMUTABLE` ne pouvaient pas modifier la base de données n'était pas contraint par le système. La version 8.0 le contraint en réclamant que les fonctions SQL et les fonctions de langages de procédures de ces catégories ne contiennent pas de commandes SQL autre que `SELECT`. (Ceci n'a pas été complètement testé car de telles fonctions pourraient toujours appeler des fonctions `VOLATILE` qui modifient la base de données. Si vous le faites, vous trouverez que la fonction `STABLE` ou `IMMUTABLE` n'est pas au courant des modifications effectuées sur la base de données par la fonction appelée.)

## 31.7. Fonctions en langage de procédures

PostgreSQL autorise l'écriture de fonctions définies par l'utilisateur dans d'autres langages que SQL et C. Ces autres langages sont appelés des *langages de procédure* (PL). Les langages de procédures ne sont pas compilés dans le serveur PostgreSQL ; ils sont fournis comme des modules chargeables. Voir Chapitre 34 et les chapitres suivantes pour plus d'informations.

Il y a actuellement quatre langages de procédures disponibles dans la distribution PostgreSQL standard : PL/pgSQL, PL/Tcl, PL/Perl et PL/Python. Référez−vous au Chapitre 34 pour plus d'informations. D'autres langages peuvent être définis par les utilisateurs. Les bases du développement d'un nouveau langage de procédures sont traitées dans le Chapitre 45.

## 31.8. Fonctions internes

Les fonctions internes sont des fonctions écrites en C qui ont été liées de façon statique dans le serveur PostgreSQL. Le << corps >> de la définition de la fonction spécifie le nom en langage C de la fonction, qui n'est pas obligatoirement le même que le nom déclaré pour l'utilisation en SQL. (Pour des raisons de rétro compatibilité, un corps vide est accepté pour signifier que le nom de la fonction en langage C est le même que le nom SQL).

Normalement, toutes les fonctions internes présentes dans le serveur sont déclarées pendant l'initialisation du groupe de base de données (`initdb`), mais un utilisateur peut utiliser la commande `CREATE FUNCTION` pour créer des noms d'alias supplémentaires pour une fonction interne. Les fonctions internes sont déclarées dans la commande `CREATE FUNCTION` avec le nom de langage `internal`. Par exemple, pour créer un alias de la fonction `sqrt` :

```
CREATE FUNCTION racine_carree(double precision) RETURNS double precision    AS
'dsqrt'
LANGUAGE internal    STRICT;
```

(La plupart des fonctions internes doivent être déclarées << STRICT >>.)

> **Note :** Toutes les fonctions << prédéfinies >> ne sont pas internes (au sens explicité ci−dessus). Quelques fonctions prédéfinies sont écrites en SQL.

# 31.9. Fonctions en langage C

Les fonctions définies par l'utilisateur peuvent être écrites en C (ou dans un langage pouvant être rendu compatible avec C, comme C++). Ces fonctions sont compilées en objets dynamiques chargeables (encore appelés bibliothèques partagées) et sont chargées par le serveur à la demande. Cette caractéristique de chargement dynamique est ce qui distingue les fonctions en << langage C >> des fonctions << internes >> — les véritables conventions de codage sont essentiellement les mêmes pour les deux. (C'est pourquoi la bibliothèque standard de fonctions internes est une source abondante d'exemples de code pour les fonctions C définies par l'utilisateur.)

Deux différentes conventions d'appel sont actuellement en usage pour les fonctions C. La plus récente convention d'appel, << version 1 >>, est indiquée en écrivant une macro d'appel `PG_FUNCTION_INFO_V1()` comme illustré ci−après. L'absence d'une telle macro indique une fonction écrite selon l'ancien style (<< version 0 >>). Le nom de langage spécifié dans la commande `CREATE FUNCTION` est `C` dans les deux cas. Les fonctions suivant l'ancien style sont maintenant déconseillées en raison de problèmes de portabilité et d'un manque de fonctionnalité, mais elles sont encore supportées pour des raisons de compatibilité.

## 31.9.1. Chargement dynamique

La première fois qu'une fonction définie par l'utilisateur dans un fichier objet particulier chargeable est appelée dans une session, le chargeur dynamique charge ce fichier objet en mémoire de telle sorte que la fonction peut être appelée. La commande `CREATE FUNCTION` pour une fonction en C définie par l'utilisateur doit par conséquent spécifier deux éléments d'information pour la fonction : le nom du fichier objet chargeable et le nom en C (lien symbolique) de la fonction spécifique à appeler à l'intérieur de ce fichier objet. Si le nom en C n'est pas explicitement spécifié, il est supposé être le même que le nom de la fonction SQL.

L'algorithme suivant, basé sur le nom donné dans la commande CREATE FUNCTION, est utilisé pour localiser le fichier objet partagé:

1. Si le nom est un chemin absolu, le fichier est chargé.
2. Si le nom commence par la chaîne $libdir, cette chaîne est remplacée par le nom du répertoire de la bibliothèque du paquetage PostgreSQL, qui est déterminé au moment de la compilation.
3. Si le nom ne contient pas de partie répertoire, le fichier est recherché par le chemin spécifié dans la variable de configuration dynamic_library_path.
4. Dans les autres cas, (nom de fichier non trouvé dans le chemin ou ne contenant pas de partie répertoire non absolu), le chargeur dynamique essaiera d'utiliser le nom donné, ce qui échouera très vraisemblablement. (Il n'est pas fiable de dépendre du répertoire de travail en cours.)

Si cette séquence ne fonctionne pas, l'extension pour les noms de fichier des bibliothèques partagées spécifique à la plateforme (souvent .so) est ajoutée au nom attribué et la séquence est à nouveau tentée. En cas de nouvel échec, le chargement échoue.

L'identifiant utilisateur sous lequel fonctionne le serveur PostgreSQL doit pouvoir suivre le chemin jusqu'au fichier que vous essayez de charger. C'est une erreur fréquente de définir le fichier ou un répertoire supérieur comme non lisible et/ou non exécutable par l'utilisateur postgres.

Dans tous les cas, le nom de fichier donné dans la commande CREATE FUNCTION est enregistré littéralement dans les catalogues systèmes, de sorte que, si le fichier doit être à nouveau chargé, la même procédure sera appliquée.

> **Note :** PostgreSQL ne compilera pas une fonction C automatiquement. Le fichier objet doit être compilé avant d'être référencé dans une commande CREATE FUNCTION. Voir la Section 31.9.6 pour des informations complémentaires.

Après avoir été utilisé pour la première fois, un fichier objet chargé dynamiquement est conservé en mémoire. Les futurs appels de fonction(s) dans ce fichier pendant la même session provoquerons seulement une légère surcharge due à la consultation d'une table de symboles. Si vous devez forcer le chargement d'un fichier objet, par exemple après une recompilation, utilisez la commande LOAD ou commencez une nouvelle session.

Il est recommandé de localiser les bibliothèques dynamiques soit relativement à $libdir soit au moyen du chemin des bibliothèques dynamiques. Ceci simplifie la mise à jour dans le cas d'une nouvelle installation dans un endroit différent. Le répertoire effectif représenté par $libdir peut être retrouvé avec la commande pg_config --pkglibdir.

Avant la version 7.2 de PostgreSQL, seuls les chemins absolus exacts pouvaient être spécifiés dans la commande CREATE FUNCTION. Cette approche est maintenant déconseillée car elle rend la définition de la fonction inutilement non portable. Il vaut mieux seulement spécifier le nom de la bibliothèque partagée sans chemin ni extension, et laisser le mécanisme de recherche générer cette information.

## 31.9.2. Types de base dans les fonctions en langage C

Pour savoir comment écrire des fonctions en langage C, vous devez savoir comment PostgreSQL représente en interne les types de données de base et comment elles peuvent être passés vers et depuis les fonctions. En interne, PostgreSQL considère un type de base comme un << blob de mémoire >>. Les fonctions que vous définissez sur un type définissent à leur tour la façon pour PostgreSQL d'opérer sur lui. C'est–à–dire que

PostgreSQL ne fera que conserver et retrouver les données sur le disque et utilisera votre fonction pour entrer, traiter et restituer les données.

Les types de base peuvent avoir un des trois formats internes suivants :

- passage par valeur, longueur fixe ;
- passage par référence, longueur fixe ;
- passage par référence, longueur variable.

Les types par valeur peuvent seulement avoir une longueur de 1, 2 ou 4 octets (également 8 octets si `sizeof(Datum)` est de huit octets sur votre machine). Vous devriez être attentif lors de la définition de vos types de sorte à qu'ils aient la même taille sur toutes les architectures. Par exemple, le type `long` est dangereux car il a une taille de quatre octets sur certaines machines et huit octets sur d'autres, alors que le type `int` est de quatre octets sur la plupart des machines Unix. Une implémentation raisonnable du type `int4` sur une machine Unix pourrait être

```
/* entier sur quatre octets, passé par valeur */
typedef int int4;
```

D'autre part, les types à longueur fixe d'une taille quelconque peuvent être passés par référence. Par exemple, voici l'implémentation d'un type PostgreSQL :

```
/* structure de 16 octets, passée par référence */
typedef struct
{
    double  x, y;
} Point;
```

Seuls des pointeurs vers de tels types peuvent être utilisés en les passant dans et hors des fonctions PostgreSQL. Pour renvoyer une valeur d'un tel type, allouez la quantité appropriée de mémoire avec `palloc`, remplissez la mémoire allouée et renvoyez un pointeur vers elle. (Vous pouvez aussi renvoyer directement une valeur d'entrée qui a le même type que la valeur de retour en renvoyant le pointeur vers la valeur d'entrée. Cependant, *ne jamais* modifier le contenu d'une valeur d'entrée passée par référence.)

Enfin, tous les types à longueur variable doivent aussi être passés par référence. Tous les types à longueur variable doivent commencer avec un champ d'une longueur d'exactement quatre octets et toutes les données devant être stockées dans ce type doivent être localisées dans la mémoire à la suite immédiate de ce champ longueur. Le champ longueur contient la longueur totale de la structure, c'est–à–dire incluant la longueur du champ longueur lui–même.

Comme exemple, nous pouvons définir le type `text` comme ceci :

```
typedef struct {
    int4 length;
    char data[1];
} text;
```

Il est évident que le champ déclaré ici n'est pas assez long pour contenir toutes les chaînes possibles. Comme il est impossible de déclarer une structure de taille variable en C, nous nous appuyons sur le fait que le compilateur C ne vérifie pas la plage des indices de tableau. Nous allouons juste la quantité d'espace nécessaire et ensuite nous accédons au tableau comme s'il avait été déclaré avec la bonne longueur. (C'est une astuce courante que vous pouvez trouver dans beaucoup de manuels de C).

En manipulant les types à longueur variable, nous devons être attentifs à allouer la quantité correcte de mémoire et à fixer correctement le champ longueur. Par exemple, si nous voulons stocker 40 octets dans une structure `text`, nous devrions utiliser un fragment de code comme celui−ci :

```
#include "postgres.h"
...
char buffer[40]; /* notre donnée source */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
destination->length = VARHDRSZ + 40;
memcpy(destination->data, buffer, 40);
...
```

`VARHDRSZ` est équivalent à `sizeof(int4)` mais est considéré comme une meilleure tournure de référence à la taille de l'overhead pour un type à longueur variable.

Tableau 31−1 spécifie la correspondance entre les types C et les types SQL quand on écrit une fonction en langage C utilisant les types intégrés à PostgreSQL. La colonne << Défini dans >> donne le fichier d'en−tête devant être inclus pour accéder à la définition du type. (La définition effective peut se trouver dans un fichier différent inclus dans le fichier indiqué. Il est recommandé que les utilisateurs s'en tiennent à l'interface définie.) Notez que vous devriez toujours inclure `postgres.h` en premier dans tout fichier source car il déclare un grand nombre d'éléments dont vous aurez besoin de toute façon.

**Tableau 31−1. Équivalence des types C et des types SQL intégrés**

| Type SQL | Type C | Défini dans |
|---|---|---|
| `abstime` | `AbsoluteTime` | `utils/nabstime.h` |
| `boolean` | `bool` | `postgres.h` (intégration au compilateur) |
| `box` | `BOX*` | `utils/geo_decls.h` |
| `bytea` | `bytea*` | `postgres.h` |
| `"char"` | `char` | (intégré au compilateur) |
| `character` | `BpChar*` | `postgres.h` |
| `cid` | `CommandId` | `postgres.h` |
| `date` | `DateADT` | `utils/date.h` |
| `smallint` (`int2`) | `int2` or `int16` | `postgres.h` |
| `int2vector` | `int2vector*` | `postgres.h` |
| `integer` (`int4`) | `int4` or `int32` | `postgres.h` |
| `real` (`float4`) | `float4*` | `postgres.h` |
| `double precision` (`float8`) | `float8*` | `postgres.h` |
| `interval` | `Interval*` | `utils/timestamp.h` |
| `lseg` | `LSEG*` | `utils/geo_decls.h` |
| `name` | `Name` | `postgres.h` |
| `oid` | `Oid` | `postgres.h` |
| `oidvector` | `oidvector*` | `postgres.h` |
| `path` | `PATH*` | `utils/geo_decls.h` |
| `point` | `POINT*` | `utils/geo_decls.h` |

| | | |
|---|---|---|
| regproc | regproc | postgres.h |
| reltime | RelativeTime | utils/nabstime.h |
| text | text* | postgres.h |
| tid | ItemPointer | storage/itemptr.h |
| time | TimeADT | utils/date.h |
| time with time zone | TimeTzADT | utils/date.h |
| timestamp | Timestamp* | utils/timestamp.h |
| tinterval | TimeInterval | utils/nabstime.h |
| varchar | VarChar* | postgres.h |
| xid | TransactionId | postgres.h |

Maintenant que nous avons passé en revue toutes les structures possibles pour les types de base, nous pouvons donner quelques exemples de vraies fonctions.

## 31.9.3. Conventions d'appel de la version 0 pour les fonctions en langage C

Nous présentons l'<< ancien style >> de convention d'appel en premier — bien que cette approche soit maintenant déconseillée, elle est plus facile à maîtriser au début. Dans la méthode version−0, les arguments et résultats de la fonction C sont simplement déclarés dans le style C normal mais en faisant attention à utiliser la représentation C de chaque type de données SQL comme montré ci−dessus.

Voici quelques exemples :

```
#include "postgres.h"
#include <string.h>

/* par valeur */

int
add_one(int arg)
{
    return arg + 1;
}

/* par référence, taille fixe */

float8 *
add_one_float8(float8 *arg)
{
    float8    *result = (float8 *) palloc(sizeof(float8));

    *result = *arg + 1.0;

    return result;
}

Point *
makepoint(Point *pointx, Point *pointy)
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;
```

```
    return new_point;
}

/* par référence, taille variable */

text *
copytext(text *t)
{
    /*
     * VARSIZE est la taille totale de la structure en octets.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(new_t) = VARSIZE(t);
    /*
     * VARDATA est un pointeur sur la région de données de la structure.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t)-VARHDRSZ);    /* nombre d'octets */
    return new_t;
}

text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    VARATT_SIZEP(new_text) = new_text_size;
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    return new_text;
}
```

En supposant que le code ci–dessus ait été écrit dans le fichier `funcs.c` et compilé en objet partagé, nous pourrions définir les fonctions pour PostgreSQL avec des commandes comme ceci :

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'DIRECTORY/funcs', 'add_one'
     LANGUAGE C STRICT;

-- notez la surcharge du nom de la fonction SQL "add_one"
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'DIRECTORY/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'DIRECTORY/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'DIRECTORY/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'DIRECTORY/funcs', 'concat_text',
     LANGUAGE C STRICT;
```

Ici, *DIRECTORY* représente le répertoire contenant le fichier de la bibliothèque partagée (par exemple le répertoire du tutoriel de PostgreSQL, qui contient le code des exemples utilisés dans cette section). (Un

meilleur style aurait été d'écrire seulement `'funcs'` dans la clause `AS`, après avoir ajouté *`DIRECTORY`* au chemin de recherche. Dans tous les cas, nous pouvons omettre l'extension spécifique au système pour les bibliothèques partagées, communément `.so` ou `.sl`.)

Remarquez que nous avons spécifié la fonction comme << STRICT >>, ce qui signifie que le système devra automatiquement supposer un résultat NULL si n'importe quelle valeur d'entrée est NULL. Ainsi, nous évitons d'avoir à vérifier l'existence d'entrées NULL dans le code de la fonction. Sinon, nous aurions dû contrôler explicitement les valeurs NULL en testant un pointeur NULL pour chaque argument passé par référence. (Pour les arguments passés par valeur, nous n'aurions même aucun moyen de contrôle!)

Bien que cette convention d'appel soit simple à utiliser, elle n'est pas très portable ; sur certaines architectures, il y a des problèmes pour passer de cette manière des types de données plus petits que `int`. De plus, il n'y a pas de moyen simple de renvoyer un résultat NULL, ni de traiter des arguments NULL autrement qu'en rendant la fonction strict. La convention version−1, présentée ci−après, permet de surmonter ces objections.

## 31.9.4. Conventions d'appel de la version 1 pour les fonctions en langage C

La convention d'appel version−1 repose sur des macros pour supprimer la plus grande partie de la complexité du passage d'arguments et de résultats. La déclaration C d'une fonction en version−1 est toujours :

```
Datum funcname(PG_FUNCTION_ARGS)
```

De plus, la macro d'appel

```
PG_FUNCTION_INFO_V1(funcname);
```

doit apparaître dans le même fichier source. (Par convention, elle est écrite juste avant la fonction elle−même.) Cette macro n'est pas nécessaire pour les fonctions `internal` puisque PostgreSQL assume que toutes les fonctions internes utilisent la convention version−1. Elle est toutefois requise pour les fonctions chargées dynamiquement.

Dans une fonction version−1, chaque argument existant est traité par une macro `PG_GETARG_`*`xxx`*`()` correspondant au type de donnée de l'argument et le résultat est renvoyé par une macro `PG_RETURN_`*`xxx`*`()` correspondant au type renvoyé. `PG_GETARG_`*`xxx`*`()` prend comme argument le nombre d'arguments de la fonction à parcourir, le compteur commençant à 0. `PG_RETURN_`*`xxx`*`()` prend comme argument la valeur effective à renvoyer.

Voici la même fonction que précédemment, codée en style version−1

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"

/* par valeur */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);
```

```
    PG_RETURN_INT32(arg + 1);
}

/* par référence, longueur fixe */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* Les macros pou FLOAT8 cache sa nature de passage par référence. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Ici, la nature de passage par référence de Point n'est pas cachée. */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* par référence, longueur variable */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE est la longueur totale de la structure en octets.
     */
    text     *new_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(new_t) = VARSIZE(t);
    /*
     * VARDATA est un pointeur vers la région de données de la structure.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t)-VARHDRSZ);    /* nombre d'octets */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_P(0);
    text  *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
```

```
    text *new_text = (text *) palloc(new_text_size);

    VARATT_SIZEP(new_text) = new_text_size;
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

Les commandes `CREATE FUNCTION` sont les mêmes que pour leurs équivalents dans la version−0.

Au premier coup d'oeil, les conventions de codage de la version−1 peuvent sembler inutilement obscures. Pourtant, elles offrent nombre d'améliorations car les macros peuvent cacher les détails superflus. Un exemple est donné par la fonction `add_one_float8` où nous n'avons plus besoin de prêter attention au fait que le type `float8` est passé par référence. Un autre exemple de simplification est donné par les macros pour les types à longueur variable `GETARG` qui permettent un traitement plus efficace des valeurs << toasted >> (compressées ou hors−ligne).

Une des grandes améliorations dans les fonctions version−1 est le meilleur traitement des entrées et des résultats NULL. La macro `PG_ARGISNULL(`*`n`*`)` permet à une fonction de tester si chaque entrée est NULL. (Évidemment, ceci n'est nécessaire que pour les fonctions non déclarées << STRICT >>.) Comme avec les macros `PG_GETARG_`*`xxx`*`()`, les arguments d'entrée sont comptés à partir de zéro. Notez qu'on doit se garder d'exécuter `PG_GETARG_`*`xxx`*`()` jusqu'à ce qu'on ait vérifié que l'argument n'est pas NULL. Pour renvoyer un résultat NULL, exécutez la fonction `PG_RETURN_NULL()` ; ceci convient aussi bien dans les fonctions STRICT que non STRICT.

Les autres options proposées dans l'interface de nouveau style sont deux variantes des macros `PG_GETARG_`*`xxx`*`()`. La première d'entre elles, `PG_GETARG_`*`xxx`*`_COPY()`, garantit le renvoi d'une copie de l'argument spécifié, où nous pouvons écrire en toute sécurité. (Les macros normales peuvent parfois renvoyer un pointeur vers une valeur physiquement mise en mémoire dans une table qui ne doit pas être modifiée. En utilisant les macros `PG_GETARG_`*`xxx`*`_COPY()`, on garantit l'écriture du résultat.) La seconde variante se compose des macros `PG_GETARG_`*`xxx`*`_SLICE()` qui prennent trois arguments. Le premier est le nombre d'argument de la fonction (comme ci−dessus). Le second et le troisième sont le décalage et la longueur du segment qui doit être renvoyé. Les décalages sont comptés à partir de zéro et une longueur négative demande le renvoi du reste de la valeur. Ces macros procurent un accès plus efficace à des parties de valeurs à grande dimension dans le cas où elles ont un type de stockage en mémoire << external >>. (Le type de stockage d'une colonne peut être spécifié en utilisant `ALTER TABLE` *`tablename`* `ALTER COLUMN` *`colname`* `SET STORAGE` *`typestockage`*. *`typestockage`* est un type parmi `plain`, `external`, `extended` ou `main`.)

Enfin, les conventions d'appels de la version−1 rendent possible le renvoi de résultats d'ensemble (Section 31.9.10), l'implémentation de fonctions trigger (déclencheur) (Chapitre 32) et d'opérateurs d'appel de langage procédural (Chapitre 45). Le code version−1 est aussi plus portable que celui de version−0 car il ne contrevient pas aux restrictions du protocole d'appel de fonction en C standard. Pour plus de détails, voyez `src/backend/utils/fmgr/README` dans les fichiers sources de la distribution.

## 31.9.5. Écriture du code

Avant de nous intéresser à des sujets plus avancés, nous devons discuter de quelques règles de codage des fonctions en langage C de PostgreSQL. Bien qu'il soit possible de charger des fonctions écrites dans des langages autre que le C dans PostgreSQL, c'est habituellement difficile (quand c'est possible) parce que les autres langages comme C++, FORTRAN ou Pascal ne suivent pas fréquemment les mêmes conventions de

nommage que le C. C'est−à−dire que les autres langages ne passent pas les arguments et ne renvoient pas les valeurs entre fonctions de la même manière. Pour cette raison, nous supposerons que nos fonctions en langage C sont réellement écrites en C.

Les règles de base pour l'écriture de fonctions C sont les suivantes :

- Utilisez `pg_config --includedir-server` pour découvrir où sont installés les fichiers d'en−tête du serveur PostgreSQL sur votre système (ou sur le système de vos utilisateurs). Cette option est apparue dans PostgreSQL version 7.2. Pour PostgreSQL 7.1, vous devez utiliser l'option `--includedir`. (La commande `pg_config` terminera avec un statut différent de zéro si elle rencontre une option inconnue.) Pour les versions antérieures à la 7.1, vous devez deviner, mais puisque cela remonte avant l'introduction des conventions d'appel actuelles, il y a peu de chance que vous vouliez supporter ces versions.
- Quand vous allouez de la mémoire, utilisez les fonctions PostgreSQL `palloc` et `pfree` au lieu des fonctions correspondantes `malloc` et `free` de la bibliothèque C. La mémoire allouée par `palloc` sera libérée automatiquement à la fin de chaque transaction, empêchant des débordements de mémoire.
- Remettez toujours à zéro les octets de vos structures en utilisant `memset`. Sinon, il est difficile de supporter les index ou les jointures de découpage car vous devez retenir seulement les bits significatifs de votre structure de donnée pour calculer un découpage. Même si vous initialisez tous les champs de votre structure, il peut y avoir des remplissages d'alignement (trous dans la structure) pouvant contenir des valeurs parasites.
- La plupart des types internes PostgreSQL sont déclarés dans `postgres.h` alors que les interfaces de gestion des fonctions (`PG_FUNCTION_ARGS`, etc.) sont dans `fmgr.h`. Du coup, vous aurez besoin d'inclure au moins ces deux fichiers. Pour des raisons de portabilité, il vaut mieux inclure `postgres.h` *en premier*, avant tout autre fichier d'en−tête système ou utilisateur. En incluant `postgres.h`, il inclura également `elog.h` et `palloc.h` pour vous.
- Les noms de symboles définis dans les objets ne doivent pas entrer en conflit entre eux ou avec les symboles définis dans les exécutables du serveur PostgreSQL. Vous aurez à renommez vos fonctions ou variables si vous recevez un message d'erreur à cet effet.
- Compiler ou lier votre code pour qu'il soit chargé dynamiquement dans PostgreSQL exige toujours des options spéciales. Voir la Section 31.9.6 pour une explication détaillée sur la meilleure façon de le faire sur votre système particulier.

## 31.9.6. Compiler et lier des fonctions chargées dynamiquement

Avant d'être en mesure d'utiliser des fonctions d'extension écrites en C dans PostgreSQL, elles doivent être compilées et liées d'une manière particulière afin de produire un fichier pouvant être chargé dynamiquement par le serveur. Pour être plus précis, une *bibliothèque partagée* doit être créée.

Pour obtenir plus d'informations sur ce qui est abordé dans cette section, vous devrez lire la documentation de votre système d'exploitation, en particulier les pages traitant du compilateur C, `cc`, ainsi que celle sur l'éditeur de lien, `ld`. Par ailleurs, le code source de PostgreSQL contient plusieurs exemples fonctionnels contenu dans le répertoire `contrib`. Néanmoins, en vous appuyant sur ces exemples, vous créerez des modules dépendants de la disponibilité du code source de PostgreSQL.

La création de bibliothèques partagées est un processus analogue à celui utilisé pour lier des exécutables : en premier lieu, les sources sont compilées en fichiers objets puis sont liées ensemble. Les fichiers objets doivent être compilés sous la forme de *code à position indépendante* (PIC, acronyme de *position−independent code*). Conceptuellement, cela correspond au fait qu'ils peuvent être placés à une position arbitraire de la mémoire

lorsqu'ils sont chargés par l'exécutable. (Les fichiers objets destinés aux exécutables ne sont généralement pas compilés de cette manière.). La commande permettant de lier des bibliothèques partagées nécessite des options spéciales permettant de la distinguer d'une liaison pour un exécutable. Enfin, ceci est la théorie. La réalité est moins belle sur certains systèmes.

Dans les exemples suivants nous supposons que le code source est un fichier `foo.c` et que nous souhaitons créer une bibliothèque partagée `foo.so`. Le fichier objet intermédiaire sera appelé `foo.o` sauf si précisé autrement. Une bibliothèque partagée peut contenir plus d'un fichier objet. Ceci dit, nous n'en utiliserons qu'un ici.

BSD/OS
: L'option du compilateur pour créer des PIC est `-fpic`. L'option de l'éditeur de liens pour créer des bibliothèques partagées est `-shared`.

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

Ceci est applicable à partir de la version 4.0 de BSD/OS.

FreeBSD
: L'option du compilateur pour créer des PIC est `-fpic`. L'option de l'éditeur de liens pour créer des bibliothèques partagées est `-shared`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

Ceci est applicable à partir de la version 3.0 de FreeBSD.

HP-UX
: L'option du compilateur système pour créer des PIC est `+z`. Lorsque vous utilisez GCC, l'option est `-fpic`. Le commutateur de l'éditeur de liens pour les bibliothèques partagées est `-b`. Ainsi

```
cc +z -c foo.c
```

ou

```
gcc -fpic -c foo.c
```

puis

```
ld -b -o foo.sl foo.o
```

HP-UX utilise l'extension `.sl` pour les bibliothèques partagées, à la différence de la plupart des autres systèmes.

IRIX
: PIC est l'option par défaut. Aucune option de compilation particulière n'est nécessaire. Le commutateur de l'éditeur de liens pour produire des bibliothèques partagées est `-shared`.

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

Linux
: L'option de compilation pour créer des PIC est `-fpic`. Sur certaines plateformes et dans certaines situations, `-fPIC` doit être utilisé si `-fpic` ne fonctionne pas. Reportez vous au manuel de GCC pour plus d'informations. L'option de compilation pour créer des bibliothèques partagées est

`-shared`. Un exemple complet devrait ressembler à ce qui suit :

```
cc -fpic -c foo.c
cc -shared -o foo.so foo.o
```

MacOS X

Voici un exemple. Il suppose que les outils de développement sont installés.

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

L'option de compilation pour créer des PIC est `-fpic`. Pour les systèmes ELF, l'option de compilation pour lier les bibliothèques partagées est `-shared`. Sur les systèmes plus anciens n'étant pas ELF, on utilise `ld -Bshareable`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

L'option de compilation pour créer des PIC est `-fpic`. Les bibliothèques partagées peuvent être créées avec la commande suivante `ld -Bshareable`.

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

L'option de compilation pour créer des PIC est `-KPIC` avec le compilateur de Sun et `-fpic` avec GCC. Pour lier les bibliothèques partagées, l'option de compilation est respectivement `-G` ou `-shared`.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

ou

```
gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

Tru64 UNIX

Par défaut, on compile des PIC. Ainsi, aucune directive particulière n'est à fournir pour la compilation. Pour lier, des options spécifiques sont à fournir à `ld` :

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

Une procédure identique doit être employée dans le cas où GCC serait utilisé à la place du compilateur du système ; aucune option particulière n'est nécessaire.

UnixWare

L'option de compilation pour créer des PIC est `-KPIC` avec le compilateur SCO et `-fpic` avec GCC. Pour lier des bibliothèques partagées, les options respectives sont `-G` et `-shared`.

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

ou

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

> **Astuce :** Si cela vous semble compliqué, vous pouvez tenter d'utiliser GNU Libtool. Cet outil permet de s'affranchir des différences entre les nombreux systèmes au travers d'une interface uniformisée.

La bibliothèque partagée résultante peut être chargée dans PostgreSQL. Lorsque l'on spécifie le nom du fichier dans la commande `CREATE FUNCTION`, on doit lui donner le nom de la bibliothèque partagée et non celui du fichier objet intermédiaire. Notez bien que l'extension standard pour les bibliothèques partagées (en général `.so` ou `.sl`) peut être omise dans la commande `CREATE FUNCTION` et doit l'être pour obtenir une meilleure portabilité.

Référez−vous à la Section 31.9.1 pour savoir à quel endroit le serveur s'attend à trouver les fichiers de bibliothèques partagées.

## 31.9.7. Infrastructure de construction d'extensions

Si vous pensez distribuer vos modules d'extension PostgreSQL, configurer un système de construction portable peut être assez compliqué. Du coup, l'installation de PostgreSQL fournit une infrastructure de construction pour les extensions, appelée PGXS, pour que les modules d'extension simples puissent être construit simplement avec un serveur déjà installé. Notez que cette infrastructure n'a pas pour but d'être un ensemble de travail universel pouvant être utilisé pour construire tous les logiciels s'interfaçant avec PostgreSQL ; il automatise simplement les règles de construction communes pour les modules simples d'extension du serveur. Pour des paquetages plus complexes, vous aurez besoin d'écrire votre propre système de construction.

Pour utiliser l'infrastructure de votre extension, vous devez écrire un simple fichier makefile. Dans ce fichier, vous devez configurer quelques variables et inclure enfin le makefile global PGXS. Voici un exemple qui construit un module d'extension nommé `isbn_issn` consistant en une bibliothèque partagée, un script SQL et un fichier texte de documentation :

```
MODULES = isbn_issn
DATA_built = isbn_issn.sql
DOCS = README.isbn_issn

PGXS := $(shell pg_config --pgxs)
include $(PGXS)
```

Les deux dernières lignes devraient toujours être identiques. Plus tôt dans le fichier, vous affectez des variables ou ajoutez vos propres règles pour make.

Les variables suivantes peuvent être configurées :

`MODULES`
: liste des objets partagés à construire à partir du fichier source de même base (ne pas inclure le suffixe dans cette liste)

`DATA`
: files spécifiques à installer dans *`prefix`*`/share/contrib`

`DATA_built`
: fichiers spécifiques à installer dans *`prefix`*`/share/contrib`, qui ont d'abord besoin d'être construit

`DOCS`

fichiers spécifiques à installer dans *prefix*/doc/contrib

SCRIPTS
: fichiers script (pas des binaires) à installer dans *prefix*/bin

SCRIPTS_built
: fichiers script (pas des binaires) à installer dans *prefix*/bin, qui ont d'abord besoin d'être construit

REGRESS
: liste des cas de tests de regression (sans suffixe)

ou au moins un parmi ces deux–là :

PROGRAM
: un programme binaire à construire (liste des fichiers objets dans OBJS)

MODULE_big
: un objet partagé à construire (liste des fichiers objets dans OBJS)

Ce qui suit peut être configuré :

EXTRA_CLEAN
: fichiers supplémentaires à supprimer dans make clean

PG_CPPFLAGS
: sera ajouté à CPPFLAGS

PG_LIBS
: sera ajouté à la ligne de liens PROGRAM

SHLIB_LINK
: sera ajouté à la ligne de lien MODULE_big

Nommez ce fichier Makefile et placez le dans le répertoire qui contient votre extension. Ensuite, vous pouvez lancer la compilation avec make, et plus tard make install pour installer votre module. L'extension est compilée et installée pour l'installation de PostgreSQL qui correspond à la première commande pg_config trouvée dans votre chemin.

## 31.9.8. Arguments de type composite dans les fonctions en langage C

Les types composites n'ont pas une organisation fixe comme les structures en C. Des instances d'un type composite peuvent contenir des champs NULL. De plus, les types composites faisant partie d'une hiérarchie d'héritage peuvent avoir des champs différents des autres membres de la même hiérarchie. En conséquence, PostgreSQL propose une interface de fonction pour accéder depuis le C aux champs des types composites.

Supposons que nous voulions écrire une fonction pour répondre à la requête

```
SELECT nom, c_surpaye(emp, 1500) AS surpaye
    FROM emp
    WHERE nom = 'Bill' OR nom = 'Sam';
```

En utilisant les conventions d'appel de la version 0, nous pouvons définir c_surpaye comme :

```
#include "postgres.h"
#include "executor/executor.h"  /* pour GetAttributeByName() */

bool
c_surpaye(HeapTupleHeader *t, /* la ligne courante d'emp */
          int32 limite)
```

```
{
    bool isNULL;
    int32 salaire;

    salaire = DatumGetInt32(GetAttributeByName(t, "salaire", &isNULL));
    if (isNULL)
        return false;
    return salaire > limite;
}
```

Dans le codage version−1, le code ci−dessus devient :

```
#include "postgres.h"
#include "executor/executor.h"  /* pour GetAttributeByName() */

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  *t = (HeapTupleHeader *) PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limite = PG_GETARG_INT32(1);
    bool isNULL;
    Datum salaire;

    salaire = GetAttributeByName(t, "salaire", &isNULL);
    if (isNULL)
        PG_RETURN_BOOL(false);
    /* Autrement, nous pourrions préférer de lancer PG_RETURN_NULL() pour un
       salaire NULL.
    */

    PG_RETURN_BOOL(DatumGetInt32(salaire) > limite);
}
```

`GetAttributeByName` est la fonction système PostgreSQL qui renvoie les attributs depuis une colonne spécifiée. Elle a trois arguments : l'argument de type `HeapTupleHeader` passé à la fonction, le nom de l'attribut recherché et un paramètre de retour qui indique si l'attribut est NULL. `GetAttributeByName` renvoie une valeur de type `Datum` que vous pouvez convertir dans un type voulu en utilisant la macro appropriée `DatumGet`*`XXX`*`()`. Notez que la valeur de retour est insignifiante si le commutateur NULL est positionné ; il faut toujours vérifier le commutateur NULL avant de commencer à faire quelque chose avec le résultat.

Il y a aussi `GetAttributeByNum`, qui sélectionne l'attribut cible par le numéro de colonne au lieu de son nom.

La commande suivante déclare la fonction `c_surpaye` en SQL :

```
CREATE FUNCTION c_surpaye(emp, integer) RETURNS boolean
    AS 'DIRECTORY/funcs', 'c_surpaye'
    LANGUAGE C STRICT;
```

Notez que nous avons utilisé `STRICT` pour que nous n'ayons pas à vérifier si les arguments en entrée sont NULL.

## 31.9.9. Renvoi de lignes (types composites) à partir de fonctions en langage C

Pour renvoyer une ligne ou une valeur de type composite à partir d'une fonction en langage C, vous pouvez utiliser une API spéciale qui fournit les macros et les fonctions dissimulant en grande partie de la complexité liée à la construction de types de données composites. Pour utiliser cette API, le fichier source doit inclure :

```
#include "funcapi.h"
```

Il existe deux façons de construire une valeur de données composites (autrement dit un << tuple >>) : vous pouvez le construire à partir d'un tableau de valeurs Datum ou à partir d'un tableau de chaînes C qui peuvent passer dans les fonctions de conversion des types de données du tuple. Quelque soit le cas, vous avez d'abord besoin d'obtenir et de construire un descripteur `TupleDesc` pour la structure du tuple. En travaillant avec des Datums, vous passez le `TupleDesc` à `BlessTupleDesc`, puis vous appelez `heap_formtuple` pour chaque ligne. En travaillant avec des chaînes C, vous passez `TupleDesc` à `TupleDescGetAttInMetadata`, puis vous appelez `BuildTupleFromCStrings` pour chaque ligne. Dans le cas d'une fonction renvoyant un ensemble de tuple, les étapes de configuration peuvent toutes être entreprises une fois lors du premier appel à la fonction.

Plusieurs fonctions d'aide sont disponibles pour configurer le `TupleDesc` initial. Si vous voulez utiliser un type composite nommé, vous pouvez récupérer l'information à partir du catalogue système. Utilisez

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

pour obtenir une `TupleDesc` pour une relation nommée ou

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

pour obtenir une `TupleDesc` basée sur l'OID d'un type. Ceci peut être utilisé pour obtenir un `TupleDesc` soit pour un type de base, soit pour un type composite. Lorsque vous écrivez une fonction qui renvoie `record`, le `TupleDesc` attendu doit être passé à l'appelant.

Une fois que vous avez un `TupleDesc`, appelez

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

Si vous pensez travailler avec des Datums ou

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

Si vous pensez travailler avec des chaînes C. Si vous écrivez une fonction renvoyant un ensemble, vous pouvez sauvegarder les résultats de ces fonctions dans la structure dans le `FuncCallContext` — utilisez le champ `tuple_desc` ou `attinmeta` respectivement.

Lorsque vous fonctionnez avec des Datums, utilisez

```
HeapTuple heap_formtuple(TupleDesc tupdesc, Datum *values, char *NULLs)
```

pour construire une donnée utilisateur `HeapTuple` indiquée dans le format Datum.

Lorsque vous travaillez avec des chaînes C, utilisez

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

pour construire une donnée utilisateur `HeapTuple` indiquée dans le format des chaînes C. `values` est un tableau de chaîne C, une pour chaque attribut de la ligne renvoyée. Chaque chaîne C doit être de la forme attendue par la fonction d'entrée du type de donnée de l'attribut. Afin de renvoyer une valeur NULL pour un des attributs, le pointeur correspondant dans le tableau de valeurs (`values`) doit être fixé à `NULL`. Cette fonction demandera à être appelée pour chaque ligne que vous renvoyez.

Une fois que vous avez construit un tuple devant être renvoyé par votre fonction, vous devez le convertir en type `Datum`. Utilisez

```
HeapTupleGetDatum(HeapTuple tuple)
```

pour convertir un type `HeapTuple` en un Datum valide. Ce `Datum` peut être renvoyé directement si vous envisagez de renvoyer juste une simple ligne ou bien il peut être utilisé pour renvoyer la valeur courante dans une fonction renvoyant un ensemble.

Un exemple figure dans la section suivante.

## 31.9.10. Renvoi d'ensembles depuis les fonctions en langage C

Il existe aussi une API spéciale procurant le moyen de renvoyer des ensembles (lignes multiples) depuis une fonction en langage C. Une fonction renvoyant un ensemble doit suivre les conventions d'appel de la version−1. Aussi, les fichiers source doivent inclure l'en−tête `funcapi.h`, comme ci−dessus.

Une fonction renvoyant un ensemble (SRF : << set returning function >>) est appelée une fois pour chaque élément qu'elle renvoie. La SRF doit donc sauvegarder suffisamment l'état pour se rappeler ce qu'elle était en train de faire et renvoyer le prochain élément à chaque appel. La structure `FuncCallContext` est offerte pour assister le contrôle de ce processus. À l'intérieur d'une fonction, `fcinfo->flinfo->fn_extra` est utilisée pour conserver un pointeur vers `FuncCallContext` au cours des appels successifs.

```
typedef struct
{
    /*
     * Number of times we've been called before
     *
     * call_cntr is initialized to 0 for you by SRF_FIRSTCALL_INIT(), and
     * incremented for you every time SRF_RETURN_NEXT() is called.
     */
    uint32 call_cntr;

    /*
     * OPTIONAL maximum number of calls
     *
     * max_calls is here for convenience only and setting it is optional.
     * If not set, you must provide alternative means to know when the
     * function is done.
     */
    uint32 max_calls;

    /*
     * OPTIONAL pointer to result slot
     *
     * This is obsolete and only present for backwards compatibility, viz,
     * user-defined SRFs that use the deprecated TupleDescGetSlot().
```

```
     */
    TupleTableSlot *slot;

    /*
     * OPTIONAL pointer to miscellaneous user-provided context information
     *
     * user_fctx is for use as a pointer to your own data to retain
     * arbitrary context information between calls of your function.
     */
    void *user_fctx;

    /*
     * OPTIONAL pointer to struct containing attribute type input metadata
     *
     * attinmeta is for use when returning tuples (i.e., composite data types)
     * and is not used when returning base data types. It is only needed
     * if you intend to use BuildTupleFromCStrings() to create the return
     * tuple.
     */
    AttInMetadata *attinmeta;

    /*
     * memory context used for structures that must live for multiple calls
     *
     * multi_call_memory_ctx is set by SRF_FIRSTCALL_INIT() for you, and used
     * by SRF_RETURN_DONE() for cleanup. It is the most appropriate memory
     * context for any memory that is to be reused across multiple calls
     * of the SRF.
     */
    MemoryContext multi_call_memory_ctx;

    /*
    * OPTIONAL pointer to struct containing tuple description
    *
    * tuple_desc is for use when returning tuples (i.e. composite data types)
    * and is only needed if you are going to build the tuples with
    * heap_formtuple() rather than with BuildTupleFromCStrings().  Note that
    * the TupleDesc pointer stored here should usually have been run through
    * BlessTupleDesc() first.
    */
    TupleDesc tuple_desc;

} FuncCallContext;
```

Une SRF utilise plusieurs fonctions et macros qui manipulent automatiquement la structure `FuncCallContext` (et s'attendent à la trouver via `fn_extra`). Utilisez

```
SRF_IS_FIRSTCALL()
```

pour déterminer si votre fonction est appelée pour la première fois. Au premier appel, utilisez

```
SRF_FIRSTCALL_INIT()
```

pour initialiser la structure `FuncCallContext`. À chaque appel de fonction, y compris le premier, utilisez

```
SRF_PERCALL_SETUP()
```

pour une mise à jour correcte en vue de l'utilisation de `FuncCallContext` et pour nettoyer toutes les données renvoyées précédemment et conservées depuis le dernier passage de la fonction.

Si votre fonction a des données à renvoyer, utilisez

```
SRF_RETURN_NEXT(funcctx, result)
```

pour les renvoyer à l'appelant. (`result` doit être de type `Datum`, soit une valeur simple, soit un tuple préparé comme décrit ci–dessus.) Enfin, quand votre fonction a fini de renvoyer des données, utilisez

```
SRF_RETURN_DONE(funcctx)
```

pour nettoyer et terminer la SRF.

Le contexte mémoire courant lors de l'appel de la SRF est un contexte transitoire qui est effacé entre les appels. Cela signifie que vous n'avez pas besoin d'appeler `pfree` sur tout ce que vous avez alloué en utilisant `palloc` ; ce sera supprimé de toute façon. Toutefois, si vous voulez allouer des structures de données devant persister tout au long des appels, vous avez besoin de les conserver quelque part ailleurs. Le contexte mémoire référencé par `multi_call_memory_ctx` est un endroit approprié pour toute donnée devant survivre jusqu'à l'achèvement de la fonction SRF. Dans la plupart des cas, cela signifie que vous devrez basculer vers `multi_call_memory_ctx` au moment de la préparation du premier appel.

Un exemple complet de pseudo–code ressemble à ceci :

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    MemoryContext     oldcontext;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* One-time setup code appears here: */
        user code
        if returning composite
            build TupleDesc, and perhaps
AttInMetadata
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }

    /* Each-time setup code appears here: */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* this is just one way we might test whether we are done: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* Here we want to return another item: */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
```

```
        /* Here we are done returning items and just need to clean up: */
        user code
        SRF_RETURN_DONE(funcctx);
    }
}
```

Un exemple complet d'une simple SRF retournant un type composite ressemble à ceci :

```
PG_FUNCTION_INFO_V1(testpassbyval);

Datum
testpassbyval(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;

     /* stuff done only on the first call of the function */
     if (SRF_IS_FIRSTCALL())
     {
        MemoryContext   oldcontext;

        /* create a function context for cross-call persistence */
        funcctx = SRF_FIRSTCALL_INIT();

        /* switch to memory context appropriate for multiple function calls */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* total number of tuples to be returned */
        funcctx->max_calls = PG_GETARG_UINT32(0);

        /* Build a tuple description for a __testpassbyval tuple */
        tupdesc = RelationNameGetTupleDesc("__testpassbyval");

        /*
         * generate attribute metadata needed later to produce tuples from raw
         * C strings
         */
        attinmeta = TupleDescGetAttInMetadata(tupdesc);
        funcctx->attinmeta = attinmeta;

        MemoryContextSwitchTo(oldcontext);
    }

    /* stuff done on every call of the function */
    funcctx = SRF_PERCALL_SETUP();

    call_cntr = funcctx->call_cntr;
    max_calls = funcctx->max_calls;
    attinmeta = funcctx->attinmeta;

    if (call_cntr < max_calls)    /* do when there is more left to send */
    {
        char       **values;
        HeapTuple    tuple;
        Datum        result;

        /*
        * Prepare a values array for building the returned tuple.
         * This should be an array of C strings which will
```

```
         * be processed later by the type input functions.
         */
        values = (char **) palloc(3 * sizeof(char *));
        values[0] = (char *) palloc(16 * sizeof(char));
        values[1] = (char *) palloc(16 * sizeof(char));
        values[2] = (char *) palloc(16 * sizeof(char));

        snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
        snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
        snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

        /* build a tuple */
        tuple = BuildTupleFromCStrings(attinmeta, values);

        /* make the tuple into a datum */
        result = HeapTupleGetDatum(tuple);

        /* clean up (this is not really necessary) */
        pfree(values[0]);
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* do when there is no more left */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

Le code SQL pour déclarer cette fonction est le suivant :

```
CREATE TYPE __testpassbyval AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION testpassbyval(integer, integer) RETURNS SETOF
__testpassbyval
    AS 'filename', 'testpassbyval'
    LANGUAGE C IMMUTABLE STRICT;
```

Le répertoire `contrib/tablefunc` situé dans les fichiers source de la distribution contient d'autres exemples de fonctions renvoyant des ensembles.

## 31.9.11. Arguments polymorphes et types renvoyés

Les fonctions en langage C peuvent être déclarées pour accepter et renvoyer les types << polymorphes >> `anyelement` et `anyarray`. Voyez la Section 31.2.1 pour une explication plus détaillée des fonctions polymorphes. Si les types des arguments ou du renvoi de la fonction sont définis comme polymorphes, l'auteur de la fonction ne peut pas savoir à l'avance quel type de données sera appelé ou bien quel type doit être renvoyé. Il y a deux routines offertes par `fmgr.h` qui permettent à une fonction en version−1 de découvrir les types de données effectifs de ses arguments et le type qu'elle doit renvoyer. Ces routines s'appellent `get_fn_expr_rettype(FmgrInfo *flinfo)` et `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)`. Elles renvoient l'OID du type du résultat ou de l'argument ou InvalidOID si l'information n'est pas disponible. L'accès à la structure `flinfo` se fait normalement avec `fcinfo->flinfo`. Le paramètre `argnum` est basé à partir de zéro.

Par exemple, supposons que nous voulions écrire une fonction qui accepte un argument de n'importe quel type et qui renvoie un tableau uni−dimensionnel de ce type :

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* get the provided element */
    element = PG_GETARG_DATUM(0);

    /* we have one dimension */
    ndims = 1;
    /* and one element */
    dims[0] = 1;
    /* and lower bound is 1 */
    lbs[0] = 1;

    /* get required info about the element type */
    get_typlenbyvalalign(element_type, &typlen, &typbyval,
&typalign);

    /* now build the array */
    result = construct_md_array(&element, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

La commande suivante déclare la fonction `make_array` en SQL :

```
CREATE FUNCTION make_array(anyelement)
    RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE 'C' STRICT;
```

Notez l'utilisation de STRICT ; ceci est primordial car le code ne se préoccupe pas de tester une entrée NULL.

## 31.9.12. Arguments et codes de retour polymorphiques

Les fonctions en langage C peuvent être déclarées pour accepter et recevoir des types polymorphiques `anyelement` et `anyarray`. Voir la Section 31.2.6 pour une explication plus détaillée des fonctions polymorphiques. Lorsque les arguments ou les codes de retour de la fonction sont définis comme des types polymorphiques, l'auteur de la fonction ne sait pas à l'avance avec quels types de données celle−ci sera appelée, ou quel type elle aura besoin de renvoyer. Il existe deux routines apportées par `fmgr.h` permettant à une fonction C en version 1 de découvrir les réels types de données des arguments et celui attendu en sortie.

Les routines s'appellent `get_fn_expr_rettype(FmgrInfo *flinfo)` et `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)`. Elles renvoient le type de l'OID du résultat ou de l'argument. Si l'information n'est pas disponible, elles renvoient `InvalidOid`. La structure `flinfo` est normalement accédée comme `fcinfo->flinfo`. Le paramètre `argnum` commence à zéro.

Par exemple, supposons que nous voulons écrire une fonction acceptant un seul élément de tout type et renvoyant un tableau à une dimension de ce type :

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* get the provided element */
    element = PG_GETARG_DATUM(0);

    /* we have one dimension */
    ndims = 1;
    /* and one element */
    dims[0] = 1;
    /* and lower bound is 1 */
    lbs[0] = 1;

    /* get required info about the element type */
    get_typlenbyvalalign(element_type, &typlen, &typbyval,
&typalign);

    /* now build the array */
    result = construct_md_array(&element, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

La commande suivante déclare la fonction `make_array` en SQL :

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE C STRICT;
```

Notez l'utilisation de `STRICT` ; c'est essentiel car le code ne s'occupe pas de tester une entrée NULL.

# 31.10. Agrégats définis par l'utilisateur

Dans PostgreSQL, les fonctions d'agrégat sont exprimées comme des *valeurs d'état* et des *fonctions d'état de transition*. Autrement dit, un agrégat peut être défini en termes d'état modifié chaque fois qu'une entrée est traitée. Pour définir une nouvelle fonction d'agrégat, on choisit un type de donnée pour la valeur d'état, une valeur initiale pour l'état et une fonction de transition d'état. La fonction de transition d'état est simplement une fonction ordinaire qui pourrait aussi bien être utilisée hors du contexte de l'agrégat. Une *fonction finale* peut également être spécifiée, au cas où le résultat désiré pour l'agrégat est différent des données devant être conservées comme valeur courante de l'état.

Ainsi, en plus des types de données de l'argument et du résultat vus par l'utilisateur, il existe un type de donnée pour la valeur d'état interne qui peut être différent de ces deux derniers.

Si nous définissons un agrégat qui n'utilise pas de fonction finale, nous avons un agrégat qui calcule pour chaque ligne une fonction des valeurs de colonnes. `sum` est un exemple de cette sorte d'agrégat. `sum` commence à zéro et ajoute toujours la valeur de la ligne courante à son total en cours. Par exemple, si nous voulons faire un agrégat `sum` pour opérer sur un type de donnée pour des nombres complexes, nous avons seulement besoin de la fonction d'addition pour ce type de donnée. La définition de l'agrégat sera :

```
CREATE AGGREGATE somme_complexe (
    sfunc = ajout_complexe,
    basetype = complexe,
    stype = complexe,
    initcond = '(0,0)'
);

SELECT somme_complexe(a) FROM test_complexe;

 somme_complexe
----------------
     (34,53.9)
```

(Dans la pratique, nous aurions seulement nommé l'agrégat `sum` et laissé PostgreSQL déterminer quel genre de somme appliquer à une colonne de type `complexe`.)

La définition précédente de `sum` renverra zéro (la condition d'état initial) s'il n'y a pas de valeurs d'entrée non NULL. Peut−être désirons−nous que dans ce cas elle retourne NULL — le standard SQL prévoit que la fonction `sum` se comporte ainsi. Nous pouvons faire ceci simplement en omettant l'instruction `initcond`, de sorte que la condition d'état initial soit NULL. Ordinairement, ceci signifierait que `sfunc` aurait à vérifier l'entrée d'une condition d'état NULL, mais pour la fonction `sum` et quelques autres agrégats simples comme `max` et `min`, il suffit d'insérer la première valeur d'entrée non NULL dans la variable d'état et ensuite de commencer à appliquer la fonction de transition d'état à la seconde valeur non NULL. PostgreSQL fera cela automatiquement si la condition initiale est NULL et si la fonction de transition est marquée << strict >> (c'est−à−dire qu'elle ne doit pas être appelée pour des entrées NULL).

Un autre comportement par défaut d'une fonction de transition << strict >> est que la valeur d'état précédente est gardée inchangée chaque fois qu'une entrée NULL est rencontrée. Ainsi, les valeurs NULL sont ignorées. Si vous avez besoin d'un autre comportement pour les entrées NULL, vous devez juste ne pas définir votre fonction de transition comme << strict >> et la coder pour vérifier les entrées NULL et faire le nécessaire.

`avg` (average = moyenne) est un exemple plus compliqué d'agrégat. Il demande deux état courants : la somme

des entrées et le compte du nombre d'entrées. Le résultat final est obtenu en divisant ces quantités. La moyenne est typiquement implémentée en utilisant comme valeur d'état un tableau de deux éléments. Par exemple, l'implémentation intégrée de `avg(float8)` ressemble à :

```
CREATE AGGREGATE avg (
    sfunc = float8_accum,
    basetype = float8,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0}'
);
```

Les fonctions d'agrégat peuvent utiliser des fonctions d'état de transition ou des fonctions finales polymorphes, de sorte que les mêmes fonctions peuvent être utilisées pour implémenter de multiples agrégats. Voir la Section 31.2.1 pour une explication des fonctions polymorphes. Pour aller encore plus loin, la fonction d'agrégat elle−même peut être spécifiée avec un type de base et un type d'état polymorphes, permettant ainsi à une unique définition de fonction de servir pour de multiples types de données d'entrée. Voici un exemple d'agrégat polymorphe :

```
CREATE AGGREGATE array_accum (
    sfunc = array_append,
    basetype = anyelement,
    stype = anyarray,
    initcond = '{}'
);
```

Ici, le type d'état effectif pour n'importe quel appel d'agrégat est le type tableau, ayant comme éléments le type effectif d'entrée.

Voici le résultat quand on utilise deux types de donnée effectifs différents comme arguments :

```
SELECT attrelid::regclass, array_accum(attname)
FROM pg_attribute WHERE attnum > 0
AND attrelid = 'pg_user'::regclass GROUP BY attrelid;
 attrelid |                                array_accum
----------+-------------------------------------------------------------------------
 pg_user  | {usename,usesysid,usecreatedb,usesuper,usecatupd,passwd,valuntil,useconfig}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid)
FROM pg_attribute WHERE attnum > 0
AND attrelid = 'pg_user'::regclass GROUP BY attrelid;
 attrelid |         array_accum
----------+-----------------------------
 pg_user  | {19,23,16,16,16,25,702,1009}
(1 row)
```

Pour plus de détails, voyez la commande *CREATE AGGREGATE*.

# 31.11. Types définis par l'utilisateur

Décrit dans la Section 31.2, PostgreSQL peut être étendu pour supporter de nouveaux types de données. Cette section décrit comment définir de nouveaux types de base, qui sont des types de données définis en−dessous du niveau du langage SQL. Créer un nouveau type de base requiert l'implémentation de fonctions pour opérer

sur le type dans un langage de base du niveau du C.

Les exemples de cette section sont disponibles dans `complex.sql` et `complex.c` du répertoire `src/tutorial` de la distribution des sources. Voir le fichier `README` de ce répertoire pour les instructions d'exécution des exemples.

Un type défini par l'utilisateur doit toujours avoir des fonctions d'entrée et de sortie. Ces fonctions déterminent comment le type apparaît dans les chaînes de caractères (pour l'entrée par l'utilisateur et le renvoi à l'utilisateur) et comment ce type est organisé en mémoire. La fonction d'entrée prend comme argument une chaîne de caractères terminée par NULL et renvoie la représentation interne (en mémoire) du type. La fonction de sortie prend comme argument la représentation interne du type et renvoie une chaîne de caractères terminée par NULL. Si vous voulez faire plus avec le type que simplement l'enregistrer, vous devez apporter des fonctions supplémentaires pour implémenter toutes opérations que vous souhaitez avoir pour ce type.

Supposons que nous voulions définir un type `complex` représentant les nombres complexes. Une façon naturelle de représenter un nombre complexe en mémoire serait la structure C suivante :

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

Nous aurons besoin d'utiliser ce type par référence car il est trop important pour tenir sur une seule valeur `Datum`.

Comme représentation externe du type sous forme de chaîne, nous choisissons une chaîne de la forme `(x,y)`.

Habituellement, les fonctions d'entrée et de sortie ne sont pas compliquées à écrire, surtout la fonction de sortie. Mais en définissant la représentation externe du type par une chaîne, souvenez−vous que vous devez éventuellement écrire un analyseur complet et robuste pour cette représentation en tant que fonction d'entrée. Par exemple :

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

La fonction de sortie peut simplement s'écrire :

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

Vous devriez faire attention en écrivant des fonctions d'entrée et de sortie inverses l'une de l'autre. Sinon, vous aurez de graves problèmes quand vous aurez besoin de sauvegarder votre base de données dans un fichier et ensuite de le relire. Ceci est un problème particulièrement fréquent quand des nombres à virgule flottante sont concernés.

De manière optionnelle, un type défini par l'utilisateur peut apporter des routines d'entrée et de sortie binaires. Les entrées/sorties binaires sont normalement plus rapides mais moins portables que les entrées/sorties textuelles. Avec les entrées/sorties textuelles, c'est à vous de définir exactement la représentation binaire externe. La plupart des types de données intégrés essaient d'apporter une représentation binaire indépendante de la machine. Pour `complex`, nous allons revenir aux convertisseurs d'entrées/sorties binaires pour le type `float8` :

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

Pour définir le type `complex`, nous avons besoin de créer les fonctions d'entrées/sorties définies par l'utilisateur avant de créer le type :

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
```

```
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;
```

Notez que la déclaration des fonctions d'entrée et de sortie doit pouvoir référencer un type non encore défini. Ceci est permis mais provoque des messages d'avertissement qui peuvent être ignorés. La fonction en entrée doit d'abord apparaître.

Finalement, nous pouvons déclarer le type de données :

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

Quand vous définissez un nouveau type de base, PostgreSQL fournit automatiquement le support pour des tableaux de ce type. Pour des raisons historiques, le type tableau a le même nom que le type de base avec un caractère souligné (_) en préfixe.

Une fois que le type de données existe, nous pouvons déclarer les fonctions supplémentaires pour apporter des opérations utiles pour ce type de données. Les opérateurs peuvent alors être définis au–dessus de ces fonctions et, si nécessaire, les classes d'opérateurs peuvent aussi être créées pour apporter le support de l'indexage du type de données. Ces couches supplémentaires sont discutées dans les sections suivantes.

Si les valeurs de votre type de donnée peuvent excéder une taille de quelques centaines d'octets (sous la forme interne), vous devriez marquer le type de données comme TOAST–able. Pour cela, la représentation interne doit suivre le cadre standard des données à longueur variable : les quatre premiers octets doivent être un `int32` contenant la longueur totale en octets de la donnée (lui–même inclus). Les fonctions C opérant sur le type de données doivent faire bien attention à déballer toutes les valeurs toast des données (ce détail peut normalement être cachée dans les macros `GETARG`). Puis, quand on exécute la commande `CREATE TYPE`, spécifiez la longueur interne comme `variable` et choisissez l'option de stockage en mémoire appropriée.

Pour plus de détails, voir la description de la commande ***CREATE TYPE***.

# 31.12. Opérateurs définis par l'utilisateur

Chaque opérateur est un << sucre syntaxique >> pour l'appel d'une fonction sous−jacente qui effectue le véritable travail ; aussi devez−vous en premier lieu créer cette fonction avant de pouvoir créer l'opérateur. Toutefois, un opérateur n'est pas *simplement* un << sucre syntaxique >> car il apporte des informations supplémentaires qui aident le planificateur de requête à optimiser les requêtes utilisées par l'opérateur. La prochaine section est consacrée à l'explication de ces informations additionnelle.

PostgreSQL accepte les opérateurs unaire gauche, unaire droit et binaire. Les opérateurs peuvent être surchargés ; c'est−à−dire que le même nom d'opérateur peut être utilisé pour différents opérateurs, à condition qu'ils aient des nombres et des types différents d'opérandes. Quand une requête est exécutée, le système détermine l'opérateur à appeler en fonction du nombre et des types d'opérandes fournis.

Voici un exemple de création d'opérateur pour l'addition de deux nombres complexes. Nous supposons avoir déjà créé la définition du type `complex` (voir la Section 31.11). Premièrement, nous avons besoin d'une fonction qui fasse le travail, ensuite nous pouvons définir l'opérateur :

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C;

CREATE OPERATOR + ( leftarg = complex, rightarg = complex, procedure =
complex_add, commutator = + );
```

Maintenant nous pouvons exécuter la requête comme ceci :

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

Nous avons montré comment créer un opérateur binaire. Pour créer des opérateurs unaires, il suffit d'omettre un des `leftarg` (pour un opérateur unaire gauche) ou `rightarg` (pour un opérateur unaire droit). La clause `procedure` et les clauses argument sont les seuls éléments requis dans la commande `CREATE OPERATOR`. La clause `commutator` montrée dans l'exemple est une indication optionnelle pour l'optimiseur de requête. Des détails supplémentaires sur la clause `commutator` et d'autres compléments d'optimisation sont donnés dans la prochaine section.

# 31.13. Informations sur l'optimisation d'un opérateur

Une définition d'opérateur PostgreSQL peut inclure plusieurs clauses optionnelles qui donnent au système des informations utiles sur le comportement de l'opérateur. Ces clauses devraient être fournies chaque fois que c'est utile car elles peuvent considérablement accélérer l'exécution des requêtes utilisant cet opérateur. Mais si vous le faites, vous devez être sûr de leur justesse ! L'usage incorrect d'une clause d'optimisation peut entraîner un arrêt brutal du processus serveur, des sorties subtilement fausses ou d'autres effets pervers. Vous pouvez toujours abandonner une clause d'optimisation si vous n'êtes pas sûr d'elle ; la seule conséquence est un possible ralentissement des requêtes.

Des clauses d'optimisation additionnelles pourront être ajoutées dans les futures versions de PostgreSQL. Celles décrites ici sont toutes celles que cette version comprend.

### 31.13.1. COMMUTATOR

La clause COMMUTATOR, si elle est fournie, désigne un opérateur qui est le commutateur de l'opérateur en cours de définition. Nous disons qu'un opérateur A est le commutateur de l'opérateur B si (x A y) est égal à (y B x) pour toutes valeurs possible de x, y. Notez que B est aussi le commutateur de A. Par exemple, les opérateurs < et > pour un type particulier de données sont habituellement des commutateurs l'un pour l'autre, et l'opérateur + est habituellement commutatif avec lui−même. Mais l'opérateur − n'est habituellement commutatif avec rien.

Le type de l'opérande gauche d'un opérateur commuté est le même que l'opérande droit de son commutateur, et vice versa. Aussi PostgreSQL n'a besoin que du nom de l'opérateur commutateur pour consulter le commutateur, et c'est tout ce qui doit être fourni à la clause COMMUTATOR .

Vous avez juste à définir un opérateur auto−commutateur. Mais les choses sont un peu plus compliquées quand vous définissez une paire de commutateurs : comment peut−on définir la référence du premier au second, alors que ce dernier n'est pas encore défini ? Il y a deux solutions à ce problème :

- Une façon d'opérer est d'omettre la clause COMMUTATOR dans le premier opérateur que vous définissez et ensuite d'en insérer une dans la définition du second opérateur. Puisque PostgreSQL sait que les opérateurs commutatifs vont par paire, quand il voit la seconde définition, il retourne instantanément remplir la clause COMMUTATOR manquante dans la première définition.
- L'autre façon, plus directe, est de simplement inclure les clauses COMMUTATOR dans les deux définitions. Quand PostgreSQL traite la première définition et réalise que la clause COMMUTATOR se réfère à un opérateur inexistant, le système va créer une entrée provisoire pour cet opérateur dans le catalogue système. Cette entrée sera pourvue seulement de données valides pour le nom de l'opérateur, les types d'opérande droit et gauche et le type du résultat, puisque c'est tout ce que PostgreSQL peut déduire à ce point. La première entrée du catalogue pour l'opérateur sera liée à cette entrée provisoire. Plus tard, quand vous définirez le second opérateur, le système mettra à jour l'entrée provisoire avec les informations additionnelles fournies par la seconde définition. Si vous essayez d'utiliser l'opérateur provisoire avant qu'il ne soit complété, vous aurez juste un message d'erreur.

### 31.13.2. NEGATOR

La clause NEGATOR dénomme un opérateur qui est l'opérateur de négation de l'opérateur en cours de définition. Nous disons qu'un opérateur A est l'opérateur de négation de l'opérateur B si tous les deux renvoient des résultats booléens et si (x A y) est égal à NOT (x B y) pour toutes les entrées possible x, y. Notez que B est aussi l'opérateur de négation de A. Par exemple, < et >= forment une paire d'opérateurs de négation pour la plupart des types de données. Un opérateur ne peut jamais être validé comme son propre opérateur de négation .

Au contraire des commutateurs, une paire d'opérateurs unaires peut être validée comme une paire d'opérateurs de négation réciproques ; ce qui signifie que (A x) est égal à NOT (B x) pour tout x ou l'équivalent pour les opérateurs unaires à droite.

L'opérateur de négation d'un opérateur doit avoir les mêmes types d'opérandes gauche et/ou droit que l'opérateur à définir, aussi comme avec COMMUTATOR, seul le nom de l'opérateur doit être donné dans la clause NEGATOR.

Définir un opérateur de négation est très utile pour l'optimiseur de requêtes car il permet de simplifier des expressions telles que `NOT (x = y)` en `x <> y`. Ceci arrive plus souvent que vous ne pouvez le penser parce que les opérations `NOT` peuvent être insérées à la suite d'autres réarrangements.

Des paires d'opérateurs de négation peuvent être définies en utilisant la même méthode que pour les commutateurs.

## 31.13.3. RESTRICT

La clause `RESTRICT`, si elle est invoquée, nomme une fonction d'estimation de sélectivité de restriction pour cet opérateur. (Notez que c'est un nom de fonction, et non pas un nom d'opérateur.) Les clauses `RESTRICT` n'ont de sens que pour les opérateurs binaires qui renvoient un type `boolean`. Un estimateur de sélectivité de restriction repose sur l'idée de prévoir quelle fraction des lignes dans une table satisfera une condition de clause `WHERE` de la forme

```
colonne OP constante
```

pour l'opérateur courant et une valeur constante particulière. Ceci aide l'optimiseur en lui donnant une idée du nombre de lignes qui sera éliminé par les clauses `WHERE` qui ont cette forme. (Vous pouvez vous demander, qu'arrivera−t−il si la constante est à gauche ? hé bien, c'est une des choses à laquelle sert le `COMMUTATOR` ...)

L'écriture de nouvelles fonctions d'estimation de restriction de sélectivité est éloignée des objectifs de ce chapitre, mais heureusement, vous pouvez habituellement simplement utiliser un des estimateurs standard du système pour beaucoup de vos propres opérateurs. Voici les estimateurs standard de restriction :

`eqsel` pour =
`neqsel` pour <>
`scalarltsel` pour < ou <=
`scalargtsel`pour > ou >=

Ces catégories peuvent sembler un peu curieuses, mais cela prend un sens si vous y réfléchissez. = acceptera typiquement seulement une petite fraction des lignes d'une table ; <> rejettera typiquement seulement une petite fraction des lignes de la table. < acceptera une fraction des lignes en fonction de la situation de la constante donnée dans la gamme de valeurs de la colonne pour cette table (ce qui est justement l'information collectée par la commande `ANALYZE` et rendue disponible pour l'estimateur de sélectivité). <= acceptera une fraction légèrement plus grande que < pour la même constante de comparaison mais elles sont assez proches pour ne pas valoir la peine d'être distinguées puisque nous ne risquons pas de toute façon de faire mieux qu'une grossière estimation. La même remarque s'applique à > et >=.

Vous pouvez fréquemment vous en sortir à bon compte en utilisant soit `eqsel` ou `neqsel` pour des opérateurs qui ont une très grande ou une très faible sélectivité, même s'ils ne sont pas réellement égalité ou inégalité. Par exemple, les opérateurs géométrique d'égalité approchée utilisent `eqsel` en supposant habituellement qu'ils ne correspondent qu'à une petite fraction des entrées dans une table.

Vous pouvez utiliser `scalarltsel` et `scalargtsel` pour des comparaisons de types de données qui possèdent un moyen de conversion en scalaires numériques pour les comparaisons de rang. Si possible, ajoutez le type de données à ceux acceptés par la fonction `convert_to_scalar()` dans `src/backend/utils/adt/selfuncs.c`. (Finalement, cette fonction devrait être remplacée par des fonctions pour chaque type de données identifié grâce à une colonne du catalogue système `pg_type` ; mais cela n'a pas encore été fait.) Si vous ne faites pas ceci, les choses fonctionneront mais les estimations de

l'optimiseur ne seront pas aussi bonnes qu'elles pourraient l'être.

D'autres fonctions d'estimation de sélectivité conçues pour les opérateurs géométriques sont placées dans `src/backend/utils/adt/geo_selfuncs.c` : `areasel`, `positionsel` et `contsel`. Lors de cette rédaction, ce sont seulement des fragments mais vous pouvez vouloir les utiliser (ou mieux les améliorer).

### 31.13.4. JOIN

La clause `JOIN`, si elle est invoquée, nomme une fonction d'estimation de sélectivité de jointure pour l'opérateur. (Notez que c'est un nom de fonction, et non pas un nom d'opérateur.) Les clauses `JOIN` n'ont de sens que pour les opérateurs binaires qui renvoient un type `boolean`. Un estimateur de sélectivité de jointure repose sur l'idée de prévoir quelle fraction des lignes dans une paire de tables satisfera une condition de clause `WHERE` de la forme

```
table1.column1 OP table2.column2
```

pour l'opérateur courant. Comme pour la clause `RESTRICT`, ceci aide considérablement l'optimiseur en lui indiquant parmi plusieurs séquences de jointure possibles laquelle prendra vraisemblablement le moins de travail.

Comme précédemment, ce chapitre n'essaiera pas d'expliquer comment écrire une fonction d'estimation de sélectivité de jointure mais suggérera simplement d'utiliser un des estimateurs standard s'il est applicable :

`eqjoinsel` pour =
`neqjoinsel` pour <>
`scalarltjoinsel` pour < ou <=
`scalargtjoinsel` pour > ou >=
`areajoinsel` pour des comparaisons basées sur une aire 2D
`positionjoinsel` pour des comparaisons basées sur une position 2D
`contjoinsel` pour des comparaisons basées sur un appartenance 2D

### 31.13.5. HASHES

La clause `HASHES` indique au système qu'il est permis d'utiliser la méthode de jointure–découpage pour une jointure basée sur cet opérateur. `HASHES` n'a de sens que pour un opérateur binaire qui renvoie un `boolean` et en pratique l'opérateur égalité serait mieux approprié pour certains types de données

La jointure–découpage repose sur l'hypothèse que l'opérateur de jointure peut seulement renvoyer la valeur vrai pour des paires de valeurs droite et gauche qui correspondent au même code de découpage. Si deux valeurs sont placées dans deux différents paquets ("buckets"), la jointure ne pourra jamais les comparer avec la supposition implicite que le résultat de l'opérateur de jointure doit être faux. Ainsi, il n'y a aucun sens à spécifier `HASHES` pour des opérateurs qui ne représentent pas l'égalité.

Pour être marqué `HASHES`, l'opérateur de jointure doit apparaître dans une classe d'opérateurs d'index de découpage. Ceci n'est pas rendu obligatoire quand vous créez l'opérateur, puisque évidemment la classe référençant l'opérateur peut ne pas encore exister. Mais les tentatives d'utilisation de l'opérateur dans les jointure–découpage échoueront à l'exécution si une telle classe d'opérateur n'existe pas. Le système a besoin de la classe d'opérateur pour définir la fonction de découpage spécifique au type de données d'entrée de

l'opérateur. Bien sûr, vous devez également fournir une fonction de découpage appropriée avant de pouvoir créer la classe d'opérateur.

On doit apporter une grande attention à la préparation des fonctions de découpage, parce qu'il y a des processus dépendants de la machine qui peuvent ne pas faire les choses correctement. Par exemple, si votre type de données est une structure dans laquelle peuvent se trouver des bits de remplissage sans intérêt, vous ne pouvez pas simplement passer la structure complète à la fonction `hash_any`. (À moins d'écrire vos autres opérateurs et fonctions de façon à s'assurer que les bits inutilisés sont toujours zéro, ce qui est la stratégie recommandée.) Un autre exemple est fourni sur les machines qui respectent le standard de virgule−flottante IEEE, le zéro négatif et le zéro positif sont des valeurs différentes (les motifs de bit sont différents) mais ils sont définis pour être égaux. Si une valeur flottante peut contenir un zéro négatif, alors une étape supplémentaire est nécessaire pour s'assurer qu'elle génère la même valeur de découpage qu'un zéro positif.

> **Note :** La fonction sous−jacente à un opérateur de jointure−découpage doit être marquée immuable ou stable. Si elle est volatile, le système n'essaiera jamais d'utiliser l'opérateur pour une jointure hachage.

> **Note :** Si un opérateur de jointure−hachage a une fonction sous−jacente marquée stricte, la fonction doit également être complète : cela signifie qu'elle doit renvoyer TRUE ou FALSE, jamais NULL, pour n'importe quelle double entrée non NULL. Si cette règle n'est pas respectée, l'optimisation de découpage des opérations `IN` peut générer des résultats faux. (Spécifiquement, `IN` devrait renvoyer FALSE quand la réponse correcte devrait être NULL ; ou bien il devrait renvoyer une erreur indiquant qu'il ne s'attendait pas à un résultat NULL.)

## 31.13.6. MERGES (SORT1, SORT2, LTCMP, GTCMP)

La clause `MERGES`, si elle est présente, indique au système qu'il est permis d'utiliser la méthode de jointure−union pour une jointure basée sur cet opérateur. `MERGES` n'a de sens que pour un opérateur binaire qui renvoie un `boolean`, et en pratique, cet opérateur doit représenter l'égalité pour des types de données ou des paires de types de données.

La jointure−union est fondée sur le principe d'ordonner les tables gauche et droite et ensuite de les comparer en parallèle. Ainsi, les deux types de donnée doivent être capable d'être pleinement ordonnées, et l'opérateur de jointure doit pouvoir réussir seulement pour des paires de valeurs tombant à la << même place >> dans l'ordre de tri. En pratique, cela signifie que l'opérateur de jointure doit se comporter comme l'opérateur égalité. Mais contrairement à la jointure−hachage, où il vaut mieux que les types de donnée droite et gauche soit les mêmes (ou au moins soient bitwise équivalent), il est possible de faire une jointure−union sur deux types de données distincts, tant qu'ils sont logiquement compatibles. Par exemple, l'opérateur d'égalité `smallint`−contre−`integer` est susceptible d'opérer une jointure−union. Nous avons seulement besoin d'opérateurs de tri qui organisent les deux types de données en séquences logiquement comparables.

L'exécution d'un jointure−union exige que le système soit capable d'identifier quatre opérateurs rattachés à l'opérateur de jointure−union: la comparaison less−than pour le type de donnée de l'opérande gauche, la comparaison less−than pour le type de donnée de l'opérande droit, la comparaison less−than entre les deux types de donnée et la comparaison greater−than entre les deux types de donnée. (Il y a en fait quatre opérateurs distincts si l'opérateur de jointure−union a deux types de données d'opérande différents; mais quand les types d'opérande sont les mêmes, les trois opérateurs less−than sont tous le même opérateur.) Il est possible de spécifier ces opérateurs individuellement par leur nom, comme les options respectives `SORT1`, `SORT2`, `LTCMP`, et `GTCMP`. Le système remplira respectivement par défaut les noms <, <, <, > si n'importe lequel d'entre eux est omis quand `MERGES` est spécifié De même, `MERGES` sera supposé être indiqué si

n'importe laquelle de ces quatre options apparaît, il est donc possible de seulement spécifier quelques unes de ces options et de laisser le système compléter le reste.

Les types de données des opérandes des quatre opérateurs de comparaison peuvent être déduits des types d'opérandes de l'opérateur de jointure−union, aussi, exactement comme avec `COMMUTATOR`, seuls les noms d'opérateurs ont besoin d'être donnés dans ces clauses. A moins que vous ne fassiez des choix particuliers de noms d'opérateurs, il suffit d'écrire `MERGES` et laisser le système remplir les détails. (Comme avec `COMMUTATOR` et `NEGATOR`, le système est capable de faire des entrées d'opérateur fictives si il vous arrive de définir l'opérateur égalité avant les autres.)

Il existe des restrictions additionnelles sur les opérateurs que vous marquez comme jointure−union. Ces restrictions ne sont pas actuellement contrôlées par la commande `CREATE OPERATOR`, mais des erreurs peuvent intervenir lors de l'utilisation de l'opérateur si un des points suivants n'est pas vérifié:

- Un opérateur d'égalité capable de jointure−union doit avoir un commutateur capable de jointure−union (qui peut être lui−même si les deux types de donnée d'opérande sont les mêmes, ou un opérateur d'égalité apparenté si ils sont différents).
- Si il existe un opérateur capable de jointure−union reliant deux types de données A et B, et un autre opérateur capable de jointure−union reliant B à un troisième type de donnée C, alors A et C doivent aussi avoir un opérateur capable de jointure−union ; en d'autres mots, avoir un opérateur de jointure−union doit être une propriété transitive.
- Des résultats bizarres apparaîtront lors de l'exécution si les quatre opérateurs de comparaison que vous nommez ne trient pas les valeurs de façon compatible.

**Note :** La fonction sous−jacente à un opérateur de jointure−union doit être marquée immuable ou stable. Si elle est volatile, le système n'essaiera jamais d'utiliser l'opérateur pour une jointure union.

**Note :** Dans les versions de PostgreSQL antérieure à 7.3, `MERGES` n'était pas disponible : pour faire un opérateur de jointure union, on devait explicitement écrire `SORT1` et `SORT2`. De plus, les options `LTCMP` et `GTCMP` n'existaient pas ; les noms de ces opérateurs ont été rattachés respectivement à < et >.

## 31.14. Interfacer des extensions d'index

Les procédures décrites jusqu'à maintenant permettent de définir de nouveaux types, de nouvelles fonctions et de nouveaux opérateurs. Néanmoins, nous ne pouvons pas encore définir un index sur une colonne d'un nouveau type de données. Pour cela, nous devons définir une *classe d'opérateur* pour le nouveau type de données. Plus loin dans cette section, nous illustrerons ce concept avec un exemple : une nouvelle classe d'opérateur pour la méthode d'indexation B−tree qui enregistre et trie des nombres complexes dans l'ordre ascendant des valeurs absolues.

**Note :** Avant PostgreSQL version 7.3, il était nécessaire de faire manuellement quelques ajouts aux catalogues système `pg_amop`, `pg_amproc` et `pg_opclass` afin de créer une classe d'opérateur définie par l'utilisateur. Cette approche est maintenant obsolète, au bénéfice de la commande `CREATE OPERATOR CLASS`, qui est beaucoup plus simple et permet d'éviter les erreurs lors de la création nécessaire des entrées de ces catalogues.

## 31.14.1. Méthodes d'indexation et classes d'opérateurs

La table pg_am contient une ligne pour chaque méthode d'indexation (connue en interne comme méthode d'accès). Le support pour l'accès normal aux tables est implémenté dans PostgreSQL mais toutes les méthodes d'index sont décrites dans pg_am. Il est possible d'ajouter une nouvelle méthode d'indexation en définissant les routines d'interface nécessaires et en créant ensuite une ligne dans la table pg_am — mais ceci est bien au−delà du sujet de ce chapitre.

Les routines d'une méthode d'indexation ne connaissent pas directement les types de données sur lesquels opère la méthode. Au lieu de cela, une *classe d'opérateur* identifie l'ensemble des opérations que la méthode d'indexation doit utiliser pour fonctionner avec un type particulier de données. Les classes d'opérateurs sont ainsi dénommées parce qu'une de leur tâche est de spécifier l'ensemble des opérateurs de la clause WHERE utilisables avec un index (c'est−à−dire, qui peuvent être requalifiés en parcours d'index). Une classe d'opérateur peut également spécifier des *procédures d'appui*, nécessaires pour les opérations internes de la méthode d'indexation mais sans correspondance directe avec un quelconque opérateur de clause WHERE pouvant être utilisé avec l'index.

Il est possible de définir plusieurs classes d'opérateurs pour le même type de données et la même méthode d'indexation. Ainsi, de multiples ensembles de sémantiques d'indexation peuvent être définis pour un seul type de données. Par exemple, un index B−tree exige qu'un tri ordonné soit défini pour chaque type de données auquel il peut s'appliquer. Il peut être utile pour un type de donnée nombre complexe de disposer d'une classe d'opérateur B−tree qui trie les données selon la valeur absolue complexe, une autre selon la partie réelle, etc. Typiquement, une des classes d'opérateur sera considérée comme plus utile et sera marquée comme l'opérateur par défaut pour ce type de données et cette méthode d'indexation.

Le même nom de classe d'opérateur peut être utilisé pour plusieurs méthodes d'indexation différentes (par exemple, les méthodes d'index B−tree et hash ont toutes les deux des classes d'opérateur nommées int4_ops) mais chacune de ces classes est une entité indépendante et doit être définie séparément.

---

## 31.14.2. Stratégies des méthode d'indexation

Les opérateurs associés avec une classe d'opérateur sont identifiés par des << numéros de stratégie >>, servant à identifier la sémantique de chaque opérateur dans le contexte de sa classe d'opérateur. Par exemple, les B−trees imposent un classement strict selon les clés, du plus petit au plus grand. Ainsi, des opérateurs comme << plus petit que >> et << plus grand que >> sont intéressants pour un B−tree. Comme PostgreSQL permet à l'utilisateur de définir des opérateurs, PostgreSQL ne peut pas rechercher le nom d'un opérateur (par exemple, < ou >=) et rapporter de quelle comparaison il s'agit. Au lieu de cela, la méthode d'indexation définit un ensemble de << stratégies >>, qui peuvent être comprises comme des opérateurs généralisés. Chaque classe d'opérateur spécifie l'opérateur effectif correspondant à chaque stratégie pour un type de donnée particulier et pour une interprétation de la sémantique d'index.

La méthode d'indexation B−tree définit cinq stratégies, exposées dans le Tableau 31−2.

**Tableau 31−2. Stratégies B−tree**

| Opération | Numéro de stratégie |
|---|---|
| plus petit que | 1 |
| plus petit ou égal | 2 |

| | |
|---|---|
| égal | 3 |
| plus grand ou égal | 4 |
| plus grand que | 5 |

Les index de découpage expriment seulement une égalité bit à bit et utilisent ainsi une seule stratégie exposée dans le Tableau 31−3.

**Tableau 31−3. Stratégies de découpage**

| Opération | Numéro de stratégie |
|---|---|
| égal à | 1 |

Les index R−tree expriment des relations d'appartenance à un rectangle. Ils utilisent huit stratégies, exposées dans le Tableau 31−4.

**Tableau 31−4. Stratégies R−tree**

| Opération | Numéro de stratégie |
|---|---|
| à gauche de | 1 |
| à gauche ou imbriqué | 2 |
| imbriqué | 3 |
| à droite ou imbriqué | 4 |
| à droite de | 5 |
| identique | 6 |
| contient | 7 |
| contenu par | 8 |

Les index GiST sont encore plus flexibles : ils n'ont pas du tout d'ensemble fixe de stratégies. En lieu et place, la routine d'appui << consistency >> de chaque classe d'opérateur GiST particulière interprète les numéros de stratégie comme elle l'entend.

Notez que tous les opérateurs de stratégie renvoient des valeurs de type booléen. Dans la pratique, tous les opérateurs définis comme stratégies de méthode d'indexation doivent renvoyer un type `boolean` puisqu'ils doivent apparaître au plus haut niveau d'une clause `WHERE` pour être utilisés avec un index.

A ce propos, la colonne `amorderstrategy` dans `pg_am` indique si la méthode d'indexation supporte les balayages ordonnés. Zéro indique qu'elle ne les supporte pas ; si elle les supporte, `amorderstrategy` est le numéro de stratégie qui correspond à l'opérateur de classement. Par exemple, B−tree a `amorderstrategy` = 1, qui est son numéro de stratégie pour << plus petit que >>.

## 31.14.3. Routines d'appui des méthodes d'indexation

Généralement, les stratégies n'apportent pas assez d'informations au système pour indiquer comment utiliser un index. Dans la pratique, les méthodes d'indexation demandent des routines d'appui additionnelles pour fonctionner. Par exemple, les méthodes d'index B−tree doivent être capables de comparer deux clés et de déterminer laquelle est supérieure, égale ou inférieure à l'autre. De la même façon, la méthode d'indexation R−tree doit être capable de calculer les intersections, unions et dimensions des rectangles. Ces opérations ne correspondent pas à des opérateurs utilisés dans les commandes SQL ; ce sont des routines administratives

utilisées en interne par des méthodes d'index.

Comme pour les stratégies, la classe d'opérateur énumère les fonctions spécifiques et le rôle qu'elles doivent jouer pour un type de donnée donné et une interprétation sémantique donnée. La méthode d'indexation définit l'ensemble des fonctions dont elle a besoin et la classe d'opérateur identifie les fonctions exactes à utiliser en les assignant aux << numéros de fonction d'appui >>.

Les B−trees demandent une seule fonction d'appui, exposée dans le Tableau 31−5.

**Tableau 31−5. Fonctions d'appui de B−tree**

| Fonction | Numéro d'appui |
|---|---|
| Comparer deux clés et renvoyer un entier inférieure à zéro, zéro, ou supérieure à zéro, indiquant si la première clé est inférieure, égale ou supérieure à la deuxième. | 1 |

Pareillement, les index de découpage requièrent une fonction d'appui, exposée dans le Tableau 31−6.

**Tableau 31−6. Fonctions d'appui pour découpage**

| Fonction | Numéro de l'appui |
|---|---|
| Calculer la valeur de découpage pour une clé | 1 |

Les index R−tree requièrent trois fonctions d'appui, exposées dans le Tableau 31−7.

**Tableau 31−7. Fonctions d'appui pour R−tree**

| Fonction | Numéro d'appui |
|---|---|
| union | 1 |
| intersection | 2 |
| taille | 3 |

Les index GiST requièrent sept fonctions d'appui, exposées dans le Tableau 31−8.

**Tableau 31−8. Fonctions de support GiST**

| Fonction | Numéro de support |
|---|---|
| cohérence | 1 |
| union | 2 |
| compression | 3 |
| décompression | 4 |
| coût | 5 |
| séparation | 6 |
| égalité | 7 |

Contrairement aux opérateurs de stratégie, les fonctions d'appui renvoient le type de donnée, quelqu'il soit, que la méthode d'indexation particulière attend, par exemple, dans le cas de la fonction de comparaison des B−trees, un entier signé.

## 31.14.4. Exemple

Maintenant que nous avons vu les idées, voici l'exemple promis de création d'une nouvelle classe d'opérateur. Cette classe d'opérateur encapsule les opérateurs qui trient les nombres complexes selon l'ordre de la valeur absolue, aussi avons−nous choisi le nom de `complex_abs_ops`. En premier lieu, nous avons besoin d'un ensemble d'opérateurs. La procédure pour définir des opérateurs a été discutée dans la Section 31.12. Pour une classe d'opérateur sur les B−trees, nous avons besoin des opérateurs :

- absolute−value less−than (stratégie 1) ;
- absolute−value less−than−or−equal (stratégie 2) ;
- absolute−value equal (stratégie 3) ;
- absolute−value greater−than−or−equal (stratégie 4) ;
- absolute−value greater−than (stratégie 5) ;

Le code C de l'opérateur d'égalité ressemble à ceci :

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

bool
complex_abs_eq(Complex *a, Complex *b)
{
    double amag = Mag(a), bmag = Mag(b);
    return (amag == bmag);
}
```

Les quatre autres opérateurs sont très similaires. Vous pouvez trouver leur code dans `src/tutorial/complex.c` et `src/tutorial/complex.sql` dans la distribution des sources.

Maintenant, déclarons les fonctions et les opérateurs basés sur ces fonctions :

```
CREATE FUNCTION complex_abs_eq(complex, complex) RETURNS boolean
    AS 'nom_fichier', 'complex_abs_eq'
    LANGUAGE C;

CREATE OPERATOR = (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_abs_eq,
    restrict = eqsel,
    join = eqjoinsel
);
```

Il est important de spécifier les fonctions de sélectivité de restriction et de jointure, sinon l'optimiseur sera incapable de faire un usage effectif de l'index. Notez que les cas 'less−than', 'equal' et 'greater−than' doivent utiliser des fonctions de sélectivité différentes.

Voici d'autres choses importantes à noter :

- Il ne peut y avoir qu'un seul opérateur nommé, disons, = et acceptant un type `complex` pour ses deux opérandes. Dans le cas présent, nous n'avons aucun autre opérateur = pour `complex`, mais si nous

construisons un type de donnée fonctionnel, nous aurions certainement désiré que = soit l'opération ordinaire d'égalité pour les nombres complexes (et non pour l'égalité de leurs valeurs absolues). Dans ce cas, nous aurions eu besoin d'utiliser un autre nom d'opérateur pour notre fonction `complex_abs_eq`.

- Bien que PostgreSQL puisse se débrouiller avec des fonctions ayant le même nom, tant qu'elles ont en argument des types de données différents, en C il ne peut exister qu'une fonction globale pour un nom donné. Aussi ne devons−nous pas donner un nom simple comme `abs_eq`. Habituellement, c'est une bonne habitude d'inclure le nom du type de données dans le nom de la fonction C, pour ne pas provoquer de conflit avec des fonctions pour d'autres types de donnée.
- Nous aurions pu faire de `abs_eq` le nom PostgreSQL de la fonction, en laissant à PostgreSQL le soin de la distinguer de toute autre fonction PostgreSQL de même nom par les types de données en argument. Pour la simplicité de l'exemple, nous donnerons à la fonction le même nom au niveau de C et au niveau de PostgreSQL.

La prochaine étape est l'enregistrement de la routine d'appui nécessaire pour les B−trees. Le code exemple C qui implémente ceci est dans le même fichier qui contient les fonctions d'opérateur. Voici comment déclarer la fonction :

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C;
```

Maintenant que nous avons les opérateurs requis et la routine d'appui, nous pouvons enfin créer la classe d'opérateur.

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

Et c'est fait ! Il devrait être possible maintenant de créer et d'utiliser les index B−tree sur les colonnes `complex`.

Nous aurions pu écrire les entrées de l'opérateur de façon plus explicite comme dans

```
        OPERATOR        1       < (complex, complex) ,
```

mais il n'y a pas besoin de faire ainsi quand les opérateurs prennent le même type de donnée que celui pour lequel la classe d'opérateur a été définie.

Les exemples ci−dessus supposent que vous voulez que cette nouvelle classe d'opérateur soit la classe d'opérateur B−tree par défaut pour le type de donnée `complex`. Si vous ne voulez pas, supprimez simplement le mot `DEFAULT`.

## 31.14.5. Classes d'opérateur inter−type

Jusqu'à maintenant, nous avons supposé implicitement qu'une classe d'opérateur s'occupe d'un seul type de données. Bien qu'il ne peut y avoir qu'un seul type de données dans une colonne d'index particulière, il est souvent utile d'indexer les opérations qui comparent une colonne indexée à une valeur d'un type de données différent. Ceci est actuellement supporté par les méthodes d'indexation B−tree et GiST.

Les index B−trees requièrent que l'opérande côté gauche de chaque opérateur soit le type de donnée indexé mais l'opérande côté droit peut être d'un type différent. Il doit exister une fonction de support disposant d'une signature correspondante. Par exemple, la classe d'opérateur intégrée pour le type `bigint` (`int8`) permet les comparaisons inter−type vers `int4` et `int2`. Cela pourrait être dupliqué par cette définition :

```
CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree AS
-- comparaisons int8 standard
OPERATOR 1 < ,
OPERATOR 2 <= ,
OPERATOR 3 = ,
OPERATOR 4 >= ,
OPERATOR 5 > ,
FUNCTION 1 btint8cmp(int8, int8) ,

-- comparaisons entre types vers int2 (smallint)
OPERATOR 1 < (int8, int2) ,
OPERATOR 2 <= (int8, int2) ,
OPERATOR 3 = (int8, int2) ,
OPERATOR 4 >= (int8, int2) ,
OPERATOR 5 > (int8, int2) ,
FUNCTION 1 btint82cmp(int8, int2) ,

-- comparaisons entre types vers int4 (integer)
OPERATOR 1 < (int8, int4) ,
OPERATOR 2 <= (int8, int4) ,
OPERATOR 3 = (int8, int4) ,
OPERATOR 4 >= (int8, int4) ,
OPERATOR 5 > (int8, int4) ,
FUNCTION 1 btint84cmp(int8, int4) ;
```

Notez que cette définition << surcharge >> la stratégie de l'opérateur et supporte les numéros de fonction. Ceci est autorisé (seulement pour les classes d'opérateur B−tree) tant que chaque instance d'un nombre particulier a un type de données côté droit différent. Les instances qui ne sont pas inter−type sont la valeur par défaut ou les opérateurs principaux de la classe d'opérateur.

Les index GiST n'autorisent pas le surchargement de stratégie ou les nombres de fonctions de support mais il est toujours possible d'obtenir l'effet d'un support de plusieurs types de données sur le côté droit en affectant un numéro de stratégie distinct à chaque opérateur qui a besoin d'être supporté. La fonction de support `cohérente` doit déterminer ce qu'elle a besoin de faire suivant le numéro de stratégie et doit être préparée à accepter des valeurs de comparaisons des types de données appropriés.

## 31.14.6. Dépendances du système pour les classes d'opérateur

PostgreSQL utilise les classe d'opérateur pour inférer les propriétés des opérateurs de plusieurs autres façons que le seul usage avec les index. Donc, vous pouvez créer des classes d'opérateur même si vous n'avez pas l'intention d'indexer une quelconque colonne de votre type de donnée.

En particulier, il existe des caractéristiques de SQL telles que `ORDER BY` et `DISTINCT` qui requièrent la comparaison et le tri des valeurs. Pour implémenter ces caractéristiques sur un type de donnée défini par l'utilisateur, PostgreSQL recherche la classe d'opérateur B−tree par défaut pour le type de donnée. Le membre << equals >> de cette classe d'opérateur définit pour le système la notion d'égalité des valeurs pour `GROUP BY` et `DISTINCT`, et le tri ordonné imposé par la classe d'opérateur définit le `ORDER BY` par défaut.

La comparaison des tableaux de types définis par l'utilisateur repose sur les sémantiques définies par la classe d'opérateur B−tree par défaut.

S'il n'y a pas de classe d'opérateur B−tree par défaut pour le type de donnée, le système cherchera une classe d'opérateur de découpage. Mais puisque cette classe d'opérateur ne fournit que l'égalité, c'est en pratique seulement suffisant pour établir l'égalité de tableau.

Quand il n'y a pas de classe d'opérateur par défaut pour un type de donnée, vous obtenez des erreurs telles que << could not identify an ordering operator >> si vous essayez d'utiliser ces caractéristiques SQL avec le type de donnée.

> **Note :** Dans les versions de PostgreSQL antérieures à la 7.4, les opérations de tri et de groupement utilisaient implicitement les opérateurs nommés =, < et >. Le nouveau comportement qui repose sur les classes d'opérateurs par défaut évite d'avoir à faire une quelconque supposition sur le comportement des opérateurs avec des noms particuliers.

## 31.14.7. Caractéristiques spéciales des classes d'opérateur

Il y a deux caractéristiques spéciales des classes d'opérateur dont nous n'avons pas encore parlées, essentiellement parce qu'elles ne sont pas utiles avec les méthodes d'index les plus communément utilisées.

Normalement, déclarer un opérateur comme membre d'une classe d'opérateur signifie que la méthode d'indexation peut retrouver exactement l'ensemble de lignes qui satisfait la condition `WHERE` utilisant cet opérateur. Par exemple,

```
SELECT * FROM table WHERE colonne_entier < 4;
```

peut être accompli exactement par un index B−tree sur la colonne entière. Mais il y a des cas où un index est utile comme un guide inexact vers la colonne correspondante. Par exemple, si un index R−tree enregistre seulement les rectangles limite des objets, alors il ne peut pas exactement satisfaire une condition `WHERE` qui teste le chevauchement entre des objets non rectangulaires comme des polygones. Cependant, nous pourrions utiliser l'index pour trouver des objets dont les rectangles limites chevauchent les limites de l'objet cible. Dans ce cas, l'index est dit être à perte pour l'opérateur, et nous ajoutons `RECHECK` à la clause `OPERATOR` dans la commande `CREATE OPERATOR CLASS`. `RECHECK` est valide si l'index garantie de retourner toutes les lignes requises, plus peut−être des lignes supplémentaires pouvant être éliminées par le recours à l'opérateur original.

Considérons à nouveau la situation où nous gardons seulement dans l'index le rectangle délimitant un objet complexe comme un polygone. Dans ce cas, il n'est pas très intéressant de conserver le polygone entier dans l'index − nous pouvons aussi bien conserver seulement un objet simple du type `box`. Cette situation est exprimée par l'option `STORAGE` dans la commande `CREATE OPERATOR CLASS` : nous aurons à écrire quelque chose comme

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
```

```
...
STORAGE box;
```

Actuellement, seule la méthode d'indexation GiST supporte un type `STORAGE` qui soit différent du type de donnée de la colonne. Les routines d'appui de GiST pour la `compression` et la `décompression` doivent s'occuper de la conversion du type de donnée quand `STORAGE` est utilisé.

---

# Chapitre 32. Déclencheurs (triggers)

Ce chapitre décrit l'écriture des fonctions déclencheurs. Ces fonctions peuvent être écrites en C ou dans n'importe quel autre langage procédural disponible. Il n'est actuellement pas possible d'écrire une fonction déclencheur en langage SQL.

## 32.1. Survol du comportement des déclencheurs

Une fonction déclencheur peut être définie pour s'exécuter avant ou après une commande `INSERT`, `UPDATE` ou `DELETE`, soit une fois par ligne modifiée soit une fois par expression SQL. Si un élément déclencheur se produit, le gestionnaire de déclencheurs est appelé au bon moment pour gérer l'évènement.

La fonction déclencheur doit être définie avant que le déclencheur lui−même puisse être créé. La fonction déclencheur doit être déclarée comme une fonction ne prenant aucun argument et retournant un type `trigger`. (La fonction déclencheur reçoit ses entrées via une structure `TriggerData` passée spécifiquement, et pas sous la forme d'arguments de fonctions ordinaires.)

Une fois qu'une fonction déclencheur est créée, le déclencheur (trigger) est établi avec *CREATE TRIGGER*. La même fonction déclencheur est utilisable par plusieurs déclencheurs.

Il existe deux types de déclencheurs : les déclencheurs en mode ligne et ceux en mode instruction. Dans un déclencheur mode ligne, la fonction du déclencheur est appelée une fois pour chaque ligne affectée par l'instruction qui a lancé le déclencheur. Au contraire, un déclencheur mode instruction n'est appelé qu'une seule fois lorsqu'une instruction appropriée est exécutée, quelque soit le nombre de lignes affectées par cette instruction. En particulier, une instruction n'affectant aucune ligne résultera toujours en l'exécution de tout déclencheur mode instruction applicable. Ces deux types sont quelque fois appelés respectivement des << déclencheurs niveau ligne >> et des << déclencheurs niveaux instruction >>.

Les déclencheurs << avant >> en mode instruction se lancent naturellement avant que l'instruction ait fait une modification alors que le déclencheur << après >> en mode instruction se lance à la fin de l'instruction. Les déclencheurs << avant >> en mode ligne s'exécutent immédiatement avant qu'une ligne particulière ne soit traitée alors que les déclencheurs << après >> en mode ligne se déclenchent à la fin de l'instruction (mais avant tout déclencheur << après >> en mode instruction).

Les fonctions déclencheurs appelées par des déclencheurs niveau instruction devraient toujours renvoyer `NULL`. Les fonctions déclencheurs appelées par des déclencheurs niveau ligne peuvent renvoyer une ligne de la table (une valeur de type `HeapTuple`) vers l'exécuteur appelant, s'ils le veulent. Un déclencheur niveau ligne exécuté avant une opération a les choix suivants :

- Il peut retourner un pointeur `NULL` pour sauter l'opération pour la ligne courante. Ceci instruit l'exécuteur de ne pas exécuter l'opération niveau ligne qui a lancé le déclencheur (l'insertion ou la modification d'une ligne particulière de la table).
- Pour les seuls déclencheurs `INSERT` et `UPDATE` au niveau ligne, la valeur de retour devient la ligne qui sera insérée ou remplacera la ligne en cours de mise à jour. Ceci permet à la fonction déclencheur de modifier la ligne en cours d'insertion ou de mise à jour.

Un déclencheur avant niveau ligne, qui n'est pas conçu pour avoir l'un de ces comportements, doit prendre garde à retourner la même ligne que celle qui lui a été passée comme nouvelle ligne (c'est−à−dire, la nouvelle

(NEW) ligne pour des déclencheurs INSERT et UPDATE, l'ancienne (OLD) ligne pour les déclencheurs DELETE).

La valeur de retour est ignorée pour les déclencheurs niveau ligne lancés après une opération. Ils pourraient donc très bien renvoyer la valeur NULL.

Si plus d'un déclencheur est défini pour le même évènement sur la même relation, les déclencheurs seront lancés dans l'ordre alphabétique de leur nom. Dans le cas de déclencheurs avant, les lignes, susceptibles d'être modifiées, renvoyées par chaque déclencheur deviennent l'argument du prochain déclencheur. Si un des déclencheurs avant renvoie un pointeur NULL, l'opération est abandonnée et les déclencheurs suivants ne sont pas lancés.

Typiquement, les déclencheurs avant en mode ligne sont utilisés pour vérifier ou modifier les données qui seront insérées ou mises à jour. Par exemple, un déclencheur avant pourrait être utilisé pour insérer l'heure actuelle dans une colonne de type timestamp ou pour vérifier que deux éléments d'une ligne sont cohérents. Les déclencheurs après en mode ligne sont pour la plupart utilisés pour propager des mises à jour vers d'autres tables ou pour réaliser des tests de cohérence avec d'autres tables. La raison de cette division du travail est qu'un déclencheur après peut être certain qu'il voit la valeur finale de la ligne alors qu'un déclencheur avant ne l'est pas ; il pourrait exister d'autres déclencheurs avant qui seront exécutés après lui. Si vous avez aucune raison spéciale pour le moment du déclenchement, le cas avant est plus efficace car l'information sur l'opération n'a pas besoin d'être sauvegardée jusqu'à la fin du traitement.

Si une fonction déclencheur exécute des commandes SQL, alors ces commandes peuvent relancer des déclencheurs. On appelle ceci un déclencheur en cascade. Il n'y a pas de limitation directe du nombre de niveaux de cascade. Il est possible que les cascades causent une invocation récursive du même déclencheur ; par exemple, un déclencheur INSERT pourrait exécuter une commande qui insère une ligne supplémentaire dans la même table, entraînant un nouveau lancement du déclencheur INSERT. Il est de la responsabilité du programmeur d'éviter les récursions infinies dans de tels scénarios.

Quand un  déclencheur est défini, des arguments peuvent être spécifiés pour lui. L'objectif de l'inclusion d'arguments dans la définition du déclencheur est de permettre à différents déclencheurs ayant des exigences similaires d'appeler la même fonction. Par exemple, il pourrait y avoir une fonction déclencheur généralisée qui prend comme arguments deux noms de colonnes et place l'utilisateur courant dans l'une et un horodatage dans l'autre. Correctement écrit, cette fonction déclencheur serait indépendante de la table particulière sur laquelle il se déclenche. Ainsi, la même fonction pourrait être utilisée pour des événements INSERT sur n'importe quelle table ayant des colonnes adéquates, pour automatiquement suivre les créations d'enregistrements dans une table de transactions par exemple. Elle pourrait aussi être utilisée pour suivre les dernières mises à jours si définie comme un déclencheur UPDATE.

Chaque langage de programmation supportant les déclencheurs a sa propre méthode pour rendre disponible les données en entrée à la fonction du déclencheur. Cette donnée en entrée inclut le type d'événement du déclencheur (c'est–à–dire INSERT ou UPDATE) ainsi que tous les arguments listés dans CREATE TRIGGER. Pour un déclencheur niveau ligne, la donnée en entrée inclut aussi la ligne NEW pour les déclencheurs INSERT et UPDATE et/ou la ligne OLD pour les déclencheurs UPDATE et DELETE. Les déclencheurs niveau instruction n'ont actuellement aucun moyen pour examiner le(s) ligne(s) individuelle(s) modifiées par l'instruction.

## 32.2. Visibilité des modifications des données

Si vous exécutez des commandes SQL dans votre fonction SQL et que ces commandes accèdent à la table pour laquelle vous créez ce déclencheur, alors vous avez besoin de connaître les règles de visibilité du déclencheur car elles déterminent si les commandes SQL voient les données changées pour lesquelles est exécuté le déclencheur. Brièvement :

- Les déclencheurs niveau instruction suivent des règles de visibilité simples : aucune des modifications réalisées par une instruction n'est visible aux déclencheurs niveau instruction appelés avant l'instruction alors que toutes les modifications sont visibles aux déclencheurs après niveau instruction.
- Les modifications de données (insertion, mise à jour ou suppression) lançant le déclencheur ne sont naturellement *pas* visibles aux commandes SQL exécutées dans un déclencheur avant en mode ligne parce qu'elles ne sont pas encore survenues.
- Néanmoins, les commandes SQL exécutées par un déclencheur avant en mode ligne *verront* les effets des modifications de données pour les lignes précédemment traitées dans la même commande externe. Ceci requiert une attention car l'ordre des événements des modifications ne sont en général pas prévisibles ; une commande SQL affectant plusieurs lignes pourrait visiter les lignes dans n'importe quel ordre.
- Quand un déclencheur après en mode ligne est exécuté, toutes les modifications de données réalisées par la commande externe sont déjà terminées et sont visibles à la fonction appelée par le déclencheur.

Il existe plus d'informations sur les règles de visibilité des données dans Section 39.4. L'exemple dans Section 32.4 contient une démonstration de ces règles.

## 32.3. Écrire des fonctions déclencheurs en C

Cette section décrit les détails de bas niveau de l'interface d'une fonction déclencheur. Ces informations ne sont nécessaires que lors de l'écriture d'une fonction déclencheur en C. Si vous utilisez un langage de plus haut niveau, ces détails sont gérés pour vous. La documentation de chaque langage procédural explique comment écrire un déclencheur dans ce langage.

Les fonctions déclencheurs doivent utiliser la << version 1 >> de l'interface du gestionnaire de fonctions.

Quand une fonction est appelée par le gestionnaire de déclencheur, elle ne reçoit aucun argument classique, mais un pointeur de << contexte >> pointant sur une structure `TriggerData`. Les fonctions C peuvent vérifier si elles sont appelées par le gestionnaire de déclencheurs ou pas en exécutant la macro

```
CALLED_AS_TRIGGER(fcinfo)
```

qui se décompose en

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

Si elle retourne la valeur vraie, alors il est bon de convertir `fcinfo->context` en type `TriggerData *` et de faire usage de la structure pointée `TriggerData`. La fonction *ne* doit *pas* modifier la structure `TriggerData` ou une donnée quelconque vers laquelle elle pointe.

`struct TriggerData` est définie dans `commands/trigger.h` :

```
typedef struct TriggerData
{
    NodeTag       type;
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
    Buffer        tg_trigtuplebuf;
    Buffer        tg_newtuplebuf;
} TriggerData;
```

où les membres sont définis comme suit :

`type`
: Toujours `T_TriggerData`.

`tg_event`
: Décrit l'évènement pour lequel la fonction est appelée. Vous pouvez utiliser les macros suivantes pour examiner `tg_event`:

  `TRIGGER_FIRED_BEFORE(tg_event)`
  : Renvoie vrai si le déclencheur est lancé avant l'opération.

  `TRIGGER_FIRED_AFTER(tg_event)`
  : Renvoie vrai si le déclencheur est lancé après l'opération.

  `TRIGGER_FIRED_FOR_ROW(tg_event)`
  : Renvoie vrai si le déclencheur est lancé pour un évènement de mode ligne.

  `TRIGGER_FIRED_FOR_STATEMENT(tg_event)`
  : Renvoie vrai si le déclencheur est lancé pour un évènement de mode instruction.

  `TRIGGER_FIRED_BY_INSERT(tg_event)`
  : Retourne vrai si le déclencheur est lancé par une commande `INSERT`.

  `TRIGGER_FIRED_BY_UPDATE(tg_event)`
  : Retourne vrai si le déclencheur est lancé par une commande `UPDATE`.

  `TRIGGER_FIRED_BY_DELETE(tg_event)`
  : Retourne vrai si le déclencheur est lancé par une commande `DELETE`.

`tg_relation`
: Un pointeur vers une structure décrivant la relation pour laquelle le déclencheur est lancé. Voir `utils/rel.h` pour les détails sur cette structure. Les choses les plus intéressantes sont `tg_relation->rd_att` (descripteur de nuplets de la relation) et `tg_relation->rd_rel->relname` (nom de la relation ; le type n'est pas `char*` mais `NameData`; utilisez `SPI_getrelname(tg_relation)` pour obtenir un `char*` si vous avez besoin d'une copie du nom).

`tg_trigtuple`
: Un pointeur vers la ligne pour laquelle le déclencheur a été lancé. Il s'agit de la ligne étant insérée, mise à jour ou effacée. Si ce déclencheur a été lancé pour une commande `INSERT` ou `DELETE`, c'est cette valeur que la fonction doit retourner si vous ne voulez pas remplacer la ligne par une ligne différente (dans le cas d'un `INSERT`) ou sauter l'opération.

`tg_newtuple`
: Un pointeur vers la nouvelle version de la ligne, si le déclencheur a été lancé pour un `UPDATE` et `NULL` si c'est pour un `INSERT` ou un `DELETE`. C'est ce que la fonction doit retourner si l'évènement est un `UPDATE` et que vous ne voulez pas remplacer cette ligne par une ligne différente ou bien sauter l'opération.

`tg_trigger`
: Un pointeur vers une structure de type `Trigger`, définie dans `utils/rel.h`:

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    bool        tgenabled;
    bool        tgisconstraint;
    Oid         tgconstrrelid;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgattr[FUNC_MAX_ARGS];
    char      **tgargs;
} Trigger;
```

où `tgname` est le nom du déclencheur, `tgnargs` est le nombre d'arguments dans `tgargs` et `tgargs` est un tableau de pointeurs vers les arguments spécifiés dans l'expression contenant la commande `CREATE TRIGGER`. Les autres membres ne sont destinés qu'à un usage interne.

`tg_trigtuplebuf`
: Le tampon contenant `tg_trigtuple` ou `InvalidBuffer` s'il n'existe pas une telle ligne ou si elle n'est pas stockée dans un tampon du disque.

`tg_newtuplebuf`
: Le tampon contenant `tg_newtuple` ou `InvalidBuffer` s'il n'existe pas une telle ligne ou si elle n'est pas stockée dans un tampon du disque.

Une fonction déclencheur doit retourner soit un pointeur `HeapTuple` soit un pointeur `NULL` (*pas* une valeur SQL NULL, donc ça ne place pas `isNull` à true). Faites attention de renvoyer soit un `tg_trigtuple` soit un `tg_newtuple`, comme approprié, si vous ne voulez pas changer la ligne en cours de modification.

## 32.4. Un exemple complet

Voici un exemple très simple de fonction déclencheur écrite en C. (Les exemples de déclencheurs écrits avec différents langages procéduraux pourraient être trouvés dans la documentation de ceux−ci.)

La fonction `trigf` indique le nombre de lignes de la table `ttest` et saute l'opération si la commande tente d'insérer une valeur NULL dans la colonne `x`. (Ainsi le déclencheur agit comme une contrainte non NULL mais n'annule pas la transaction.)

Tout d'abord, la définition des tables :

```
CREATE TABLE ttest (
    x integer
);
```

Voici le code source de la fonction tigger :

```
#include "postgres.h"
#include "executor/spi.h"       /* nécessaire pour fonctionner avec SPI */
#include "commands/trigger.h"   /* ... et les déclencheurs */

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);
```

```
Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checkNULL = false;
    bool        isNULL;
    int         ret, i;

    /* on s'assure que la fonction est appelée en tant que déclencheur */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* nuplet à retourner à l'exécuteur */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* vérification des valeurs NULL */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checkNULL = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connexion au gestionnaire SPI */
    if ((ret = SPI_connect()) < 0)
        elog(INFO, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* obtient le nombre de lignes dans la table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(NOTICE, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) renvoie int8, prenez garde à bien convertir */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isNULL));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checkNULL)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isNULL);
        if (isNULL)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

Après avoir compilé le code source, déclarez la fonction et les déclencheurs :

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'nomfichier'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();
```

A présent, testez le fonctionnement du déclencheur :

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion supprimé et déclencheur APRES non exécuté

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                       ^^^^^^^^
                        souvenez vous de ce que nous avons dit sur la visibilité.
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                       ^^^^^^
                        souvenez vous de ce que nous avons dit sur la visibilité.
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
```

```
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                       ^^^^^^
                       souvenez vous de ce que nous avons dit sur la visibilité.
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

Vous trouverez des exemples plus complexes dans `src/test/regress/regress.c` et dans `contrib/spi`.

# Chapitre 33. Système de règles

Ce chapitre discute du système de règles dans PostgreSQL. Les systèmes de règles de production sont simples conceptuellement mais il existe de nombreux points subtils impliqués dans leur utilisation.

Certains autres systèmes de bases de données définissent des règles actives pour la base de données, conservées habituellement en tant que procédures stockées et déclencheurs. Avec PostgreSQL, elles peuvent aussi être implémentées en utilisant des fonctions et des déclencheurs.

Le système de règles (plus précisément, le système de règles de réécriture de requêtes) est totalement différent des procédures stockées et des déclencheurs. Il modifie les requêtes pour prendre en considération les règles puis passe la requête modifiée au planificateur de requêtes pour planification et exécution. Il est très puissant et peut être utilisé pour beaucoup de choses comme des procédures en langage de requêtes, des vues et des versions. Les fondations théoriques et la puissance de ce système de règles sont aussi discutées dans *On Rules, Procedures, Caching and Views in Database Systems* et *A Unified Framework for Version Modeling Using Production Rules in a Database System*.

## 33.1. Arbre de requêtes

Pour comprendre comment fonctionne le système de règles, il est nécessaire de connaître quand il est appelé et quelles sont ses entrées et sorties.

Le système de règles est situé entre l'analyseur et le planificateur. Il prend la sortie de l'analyseur, un arbre de requête et les règles de réécriture définies par l'utilisateur qui sont aussi des arbres de requêtes avec quelques informations supplémentaires, et crée zéro ou plus arbres de requêtes comme résultat. Donc, son entrée et sortie sont toujours des éléments que l'analyseur lui−même pourrait avoir produit et, du coup, tout ce qu'il est voit est basiquement représentable comme une instruction SQL.

Maintenant, qu'est−ce qu'un arbre de requêtes ? C'est une représentation interne d'une instruction SQL où les parties qui le forment sont stockées séparément. Ces arbres de requêtes sont affichables dans le journal de traces du serveur si vous avez configuré les paramètres `debug_print_parse`, `debug_print_rewritten`, ou `debug_print_plan`. Les actions de règles sont aussi enregistrées comme arbres de requêtes dans le catalogue système `pg_rewrite`. Elles ne sont pas formatées comme la sortie de traces mais elles contiennent exactement la même information.

Lire un arbre de requête brut requiert un peu d'expérience. Mais comme les représentations SQL des arbres de requêtes sont suffisantes pour comprendre le système de règles, ce chapitre ne vous apprendra pas à les lire.

Lors de la lecture des représentations SQL des arbres de requêtes dans ce chapitre, il est nécessaire d'être capable d'identifier les morceaux cassés de l'instruction lorsqu'il est dans la structure de l'arbre de requête. Les parties d'un arbre de requêtes sont

le type de commande
: C'est une simple valeur indiquant quelle commande (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) produirait l'arbre de requêtes.

la table d'échelle

La table d'échelle est une liste des relations utilisées dans la requête. Dans une instruction SELECT, ce sont les relations données après le mot clé FROM.

Chaque entrée de la table d'échelle identifie une table ou une vue et indique par quel nom elle est désignée dans les autres parties de la requête. Dans l'arbre de requêtes, les entrées de la table d'échelle sont référencées par des numéros plutôt que par des noms, donc il importe peu, ici, de savoir s'il y a des noms dupliqués comme cela serait le cas avec une instruction SQL. Ceci peut arriver après que les tables d'échelle des règles aient été assemblées. Les exemples dans ce chapitre n'auront pas cette situation.

la relation résultat

C'est un index dans la table d'échelle qui identifie la relation où iront les résultats de la requête.

Les requêtes SELECT n'ont normalement pas de relation résultat. Le cas spécial d'un SELECT INTO est pratiquement identique à un CREATE TABLE suivi par un INSERT ... SELECT et n'est pas discuté séparément ici.

Pour les commandes INSERT, UPDATE, et DELETE, la relation de résultat est la table (ou vue !) où les changements prennent effet.

la liste cible

La liste cible est une liste d'expressions définissant le résultat d'une requête. Dans le cas d'un SELECT, ces expressions sont celles qui construisent la sortie finale de la requête. Ils correspondent aux expressions entre les mots clés SELECT et FROM. (* est seulement une abréviation pour tous les noms de colonnes d'une relation. Il est étendu par l'analyseur en colonnes individuelles, pour que le système de règles ne le voit jamais.)

Les commandes DELETE n'ont pas besoin d'une liste de colonnes car elles ne produisent aucun résultat. En fait, le planificateur ajoutera une entrée spéciale CTID pour aller jusqu'à la liste de cibles mais c'est après le système de règles ; pour le système de règles et sera discuté après ; pour le système de règles, la liste de cibles est vide.

Pour les commandes INSERT, la liste cible décrit les nouvelles lignes devant aller dans la relation résultat. Elle consiste en des expressions de la clause VALUES ou en celles de la clause SELECT dans INSERT ... SELECT. La première étape du processus de réécriture ajoute les entrées de la liste cible pour les colonnes n'ont affectées par la commande originale mais ayant des valeurs par défaut. Toute colonne restante (avec soit une valeur donnée soit une valeur par défaut) sera remplie par le planificateur avec une expression NULL constante.

Pour les commandes UPDATE, la liste cible décrit les nouvelles lignes remplaçant les anciennes. Dans le système des règles, elle contient seulement les expressions de la partie SET colonne = expression de la commande. Le planificateur gérera les colonnes manquantes en insérant des expressions qui copient les valeurs provenant de l'ancienne ligne dans la nouvelle. et elle ajoutera l'ancienne entrée CTID dans la nouvelle ligne comme pour le DELETE.

Chaque entrée de la liste cible contient une expression qui peut être une valeur constante, une variable pointant vers une colonne d'une des relations de la table d'échelle, un paramètre ou un arbre d'expressions réalisé à partir d'appels de fonctions, de constantes, de variables, d'opérateurs, etc.

la qualification

La qualification de la requête est une expression ressemblant à une de celles contenues dans les entrées de la liste cible. La valeur résultant de cette expression est un booléen indiquant si l'opération (INSERT, UPDATE, DELETE ou SELECT) pour la ligne de résultat final devrait être exécutée ou non. Elle correspond à la clause WHERE d'une instruction SQL.

l'arbre de jointure

L'arbre de jointure de la requête affiche la structure de la clause FROM. Pour une simple requête comme SELECT ... FROM a, b, c, l'arbre de jointure est une simple liste d'éléments de FROM parce que nous sommes autorisés à les joindre dans tout ordre. Mais quand des expressions JOIN, et plus particulièrement les jointures externes, sont utilisées, nous devons les joindre dans l'ordre affiché par les jointures. Dans ce cas, l'arbre de jointure affiche la structure des expressions JOIN. Les restrictions associées avec ces clauses JOIN particulières (à partir d'expressions ON ou USING) sont enregistrées comme des expressions de qualification attachées aux nœuds de l'arbre de jointure. Il s'avère agréable d'enregistrer l'expression de haut niveau WHERE comme une qualification attachée à l'élément de l'arbre de jointure de haut niveau. Donc, réellement, l'arbre de jointure représente à la fois les clauses FROM et WHERE d'un SELECT.

le reste

Les autres parties de l'arbre de requête comme la clause ORDER BY n'ont pas d'intérêt ici. Le système de règles substitue quelques entrées lors de l'application des règles mais ceci n'a pas grand chose à voir avec les fondamentaux du système de règles.

# 33.2. Vues et système de règles

Avec PostgreSQL, les vues sont implémentées en utilisant le système de règles. En fait, il n'y a essentiellement pas de différences entre

```
CREATE VIEW mavue AS SELECT * FROM matable;
```

et ces deux commandes

```
CREATE TABLE mavue (liste de colonnes identique à
celle de matable);
CREATE RULE "_RETURN" AS ON SELECT TO mavue DO INSTEAD
    SELECT * FROM matable;
```

parce que c'est exactement ce que fait la commande CREATE VIEW en interne. Ceci a quelques effets de bord. L'un d'entre eux est que l'information sur une vue dans les catalogues système PostgreSQL est exactement le même que pour celui d'une table. Donc, pour l'analyseur, il n'y a aucune différence entre une table et une vue. Elles sont la même chose : des relations.

## 33.2.1. Fonctionnement des règles SELECT

Les règles ON SELECT sont appliquées à toutes les requêtes comme la dernière étape, même si la commande donnée est un INSERT, UPDATE ou DELETE. Et ils ont des sémantiques différentes à partir des règles sur les autres types de commandes dans le fait qu'elles modifient l'arbre de requêtes en place au lieu d'en créer un nouveau. Donc, les règles SELECT sont décrites avant.

Actuellement, il n'existe qu'une action dans une règle ON SELECT et elle doit être une action SELECT inconditionnelle qui est INSTEAD. Cette restriction était requise pour rendre les règles assez sûres pour les ouvrir aux utilisateurs ordinaires et cela restreint les règles ON SELECT à agir comme des vues.

Les exemples pour ce chapitre sont deux vues jointes, réalisant quelques calculs, et quelques vues supplémentaires les utilisant à leur tour. Une des deux premières vues est personnalisée plus tard en ajoutant

des règles pour des opérations INSERT, UPDATE et DELETE de façon à ce que le résultat final sera une vue qui se comporte comme une vraie table avec quelques fonctionnalités magiques. Il n'existe pas une tel exemple pour commencer et ceci rend les choses plus difficiles à obtenir. Mais il est mieux d'avoir un exemple couvrant tous les points discutés étape par étape plutôt que plusieurs exemples, rendant la compréhension plus difficile.

Pour cet exemple, nous avons besoin d'une petite fonction min, renvoyant la valeur la plus basse entre deux entiers. Nous la créons comme

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$' LANGUAGE SQL STRICT;
```

Les tables réelles dont nous avons besoin dans les deux premières descriptions du système de règles sont les suivantes :

```
CREATE TABLE shoe_data (
    shoename   text,          -- clé primaire
    sh_avail   integer,       -- nombre de pairs disponibles
    slcolor    text,          -- couleur de lacet préférée
    slminlen   real,          -- longueur minimum du lacet
    slmaxlen   real,          -- longueur maximum du lacet
    slunit     text           -- unité de longueur
);

CREATE TABLE shoelace_data (
    sl_name    text,          -- clé primaire
    sl_avail   integer,       -- nombre de pairs disponibles
    sl_color   text,          -- couleur du lacet
    sl_len     real,          -- longueur du lacet
    sl_unit    text           -- unité de longueur
);

CREATE TABLE unit (
    un_name    text,          -- clé primaire
    un_fact    real           -- facteur pour le transformer en cm
);
```

Comme vous pouvez le constater, elles représentent les données d'un magasin de chaussures.

Les vues sont créées avec

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
```

```
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

La commande `CREATE VIEW` pour la vue `shoelace` (qui est la plus simple que nous avons) écrira une relation `shoelace` et une entrée dans `pg_rewrite` indiquant la présence d'une règle de réécriture devant être appliquée à chaque fois que la relation `shoelace` est référencée dans une table d'échelle de la requête. La règle n'a aucune qualification de règle (discuté plus tard, avec les règles non `SELECT` car les règles `SELECT` ne le ont pas encore) et qu'il s'agit de `INSTEAD`. Notez que les qualifications de règles ne sont pas identiques aux qualifications de requêtes. L'action de notre règle a une qualification de requête. L'action de la règle a un arbre de requête qui est une copie de l'instruction `SELECT` dans la commande de création de la vue.

> **Note :** Les deux entrées supplémentaires de la table d'échelle pour `NEW` et `OLD` (nommées `*NEW*` et `*OLD*` pour des raisons historiques dans l'arbre de requête affiché) que vous pouvez voir dans l'entrée de `pg_rewrite` ne sont d'aucun intérêt pour les règles `SELECT`.

Maintenant, nous remplissons `unit`, `shoe_data` et `shoelace_data`, et lançons une requête simple sur une vue :

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
```

```
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

C'est la requête `SELECT` la plus simple que vous pouvez lancer sur nos vues, donc nous prenons cette opportunité d'expliquer les bases des règles de vues. `SELECT * FROM shoelace` a été interprété par l'analyseur et a produit l'arbre de requête

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

et ceci est donné au système de règles. Ce système traverse la table d'échelle et vérifie s'il existe des règles pour chaque relation. Lors du traitement d'une entrée de la table d'échelle pour `shoelace` (la seule jusqu'à maintenant), il trouve la règle `_RETURN` avec l'arbre de requête

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

Pour étendre la vue, la réécriture crée simplement une entrée de la table d'échelle de sous−requête contenant l'arbre de requête de l'action de la règle et substitue cette entrée par l'original référencé dans la vue. L'arbre d'échelle résultant de la réécriture est pratiquement identique à celui que vous avez saisi

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

Néanmoins, il y a une différence : la table d'échelle de la sous−requête a deux entrées supplémentaires, `shoelace *OLD*` et `shoelace *NEW*`. Ces entrées ne participent pas directement dans la requête car elles ne sont pas référencées par l'arbre de jointure de la sous−requête ou par la liste cible. La réécriture les utilise pour enregistrer l'information de vérification des droits d'accès qui étaient présents à l'origine dans l'entrée de table d'échelle référencée par la vue. De cette façon, l'exécution vérifiera toujours que l'utilisateur a les bons droits pour accéder à la vue même s'il n'y a pas d'utilisation directe de la vue dans la requête réécrite.

C'était la première règle appliquée. Le système de règles continuera de vérifier les entrées restantes de la table d'échelle dans la requête principale (dans cet exemple, il n'en existe pas plus), et il vérifiera récursivement les entrées de la table d'échelle dans la sous−requête ajoutée pour voir si une d'elle référence les vues. (Mais il n'étendra ni `*OLD*` ni `*NEW*` — sinon nous aurions une récursion infinie !) Dans cet exemple, il n'existe pas de règles de réécriture pour `shoelace_data` ou `unit`, donc la réécriture est terminée et ce qui est ci−dessus est le résultat final donné au planificateur.

Maintenant, nous voulons écrire une requête qui trouve les chaussures actuellement dans le magasin dont nous avons les lacets correspondant (couleur et longueur) et pour lesquels le nombre total de pairs correspondant exactement est supérieur ou égal à deux.

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

Cette fois, la sortie de l'analyseur est l'arbre de requête

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

La première règle appliquée sera celle de la vue `shoe_ready` et cela résultera avec cet arbre de requête

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
          AND rsl.sl_len_cm >= rsh.slminlen_cm
          AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

De façon similaire, les règles pour `shoe` et `shoelace` sont substituées dans la table d'échelle de la sous−requête, amenant à l'arbre de requête final à trois niveaux :

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
                       sh.slcolor,
                       sh.slminlen,
                       sh.slminlen * un.un_fact AS slminlen_cm,
                       sh.slmaxlen,
                       sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                       sh.slunit
                  FROM shoe_data sh, unit un
                 WHERE sh.slunit = un.un_name) rsh,
               (SELECT s.sl_name,
                       s.sl_avail,
```

```
                        s.sl_color,
                        s.sl_len,
                        s.sl_unit,
                        s.sl_len * u.un_fact AS sl_len_cm
                    FROM shoelace_data s, unit u
                    WHERE s.sl_unit = u.un_name) rsl
          WHERE rsl.sl_color = rsh.slcolor
            AND rsl.sl_len_cm >= rsh.slminlen_cm
            AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail > 2;
```

Il s'avère que le planificateur réduira cet arbre en un arbre de requête à deux niveaux : les commandes `SELECT` du bas seront << remontées >> dans le `SELECT` du milieu car il n'est pas nécessaire de les traiter séparément. Mais le `SELECT` du milieu restera séparé du haut car il contient des fonctions d'agrégat. Si nous les avions monté, cela aurait modifié le comportement du `SELECT` de haut niveau, ce qui n'est pas ce que nous voulons. Néanmoins, réduire l'arbre de requête est une optimisation qui ne concerne pas le système de réécriture.

> **Note :** Actuellement, il n'existe pas de mécanisme d'arrêts de récursion pour les règles de vues dans le système de règles (uniquement pour les autres types de règles). Ceci n'est pas trop gênant car la seule façon de placer ce système dans une boucle sans fin (massacrant le processus serveur jusqu'à ce qu'il atteigne la limite en mémoire) est de créer des tables et de configurer les règles de vue à la main avec `CREATE RULE` de telle façon que une sélection à partir de l'autre qui sélectionne à partir de la première. Cela pourrait ne jamais arriver si `CREATE VIEW` est utilisé parce que pour le premier `CREATE VIEW`, la deuxième relation n'existera pas et, du coup, la première vue ne pourra pas sélectionner quelque chose de la seconde.

## 33.2.2. Règles de vue dans des instructions non `SELECT`

Deux détails de l'arbre de requête n'ont pas été abordés dans la description des règles de vue ci–dessus. Ce sont le type de commande et le relation résultante. En fait, les règles de vue n'ont pas besoin de cette information.

Il existe seulement quelques différences entre un arbre de requête pour un `SELECT` et un pour une autre commande. De façon évidente, ils ont un type de commande différent et pour une commande autre qu' un `SELECT`, la relation résultante pointe vers l'entrée de table d'échelle où le résultat devrait arriver. Tout le reste est absolument identique. Donc, avec deux tables `t1` et `t2` avec les colonnes `a` et `b`, les arbres de requêtes pour les deux commandes

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b WHERE t1.a = t2.a;
```

sont pratiquement identiques. En particulier :

- Les tables d'échelle contiennent des entrées pour les tables `t1` et `t2`.
- Les listes cibles contiennent une variable pointant vers la colonne `b` de l'entrée de la table d'échelle pour la table `t2`.
- Les expressions de qualification comparent les colonnes `a` des deux entrées de table d'échelle pour une égalité.
- Les arbres de jointure affichent une jointure simple entre `t1` et `t2`.

La conséquence est que les deux arbres de requête résultent en des plans d'exécution similaires : ce sont tous les deux des jointures sur les deux tables. Pour l'UPDATE, les colonnes manquantes de t1 sont ajoutées à la liste cible par le planificateur et l'arbre de requête final sera lu de cette façon

```
UPDATE t1 SET a = t1.a, b = t2.b WHERE t1.a = t2.a;
```

et, du coup, l'exécuteur lancé sur la jointure produira exactement le même résultat qu'un

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

le ferait. Mais il y a un petit problème avec UPDATE : l'exécuteur ne fait pas attention au but du résultat de la jointure. Il produit simplement un ensemble de lignes composant le résultat. La différence entre une commande SELECT et une commande UPDATE est gérée par celui qui a appelé l'exécuteur. L'appelant sait toujours (en regardant dans l'arbre de requêtes) qu'il s'agit d'un UPDATE, et il sait que le résultat ira dans la table t1. Mais quelles lignes disponibles maintenant seront remplacées par les nouvelles lignes ?

Pour résoudre ce problème, une autre entrée est ajoutée dans la liste cible de l'UPDATE (et aussi dans les instructions DELETE) : l'identifiant actuel du tuple (CTID, acronyme de *current tuple ID*). Cette colonne système contient le numéro de bloc du fichier et la position dans le bloc pour cette ligne. Connaissant la table, le CTID peut être utilisé pour récupérer la ligne originale de t1 à mettre à jour. Après avoir ajouté le CTID dans la liste cible, la requête ressemble à ceci

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Maintenant, un autre détail de PostgreSQL entre en jeu. Les anciennes lignes de la table ne sont pas surchargées et cela explique pourquoi ROLLBACK est rapide. Avec un UPDATE, la nouvelle ligne résultat est insérée dans la table (après avoir enlevé le CTID) et dans le nouvel en−tête de ligne de l'ancienne ligne, vers où pointe le CTID, les entrées cmax et xmax sont configurés par le compteur de commande actuel et par l'identifiant de transaction actuel. Du coup, l'ancienne ligne est cachée et, après validation de la transaction, le nettoyeur (vacuum) peut réellement la déplacer.

Connaissant tout ceci, nous pouvons simplement appliquer les règles de vues de la même façon que toute autre commande. Il n'y a pas de différence.

## 33.2.3. Puissance des vues dans PostgreSQL

L'exemple ci−dessus démontre l'incorporation des définitions de vues par le système de règles dans l'arbre de requête original. Dans le deuxième exemple, un simple SELECT d'une vue a créé un arbre de requête final qui est une jointure de quatre tables (unit a été utilisé deux fois avec des noms différents).

Le bénéfice de l'implémentation des vues avec le système de règles est que le planificateur a toute l'information sur les tables à parcourir et sur les relations entre ces tables et les qualifications restrictives à partir des vues et les qualifications à partir de la requête originale dans un seule arbre de requête. Et c'est toujours la situation quand la requête originale est déjà une jointure sur des vues. Le planificateur doit décider du meilleur chemin pour exécuter la requête et plus le planificateur a d'informations, meilleure sera la décision. Le système de règles implémenté dans PostgreSQL s'en assure, c'est toute l'information disponible sur la requête à ce moment.

### 33.2.4. Mise à jour d'une vue

Qu'arrive−t'il si une vue est nommée comme la relation cible d'un INSERT, UPDATE ou DELETE ? Après avoir réalisées les substitutions décrites ci−dessus, nous aurons un arbre de requête dans lequel la relation résultante pointe vers une entrée de la table d'échelle de la sous−requête. Ceci ne fonctionnera pas car la réécriture renvoie une erreur si elle aperçoit qu'elle a produit une telle chose.

Pour modifier ceci, nous pouvons définir des règles modifiant le comportement de ce type de commandes. C'est donc le thème de la prochaine section.

## 33.3. Règles sur INSERT, UPDATE et DELETE

Les règles définies sur INSERT, UPDATE et DELETE sont significativement différentes des règles de vue décrites dans la section précédente. Tout d'abord, leur commande CREATE RULE permet plus de choses :

- Elles peuvent n'avoir aucune action.
- Elles peuvent avoir plusieurs actions.
- Elles peuvent être de type INSTEAD ou ou ALSO (valeur par défaut).
- Les pseudo relations NEW et OLD deviennent utiles.
- Elles peuvent avoir des qualifications de règles.

Ensuite, elles ne modifient pas l'arbre de requête en place. À la place, elles créent de nouveaux arbres de requêtes et peuvent abandonner l'original.

### 33.3.1. Fonctionnement des règles de mise à jour

Gardez la syntaxe

```
CREATE RULE rule_name AS ON event
    TO object [WHERE rule_qualification]
    DO [ALSO|INSTEAD] [action | (actions) | NOTHING];
```

en tête. Dans la suite, *règles de mise à jour* signifie les règles qui sont définies sur INSERT, UPDATE ou DELETE.

Les règles de mise à jour sont appliquées par le système de règles lorsque la relation résultante et le type de commande d'un arbre de requête sont égaux pour l'objet et l'événement donné dans la commande CREATE RULE. Pour les règles de mise à jour, le système de règles crée une liste d'arbres de requêtes. Initialement, la liste d'arbres de requêtes est vide. Il peut y avoir aucune (mot clé NOTHING), une ou plusieurs actions. Pour simplifier, nous verrons une règle avec une action. Cette règle peut avoir une qualification et peut être de type INSTEAD ou ALSO (valeur par défaut).

Qu'est−ce qu'une qualification de règle ? C'est une restriction indiquant le moment où doivent être réalisés les actions de la règle. Cette qualification peut seulement référencer les pseudo relations NEW et/ou OLD, qui représentent basiquement la relation qui a été donné comme objet (mais avec une signification spéciale).

Donc, nous avons quatre cas qui produisent les arbres de requêtes suivant pour une règle à une seule action.

Sans qualification et ALSO

l'arbre de requête à partir de l'action de la règle avec l'ajout de la qualification de l'arbre de requête

Sans qualification mais avec `INSTEAD`

l'arbre de requête à partir de l'action de la règle avec l'ajout de la qualification de l'arbre de requête original

Qualification donnée et `ALSO`

l'arbre de requête à partir de l'action de la règle avec l'ajout de la qualification de la règle et de la qualification de l'arbre de requête original

Qualification donnée avec `INSTEAD`

l'arbre de requête à partir de l'action de la règle avec la qualification de la requête et la qualification de l'arbre de requête original ; et l'ajout de l'arbre de requête original avec la qualification inverse de la règle

Enfin, si la règle est `ALSO`, l'arbre de requête original est ajouté à la liste. Comme seules les règles qualifiées `INSTEAD` ont déjà ajouté l'arbre de requête original, nous finissons avec un ou deux arbres de requête en sortie pour une règle avec une action.

Pour les règles `ON INSERT`, la requête originale (si elle n'est pas supprimée par `INSTEAD`) est réalisée avant toute action ajoutée par les règles. Ceci permet aux actions de voir les lignes insérées. Mais pour les règles `ON UPDATE` et `ON DELETE`, la requête originale est réalisée après les actions ajoutées par les règles. Ceci nous assure que les actions pourront voir les lignes à mettre à jour ou à supprimer ; sinon, les actions pourraient ne rien faire parce qu'elles ne trouvent aucune ligne correspondant à leurs qualifications.

Les arbres de requêtes générés à partir des actions de règles sont envoyés de nouveau dans le système de réécriture et peut−être que d'autres règles seront appliquées résultant en plus ou moins d'arbres de requêtes. Donc, les actions d'une règle doivent avoir soit un type de commande différent soit une relation résultante différente de celle où la règle elle−même est active, sinon ce processus récursif se terminera dans une boucle infinie. (L'expansion récursive d'une règle sera détectée et rapportée comme une erreur.)

Les arbres de requête trouvés dans les actions du catalogue système `pg_rewrite` sont seulement des modèles. Comme ils peuvent référencer les entrées de la table d'échelle pour `NEW` et `OLD`, quelques substitutions ont dû être faites avant qu'elles ne puissent être utilisées. Pour toute référence de `NEW`, une entrée correspondante est recherchée dans la liste cible de la requête originale. Si elle est trouvée, cette expression de l'entrée remplace la référence. Sinon, `NEW` signifie la même chose que `OLD` (pour un `UPDATE`) ou est remplacé par une valeur NULL (pour un `INSERT`). Toute référence à `OLD` est remplacée par une référence à l'entrée de la table d'échelle qui est la relation résultante.

Après que le système a terminé d'appliquer des règles de mise à jour, il applique les règles de vues pour le(s) arbre(s) de requête produit(s). Les vues ne peuvent pas insérer de nouvelles actions de mise à jour, donc il n'est pas nécessaire d'appliquer les règles de mise à jour à la sortie d'une réécriture de vue.

#### 33.3.1.1. Une première requête étape par étape

Disons que nous voulons tracer les modifications dans la colonne `sl_avail` de la relation `shoelace_data`. Donc, nous allons configurer une table de traces et une règle qui va écrire une entrée lorsqu'un `UPDATE` est lancé sur `shoelace_data`.

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- modification de shoelace
    sl_avail   integer,       -- nouvelle valeur disponible
    log_who    text,          -- qui l'a modifié
    log_when   timestamp      -- quand
);
```

```
CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

Maintenant, quelqu'un fait :

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

et nous regardons la table des traces :

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

C'est ce à quoi nous nous attendions. Voici ce qui s'est passé en tâche de fond. L'analyseur a créé l'arbre de requête

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

Il existe une règle `log_shoelace` qui est `ON UPDATE` avec l'expression de qualification de la règle

```
NEW.sl_avail <> OLD.sl_avail
```

et l'action

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*;
```

(Ceci semble un peu étrange car vous ne pouvez pas normalement écrire `INSERT ... VALUES ... FROM`. Ici, la clause `FROM` indique seulement qu'il existe des entrées de la table d'échelle dans l'arbre de requête pour `*NEW*` et `*OLD*`. Elles sont nécessaires pour qu'elles puissent être référencées par des variables dans l'arbre de requête de la commande `INSERT`.)

La règle est une règle qualifiée `ALSO` de façon à ce que le système de règles doit renvoyer deux arbres de requêtes : l'action de la règle modifiée et l'arbre de requête original. Dans la première étape, la table d'échelle de la requête original est incorporée dans l'arbre de requête d'action de la règle. Ceci a pour résultat :

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data;
```

Pour la deuxième étape, la qualification de la règle lui est ajoutée, donc l'ensemble de résultat est restreint aux lignes où `sl_avail` a changé :

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE *NEW*.sl_avail <> *OLD*.sl_avail;
```

(Ceci semble encore plus étrange car `INSERT ... VALUES` n'a pas non plus une clause `WHERE` mais le planificateur et l'exécuteur n'auront pas de difficultés avec ça. Ils ont besoin de supporter cette même fonctionnalité pour `INSERT ... SELECT`.)

À l'étape 3, la qualification de l'arbre de requête original est ajoutée, restreignant encore plus l'ensemble de résultats pour les seules lignes qui auront été modifiées par la requête originale :

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE *NEW*.sl_avail <> *OLD*.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

La quatrième étape remplace les références à `NEW` par les entrées de la liste cible à partir de l'arbre de requête original ou par les références de la variable correspondante à partir de la relation résultat :

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE 6 <> *OLD*.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

L'étape 5 modifie les références `OLD` en référence de la relation résultat :

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

C'est tout. Comme la règle est de type `ALSO`, nous affichons aussi l'arbre de requêtes original. En bref, l'affichage à partir du système de règles est une liste de deux arbres de requêtes est une liste de deux arbres de requêtes correspondant à ces instructions :

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

```
UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

Elles sont exécutées dans cet ordre et c'est exactement le but de la règle.

Les substitutions et les qualifications ajoutées nous assurent que, si la requête originale était,

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

aucune trace ne serait écrire. Dans ce cas, l'arbre de requête original ne contient pas une entrée dans la liste cible pour `sl_avail`, donc `NEW.sl_avail` sera remplacé par `shoelace_data.sl_avail`. Du coup, la commande supplémentaire générée par la règle est

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

et la qualification ne sera jamais vraie.

Si la requête originale modifie plusieurs lignes, cela fonctionne aussi. Donc, si quelqu'un a lancé la commande

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

en fait, quatre lignes sont modifiées (`sl1`, `sl2`, `sl3` et `sl4`). Mais `sl3` a déjà `sl_avail = 0`. Dans ce cas, la qualification des arbres de requêtes originaux sont différents et cela produit un arbre de requête supplémentaire

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

à générer par la règle. Cet arbre de requête aura sûrement insérer trois nouvelles lignes de traces. Et c'est tout à fait correct.

Ici, nous avons vu pourquoi il est important que l'arbre de requête original soit exécuté en premier. Si l'`UPDATE` a été exécutée avant, toutes les lignes pourraient aussi être initialisées à zéro, donc le `INSERT` tracé ne trouvera aucune ligne à `0 <> shoelace_data.sl_avail`.

## 33.3.2. Coopération avec les vues

Une façon simple de protéger les relations d'une vue de la possibilité mentionnée que quelqu'un essaie de lancer des `INSERT`, `UPDATE` ou `DELETE` sur elles est de laisser s'abandonner ces arbres de requête. Donc, nous créons les règles

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
```

```
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

Maintenant, si quelqu'un essaie de faire une de ces opérations sur la vue `shoe`, le système de règles appliquera ces règles. Comme les règles n'ont pas d'action et sont de type `INSTEAD`, la liste résultant des arbres de requêtes sera vide et la requête entière deviendra vide car il ne reste rien à optimiser ou exécuter après que le système de règles en ait terminé avec elle.

Une façon plus sophistiquée d'utiliser le système de règles est de créer les règles qui réécrivent l'arbre de requête en un faisant la bonne opération sur les vraies tables. Pour réaliser cela sur la vue `shoelace`, nous créons les règles suivantes :

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

Maintenant, supposons que quelque fois, un paquet de lacets arrive au magasin avec une grosse liste. Mais vous ne voulez pas mettre à jour manuellement la vue `shoelace` à chaque fois. À la place, nous configurons deux petites tables  une où vous pouvez insérer les éléments de la liste et une avec une astuce spéciale. Voici les commandes de création :

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
```

```
    WHERE sl_name = NEW.ok_name;
```

Maintenant, vous pouvez remplir la table `shoelace_arrive` avec les données de la liste :

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

Jetez un œil rapidement aux données actuelles :

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

Maintenant, déplacez les lacets arrivés dans :

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

et vérifiez le résultat :

```
SELECT * FROM shoelace ORDER BY sl_name;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+----------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

C'est un long chemin du `INSERT ... SELECT` à ces résultats. Et la description de la transformation de l'arbre de requêtes sera la dernière dans ce chapitre. Tout d'abord, voici la sortie de l'analyseur

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

Maintenant, la première règle `shoelace_ok_ins` est appliquée et transforme ceci en

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

et jette l'`INSERT` actuel sur `shoelace_ok`. La requête réécrite est passée de nouveau au système de règles et la seconde règle appliquée `shoelace_upd` produit

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

De nouveau, il s'agit d'une règle `INSTEAD` et l'arbre de requête précédent est jeté. Notez que cette requête utilise toujours la vue `shoelace`. Mais le système de règles n'a pas fini cette étape, donc il continue et lui applique la règle `_RETURN`. Nous obtenons

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

Enfin, la règle `log_shoelace` est appliquée, produisant l'arbre de requête supplémentaire

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
```

```
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u,
       shoelace_data *OLD*, shoelace_data *NEW*
       shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

Une fois que le système de règles tombe en panne de règles et renvoie les arbres de requêtes générés.

Donc, nous finissons avec deux arbres de requêtes finaux qui sont équivalents aux instructions SQL

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

Le résultat est que la donnée provenant d'une relation insérée dans une autre, modifiée en mise à jour dans une troisième, modifiée en mise à jour dans une quatrième, cette dernière étant tracée dans une cinquième, se voit réduite à deux requêtes.

Il y a un petit détail assez horrible. En regardant les deux requêtes, nous nous apercevons que la relation `shoelace_data` apparaît deux fois dans la table d'échelle où cela pourrait être réduit à une seule occurrence. Le planificateur ne gère pas ceci et, du coup, le plan d'exécution de la sortie du système de règles pour `INSERT` sera

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

alors qu'omettre la table d'échelle supplémentaire résulterait en un

```
Merge Join
  ->  Seq Scan
```

```
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

qui produit exactement les mêmes entrées dans la table des traces. Du coup, le système de règles a causé un parcours supplémentaire dans la table `shoelace_data` qui n'est absolument pas nécessaire. Et le même parcours redondant est fait une fois de plus dans l'`UPDATE`. Mais ce fut réellement un travail difficile de rendre tout ceci possible.

Maintenant, nous faisons une démonstration finale du système de règles de PostgreSQL et de sa puissance. Disons que nous ajoutons quelques lacets avec des couleurs extraordinaires à votre base de données :

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Nous voulons créer une vue vérifiant les entrées `shoelace` qui ne correspondent à aucune chaussure pour la couleur. Voici la vue

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Sa sortie est

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

Maintenant, nous voulons le configurer pour que les lacets qui ne correspondent pas et qui ne sont pas en stock sont supprimés de la base de données. Pour rendre la chose plus difficile à PostgreSQL, nous ne les supprimerons pas directement. À la place, nous créons une vue supplémentaire

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

et le faisons de cette façon :

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
             WHERE sl_name = shoelace.sl_name);
```

*Voilà*:

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |        80
 sl2     |        6 | black    |    100 | cm      |       100
 sl7     |        6 | brown    |     60 | cm      |        60
 sl4     |        8 | black    |     40 | inch    |     101.6
```

```
 sl3     |       10 | black    |     35 | inch    |      88.9
 sl8     |       21 | brown    |     40 | inch    |     101.6
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
 sl5     |        4 | brown    |      1 | m       |       100
 sl6     |       20 | brown    |    0.9 | m       |        90
(9 rows)
```

Un `DELETE` sur une vue, avec une qualification de sous−requête qui utilise au total quatre vues imbriquées/jointes, où l'entre d'entre elles a une qualification de sous−requête contenant une vue et où les colonnes des vues calculées sont utilisées, est réécrite en un seul arbre de requête qui supprime les données demandées sur la vraie table.

Il existe probablement seulement quelques situations dans le vrai monde où une telle construction est nécessaire. Mais, vous vous sentez mieux quand cela fonctionne.

## 33.4. Règles et droits

À cause de la réécriture des requêtes par le système de règles de PostgreSQL, d'autres tables/vues que celles utilisées dans la requête originale pourraient être accédées. Lorsque des règles de mise à jour sont utilisées, ceci peut inclure des droits d'écriture sur les tables.

Les règles de réécriture n'ont pas de propriétaire séparé. Le propriétaire d'une relation (table ou vue) est automatiquement le propriétaire des règles de réécriture qui lui sont définies. Le système de règles de PostgreSQL modifie le comportement du système de contrôle d'accès par défaut. Les relations qui sont utilisées à cause des règles se voient vérifier avec les droits du propriétaire de la règle, et non avec ceux de l'utilisateur appelant cette règle. Ceci signifie qu'un utilisateur a seulement besoin des droits requis pour les tables/vues qu'il nomme explicitement dans ses requêtes.

Par exemple : un utilisateur a une liste de numéros de téléphone dont certains sont privés, les autres étant d'intérêt pour la secrétaire du bureau. Il peut construire de cette façon :

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE NOT private;
GRANT SELECT ON phone_number TO secretary;
```

Personne sauf lui (et les superutilisateurs de la base de données) ne peut accéder à la table `phone_data`. Mais, à cause du `GRANT`, la secrétaire peut lancer un `SELECT` sur la vue `phone_number`. Le système de règles réécrira le `SELECT` sur `phone_number` en un `SELECT` sur `phone_data` et ajoutera la qualification que seules les entrées non privées (donc où `private` est faux) sont désirées. Comme l'utilisateur est le propriétaire de `phone_number` et du coup le propriétaire de la règle, le droit de lecture de `phone_data` est maintenant vérifié avec ses propres privilèges et la requête est autorisée. La vérification de l'accès à `phone_number` est aussi réalisée mais ceci est fait avec l'utilisateur appelant, donc personne sauf l'utilisateur et la secrétaire ne peut l'utiliser.

Les droits sont vérifiés règle par règle. Donc, la secrétaire est actuellement la seule à pouvoir voir les numéros de téléphone publiques. Mais la secrétaire peut configurer une autre vue et autoriser l'accès au public. Du coup, tout le monde peut voir les données de `phone_number` via la vue de la secrétaire. Ce que la secrétaire ne peut pas faire est de créer une vue qui accède directement à `phone_data`. (En fait, elle le peut mais cela ne fonctionnera pas car tous les accès seront refusés lors de la vérification des droits.) Dès que l'utilisateur s'en rendra compte, du fait que la secrétaire a ouvert la vue `phone_number` à tout le monde, il peut révoquer son

accès. Immédiatement, tous les accès de la vue de la secrétaire échoueront.

Il pourrait être dit que cette vérification règle par règle est une brèche de sécurité mais ce n'est pas le cas. Si cela ne fonctionne pas de cette façon, la secrétaire pourrait copier une table avec les mêmes colonnes que `phone_number` et y copier les données une fois par jour. Du coup, ce sont ces propres données et il peut accorder l'accès à tout le monde s'il le souhaite. Une commande `GRANT` signifie, << J'ai confiance en vous >>. Si quelqu'un en qui vous avez confiance se comporte ainsi, il est temps d'y réfléchir et d'utiliser `REVOKE`.

Ce mécanisme fonctionne aussi pour les règles de mise à jour. Dans les exemples de la section précédente, le propriétaire des tables de la base de données d'exemple pourrait accorder les droits `SELECT`, `INSERT`, `UPDATE` et `DELETE` sur la vue `shoelace` à quelqu'un d'autre mais seulement `SELECT` sur `shoelace_log`. L'action de la règle pourrait écrire des entrées de trace qui seraient toujours exécutées avec succès et que l'autre utilisateur pourrait voir. Mais il ne peut pas créer d'entrées fausses, pas plus qu'il ne peut manipuler ou supprimer celles qui existent.

## 33.5. Règles et statut de commande

Le serveur PostgreSQL renvoie une chaîne de statut de commande, comme `INSERT 149592 1`, pour chaque commande qu'il reçoit. C'est assez simple lorsqu'il n'y a pas de règles impliquées. Mais qu'arrive–t'il lorsque la requête est réécrite par des règles ?

Les règles affectent le statut de la commande de cette façon :

- S'il n'y a pas de règle `INSTEAD` inconditionnelle pour la requête, alors la requête donnée originellement sera exécutée et son statut de commande sera renvoyé comme d'habitude. (Mais notez que s'il y avait des règles `INSTEAD` conditionnelles, la négation de leur qualifications sera ajouté à la requête initiale. Ceci pourrait réduire le nombre de lignes qu'il traite et, si c'est le cas, le statut rapporté en sera affecté.)
- S'il y a des règles `INSTEAD` inconditionnelles pour la requête, alors la requête originale ne sera pas exécutée du tout. Dans ce cas, le serveur renverra le statut de la commande pour la dernière requête qui a été insérée par une règle `INSTEAD` (conditionnelle ou non) et est du même type de commande (`INSERT`, `UPDATE` ou `DELETE`) que la requête originale. Si aucune requête ne rencontrant ces pré–requis n'est ajoutée à une règle, alors le statut de commande renvoyé affiche le type de requête original et annule le compteur de ligne et le champ OID.

(Ce système a été établi pour PostgreSQL 7.3. Dans les versions précédentes, le statut de commande pourrait afficher des résultats différents lorsque les règles existent.)

Le programmeur peut s'assurer que toute règle `INSTEAD` désirée est celle qui initialise le statut de commande dans le deuxième cas en lui donnant un nom de règle étant le dernier en ordre alphabétique parmi les règles actives pour qu'elle soit appliquée en dernier.

## 33.6. Règles contre déclencheurs

Beaucoup de choses pouvant se faire avec des déclencheurs peuvent aussi être implémentées en utilisant le système de règles de PostgreSQL. Un des points qui ne pourra pas être implémenté par les règles sont certains types de contraintes, notamment les clés étrangères. Il est possible de placer une règle qualifiée qui réécrit une commande en `NOTHING` si la valeur d'une colonne n'apparaît pas dans l'autre table. Mais alors les données

sont jetées et ce n'est pas une bonne idée. Si des vérifications de valeurs valides sont requises et dans le cas où il y a une erreur invalide, un message d'erreur devrait être généré et cela devra se faire avec un déclencheur.

D'un autre côté, un déclencheur qui est lancé sur un `INSERT` pour une vue peut faire la même chose qu'une règle : placez les données ailleurs et supprimez les insertions dans la vue. Mais, il ne pourra pas faire la même chose avec un `UPDATE` ou un `DELETE` parce qu'il n'y a pas de vraies données sur la vue qui pourraient être parcourues. Du coup, le déclencheur ne serait jamais appelé. Seule une règle pourra vous aider.

Pour les éléments qui peuvent être implémentés par les deux, ce qui sera le mieux dépend de l'utilisation de la base de données. Un déclencheur est exécuté une fois pour chaque ligne affectée. Une règle manipule la requête ou en génère une autre. Donc, si un grand nombre de lignes sont affectées pour une instruction, ue règle lançant une commande supplémentaire sera certainement plus rapide qu'un déclencheur appelé pour chaque ligne et qui devra exécuter ces opérations autant de fois. Néanmoins, l'approche du déclencheur est conceptuellement plus simple que l'approche de la règle et est plus facile à utiliser pour les novices.

Ici, nous montrons un exemple où le choix d'une règle ou d'un déclencheur joue sur une situation. Voici les deux tables :

```
CREATE TABLE computer (
    hostname        text,    -- indexé
    manufacturer    text     -- indexé
);

CREATE TABLE software (
    software        text,    -- indexé
    hostname        text     -- indexé
);
```

Les deux tables ont plusieurs milliers de lignes et les index sur `hostname` sont uniques. La règle ou le déclencheur devrait implémenter une contrainte qui supprime les lignes de `software` référençant un ordinateur supprimé. Le déclencheur utiliserait cette commande :

```
DELETE FROM software WHERE hostname = $1;
```

Comme le déclencheur est appelé pour chaque ligne individuelle supprimée à partir de `computer`, il peut préparer et sauvegarder le plan pour cette commande et passer la valeur `hostname` dans le paramètre. La règle devra être réécrite ainsi

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Maintenant, nous apercevons différents types de suppressions. Dans le cas d'un

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

la table `computer` est parcourue par l'index (rapide), et la commande lancée par le déclencheur pourrait aussi utiliser un parcours d'index (aussi rapide). La commande supplémentaire provenant de la règle serait

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

Comme il y a une configuration appropriée des index, le planificateur créera un plan

```
Nestloop
```

```
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

Donc, il n'y aurait pas trop de différence de performance entre le déclencheur et l'implémentation de la règle.

Avec la prochaine suppression, nous voulons nous débarrasser des 2000 ordinateurs où `hostname` commence avec `old`. Il existe deux commandes possibles pour ce faire. Voici l'une d'elle

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

La commande ajoutée par la règle sera

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
                       AND software.hostname = computer.hostname;
```

avec le plan

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

L'autre commande possible est

```
DELETE FROM computer WHERE hostname ~ '^old';
```

ce qui finira dans le plan d'exécution suivant pour la commande ajoutée par la règle :

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

Ceci monte que le planificateur ne réalise pas que la qualification pour `hostname` dans `computer` pourrait aussi être utilisée pour un parcours d'index sur `software` quand il existe plusieurs expressions de qualifications combinées avec `AND`, ce qui correspond à ce qu'il fait dans la version expression rationnelle de la commande. Le déclencheur sera appelé une fois pour chacun des 2000 anciens ordinateurs qui doivent être supprimées, et ceci résultera en un parcours d'index sur `computer` et 2000 parcours d'index sur `software`. L'implémentation de la règle le fera en deux commandes qui utilisent les index. Et cela dépend de la taille globale de la table `software`, si la règle sera toujours aussi rapide dans la situation du parcours séquentiel. 2000 exécutions de commandes à partir du déclencheur sur le gestionnaire SPI prend un peu de temps, même si tous les blocs d'index seront rapidement dans le cache.

La dernière commande que nous regardons est

```
DELETE FROM computer WHERE manufacurer = 'bim';
```

De nouveau, ceci pourrait résulter en de nombreuses lignes à supprimer de `computer`. Donc, le déclencheur lancera de nouveau de nombreuses commandes via l'exécuteur. La commande générée par la règle sera

```
DELETE FROM software WHERE computer.manufacurer = 'bim'
                       AND software.hostname = computer.hostname;
```

Le plan pour cette commande sera encore la boucle imbriquée sur les deux parcours d'index, en utilisant seulement un index différent sur `computer`:

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

Dans chacun de ces cas, les commandes supplémentaires provenant du système de règles seront plus ou moins indépendantes du nombre de lignes affectées en une commande.

Voici le résumé, les règles seront seulement significativement plus lentes que les déclencheurs si leur actions résultent en des jointures larges et mal qualifiées, une situation où le planificateur échoue.

---

# Chapitre 34. Langages de procédures

PostgreSQL permet aux utilisateurs d'écrire des fonctions et des procédures avec des langages autres que le SQL et le C. Ces autres langages sont appelés génériquement des *langages de procédures* (LP). Pour une fonction écrite dans un langage de procédures, le serveur de bases de données n'a aucune connaissance en interne de la façon d'interpréter le texte du source de la fonction. À la place, la tâche est passée à un gestionnaire spécial qui connaît les détails du langage. Le gestionnaire peut soit faire tout le travail de découpage, d'analyse syntaxique, d'exécution, etc. par lui−même, soit servir de << colle >> entre PostgreSQL et une implémentation existante d'un langage de programmation. Le gestionnaire lui−même est une fonction en langage C compilée dans une bibliothèque partagée et chargée à la demande, tout comme toute autre fonction C.

Actuellement, il existe quatre langages de procédures disponibles dans la distribution standard de PostgreSQL : PL/pgSQL (Chapitre 35), PL/Tcl (Chapitre 36), PL/Perl (Chapitre 37) et PL/Python (Chapitre 38). D'autres langages peuvent être définis par les utilisateurs. Les bases de développement d'un nouveau langage de procédures sont couvertes dans le Chapitre 45.

Certains langages de procédures supplémentaires ne sont pas inclus dans la distribution principale. Annexe H a quelques informations pour les trouver.

## 34.1. Installation de langages de procédures

Un langage de procédures doit être << installé >> dans chaque base de données où il est destiné à être utilisé. Mais les langages de procédures installés dans la base de données `template1` sont automatiquement disponibles dans toutes les bases de données créées par la suite car leurs entrées dans `template1` seront copiées par `CREATE DATABASE`. Ainsi, l'administrateur de bases de données peut décider quels langages sont disponibles dans quelles bases de données et peut rendre certains langages disponibles par défaut s'il le choisit.

Pour les langages fournis avec la distribution standard, le programme createlang peut être utilisé pour installer le langage au lieu de reporter tous les détails à la main. Par exemple, pour installer le langage PL/pgSQL dans la base de données `template1`, utilisez

```
createlang plpgsql template1
```

La procédure manuelle décrite ci−dessous n'est recommandée que pour installer des langages personnalisés que `createlang` ne connaît pas.

**Installation manuelle de langages de procédures**

Un langage de procédures s'installe dans une base de données en quatre étapes, qui doivent être effectuées par le superutilisateur de bases de données. Le programme `createlang` automatise tout sauf l'étape 1.

1. La bibliothèque partagée pour le gestionnaire de langage doit être compilée et installée dans le répertoire de bibliothèques approprié. Ceci fonctionne comme la construction et l'installation de modules à l'aide de fonctions C classiques définies par un utilisateur ; voir la Section 31.9.6. Souvent, le gestionnaire du langage dépendra d'une bibliothèque externe fournissant le moteur de langage

actuel ; dans ce cas, il doit aussi être installé.

2. Le gestionnaire doit être déclaré par la commande

```
CREATE FUNCTION nom_fonction()
    RETURNS langage
    AS 'chemin-vers-objet-partagé'
    LANGUAGE C;
```

Le type de retour spécial de `langage` indique au système de bases de données que cette fonction ne renvoie pas l'un des types de données SQL et n'est pas utilisable directement dans des expressions SQL.

3. En option, le gestionnaire de langages pourrait fournir une fonction de << validation >> qui vérifie la définition d'une fonction sans réellement l'exécuter. La fonction de validation est appelée par `CREATE FUNCTION` si elle existe. Si une fonction de validation est fournie par le gestionnaire, déclarez–la avec une commande comme

```
CREATE FUNCTION nom_fonction_validation(oid)
    RETURNS void
    AS 'chemin-vers-objet-partagé'
    LANGUAGE C;
```

4. Le LP doit être déclaré par la commande

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE
nom_langage
    HANDLER nom_fonction_gestionnaire
    [VALIDATOR
nom_fonction_valideur] ;
```

Le mot clé optionnel `TRUSTED` indique que les utilisateurs ordinaires n'ayant pas de droits de super–utilisateur devraient être autorisés à utiliser ce langage pour créer des procédures fonctions et déclencheurs. Les fonctions LP étant exécutées au sein du serveur de bases de données, le paramètre `TRUSTED` ne devrait être positionné que pour les langages n'accédant pas aux organes internes du serveur de bases ou du système de fichiers. Les langages PL/pgSQL, PL/Tcl, et PL/Perl sont considérés comme dignes de confiances ; les langages PL/TclU, PL/PerlU, et PL/PythonU sont conçus pour ne fournir que des fonctionnalités limitées et *ne* devraient *pas* être considérées dignes de confiance.

L'Exemple 34–1 montre comment fonctionne la procédure d'installation manuelle avec le langage PL/pgSQL.

**Exemple 34–1. Installation manuelle de PL/pgSQL**

La commande suivante indique au serveur de bases de données l'emplacement de la bibliothèque partagée pour la fonction de gestion des appels du langage PL/pgSQL.

```
CREATE FUNCTION plpgsql_call_handler() RETURNS language_handler AS
    '$libdir/plpgsql' LANGUAGE C;
```

PL/pgSQL a une fonction de validation, donc nous la déclarons aussi :

```
CREATE FUNCTION plpgsql_validator(oid) RETURNS void AS
```

```
    '$libdir/plpgsql' LANGUAGE C;
```

La commande

```
CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
    HANDLER plpgsql_call_handler
    VALIDATOR plpgsql_validator;
```

définit alors que la fonction déclarée précédemment devrait être invoquée pour les procédures fonctions ou déclencheurs pour lesquelles l'attribut langage est `plpgsql`.

Lors de l'installation par défaut de PostgreSQL, le gestionnaire pour le langage PL/pgSQL est compilé et installé dans le répertoire des bibliothèques (<< lib >>). Si le support de Tcl y est configuré, les gestionnaires pour PL/Tcl et PL/TclU sont aussi compilés et installés au même endroit. De même, les gestionnaires de PL/Perl et PL/PerlU sont compilés et installés si le support de Perl est configuré, et PL/PythonU est installé si le support de Python est configuré.

---

# Chapitre 35. PL/pgSQL – SQL Procedural Language

PL/pgSQL est un langage procédural chargeable pour le système de bases de données PostgreSQL. Les objectifs de la conception de PL/pgSQL ont été de créer un langage procédural chargeable qui

- peut être utilisé pour créer des procédures fonctions et déclencheurs,
- ajoute des structures de contrôle au langageSQL,
- peut effectuer des traitements complexes,
- hérite de tous les types, fonctions et opérateurs définis par les utilisateurs,
- peut être défini comme digne de confiance par le serveur,
- est facile à utiliser.

Exception faites des conversions d'entrées/sorties et des fonctions de traitement pour les types définis par l'utilisateur, tout ce qui peut être défini dans les fonctions en langage C peut aussi être fait avec PL/pgSQL. Par exemple, il est possible de créer des fonctions de traitement conditionnel complexes et, par la suite, de les utiliser pour définir des opérateurs ou de les utiliser dans des expressions d'index.

## 35.1. Survol

Le gestionnaire d'appel PL/pgSQL découpe le texte source de la fonction et produit un arbre d'instructions binaires internes la première fois que la fonction est appelée (au sein de chaque session). L'arbre d'instructions traduit complètement la structure de l'expression PL/pgSQL, mais les expressions SQL individuelles et les commandes SQL utilisées dans la fonction ne sont pas traduites immédiatement.

Chaque expression et commande SQL étant d'abord utilisée dans la fonction, l'interpréteur PL/pgSQL crée un plan d'exécution élaboré (en utilisant les fonctions `SPI_prepare` et `SPI_saveplan` du gestionnaire SPI). Les visites suivantes à cette expression ou commande réutilisent le plan élaboré. Ainsi, une fonction avec du code conditionnel qui contient de nombreuses expressions pour lesquelles des plans d'exécution pourraient être nécessaires ne feront que préparer et sauvegarder ces plans, qui ne sont réellement utilisés que durant le temps que la connexion à la base de données vivra. Ceci peut réduire substantiellement le temps total nécessaire à l'analyse syntaxique, et générer des plans d'exécution pour les expressions d'une fonction PL/pgSQL. Un inconvénient est que les erreurs d'une expression ou commande particulière peuvent ne pas être détectée jusqu'à ce que cette partie de la fonction soit atteinte au cours de l'exécution.

Une fois que PL/pgSQL a créé un plan d'exécution pour une commande de fonction particulière, il réutilisera ce plan pour le temps que durera la connexion à la base de données. C'est généralement un gain de performances, mais cela peut causer quelques problèmes si vous modifiez dynamiquement votre schéma de base de données. Par exemple

```
CREATE FUNCTION populate() RETURNS integer AS $$
DECLARE
    -- declarations
BEGIN
    PERFORM my_function();
END;
$$ LANGUAGE plpgsql;
```

Si vous exécutez la fonction ci−dessus, l'OID de `my_function()` sera référencé dans le plan d'exécution produit pour l'expression `PERFORM`. Par la suite, si vous détruisez et recréez `my_function()`, `populate()` ne sera plus en mesure de trouver `my_function()`. Vous auriez alors à recréer `populate()`, ou au moins à lancer une nouvelle connexion à la base de donnée pour faire en sorte de la compiler à nouveau. Un autre moyen d'éviter ce problème est d'utiliser `CREATE OR REPLACE FUNCTION` lors de la mise à jour de la définition de `my_function` (quand une fonction est << remplacée >>, son OID n'est pas changé).

Comme PL/pgSQL sauvegarde les plans d'exécution de cette façon, les commandes SQL qui apparaissent directement dans une fonction PL/pgSQL doivent se référer aux mêmes tables et colonnes pour chaque exception; en fait, vous ne pouvez pas utiliser un paramètre tel que le nom d'une table ou d'une colonne dans une commande SQL. Pour contourner cette restriction, vous pouvez construire des commandes dynamiques en utilisant l'expression PL/pgSQL `EXECUTE` — au prix de la construction d'un nouveau plan d'exécution pour chaque exécution.

> **Note :** L'expression PL/pgSQL `EXECUTE` n'a pas de rapport avec l'instruction SQL *EXECUTE* supportée par le serveur PostgreSQL. L'expression `EXECUTE` du serveur ne peut pas être utilisée au sein des fonctions PL/pgSQL (et n'est pas nécessaire).

## 35.1.1. Avantages de l'Utilisation de PL/pgSQL

SQL est le langage que PostgreSQL et la plupart des autres bases de données relationnelles utilisent comme langage de requête. Il est portable et facile à apprendre. Mais chaque expression SQL doit être exécutée individuellement par le serveur de bases de données.

Cela signifie que votre application client doit envoyer chaque requête au serveur de bases de données, attendre que celui−ci la traite, recevoir les résultats, faire quelques traitements, et enfin envoyer d'autres requêtes au serveur. Tout ceci induit des communications interprocessus et peut aussi induire une surcharge du réseau si votre client est sur une machine différente du serveur de bases de données.

Grâce à PL/pgSQL vous pouvez grouper un bloc de traitement et une série de requêtes *au sein* du serveur de bases de données, et bénéficier ainsi de la puissance d'un langage procédural, tout en gagnant du temps puisque vous évitez toute la charge de la communication client/serveur. Ceci peut permettre un gain de performances considérable.

Ainsi, avec PL/pgSQL vous pouvez utiliser tous les types de données, opérateurs et fonctions du SQL.

## 35.1.2. Arguments Supportés et Types de Données Résultats

Les fonctions écrites en PL/pgSQL peuvent accepter comme argument n'importe quel type de données supporté par le serveur, et peuvent renvoyer un résultat de n'importe lequel de ces types. Elles peuvent aussi accepter ou renvoyer n'importe quel type composite (type ligne) spécifié par nom. Il est aussi possible de déclarer une fonction PL/pgSQL renvoyant un type `record`, signifiant que le résultat est un type ligne dont les colonnes sont déterminées par spécification dans la requête appelante, comme traité dans Section 7.2.1.4.

Les fonctions PL/pgSQL peuvent aussi être déclarées comme acceptant et renvoyant les types << polymorphes >>, `anyelement` et `anyarray`. Le type de données réel géré par une fonction polymorphe peut varier d'appel en appel, comme traité dans Section 31.2.1. Voir l'exemple dans Section 35.4.1.

Les fonctions PL/pgSQL peuvent aussi être déclarées comme devant renvoyer un << set >> ou une table de n'importe lequel des type de données dont elles peuvent renvoyer une instance unique. De telles fonctions génèrent leur sortie en exécutant `RETURN NEXT` pour chaque élément désiré de l'ensemble résultat.

Enfin, une fonction PL/pgSQL peut être déclarée comme renvoyant `void` si elle n'a pas de valeur de retour utile.

PL/pgSQL n'a pas actuellement de support complet pour les types de domaine : il traite un domaine de la même façon qu'un type scalaire sous−jacent. Ceci signifie que les contraintes associées avec le domaine ne seront pas forcées. Ceci n'est pas un problème pour les arguments des fonctions mais c'est hasardeux de déclarer une fonction PL/pgSQL renvoyant un type domaine.

# 35.2. Astuces pour Développer en PL/pgSQL

Un bon moyen de développer en PL/pgSQL est d'utiliser l'éditeur de texte de votre choix pour créer vos fonctions, et d'utiliser psql dans une autre fenêtre pour charger et tester ces fonctions. Si vous procédez ainsi, une bonne idée est d'écrire la fonction en utilisant `CREATE OR REPLACE FUNCTION`. De cette façon vous pouvez recharger le fichier seulement pour mettre à jour la définition de la fonction. Par exemple :

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

Pendant que psql tourne, vous pouvez charger ou recharger de telles définitions de fonction avec

```
\i filename.sql
```

et alors immédiatement soumettre des commandes SQL pour tester la fonction.

Un autre bon moyen de développer en PL/pgSQL est d'utiliser un outil d'accès à la base de données muni d'une interface graphique qui facilite le développement dans un langage procédural. Un exemple d'un tel outil est PgAccess, bien que d'autres existent. Ces outils fournissent souvent des fonctionnalités pratiques telles que la détection des guillemets ouverts et facilitent la re−création et le débogage des fonctions.

## 35.2.1. Utilisation des guillemets simples (quotes)

Le code d'une fonction PL/pgSQL est spécifié dans la commande `CREATE FUNCTION` comme une chaîne de caractères. Si vous écrivez la chaîne littérale de la façon ordinaire en l'entourant de guillemets simples, alors tout guillemet simple dans le corps de la fonction doit être doublé ; de la même façon, les antislashs doivent être doublés. Doubler les guillemets devient rapidement difficile et, dans la plupart des cas compliqués, le code peut devenir rapidement incompréhensible parce que vous pouvez facilement vous trouver avoir besoin d'une douzaine, voire plus, de guillemets adjacents. À la place, il est recommandé d'écrire le corps de la fonction en tant qu'une chaîne littérale << avec guillemets dollar >> (voir Section 4.1.2.2). Dans cette approche, vous ne doublez jamais les marques de guillemets mais vous devez faire attention à choisir un délimiteur dollar différent pour chaque niveau d'imbrication dont vous avez besoin. Par exemple, vous pourriez écrire la commande `CREATE FUNCTION` en tant que

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
   ....
$PROC$ LANGUAGE plpgsql;
```

A l'intérieur de ceci, vous pourriez utiliser des guillemets pour les chaînes littérales simples dans les commandes SQL et `$$` pour délimiter les fragments de commandes SQL que vous assemblez comme des chaînes. Si vous avez besoin de mettre entre guillemets du texte qui inclut `$$`, vous pouvez utiliser `$Q$`, et ainsi de suite.

Le graphe suivant montre ce que vous devez faire lors de l'écriture de guillemets simples sans guillemets dollar. Cela pourrait être utile lors de la traduction de code avec guillemets simples en quelque chose de plus compréhensible.

1 guillemet simple
: Pour commencer et terminer le corps de la fonction, par exemple :

```
CREATE FUNCTION foo() RETURNS integer AS '
        .....
' LANGUAGE plpgsql;
```

Partout au sein du corps de la fonction entouré de guillemets simples, les guillemets simples *doivent* aller par paires.

2 guillemets simples
: Pour les chaînes de caractères à l'intérieur du corps de la fonction, par exemple :

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

Dans l'approche du guillemet dollar, vous devriez juste écrire

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

ce qui serait exactement ce que l'analyseur PL/pgSQL verrait dans les deux cas.

4 guillemets simples
: Quand vous avez besoin d'un guillemet simple dans une chaîne constante à l'intérieur du corps de la fonction , par exemple :

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

La valeur effectivement concaténée à `a_output` est : `AND name LIKE 'foobar' AND xyz`.

Dans l'approche du guillemet dollar, vous auriez écrit

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

Faites attention que chaque délimiteur en guillemet dollar ne soient pas simplement `$$`.

6 guillemets simples
: Quand un simple guillemet dans une chaîne à l'intérieur du corps d'une fonction est adjacent à la fin de cette chaîne constante, par exemple :

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

La valeur effectivement concaténée à `a_output` est alors : `AND name LIKE 'foobar'`.

Dans l'approche guillemet dollar, ceci devient

```
    a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 guillemets simples

Lorsque vous voulez 2 guillemets simples dans une chaîne constante (qui compte pour 8 guillemets simples) et qu'elle est adjacente à la fin de cette chaîne constante (2 de plus). Vous n'aurez probablement besoin de ceci que si vous écrivez une fonction qui génère d'autres fonctions comme dans Exemple 35−6. Par exemple :

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

La valeur de `a_output` sera alors :

```
if v_... like ''...'' then return ''...''; end if;
```

Dans l'approche du guillemet dollar, ceci devient

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
|| referrer_keys.key_string || $$'
then return '$$  || referrer_keys.referrer_type
|| $$'; end if;$$;
```

où nous supposons que nous avons seulement besoin de placer des marques de de guillemets simples dans `a_output` parce que les guillemets seront recalculés avant utilisation.

Une approche différente est d'échapper les guillemets du corps de la fonction avec un antislash plutôt qu'en les doublant. Avec cette méthode, vous vous verrez écrire des choses telles que `\'\'` au lieu de `''''`. Certains trouveront ceci plus lisible, d'autres non.

# 35.3. Structure de PL/pgSQL

PL/pgSQL est un langage structuré en blocs. Le texte complet de la définition d'une fonction doit être un *bloc*. Un bloc est défini comme :

```
[ <<label>> ]
[ DECLARE
    déclarations ]
BEGIN
    instructions
END;
```

Chaque déclaration et chaque expression au sein du bloc est terminé par un point−virgule. Un bloc qui apparaît à l'intérieur d'un autre bloc doit avoir un point−virgule après `END`, comme montré ci−dessus ; néanmoins, le `END` final qui conclut le corps d'une fonction n'a pas besoin de point−virgule.

Tous les mots clés et identifiants peuvent être écrits en majuscules et minuscules mélangées. Les identifiants sont implicitement convertis en minuscule à moins d'être entourés de guillemets doubles.

Il y a deux types de commentaires dans PL/pgSQL. Un double tiret (`--`) débute une ligne de commentaire qui s'étend jusqu'à la fin de la ligne. Un `/*` débute un bloc de commentaire qui s'étend jusqu'à la prochaine occurrence de `*/`. Les blocs de commentaires ne peuvent pas être imbriqués, mais les commentaires de lignes

(double tiret) peuvent être contenus dans un bloc de commentaire et un double tiret peut cacher les délimiteurs du bloc de commentaire `/*` et `*/`.

Chaque expression de la section expression d'un bloc peut être un *sous−bloc*. Les sous−blocs peuvent être utilisés pour des groupements logiques ou pour localiser des variables locales à un petit groupe d'instructions.

Les variables déclarées dans la section déclaration précédant un bloc sont initialisées à leur valeur par défaut chaque fois qu'on entre dans un bloc et pas seulement une fois à chaque appel de fonction. Par exemple :

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Quantity here is 30
    quantity := 50;
    --
    -- Crée un sous-bloc
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Quantity here is 80
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Quantity here is 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Il est important de ne pas confondre l'utilisation de `BEGIN`/`END` pour grouper les instructions dans PL/pgSQL avec les commandes de bases de données pour le contrôle des transactions. Les `BEGIN`/`END` de PL/pgSQL ne servent qu'au groupement ; ils ne débutent ni ne terminent une transaction. Les procédures fonctions et déclencheurs sont toujours exécutées à l'intérieur d'une transaction établie par une requête extérieure — ils ne peuvent pas être utilisés pour commencer ou valider une transaction car ils n'auraient pas de contexte pour s'exécuter. Néanmoins, un bloc contenant une clause `EXCEPTION` forme réellement une sous−transaction qui peut être annulée sans affecter la transaction externe. Pour plus d'informations sur ce point, voir Section 35.7.5.

## 35.4. Déclarations

Toutes les variables utilisées dans un bloc doivent être déclarées dans la section déclaration du bloc. (La seule exception est que la variable de boucle d'une boucle `FOR` effectuant une itération sur des valeurs entières est automatiquement déclarée comme variable entière.

Les variables PL/pgSQL peuvent être de n'importe quel type de données tels que `integer`, `varchar`, et `char`.

Voici quelques exemples de déclarations de variables :

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
```

```
arow RECORD;
```

La syntaxe générale d'une déclaration de variable est :

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];
```

La clause DEFAULT, si indiquée, spécifie la valeur initiale assignée à la variable quand on entre dans le bloc. Si la clause DEFAULT n'est pas indiquée, la variable est initialisée à la valeur SQL NULL. L'option CONSTANT empêche l'assignation de la variable, de sorte que sa valeur reste constante pour la durée du bloc. Si NOT NULL est spécifié, l'assignement d'une valeur NULL aboutira à une erreur d'exécution. Les valeurs par défaut de toutes les variables déclarées NOT NULL doivent être spécifiées non NULL.

La valeur par défaut est évaluée à chaque entrée du bloc. Ainsi, par exemple, l'assignation de now() à une variable de type timestamp donnera à la variable l'heure de l'appel de la fonction courante, et non l'heure au moment où la fonction a été précompilée.

Exemples :

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

## 35.4.1. Alias de Paramètres de Fonctions

Les paramètres passés aux fonctions sont nommés par les identifiants $1, $2, etc. Éventuellement, des alias peuvent être déclarés pour les noms de paramètres de type $*n* afin d'améliorer la lisibilité. L'alias ou l'identifiant numérique peuvent être utilisés indifféremment pour se référer à la valeur du paramètre.

Il existe deux façons de créer un alias. La façon préférée est de donner un nom au paramètre dans la commande CREATE FUNCTION, par exemple :

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
```

L'autre façon, seulement disponible depuis PostgreSQL 8.0, est de déclarer explicitement un alias en utilisant la syntaxe de déclaration

```
nom ALIAS FOR $n;
```

Le même exemple dans ce style ressemble à

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Quelques exemples de plus :

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS '
```

```
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- quelques traitements
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(in_t tablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Lorsque le type de retour d'une fonction PL/pgSQL est déclaré comme type polymorphe (`anyelement` ou `anyarray`), un paramètre spécial `$0` est crée. Son type de donnée est le type effectif de retour de la fonction, déduit d'après les types d'entrée (voir Section 31.2.1). Ceci permet à la fonction d'accéder à son type de retour réel comme on le voit ici Section 35.4.2. `$0` est initialisé à NULL et peut être modifié par la fonction, de sorte qu'il peut être utilisé pour contenir la variable de retour si besoin est, bien que ça ne soit pas requis. On peut aussi donner à `$0` un alias. Par exemple, cette fonction fonctionne comme un opérateur + pour n'importe quel type de données.

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

## 35.4.2. Copie de Types

*variable*`%TYPE`

`%TYPE` fournit le type de données d'une variable ou d'une colonne de table. Vous pouvez l'utiliser pour déclarer des variables qui contiendront des valeurs de bases de données. Par exemple, disons que vous avez une colonne nommée `user_id` dans votre table `users`. Pour déclarer une variable du même type de données que `users.user_id` vous pouvez écrire :

```
user_id users.user_id%TYPE;
```

En utilisant `%TYPE` vous n'avez pas besoin de connaître le type de données de la structure à laquelle vous faites référence, et plus important, si le type de données de l'objet référencé change dans le futur (par exemple : vous changez le type de `user_id` de `integer` à `real`), vous pouvez ne pas avoir besoin de changer votre définition de fonction.

`%TYPE` est particulièrement utile dans le cas de fonctions polymorphes, puisque les types de données nécessaires aux variables internes peuvent changer d'un appel à l'autre. Des variables appropriées peuvent être créées en appliquant `%TYPE` aux arguments de la fonction ou à la la variable fictive de résultat.

## 35.4.3. Types ligne

```
name table_name%ROWTYPE;
name composite_type_name;
```

Une variable de type composite est appelée variable *ligne* (ou variable *row−type*). Une telle variable peut contenir une ligne entière de résultat de requête `SELECT` ou `FOR`, du moment que l'ensemble de colonnes de la requête correspond au type déclaré de la variable. Les champs individuels de la valeur row sont accessibles en utilisant la notation pointée, par exemple `rowvar.field`.

Une variable ligne peut être déclarée de façon à avoir le même type que les lignes d'une table ou vue existante, en utilisant la notation *`table_name`*`%ROWTYPE` ou elle peut être déclarée en donnant un nom de type composite. (Chaque table ayant un type de données associé du même nom, il importe peu dans PostgreSQL que vous écriviez `%ROWTYPE` ou pas. Cependant la forme utilisant `%ROWTYPE` est plus portable.)

Les paramètres d'une fonction peuvent être des types composites (lignes complètes de tables). En ce cas, l'identifiant correspondant `$`*`n`* sera une variable ligne, à partir de laquelle les champs peuvent être sélectionnés, par exemple `$1.user_id`.

Seules les colonnes définies par l'utilisateur d'une ligne de table sont accessibles dans une variable de type ligne, et non l'OID ou d'autres colonnes systèmes (parce que la ligne pourrait être issue d'une vue). Les champs du type ligne héritent des tailles des champs de la table ou de leur précision pour les types de données tels que `char(`*`n`*`)`.

Voici un exemple d'utilisation des types composites :

```
CREATE FUNCTION merge_fields(t_row tablename) RETURNS text AS $$
DECLARE
    t2_row table2name%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2name WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM tablename t WHERE ... ;
```

## 35.4.4. Types Record

```
nom RECORD;
```

Les variables record sont similaires aux variables de type ligne, mais n'ont pas de structure prédéfinie. Elles empruntent la structure effective de type ligne de la ligne à laquelle elles sont assignées durant une commande `SELECT` or `FOR`. La sous−structure d'une variable record peut changer à chaque fois qu'on l'assigne. Une conséquence de cela est que jusqu'à ce qu'elle ait été assignée, elle n'a pas de sous−structure, et toutes les tentatives pour accéder à un de ses champs entraîneront une erreur d'exécution.

Notez que `RECORD` n'est pas un vrai type de données mais seulement un paramètre fictif (placeholder). Il faut aussi réaliser que lorsqu'une fonction PL/pgSQL est déclarée renvoyer un type `record`, il ne s'agit pas tout à fait du même concept qu'une variable record, même si une telle fonction peut aussi utiliser une variable record pour contenir son résultat. Dans les deux cas la structure réelle de la ligne n'est pas connue quand la fonction est écrite, mais dans le cas d'une fonction renvoyant un type `record` la structure réelle est déterminée quand

la requête appelante est analysée, alors qu'une variable record peut changer sa structure de ligne à la volée.

### 35.4.5. RENAME

```
RENAME ancien nom TO nouveau nom;
```

En utilisant la déclaration RENAME, vous pouvez changer le nom d'une variable, d'un record ou d'un row (ligne). C'est particulièrement utile si NEW ou OLD doivent être référencés par un autre nom dans une procédure déclencheur. Voir aussi ALIAS.

Exemples :

```
RENAME id TO user_id;
RENAME this_var TO that_var;
```

> **Note :** RENAME semble ne pas fonctionner dans PostgreSQL 7.3. Cette correction est de faible priorité, ALIAS couvrant la plupart des utilisations pratiques de RENAME.

## 35.5. Expressions

Toutes les expressions utilisées dans les instructions PL/pgSQL sont traitées par l'exécuteur SQL classique du serveur. En effet, une requête comme

```
SELECT expression
```

est exécutée en utilisant le gestionnaire SPI. Avant l'évaluation, les occurrences des identifiants de variables PL/pgSQL sont remplacées par des paramètres, et les valeurs réelles des variables sont passées à l'exécuteur dans le tableau des paramètres. Ceci permet au plan de requêtes pour le SELECT de n'être élaboré qu'une fois et réutilisé pour les évaluations postérieures.

L'évaluation faite par l'analyseur syntaxique principal de PostgreSQL a quelques effets de bord sur l'interprétation des valeurs constantes. Plus précisément, il y a une différence entre ce que font ces deux fonctions :

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS timestamp AS $$
BEGIN
    INSERT INTO logtable VALUES (logtxt, 'now');
    RETURN 'now';
END;
$$ LANGUAGE plpgsql;
```

et

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS timestamp AS $$
DECLARE
    curtime timestamp;
BEGIN
    curtime := 'now';
    INSERT INTO logtable VALUES (logtxt, curtime);
    RETURN curtime;
END;
$$ LANGUAGE plpgsql;
```

Dans le cas de logfunc1, l'analyseur syntaxique principal de PostgreSQL sait, quand il élabore le plan pour l' INSERT, que la chaîne 'now' doit être interprétée comme un timestamp parce que la colonne cible de logtable est de ce type. Ainsi, il en fera une constante à ce moment et cette valeur constante sera alors utilisée dans toutes les invocations de logfunc1 pendant le temps que durera la session. Il va sans dire que ce n'est pas ce que le programmeur voulait.

Dans le cas de logfunc2, l'analyseur principal de PostgreSQL ne sait pas quel type 'now' doit devenir, et par conséquent, il renvoie une valeur de type text contenant la chaîne now. Durant l'assignation consécutive de la variable locale curtime, l'interpréteur PL/pgSQL transtype cette chaîne en type timestamp en appelant les fonctions text_out et timestamp_in pour la conversion. Ainsi, l'horodateur est mis à jour à chaque exécution comme l'attend le programmeur.

La nature modifiable des variables record présente un problème lors de cette connexion. Quand les champs d'une variable record sont utilisés dans les expressions ou les instructions, les types de données des champs ne doivent pas changer entre les appels de deux expressions identiques, puisque l'expression sera planifiée en utilisant le type de données présent quand l'expression est atteinte pour la première fois. Gardez ceci à l'esprit quand vous écrivez des procédures déclencheurs qui gèrent des évènements pour plus d'une table. (EXECUTE peut être utilisé pour contourner le problème si nécessaire).

# 35.6. Instructions de base

Dans cette section ainsi que les suivantes, nous décrirons tous les types d'instructions explicitement compris par PL/pgSQL. Tout ce qui n'est pas reconnu comme l'un de ces types d'instruction est présumé être une commande SQL et est envoyé au moteur principal de bases de données pour être exécutée (après substitution de chaque variable PL/pgSQL utilisée dans l'instruction). Ainsi, par exemple, les commandes SQL INSERT, UPDATE, et DELETE peuvent être considérées comme des instructions de PL/pgSQL, mais ne sont pas spécifiquement listées ici.

## 35.6.1. Assignation

L'assignation d'une valeur à une variable ou à un champ row/record est écrite ainsi :

```
identifiant := expression;
```

Comme expliqué plus haut, l'expression dans un telle instruction est évaluée au moyen de la commande SQL SELECT envoyée au moteur principal de bases de données. L'expression ne doit manier qu'une seule valeur.

Si le type de données du résultat de l'expression ne correspond pas au type de donnée de la variable, ou que la variable a une taille ou une précision (comme char(20)), la valeur résultat sera implicitement convertie par l'interpréteur PL/pgSQL en utilisant la fonction d'écriture (output−function) du type du résultat, et la fonction d'entrée (input−function) du type de la variable. Notez que cela pourrait potentiellement conduire des erreurs d'exécution générées par la fonction d'entrée, si la forme de la chaîne de la valeur résultat n'est pas acceptable par la fonction d'entrée.

Exemples :

```
user_id := 20;
tax := subtotal * 0.06;
```

## 35.6.2. SELECT INTO

Le résultat d'une commande SELECT manipulant plusieurs colonnes (mais une seule ligne) peut être assignée à une variable de type record ou ligne, ou une liste de valeurs scalaires. Ceci est fait via :

```
SELECT INTO cible expressions FROM ...;
```

où *cible* peut être une variable record, une variable ligne, ou une liste, séparées de virgules, de simples variables de champs record/ligne. L'expression *select_expressions* et le reste de la commande sont identiques à du SQL standard.

Notez que cela est assez différent de l'interprétation normale par PostgreSQL de SELECT INTO, où la cible de INTO est une table nouvellement créée. Si vous voulez créer une table à partir du résultat d'un SELECT d'une fonction PL/pgSQL, utilisez la syntaxe CREATE TABLE ... AS SELECT.

Si une ligne ou une liste de variable est utilisée comme cible, les valeurs sélectionnées doivent correspondre exactement à la structure de la cible, ou une erreur d'exécution se produira. Quand une variable record est la cible, elle se configure seule automatiquement au type ligne formé par les colonnes résultant de la requête.

À l'exception de la clause INTO, l'instruction SELECT est identique à la commande SQL SELECT normale et peut en utiliser toute la puissance.

La clause INTO peut apparaître pratiquement partout dans l'instruction SELECT. De façon personnalisé, il est écrit soit juste après SELECT comme indiqué ci−dessus soit juste avant FROM — c'est−à−dire soit juste avant soit juste après la la liste de *select_expressions*.

Si la requête ne renvoie aucune ligne, des valeurs NULL sont assignées au(x) cibles(s). Si la requête renvoie plusieurs lignes, la première ligne est assignées au(x) cible(s) et le reste est rejeté. (Notez que << la première ligne >> n'est pas correctement définie à moins d'utiliser ORDER BY.)

Actuellement, la clause INTO peut apparaître presque n'importe où dans l'instruction SELECT, mais il est recommandé de la placer immédiatement après le mot clé SELECT comme décrit plus haut. Les version futures de PL/pgSQL pourront être moins laxistes sur le placement de la clause INTO.

Vous pouvez vérifier la variable spéciale FOUND (voir Section 35.6.6) après une instruction SELECT INTO pour déterminer si l'affectation est réussie, c'est−à−dire si au moins une ligne a été renvoyée par la requête. Par exemple :

```
SELECT INTO myrec * FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

Pour tester si un résultat record/ligne est NULL, vous pouvez utiliser la conditionnelle IS NULL. Il n'y a cependant aucun moyen de dire si une ou plusieurs lignes additionnelles ont été rejetées. Voici un exemple qui traite le cas où aucune ligne n'a été renvoyée.

```
DECLARE
    users_rec RECORD;
BEGIN
    SELECT INTO users_rec * FROM users WHERE user_id=3;
```

```
    IF users_rec.homepage IS NULL THEN
        -- l'utilisateur n'a entré aucune page, renvoyer "http://"
        RETURN 'http://';
    END IF;
END;
```

## 35.6.3. Exécuter une Expression ou Requête Sans Résultat

Quelquefois l'on souhaite évaluer une expression ou une requête mais rejeter le résultat (généralement parce que l'on appelle une fonction qui a des effets de bords utiles mais pas de résultat utile). Pour ce faire dans PL/pgSQL, utilisez l'instruction `PERFORM` :

```
PERFORM requête;
```

Ceci exécute *`requête`* et annule le résultat. Écrivez *`requête`* de la même façon que vous le feriez dans une commande SQL `SELECT` mais remplacez le mot clé initial `SELECT` par `PERFORM`. Les variables PL/pgSQL seront substituées dans la requête comme d'habitude. Par ailleurs, la variable spéciale `FOUND` est positionnée à true si la requête produit au moins une ligne ou false si elle n'en produit aucune.

> **Note :** On pourrait s'attendre à ce qu'un `SELECT` sans clause `INTO` aboutisse à ce résultat, mais en réalité la seule façon acceptée de faire cela est `PERFORM`.

Un exemple :

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

## 35.6.4. Ne rien faire du tout

Quelque fois, une instruction qui ne fait rien est utile. Par exemple, il indique qu'une partie de la chaîne if/then/else est délibérément vide. Pour cela, utilisez l'instruction :

```
NULL;
```

Par exemple, les deux fragments de code suivants sont équivalents :

```
BEGIN
y := x / 0;
EXCEPTION
WHEN division_by_zero THEN
NULL;  -- ignore l'erreur
END;
```

```
BEGIN
y := x / 0;
EXCEPTION
WHEN division_by_zero THEN  -- ignore l'erreur
END;
```

Ce qui est préférable est une histoire de goût.

**Note :** Dans Oracle's PL/SQL, les listes d'instructions vides ne sont pas autorisées et, du coup, les instructions `NULL` sont *requises* dans les situations telles que celles−ci. PL/pgSQL vous permet d'écrire simplement rien.

## 35.6.5. Exécuter des Commandes Dynamiques

Souvent vous voudrez générer des commandes dynamiques dans vos fonctions PL/pgSQL, c'est à dire, des commandes qui impliquent différentes tables ou différents types de données à chaque fois qu'elles sont exécutées. Les tentatives normales de PL/pgSQL pour garder en cache les planification des commandes ne fonctionneront pas dans de tels scénarios. Pour gérer ce type de problème, l'instruction `EXECUTE` est fournie :

```
EXECUTE chaîne-commande;
```

où *`chaîne-commande`* est une expression manipulant une chaîne (de type `text`) contenant la commande à être exécutée. Cette chaîne est littéralement donnée à manger au moteur SQL.

Notez en particulier qu'aucune substitution de variable PL/pgSQL n'est faite sur la chaîne−commande. Les valeurs des variables doivent être insérées dans la chaîne de commande lors de sa construction.

A la différence de toutes les autres commandes dans PL/pgSQL, une commande lancée par une instruction `EXECUTE` n'est pas préparée ni sauvée une seule fois pendant la durée de la session. À la place, la commande est préparée à chaque fois que l'instruction est lancée. La chaîne commande peut être dynamiquement créée à l'intérieur de la fonction pour agir sur des tables ou colonnes différentes.

Les résultats des commandes `SELECT` sont rejetés par `EXECUTE`, et `SELECT INTO` n'est pas actuellement géré à l'intérieur d'une instruction `EXECUTE`. Donc, il n'existe aucune façon d'extraire un résultat à partir d'un `SELECT` créé dynamiquement en utilisant la commande `EXECUTE` standard. Néanmoins, il existe deux autres façons de le faire : la première est d'utiliser `FOR−IN−EXECUTE` décrite dans Section 35.7.4, et la deuxième est d'utiliser un curseur avec `OPEN−FOR−EXECUTE`, comme décrit dans Section 35.8.2.

En travaillant avec des commandes dynamiques, vous aurez souvent à gérer des échappements de guillemets simples. La méthode recommandée pour mettre entre guillemets un texte fixe dans le corps de votre fonction est d'utiliser les guillemets dollar. (Si votre code n'utilise pas les guillemets dollar, référez−vous à l'aperçu dans Section 35.2.1, ce qui peut vous faire gagner des efforts lors du passage de ce code à un schéma plus raisonnable.)

Les valeurs dynamiques qui sont à insérer dans la requête construite requièrent une gestion spéciale car elles pourraient elles−même contenir des guillemets. Un exemple (ceci suppose que vous utilisez les guillemets dollar pour la fonction dans sa globalité, du coup les guillemets n'ont pas besoin d'être doublés) :

```
EXECUTE 'UPDATE tbl SET '
    || quote_ident(colname)
    || ' = '
    || quote_literal(newvalue)
    || ' WHERE key = '
    || quote_literal(keyvalue);
```

Cet exemple montre l'utilisation des fonctions `quote_ident(text)` et `quote_literal(text)`. Pour plus de sûreté, les variables contenant les identifiants des colonnes et des tables doivent être passés à la fonction `quote_ident`. Les variables contenant les valeurs devant être des chaînes dans la commande construite devraient être passées à `quote_literal`. Les deux font les étapes appropriées pour renvoyer le

texte en entrée entouré par des guillemets doubles ou simples respectivement, avec tout caractère intégré spécial proprement échappé.

Notez que les guillemets dollar sont souvent utiles pour placer un texte fixe entre guillemets. Il serait une très mauvaise idée d'essayer de faire l'exemple ci−dessus de cette façon

```
EXECUTE 'UPDATE tbl SET '
|| quote_ident(colname)
|| ' = $$'
|| newvalue
|| '$$ WHERE key = '
|| quote_literal(keyvalue);
```

car cela casserait si le contenu de `newvalue` pouvait contenir `$$`. La même objection s'appliquerait à tout délimiteur dollar que vous pourriez choisir. Donc, pour mettre un texte inconnu entre guillemets de façon sûr, vous *devez* utiliser `quote_literal`.

Un exemple bien plus important d'une commande dynamique et d'`EXECUTE` est disponible dans Exemple 35−6, qui construit et exécute une commande `CREATE FUNCTION` pour définir une nouvelle fonction.

## 35.6.6. Obtention du Statut du Résultat

Il y a plusieurs moyen de déterminer l'effet d'une commande. La première méthode est d'utiliser `GET DIAGNOSTICS`, qui a la forme suivante :

```
GET DIAGNOSTICS variable = item [ , ... ] ;
```

Cette commande permet la récupération des indicateurs de l'état du système. Chaque *`item`* est un mot clé identifiant une valeur d'état devant être assignée à la variable indiquée (qui devrait être du bon type de donnée pour pouvoir la recevoir.) Les items d'état actuellement disponibles sont `ROW_COUNT`, le nombre de lignes traitées par la dernière commande SQL envoyée au moteur SQL, et `RESULT_OID`, l'OID de la dernière ligne insérée par la commande SQL la plus récente. Notez que `RESULT_OID` n'est utile qu'après une commande `INSERT`.

Exemple :

```
GET DIAGNOSTICS var_integer = ROW_COUNT;
```

La seconde méthode pour déterminer les effets d'une commande est la variable spéciale nommée `FOUND` de type `boolean`. `FOUND` commence par être false dans chaque fonction PL/pgSQL. Elle est positionnée par chacune des types d'instructions suivants.

- Une instruction `SELECT INTO` positionne `FOUND` à true si elle renvoie une ligne, false si aucune ligne n'est renvoyée.
- Une instruction `PERFORM` positionne `FOUND` à true si elle produit (rejette) une ligne, faux si aucune ligne n'est produite.
- Les instructions `UPDATE`, `INSERT`, et `DELETE` positionnent `FOUND` à true si au moins une ligne est affectée, false si aucune ligne n'est affectée.
- Une instruction `FETCH` positionne `FOUND` à true si elle renvoie une ligne, false si aucune ligne n'est renvoyée.

- La commande FOR positionne FOUND à true si elle effectue une itération une ou plusieurs fois, sinon elle renvoie false. Ceci s'applique aux trois variantes de l'instruction FOR (boucles FOR integer, FOR record−set, et FOR record−set dynamique). FOUND n'est positionné de cette façon que quand la boucle FOR s'achève ; dans l'exécution de la chaîne, FOUND n'est pas modifiée par l'instruction FOR, bien qu'il puisse être modifié par l'exécution d'autres instructions situées dans le corps de la boucle.

FOUND est une variable locale à l'intérieur de chaque fonction PL/pgSQL ; chaque changement qui y est fait n'affecte que la fonction courante.

# 35.7. Structures de Contrôle

Les structures de contrôle sont probablement la partie la plus utile (et importante) de PL/pgSQL. Grâce aux structures de contrôle de PL/pgSQL, vous pouvez manipuler les données PostgreSQL de façons très flexible et puissante.

## 35.7.1. Retour d'une Fonction

Il y a deux commandes disponibles qui vous permettent de renvoyer des données d'une fonction : RETURN et RETURN NEXT.

### 35.7.1.1. RETURN

```
RETURN expression;
```

RETURN accompagné d'une expression termine la fonction et renvoie le valeur d'*expression* à l'appelant. Cette forme est à utiliser avec des fonctions PL/pgSQL qui ne renvoient pas d'ensemble de valeurs.

Lorsqu'elle renvoie un type scalaire, n'importe quelle expression peut être utilisée. Le résultat de l'expression sera automatiquement transtypé vers le type de retour de la fonction, comme décrit pour les assignations. Pour renvoyer une valeur composite (ligne), vous devez écrire une variable record ou ligne comme *expression*.

La valeur de retour d'une fonction ne peut pas être laissée indéfinie. Si le contrôle atteint la fin du bloc de premier niveau sans avoir rencontré d'instruction RETURN une erreur d'exécution sera lancée.

Notez que si vous avez déclaré la fonction comme renvoyant void, une instruction RETURN doit être quand même fournie ; l'expression suivant la commande RETURN est cependant optionnelle et sera ignorée dans tous les cas.

### 35.7.1.2. RETURN NEXT

```
RETURN NEXT expression;
```

Lorsqu'une fonction PL/pgSQL est déclarée renvoyer SETOF *type quelconque*, la procédure à suivre est légèrement différente. Dans ce cas, les items individuels à renvoyer sont spécifiés dans les commandes RETURN NEXT, et ensuite une commande RETURN finale, sans arguments est utilisée pour indiquer que la fonction a terminé son exécution. RETURN NEXT peut être utilisé avec des types scalaires et des types composites de données; dans ce dernier cas, une << table >> entière de résultats sera renvoyée.

Les fonctions qui utilisent RETURN NEXT devraient être appelées d'après le modèle suivant :

```
SELECT * FROM some_func();
```

En fait, la fonction doit être utilisée comme table source dans une clause FROM

RETURN NEXT n'effectue pas vraiment de renvoi; il sauvegarde simplement les valeurs des expressions. L'exécution continue alors avec la prochaine instruction dans la fonction PL/pgSQL. Lorsque des commandes RETURN NEXT successives sont renvoyées, l'ensemble des résultats est élaboré. Un RETURN final, qui ne devrait pas avoir d'argument, provoque la sortie du contrôle de la fonction.

> **Note :** L'implémentation actuelle de RETURN NEXT pour PL/pgSQL emmagasine la totalité de l'ensemble des résultats avant d'effectuer le retour de la fonction, comme vu plus haut. Cela signifie que si une fonction PL/pgSQL produit une structure résultat très grande, les performances peuvent être faibles: les données seront écrites sur le disque pour éviter un épuisement de la mémoire, mais la fonction en elle−même ne renverra rien jusqu'à ce que l'ensemble des résultats entier soit généré. Une version future de PL/pgSQL pourra permettre aux utilisateurs de définir des fonctions renvoyant des ensembles qui n'auront pas cette limitation. Actuellement le point auquel les données commencent à être écrites sur le disque est contrôlé par la variable de configuration work_mem. Les administrateurs ayant une mémoire suffisante pour enregistrer des ensembles de résultats plus importants en mémoire devraient envisager l'augmentation de ce paramètre.

## 35.7.2. Contrôles Conditionnels

Les instructions IF vous permettent d'exécuter des commandes basées sur certaines conditions. PL/pgSQL a cinq formes de IF:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF
- IF ... THEN ... ELSIF ... THEN ... ELSE
- IF ... THEN ... ELSEIF ... THEN ... ELSE

### 35.7.2.1. IF−THEN

```
IF expression-booleenne THEN
    instructions
END IF;
```

Les instructions IF−THEN sont la forme la plus simple de IF. Les instructions entre THEN et END IF seront exécutées si la condition est true. Autrement, ils seront négligés.

Exemple :

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

#### 35.7.2.2. IF-THEN-ELSE

```
IF expression-booleenne THEN
    instructions
ELSE
    instructions
END IF;
```

Les instructions IF-THEN-ELSE s'ajoutent au IF-THEN en vous permettant de spécifier un ensemble d'instructions alternatif à exécuter si la condition est évaluée à false.

Exemples :

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;

IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

---

#### 35.7.2.3. IF-THEN-ELSE IF

Les instructions IF peuvent être imbriquées, comme dans l'exemple suivant :

```
IF demo_row.sex = 'm' THEN
    pretty_sex := ''man'';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

Lorsque vous utilisez cette forme, vous imbriquez une instruction IF dans la partie ELSE d'une instruction IF extérieure. Ainsi vous avez besoin d'une instruction END IF pour chaque IF imbriqué et une pour le IF-ELSE parent. Ceci fonctionne mais devient fastidieux quand il y a de nombreuses alternatives à traiter. Considérez alors la forme suivante.

---

#### 35.7.2.4. IF-THEN-ELSIF-ELSE

```
IF expression-booleenne THEN
    instructions
[ ELSIF expression-booleenne THEN
    instructions
[ ELSIF expression-booleenne THEN
    instructions
    ...]]
[ ELSE
    instructions ]
END IF;
```

IF-THEN-ELSIF-ELSE fournit une méthode plus pratique pour vérifier de nombreuses alternatives en une instruction. Elle est équivalente formellement aux commandes IF-THEN-ELSE-IF-THEN imbriquées, mais un seul END IF est nécessaire.

Voici un exemple :

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positif';
ELSIF number < 0 THEN
    result := 'negatif';
ELSE
    -- hmm, la seule possibilité est que le nombre soit NULL
    result := 'NULL';
END IF;
```

#### 35.7.2.5. IF-THEN-ELSEIF-ELSE

ELSEIF est un alias pour ELSIF.

## 35.7.3. Boucles Simples

Grâce aux instructions LOOP, EXIT, WHILE et FOR vous pouvez faire en sorte que vos fonctions PL/pgSQL répètent une série de commandes.

#### 35.7.3.1. LOOP

```
[<<label>>]
LOOP
    instructions
END LOOP;
```

LOOP définit une boucle inconditionnelle répétée indéfiniment jusqu'à ce qu'elle soit terminée par une instruction EXIT ou RETURN. Le label optionnel peut être utilisé par les instructions EXIT dans le cas de boucles imbriquées pour définir quel niveau d'imbrication doit s'achever.

#### 35.7.3.2. EXIT

```
EXIT [ label ] [ WHEN expression ];
```

Si aucun *label* n'est donné la boucle la plus imbriquée se termine et l'instruction suivant END LOOP est exécutée ensuite. Si un *label* est donné, ce doit être le label de la boucle ou du bloc courant ou d'un niveau moins imbriqué. La boucle ou le bloc nommé se termine alors et le contrôle continue avec l'instruction située après le END de la boucle ou du bloc correspondant.

Si WHEN est présent, la sortie de boucle ne s'effectue que si les conditions spécifiées sont true, autrement le contrôle passe à l'instruction suivant le EXIT.

EXIT peut être utilisé pour causer un départ rapide de tout type de boucles ; il n'est pas limité en utilisation aux boucles sans condition.

Exemples :

```
LOOP
    -- quelques traitements
    IF count > 0 THEN
        EXIT;  -- sortie de boucle
    END IF;
END LOOP;

LOOP
    -- quelques traitements
    EXIT WHEN count > 0;
END LOOP;

BEGIN
    -- quelques traitements
    IF stocks > 100000 THEN
        EXIT;  -- cause la sortie (EXIT) du bloc BEGIN
    END IF;
END;
```

#### 35.7.3.3. WHILE

```
[<<label>>]
WHILE expression LOOP
    instructions
END LOOP;
```

L'instruction WHILE répète une séquence d'instructions aussi longtemps que l'expression conditionnelle est évaluée à vrai. La condition est vérifée juste avant chaque entrée dans le corps de la boucle.

Par exemple :

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- quelques traitements ici
END LOOP;

WHILE NOT boolean_expression LOOP
    -- quelques traitements ici
END LOOP;
```

#### 35.7.3.4. FOR (variante avec entier)

```
[<<label>>]
FOR nom IN [ REVERSE ] expression .. expression LOOP
    instruction
END LOOP;
```

Cette forme de FOR crée une boucle qui effectue une itération sur une plage de valeurs entières. La variable *nom* est automatiquement définie comme un type integer et n'existe que dans la boucle. Les deux expressions donnant les limites inférieures et supérieures de la plage sont évaluées une fois en entrant dans la boucle. Le pas de l'itération est normalement de 1, mais est −1 quand REVERSE est spécifié.

Quelques exemples de boucles FOR avec entiers :

```
FOR i IN 1..10 LOOP
    -- quelques calculs ici
```

```
    RAISE NOTICE 'i is %', i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- quelques calculs ici
END LOOP;
```

Si la limite basse est plus grande que la limite haute (ou moins grande que, dans le cas du `REVERSE` case), le corps de la boucle n'est pas exécuté du tout. Aucune erreur n'est renvoyée.

## 35.7.4. Boucler Dans les Résultats de Requêtes

En utilisant un type de `FOR` différent, vous pouvez itérer au travers des résultats d'une requête et par là−même manipuler ces données. La syntaxe est la suivante :

```
[<<label>>]
FOR record_ou_ligne IN requête LOOP
    instructions
END LOOP;
```

La variable record ou ligne est successivement assignée à chaque ligne résultant de la *`requête`* (qui doit être une commande `SELECT`) et le corps de la boucle est exécuté pour chaque ligne. Voici un exemple :

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    PERFORM cs_log('Refreshing materialized views...');

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- A présent "mviews" contient un enregistrement de cs_materialized_views

        PERFORM cs_log('Refreshing materialized view ' || quote_ident(mviews.mv_name) || '...');
        EXECUTE 'TRUNCATE TABLE  ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO ' || quote_ident(mviews.mv_name) || ' ' || mviews.mv_query;
    END LOOP;

    PERFORM cs_log('Done refreshing materialized views.');
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Si la boucle est terminée par une instruction `EXIT`, la dernière valeur ligne assignée est toujours accessible après la boucle.

L'instruction `FOR−IN−EXECUTE` est un moyen d'itérer sur des lignes :

```
[<<label>>]
FOR record_ou_ligne IN EXECUTE expression_texte LOOP
    instructions
END LOOP;
```

Ceci est identique à la forme précédente, à ceci près que l'expression `SELECT` source est spécifiée comme une expression chaîne, évaluée et replanifiée à chaque entrée dans la boucle `FOR`. Ceci permet au programmeur de choisir la vitesse d'une requête préplanifiée, ou la flexibilité d'une requête dynamique, uniquement avec la

simple instruction EXECUTE.

> **Note :** L'analyseur PL/pgSQL distingue actuellement deux types de boucles FOR (entier ou résultat d'une requête) en vérifiant si . . apparaît à l'extérieur des parenthèses entre IN et LOOP. Si . . n'est pas trouvé, la boucle est supposée être une boucle entre des lignes. Une mauvaise saisie de . . amènera donc une plainte du type << loop variable of loop over rows must be a record or row variable >> (NdT : une variable de boucle d'une boucle sur des enregistrement doit être un enregistrement ou une variable de type ligne) plutôt qu'une simple erreur de syntaxe comme vous pourriez vous y attendre.

## 35.7.5. Récupérer les erreurs

Par défaut, toute erreur survenant dans une fonction PL/pgSQL annule l'exécution de la fonction et, en fait, aussi de la transaction qui l'entoure. Vous pouvez récupérer les erreurs et les surpasser en utilisant un bloc BEGIN avec une clause EXCEPTION. La syntaxe est une extension de la syntaxe habituelle pour un bloc BEGIN :

```
[ <<label>> ]
[ DECLARE
  declarations ]
BEGIN
statements
EXCEPTION
WHEN condition [ OR condition ... ] THEN
handler_statements
[ WHEN condition [ OR condition ... ] THEN
  instructions_gestionnaire
  ... ]
END;
```

Si aucune erreur ne survient, cette forme de bloc exécute simplement toutes les *instructions* puis passent le contrôle à l'instruction suivant END. Mais si une erreur survient à l'intérieur des *instructions*, le traitement en cours des *instructions* est abandonné et le contrôle est passé à la liste d'EXCEPTION. Une recherche est effectuée sur la liste pour la première *condition* correspondant à l'erreur survenue. Si une correspondance est trouvée, les *instructions_gestionnaire* correspondantes sont exécutées puis le contrôle est passé à l'instruction suivant le END. Si aucune correspondance n'est trouvée, l'erreur se propage comme si la clause EXCEPTION n'existait pas du tout : l'erreur peut être récupérée par un bloc l'enfermant avec EXCEPTION ou, s'il n'existe pas, il annule le traitement de la fonction.

Les noms des *condition* peuvent être n'importe laquelle parmi celles listées dans l'Annexe A. Un nom de catégorie correspond à toute erreur contenue dans cette catégorie. Le nom de condition spéciale OTHERS correspond à tout type d'erreur sauf QUERY_CANCELED. (Il est possible, mais pas recommandé, de récupérer QUERY_CANCELED par son nom.) Les noms des conditions ne sont pas sensibles à la casse.

Si une nouvelle erreur survient à l'intérieur des *instructions_gestionnaire* sélectionnées, elle ne peut pas être récupérée par cette clause EXCEPTION mais est propagée en dehors. Une clause EXCEPTION l'englobant pourrait la récupérer.

Quand une erreur est récupérée par une clause EXCEPTION, les variables locales de la fonction PL/pgSQL reste comme elles étaient au moment où l'erreur est survenue mais toutes les modifications à l'état persistent de la base de données à l'intérieur du bloc sont annulées. Comme exemple, considérez ce fragment :

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
```

```
BEGIN
  UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
  x := x + 1;
  y := x / 0;
  EXCEPTION
    WHEN division_by_zero THEN
      RAISE NOTICE 'caught division_by_zero';
RETURN x;
END;
```

Quand le contrôle parvient à l'affectation de `y`, il échouera avec une erreur `division_by_zero`. Elle sera récupérée par la clause `EXCEPTION`. La valeur renvoyée par l'instruction `RETURN` sera la valeur incrémentée de `x` mais les effets de la commande `UPDATE` auront été annulés. La commande `INSERT` précédant le bloc ne sera pas annulée, du coup le résultat finale est que la base de données contient `Tom Jones` et non pas `Joe Jones`.

> **Astuce :** Un bloc contenant une clause `EXCEPTION` est significativement plus coûteuse en entrée et en sortie qu'un bloc sans. Du coup, n'utilisez pas `EXCEPTION` sans besoin.

#### Exemple 35−1. Exceptions avec `UPDATE/INSERT`

Cet exemple utilise la gestion des exceptions pour réaliser soit un `UPDATE` soit un `INSERT` suivant le cas.

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;

        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- ne fait rien
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

## 35.8. Curseurs

Plutôt que d'exécuter la totalité d'une requête à la fois, il est possible de créer un *curseur* qui encapsule la requête, puis en lit le résultat quelques lignes à la fois. Une des raisons pour faire de la sorte est d'éviter les surcharges de mémoire quand le résultat contient un grand nombre de lignes. (Cependant, les utilisateurs PL/pgSQL n'ont généralement pas besoin de se préoccuper de cela, puisque les boucles `FOR` utilisent

automatiquement un curseur en interne pour éviter les problèmes de mémoire). Un usage plus intéressant est de renvoyer une référence à un curseur qu'elle a créé, permettant à l'appelant de lire les lignes. Ceci fournit un moyen efficace de renvoyer de grands ensembles de lignes à partir des fonctions.

## 35.8.1. Déclaration de Variables Curseur

Tous les accès aux curseurs dans PL/pgSQL se font par les variables curseur, qui sont toujours du type de données spécial `refcursor`. Un des moyens de créer une variable curseur est de simplement la déclarer comme une variable de type `refcursor`. Un autre moyen est d'utiliser la syntaxe de déclaration de curseur qui est en général :

```
nom CURSOR [ ( arguments ) ] FOR requête ;
```

(`FOR` peut être remplacé par `IS` pour la compatibilité avec Oracle.) *`arguments`*, si spécifié, est une liste de paires de *`nom type-de-donnée`* qui définit les noms devant être remplacés par les valeurs des paramètres dans la requête donnée. La valeur effective à substituer pour ces noms sera spécifiée plus tard, lors de l'ouverture du curseur.

Quelques exemples :

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

Ces variables sont toutes trois du type de données `refcursor`, mais la première peut être utilisées avec n'importe quelle requête, alors que la seconde a une requête complètement spécifiée qui lui est déjà *liée*, et la dernière est liée à une requête paramétrée. (`key` sera remplacée par un paramètre de valeur entière lors de l'ouverture du curseur.) La variable `curs1` est dite *non liée* puisqu'elle n'est pas liée a une requête particulière.

## 35.8.2. Ouverture De Curseurs

Avant qu'un curseur puisse être utilisé pour rapatrier des lignes, il doit être *ouvert*. (C'est l'action équivalente de la commande SQL `DECLARE CURSOR`.) PL/pgSQL a trois formes pour l'instruction `OPEN`, dont deux utilisent des variables curseur non liées et les autres utilisent une variable curseur liée.

### 35.8.2.1. `OPEN FOR SELECT`

```
OPEN curseur_non_lié FOR SELECT ...;
```

La variable curseur est ouverte et reçoit la requête spécifiée à exécuter. Le curseur ne peut pas être déjà ouvert, et il doit avoir été déclaré comme curseur non lié. (c'est à dire comme une simple variable `refcursor`). La requête `SELECT` est traitée de la même façon que les autres instructions`SELECT` dans PL/pgSQL : les noms de variables PL/pgSQL sont remplacés, et le plan de requête est mis en cache pour une possible réutilisation.

Exemple :

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

#### 35.8.2.2. `OPEN FOR EXECUTE`

```
OPEN curseur_non_lié FOR EXECUTE chaîne_requête;
```

La variable curseur est ouverte et reçoit la requête spécifiée à exécuter. Le curseur ne peut pas être déjà ouvert, et il doit avoir été déclaré comme curseur non–lié (c'est a dire comme une simple variable `refcursor`. La requête est spécifiée comme une expression chaîne de la même façon que dans une commande `EXECUTE`. Comme d'habitude, ceci donne assez de flexibilité pour que la requête puisse changer d'une exécution à l'autre.

Exemple :

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident($1);
```

#### 35.8.2.3. Ouverture d'un Curseur Lié

```
OPEN curseur_lié [ ( arguments ) ];
```

Cette forme d'`OPEN` est utilisée pour ouvrir une variable curseur à laquelle la requête est liée au moment de la déclaration. Le curseur ne peut pas être déjà ouvert. Une liste des expressions arguments doit apparaître si et seulement si le curseur a été déclaré comme acceptant des arguments. Ces valeurs seront remplacées dans la requête. Le plan de requête pour un curseur lié est toujours considéré comme pouvant être mis en cache; il n'y a pas d'équivalent de la commande `EXECUTE` dans ce cas.

Exemples :

```
OPEN curs2;
OPEN curs3(42);
```

### 35.8.3. Utilisation des Curseurs

Une fois qu'un curseur a été ouvert, il peut être manipulé grâce aux instructions décrites ci–dessous.

Ces manipulations n'ont pas besoin de se dérouler dans la même fonction que celle qui a ouvert le curseur. Vous pouvez renvoyer une valeur `refcursor` à partir d'une fonction et laisser l'appelant opérer sur le curseur. (D'un point de vue interne, une valeur `refcursor` est simplement la chaîne de caractères du nom d'un portail contenant la requête active pour le curseur. Ce nom peut être passé à d'autres, assigné à d'autres variables `refcursor` et ainsi de suite, sans déranger le portail.)

Tous les portails sont implicitement fermés à la fin de la transaction. C'est pourquoi une valeur `refcursor` est utilisable pour référencer un curseur ouvert seulement jusqu'à la fin de la transaction.

#### 35.8.3.1. `FETCH`

```
FETCH curseur INTO target;
```

`FETCH` rapatrie le rang suivant depuis le curseur dans une cible, qui peut être une variable ligne, une variable record, ou une liste de simples variables séparées d'une virgule, exactement comme `SELECT INTO`. Comme pour `SELECT INTO`, la variable spéciale `FOUND` peut être vérifiée pour voir si une ligne a été obtenue ou pas.

Exemple :

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
```

#### 35.8.3.2. CLOSE

```
CLOSE curseur;
```

`CLOSE` ferme le portail sous−tendant un curseur ouvert. Ceci peut être utilisé pour libérer des ressources avant la fin de la transaction, ou de libérer la variable curseur pour pouvoir la réouvrir.

Exemple :

```
CLOSE curs1;
```

#### 35.8.3.3. Renvoi de Curseurs

Les fonctions PL/pgSQL peuvent renvoyer des curseurs à l'appelant. Ceci est utile pour renvoyer plusieurs lignes ou colonnes, spécialement avec des ensembles de résultats très grands. Pour cela, la fonction ouvre le curseur et renvoie le nom du curseur à l'appelant (ou simplement ouvre le curseur en utilisant un nom de portail spécifié par ou autrement connu par l'appelant). L'appelant peut alors récupérer les lignes à partir du curseur. Le curseur peut être fermé par l'appelant ou il sera fermé automatiquement à la fin de la transaction.

Le nom du portail utilisé pour un curseur peut être spécifié par le développeur ou peut être généré automatiquement. Pour spécifier un nom de portail, affectez simplement une chaîne à la variable `refcursor` avant de l'ouvrir. La valeur de la chaîne de la variable `refcursor` sera utilisée par `OPEN` comme nom du portail sous−jacent. Néanmoins, si la variable `refcursor` est NULL, `OPEN` génère automatiquement un nom qui n'entre pas en conflit avec tout portail existant et l'affecte à la variable `refcursor`.

> **Note :** Une variable curseur avec limites est initialisée avec la valeur de la chaîne représentant son nom, de façon à ce que le nom du portail soit identique au nom de la variable curseur, sauf si le développeur le surcharge par affectation avant d'ouvrir le curseur. Mais, une variable curseur sans limite aura par défaut la valeur NULL, dont il recevra un nom unique généré automatiquement sauf s'il est surchargé.

L'exemple suivant montre une façon de fournir un nom de curseur par l'appelant :

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

L'exemple suivant utilise la génération automatique du nom du curseur :

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

L'exemple suivant montre une façon de renvoyer plusieurs curseurs à une seule fonction :

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
    RETURN;
END;
$$ LANGUAGE plpgsql;

-- doit être dans une transaction pour utiliser les curseurs.
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

## 35.9. Erreurs et Messages

Utilisez l'instruction `RAISE` pour rapporter des messages et lever des erreurs.

```
RAISE niveau 'format' [, variable [, ...]];
```

Les niveaux possibles sont `DEBUG` `LOG`, `INFO`, `NOTICE`, `WARNING` et `EXCEPTION`. `EXCEPTION` lève une erreur (ce qui annule habituellement la transaction en cours). Les autres niveaux ne font que générer des messages aux différents niveaux de priorité. Quand les messages d'une priorité particulière sont indiqués par le client, écrit dans les traces du serveur, ou à la fois contrôlé par les variables de configuration log_min_messages et client_min_messages. Voir Section 16.4 pour plus d'informations.

Au sein de la chaîne de formatage, `%` est remplacé par la représentation de la chaîne argument optionnelle suivante. Écrivez `%%` pour signifier un caractère `%`. Notez que les arguments optionnels doivent actuellement

être de simples variables, non des expressions et que le format doit être une simple chaîne de caractères.

Dans cet exemple, la valeur de v_job_id remplacera le % dans la chaîne.

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

Cet exemple interrompra la transaction avec le message d'erreur donné.

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id;
```

RAISE EXCEPTION génère toujours le même code SQLSTATE, P0001, quelque soit le message avec lequel il a été invoqué. Il est possible de récupérer cette exception avec EXCEPTION ... WHEN RAISE_EXCEPTION THEN ... mais il n'existe aucun moyen de savoir quelle RAISE.

## 35.10. Procédures Déclencheur

PL/pgSQL peut être utilisé pour définir des procédures déclencheur. Une procédure déclencheur est créée grâce à la commande CREATE FUNCTION utilisée comme fonction sans arguments ayant un type de retour trigger. Notez que la fonction doit être déclarée avec aucun argument même si elle s'attend à recevoir les arguments spécifiés dans CREATE TRIGGER — les arguments trigger sont passés via TG_ARGV, comme décrit plus loin.

Quand une fonction PL/pgSQL est appelée en tant que trigger, plusieurs variables spéciales sont créées automatiquement dans le bloc de plus haut niveau. Ce sont :

NEW
: Type de données RECORD; variable contenant la nouvelle ligne de base de données pour les opérations INSERT/UPDATE dans les déclencheurs de niveau ligne. Cette variable est NULL dans un trigger de niveau instruction.

OLD
: Type de données RECORD; variable contenant l'ancienne ligne de base de données pour les opérations UPDATE/DELETE dans les triggers de niveau ligne. Cette variable est NULL dans les triggers de niveau instruction.

TG_NAME
: Type de données name; variable qui contient le nom du trigger réellement lancé.

TG_WHEN
: Type de données text; une chaîne, soit BEFORE soit AFTER selon la définition du déclencheur.

TG_LEVEL
: Type de données text; une chaîne, soit ROW soit STATEMENT selon la définition du déclencheur.

TG_OP
: Type de données text; une chaîne, INSERT, UPDATE, ou DELETE indiquant pour quelle opération le déclencheur a été lancé.

TG_RELID
: Type de données oid; l'ID de l'objet de la table qui a causé l'invocation du trigger.

TG_RELNAME
: Type de données name; le nom de la table qui a causé l'invocation du trigger.

TG_NARGS
: Type de données integer; le nombre d'arguments donnés à la procédure déclencheur dans l'instruction CREATE TRIGGER.

TG_ARGV[]
: Type de donnée text; les arguments de l'instruction CREATE TRIGGER. L'index débute à 0. Les indices invalides (inférieurs à 0 ou supérieurs ou égaux à tg_nargs) auront une valeur NULL.

Une fonction déclencheur doit renvoyer soit NULL soit une valeur record/ligne ayant exactement la structure de la table pour laquelle le déclencheur a été lancé.

Les déclencheurs de niveau ligne lancés BEFORE peuvent renvoyer NULL pour indiquer au gestionnaire de déclencheur de sauter le reste de l'opération pour cette ligne (les déclencheurs suivants ne sont pas lancés, et les INSERT/UPDATE/DELETE ne se font pas pour cette ligne). Si une valeur non NULL est renvoyée alors l'opération se déroule avec cette valeur ligne. Renvoyer une valeur ligne différente de la valeur originale de NEW modifie la ligne qui sera insérée ou mise à jour (mais n'a pas d'effet sur le cas DELETE). Pour modifier la ligne à stocker, il est possible de remplacer des valeurs seules directement dans NEW et de renvoyer NEW, ou de construire un nouveau record/ligne à renvoyer.

La valeur de retour d'un déclencheur de niveau ligne BEFORE ou AFTER ou un déclencheur de niveau ligne AFTER est toujours ignoré ; il pourrait aussi bien être NULL. Néanmoins, tous les types de déclencheurs peuvent toujours annuler l'opération complète en envoyant une erreur.

Exemple 35–2 montre un exemple d'une procédure déclencheur dans PL/pgSQL.

**Exemple 35–2. Une Procédure Déclencheur PL/pgSQL**

Cet exemple de déclencheur assure qu'à chaque moment où une ligne est insérée ou mise à jour dans la table, le nom d'utilisateur courant et l'heure sont estampillés dans la ligne. Et cela assure qu'un nom d'employé est donné et que le salaire est une valeur positive.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Verifie que empname et salary sont donnés
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be NULL';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have NULL salary', NEW.empname;
        END IF;

        -- Qui travaille pour nous quand elle doit payer pour cela ?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Rappelons nous qui a changé le payroll quand
        NEW.last_date := 'now';
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Une autre façon de tracer les modifications sur une table implique la création d'une nouvelle table qui contient une ligne pour chaque insertion, mise à jour ou suppression qui survient. Cette approche peut être vue comme un audit des modifications sur une table. Exemple 35–3 montre un exemple d'une procédure d'audit par déclencheur en PL/pgSQL.

**Exemple 35–3. Une procédure d'audit par déclencheur en PL/pgSQL**

Cet exemple de déclencheur nous assure que toute insertion, modification ou suppression d'une ligne dans la table `emp` est enregistrée (c'est–à–dire auditée) dans la table `emp_audit`. L'heure et le nom de l'utilisateur sont conservées dans la ligne avec le type d'opération réalisé.

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
    --
    -- Ajoute une ligne dans emp_audit pour refléter l'opération réalisée
    -- sur emp,
    -- utilise la variable spéciale TG_OP pour cette opération.
    --
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
        RETURN NEW;
    END IF;
    RETURN NULL; -- result is ignored since this is an AFTER trigger
END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

Une utilisation des déclencheurs est le maintien d'une table résumé d'une autre table. Le résumé résultant peut être utilisé à la place de la table originale pour certaines requêtes — souvent avec des temps d'exécution bien réduits. Cette technique est souvent utilisée pour les statistiques de données où les tables de données mesurées ou observées (appelées des tables de faits) peuvent être extrêmement grandes. Exemple 35–4

montre un exemple d'une procédure déclencheur en PL/pgSQL maintenant une table résumée pour une table de faits dans un système de données (data warehouse).

**Exemple 35−4. Une procédure déclencheur PL/pgSQL pour maintenir une table résumée**

Le schéma détaillé ici est partiellement basé sur l'exemple du *Grocery Store* provenant de *The Data Warehouse Toolkit* par Ralph Kimball.

```
--
-- Tables principales - dimension du temps de ventes.
--
CREATE TABLE time_dimension (
  time_key                    integer NOT NULL,
  day_of_week                 integer NOT NULL,
  day_of_month                integer NOT NULL,
  month                       integer NOT NULL,
  quarter                     integer NOT NULL,
  year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
  time_key                    integer NOT NULL,
  product_key                 integer NOT NULL,
  store_key                   integer NOT NULL,
  amount_sold                 numeric(12,2) NOT NULL,
  units_sold                  integer NOT NULL,
  amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Table résumé - ventes sur le temps.
--
CREATE TABLE sales_summary_bytime (
  time_key                    integer NOT NULL,
  amount_sold                 numeric(15,2) NOT NULL,
  units_sold                  numeric(12) NOT NULL,
  amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Fonction et déclencheur pour amender les colonnes résumées
-- pour un UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER AS $maint_sales_summary_b
DECLARE
  delta_time_key          integer;
  delta_amount_sold       numeric(15,2);
  delta_units_sold        numeric(12);
  delta_amount_cost       numeric(15,2);
BEGIN

  -- Travaille sur l'ajout/la suppression de montant(s).
  IF (TG_OP = 'DELETE') THEN

    delta_time_key = OLD.time_key;
    delta_amount_sold = -1 * OLD.amount_sold;
```

```
  delta_units_sold = -1 * OLD.units_sold;
  delta_amount_cost = -1 * OLD.amount_cost;

ELSIF (TG_OP = 'UPDATE') THEN

  -- interdit les mises à jour qui modifient time_key -
  -- (probablement pas trop cher, car DELETE + INSERT est la façon la plus
  -- probable de réaliser les modifications).
  IF ( OLD.time_key != NEW.time_key) THEN
    RAISE EXCEPTION 'Update of time_key : % -> % not allowed', OLD.time_key, NEW.time_key;
  END IF;

  delta_time_key = OLD.time_key;
  delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
  delta_units_sold = NEW.units_sold - OLD.units_sold;
  delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

ELSIF (TG_OP = 'INSERT') THEN

  delta_time_key = NEW.time_key;
  delta_amount_sold = NEW.amount_sold;
  delta_units_sold = NEW.units_sold;
  delta_amount_cost = NEW.amount_cost;

END IF;


-- Mise à jour de la ligne de résumé avec les nouvelles valeurs.
UPDATE sales_summary_bytime
SET amount_sold = amount_sold + delta_amount_sold,
  units_sold = units_sold + delta_units_sold,
  amount_cost = amount_cost + delta_amount_cost
  WHERE time_key = delta_time_key;


-- Il pourrait n'y avoir aucune ligne pour ce time_key
-- (donc nouvelle donnée!).
IF (NOT FOUND) THEN
  BEGIN
    INSERT INTO sales_summary_bytime (
      time_key,
      amount_sold,
      units_sold,
      amount_cost)
      VALUES (
      delta_time_key,
      delta_amount_sold,
      delta_units_sold,
      delta_amount_cost
      );
  EXCEPTION
    --
    -- Récupère le cas où deux transactions ajoutent des données pour un
    -- nouveau time_key.
    --
    WHEN UNIQUE_VIOLATION THEN
      UPDATE sales_summary_bytime
        SET amount_sold = amount_sold + delta_amount_sold,
        units_sold = units_sold + delta_units_sold,
        amount_cost = amount_cost + delta_amount_cost
        WHERE time_key = delta_time_key;
```

```
        END;
  END IF;
  RETURN NULL;

END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
  AFTER INSERT OR UPDATE OR DELETE ON sales_fact
  FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();
```

# 35.11. Portage d'Oracle PL/SQL

Cette section explicite les différences entre le PL/pgSQL de PostgreSQL et le langage PL/SQL d'Oracle, afin d'aider les développeurs qui portent des applications d'Oracle® vers PostgreSQL.

PL/pgSQL est similaire à PL/SQL sur de nombreux aspects. C'est un langage impératif structuré en blocs et toutes les variables doivent être déclarées. Les assignations, boucles, conditionnelles sont similaires. Les principales différences que vous devez garder à l'esprit quand vous portez de PL/SQL vers PL/pgSQL sont:

- Il n'y a pas de valeurs par défaut pour les paramètres dans PostgreSQL.
- Vous pouvez surcharger les fonctions dans PostgreSQL. C'est souvent utilisé pour contourner le manque de paramètres par défaut.
- Pas besoin de curseurs dans PL/pgSQL, mettez juste la requête dans l'instruction `FOR`. (Voir Exemple 35−6.)
- Dans PostgreSQL vous avez besoin d'échapper les guillemets dollar ou les guillemets simples dans le corps des fonctions. Voir Section 35.2.1.
- À la place des packages, utilisez des schémas pour organiser vos fonctions en groupes.
- Comme il n'y a pas de paquetages, il n'y a pas non plus de variables au niveau paquetage. Ceci est un peu ennuyant. Vous pourriez être capable de conserver un état par session dans les tables temporaires, à la place.

## 35.11.1. Exemples de Portages

Exemple 35−5 montre comment porter une simple fonction de PL/SQL vers PL/pgSQL.

**Exemple 35−5. Portage d'une Fonction Simple de PL/SQL vers PL/pgSQL**

Voici une fonction en PL/SQL Oracle :

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name IN varchar, v_version IN varchar)
RETURN varchar IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Parcourons cette fonction et voyons les différences avec PL/pgSQL :

- Oracle peut avoir des paramètres `IN`, `OUT`, et `INOUT` passées aux fonctions. `INOUT`, par exemple, signifie que le paramètre recevra une valeur et en renverra une autre. PostgreSQL n'a que les paramètres `IN` et, du coup, il n'y a pas de spécification pour le paramètre kind.
- Le mot clé `RETURN` dans le prototype de la fonction (pas dans le corps de la fonction) devient `RETURNS` dans PostgreSQL. De plus, `IS` devient `AS` et vous avez besoin d'ajouter une clause `LANGUAGE` parce que PL/pgSQL n'est pas le seul langage de procédures disponible.
- Dans PostgreSQL, le corps de la fonction est considéré comme une chaîne littérale, donc vous avez besoin d'utiliser les guillemets simples ou les guillemets dollar tout autour. Ceci se substitue au `/` de fin dans l'approche d'Oracle.
- La commande `show errors` n'existe pas dans PostgreSQL et n'est pas nécessaire car les erreurs sont rapportées automatiquement.

Voici de quoi aurait l'air cette fonction portée sous PostgreSQL :

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar, v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        return v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Exemple 35−6 montre comment porter une fonction qui crée une autre fonction et comment gérer les problèmes de quotes résultants.

**Exemple 35−6. Portage d'une Fonction qui Crée une Autre Fonction de PL/SQL vers PL/pgSQL**

La procédure suivante récupère des lignes d'une instruction `SELECT` et construit une grande fonction dont les résultats sont dans une instruction `IF` pour favoriser l'efficacité. Notez particulièrement les différences dans le curseur et la boucle `FOR`.

Voici la version Oracle :

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;

    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR,
        v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
        ' IF v_' || referrer_key.kind
        || ' LIKE ''' || referrer_key.key_string
        || ''' THEN RETURN ''' || referrer_key.referrer_type
        || '''; END IF;';
    END LOOP;
```

```
    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

Voici comment la fonction serait dans PostgreSQL :

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_key RECORD;  -- declare a generic record to be used in a FOR
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN' ;

    -- Remarquez comment nous parcourons les résultats d'une requête dans une boucle FOR
    -- en utilisant la construction FOR <record>.

    FOR referrer_key IN SELECT * FROM cs_referrer_keys ORDER BY try_order LOOP
        func_body := func_body ||
        ' IF v_' || referrer_key.kind
        || ' LIKE ' || quote_literal(referrer_key.key_string)
        || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
        || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
        'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
        v_domain varchar,
        v_url varchar)
        RETURNS varchar AS '
        || quote_literal(func_body)
        || ' LANGUAGE plpgsql;' ;

        EXECUTE func_cmd;
        RETURN;
END;
$func$ LANGUAGE plpgsql;
```

Notez comment le corps de la fonction est construit séparément et passé au travers de `quote_literal` pour doubler tout symbole guillemet qu'il peut contenir. Cette technique est nécessaire parce que nous ne pouvons pas utiliser à coup sûr les guillemets dollar pour définir la nouvelle fonction : nous ne sommes pas sûr de savoir quelle chaîne sera interpolée ) partir du champ `referrer_key.key_string`. (Nous supposons ici que ce `referrer_key.kind` vaut à coup sûr `host`, `domain` ou `url` mais `referrer_key.key_string` pourrait valoir autre chose, il pourrait contenir en particulier des signes dollar.) Cette fonction est en fait une amélioration de l'original Oracle parce qu'il ne générera pas de code cassé quand `referrer_key.key_string` ou `referrer_key.referrer_type` contient des guillemets.

Exemple 35–7 montre comment porter une fonction ayant des paramètres `OUT` et effectuant des manipulations de chaînes.  PostgreSQL n'a pas de fonction `instr`, mais vous pouvez contourner cela en utilisant une combinaison d'autres fonctions. Dans Section 35.11.3 il y a une implémentation PL/pgSQL d'`instr` que vous pouvez utiliser pour faciliter votre portage.

**Exemple 35−7. Portage d'une Procédure avec Manipulation de Chaînes et Paramètres `OUT` de PL/SQL vers PL/pgSQL**

La procédure Oracle suivante est utilisées pour analyser une URL et renvoyer plusieurs éléments (hôte, chemin et requête). Les fonctions PL/pgSQL ne peuvent renvoyer qu'une seule valeur. Dans PostgreSQL, les fonctions ne peuvent renvoyer qu'une seule valeur . Un moyen de contourner cela est de transformer la valeur de retour en type composite (type ligne).

Voici la version Oracle :

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- Celle ci sera passée en retour
    v_path OUT VARCHAR,  -- Celle la aussi
    v_query OUT VARCHAR) -- Et celle la
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Voici une traduction possible en PL/pgSQL :

```
CREATE TYPE cs_parse_url_result AS (
      v_host VARCHAR,
      v_path VARCHAR,
      v_query VARCHAR
      );

CREATE OR REPLACE FUNCTION cs_parse_url(v_url VARCHAR)
RETURNS cs_parse_url_result AS $$
DECLARE
```

```
    res cs_parse_url_result;
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    res.v_host := NULL;
    res.v_path := NULL;
    res.v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN res;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        res.v_host := substr(v_url, a_pos1 + 2);
        res.v_path := '/';
        RETURN res;
    END IF;

    res.v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        res.v_path := substr(v_url, a_pos2);
        RETURN res;
    END IF;

    res.v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    res.v_query := substr(v_url, a_pos1 + 1);
    RETURN res;
END;
$$ LANGUAGE plpgsql;
```

Cette fonction pourrait être utilisée ainsi :

```
      SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Exemple 35–8 montre comment porter une procédure qui utilise de nombreuses fonctionnalités spécifiques à Oracle.

**Exemple 35–8. Portage d'une Procédure de PL/SQL vers PL/pgSQL**

La version Oracle :

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;(1)
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;(2)

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock(3)
        raise_application_error(-20000, 'Unable to create a new job: a job is currently running.'
    END IF;

    DELETE FROM cs_active_job;
```

```
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
        EXCEPTION
          WHEN dup_val_on_index THEN NULL; -- ne vous inquietez pas si cela existe déjà
    END;
    COMMIT;
END;
/
show errors
```

Les procédures comme celles-ci peuvent être aisément converties en fonctions PostgreSQL renvoyant un `void`. Cette procédure en particulier est intéressante parce qu'elle peut nous apprendre diverses choses :

**(1)**

Il n'y a pas d'instruction `PRAGMA` dans PostgreSQL.

**(2)**

Si vous faites un `LOCK TABLE` dans PL/pgSQL, le verrou ne sera pas libéré jusqu'à ce que la transaction appelante soit terminée.

**(3)**

Vous ne pouvez pas lancer un `COMMIT` dans une fonction PL/pgSQL. La fonction est lancée à l'intérieur d'une transaction externe et, du coup, un `COMMIT` impliquerait simplement la fin de l'exécution de la fonction. Néanmoins, dans ce cas particulier, ce n'est de toute façon pas nécessaire parce que le verrou obtenu par `LOCK TABLE` sera libéré lors de la levée de l'erreur.

Voici comment nous pourrions porter cette procédure vers PL/pgSQL :

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
  a_running_job_count integer;
BEGIN
  LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

  SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

  IF a_running_job_count > 0 THEN
    RAISE EXCEPTION 'Unable to create a new job: a job is currently running';(1)
  END IF;

  DELETE FROM cs_active_job;
  INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

  BEGIN
    INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
  EXCEPTION
    WHEN unique_violation THEN (2)
    -- don't worry if it already exists
  END;

  RETURN;
END;
$$ LANGUAGE plpgsql;
```

**(1)**

La syntaxe de `RAISE` est considérablement différente de l'instruction Oracle similaire.

**(2)**

Les noms d'exceptions supportées par PL/pgSQL sont différents de ceux d'Oracle. L'ensemble de noms d'exceptions intégré est plus important (voir Annexe A). Il n'existe actuellement pas de façon de déclarer des noms d'exceptions définis par l'utilisateur.

La principale différence fonctionnelle entre cette procédure et l'équivalent Oracle est que le verrou exclusif sur la table `cs_jobs` sera détenu jusqu'à la fin de la transaction appelante. De plus, si l'appelant annule plus tard (par exemple à cause d'une erreur), les effets de cette procédure seront annulés.

## 35.11.2. Autres Choses A Surveiller

Cette section explique quelques autres choses à surveiller quand on effectue un portage de fonctions PL/SQL Oracle vers PostgreSQL.

### 35.11.2.1. Implicit Rollback after Exceptions

Dans PL/pgSQL, quand une exception est récupérée par une clause `EXCEPTION`, toutes les modifications de la base de données depuis le bloc `BEGIN` sont automatiquement annulées. C'est–à–dire que le comportement est identique à celui obtenu à partir d'Oracle avec

```
BEGIN
SAVEPOINT s1;
... code ici ...
EXCEPTION
WHEN ... THEN
ROLLBACK TO s1;
... code ici ...
WHEN ... THEN
ROLLBACK TO s1;
... code ici ...
END;
```

Si vous traduisez une procédure d'Oracle qui utilise `SAVEPOINT` et `ROLLBACK TO` dans ce style, votre tâche est facile : omettez `SAVEPOINT` et `ROLLBACK TO`. Si vous avez une procédure qui utilise `SAVEPOINT` et `ROLLBACK TO` d'une façon différente, alors un peu de réflexion supplémentaire sera nécessaire.

### 35.11.2.2. `EXECUTE`

La version PL/pgSQL d'`EXECUTE` fonctionne de façon similaire à la version PL/SQL, mais vous devez vous rappeler d'utiliser `quote_literal(text)` et `quote_string(text)` comme décrit dans Section 35.6.5. Les constructions de type `EXECUTE 'SELECT * FROM $1';` ne fonctionneront pas à moins d'utiliser ces fonctions.

### 35.11.2.3. Optimisation des Fonctions PL/pgSQL

PostgreSQL vous donne deux modificateurs de création de fonctions pour optimiser l'exécution : la << volatilité >> (la fonction renvoie toujours le même résultat quand on lui donne les mêmes arguments) et la << rigueur >> (une fonction renvoie NULL si tous ses arguments sont NULL). Consultez la page de référence CREATE FUNCTION pour les détails.

Pour faire usage de ces attributs d'optimisation, votre instruction `CREATE FUNCTION` devrait ressembler a ceci:

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

## 35.11.3. Annexe

Cette section contient le code d'un ensemble de fonctions `instr` compatible Oracle que vous pouvez utiliser pour simplifier vos efforts de portage.

```
--
-- fonctions instr qui reproduisent la contrepartie Oracle
-- Syntaxe: instr(string1, string2, [n], [m]) où [] signifie paramètre optionnel.
--
-- Cherche string1 en commençant par le  n-ième caractère pour la m-ième occurrence
-- de string2.  Si n est négatif, cherche en sens inverse. Si m n'est pas fourni
-- suppose 1 (la recherche commence au premier caractère).
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
DECLARE
    pos integer;
BEGIN
    pos:= instr($1, $2, 1);
    RETURN pos;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search varchar, beg_indexvarchar)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
```

```
        END LOOP;

        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
```

```
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                occur_number := occur_number + 1;

                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

# Chapitre 36. PL/Tcl – Langage procédural Tcl

PL/Tcl est un langage procédural chargeable pour le système de bases de données PostgreSQL, activant l'utilisation du langage Tcl pour l'écriture de fonctions de procédures déclencheurs.

## 36.1. Survol

PL/Tcl offre un grand nombre de fonctionnalités qu'un codeur de fonctions dispose avec le langage C, avec quelques restrictions.

La bonne restriction est que tout est exécuté dans un interpréteur Tcl sûr. En plus de l'ensemble limité de commandes d'un Tcl sûr, seules quelques commandes sont disponibles pour accéder à la base de données via SPI et pour envoyer des messages via `elog()`. Il n'existe aucun moyen d'accéder aux valeurs internes du serveur de bases de données ou pour gagner un accès niveau système d'exploitation avec les droits du processus serveur PostgreSQL comme les fonctions C vous le permettent. Du coup, un utilisateur non privilégié de la base de données pourrait avoir le droit d'utiliser ce langage.

L'autre restriction d'implémentation est que les fonctions Tcl ne peuvent pas être utilisées pour créer des fonctions d'entrées/sorties pour les nouveaux types de données.

Quelques fois, il est préférable d'écrire des fonctions Tcl non restreintes par le Tcl sûr. Par exemple, vous pourriez vouloir une fonction Tcl pour envoyer un courrier électronique. Pour gérer ces cas, il existe une variante de PL/Tcl appelée `PL/TclU` (Tcl non accrédité). C'est exactement le même langage sauf qu'un interpréteur Tcl complet est utilisé. *Si PL/TclU est utilisé, il doit être installé comme langage procédural non accrédité* de façon à ce que seuls les superutilisateurs de la base de données puissent créer des fonctions avec lui. Le codeur d'une fonction PL/TclU doit faire attention au fait que la fonction ne pourra pas être utilisé pour faire autre chose que son but initial, car il sera possible de faire tout ce qu'un administrateur de la base de données peut faire.

L'objet partagé pour les gestionnaires d'appel PL/Tcl et PL/TclU est automatiquement construit et installé dans le répertoire des bibliothèques de PostgreSQL si le support de Tcl est spécifié dans l'étape de configuration de la procédure d'installation. Pour installer PL/Tcl et/ou PL/TclU dans une base de données particulière, utilisez le programme `createlang`, par exemple `createlang pltcl` *`nom_base`* ou `createlang pltclu` *`nom_base`*.

## 36.2. Fonctions et arguments PL/Tcl

Pour créer une fonction dans le langage PL/Tcl, utilisez la syntaxe standard :

```
CREATE FUNCTION nom_fonction
(types_arguments) RETURNS
type_en_retour AS $$
    # corps de la fonction PL/Tcl
$$ LANGUAGE pltcl;
```

PL/TclU est identique sauf que le langage doit être `pltclu`.

Le corps de la fonction est simplement un bout de script Tcl. Quand la fonction est appelée, les valeurs des

arguments sont passées en tant que variables `$1` ... `$n` au script Tcl. Le résultat est renvoyé à partir du code Tcl de la façon habituelle, avec une instruction `return`.

Par exemple, une fonction renvoyant le plus grand de deux valeurs entières pourrait être définie ainsi :

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Notez la clause `STRICT`, qui nous permet d'éviter de penser aux valeurs NULL en entrées : si une valeur NULL est passée, la fonction ne sera pas appelée du tout mais renverra automatiquement un résultat nul.

Dans une fonction non stricte, si la valeur réelle d'un argument est NULL, la variable `$n` correspondante sera initialisée avec une chaîne vide. Pour détecter si un argument particulier est nul, utilisez la fonction `argisnull`. Par exemple, supposez que nous voulons `tcl_max` avec un argument nul et un non nul pour renvoyer l'argument non nul plutôt que nul :

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

Comme indiqué ci–dessus, pour renvoyer une valeur NULL à partir d'une fonction PL/Tcl, exécutez `return_null`. Ceci peut être fait que la fonction soit stricte ou non.

Les arguments de type composé sont passés à la fonction comme des tableaux Tcl. Les noms des éléments du tableau sont les noms d'attribut du type composite. Si un attribut dans la ligne passée à la valeur NULL, il n'apparaîtra pas dans le tableau. Voici un exemple :

```
CREATE TABLE employe (
    nom text,
    salaire integer,
    age integer
);

CREATE FUNCTION surpaye(employe) RETURNS boolean AS $$
    if {200000.0 < $1(salaire)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salaire)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

Il n'y a actuellement aucun support pour le retour d'une valeur résultat de type composé et pour le retour d'ensembles.

PL/Tcl n'a pas actuellement du support complet pour les types de domaine : il traite un domaine de la même façon que le type scalaire sous–jacent. Cela signifie que les contraintes associées avec le domaine ne seront

pas forcées. Ce n'est pas un problème pour les arguments de la fonction mais c'est hasardeux de déclarer une fonction PL/Tcl renvoyant un type domaine.

## 36.3. Valeurs des données avec PL/Tcl

Les valeurs des arguments fournies au code d'une fonction PL/Tcl sont simplement les arguments en entrée convertis au format texte (comme s'ils avaient été affichés par une instruction `SELECT`). De même, la commande `return` acceptera toute chaîne acceptable dans le format d'entrée du type de retour déclaré pour la fonction. Donc, à l'intérieur de la fonction PL/Tcl, toutes les valeurs de données sont simplement des chaînes de texte.

## 36.4. Données globales avec PL/Tcl

Quelque fois, il est utile d'avoir des données globales qui sont conservées entre deux appels à une fonction ou qui sont partagées entre plusieurs fonctions. Ceci peut être facilement obtenu car toutes les fonctions PL/Tcl exécutées dans une session partagent le même interpréteur Tcl sûr. Donc, toute variable globale Tcl est accessible aux appels de fonctions PL/Tcl et persisteront pour la durée de la session SQL. (Notez que les fonctions PL/TclU partagent de la même façon les données globales mais elles sont dans un interpréteur Tcl différent et ne peuvent pas communiquer avec les fonctions PL/Tcl.)

Pour aider à la protection des fonctions PL/Tcl sur les interférences non intentionnelles, un tableau global est rendu disponible pour chaque fonction via la commande `upvar`. Le nom global de cette variable est le nom interne de la fonction alors que le nom local est `GD`. Il est recommandé d'utiliser `GD` pour les données privées persistantes d'une fonction. Utilisez les variables globales Tcl uniquement pour les valeurs que vous avez l'intention de partager avec les autres fonctions.

Un exemple de l'utilisation de `GD` apparaît dans l'exemple `spi_execp` ci–dessous.

## 36.5. Accès à la base de données depuis PL/Tcl

Les commandes suivantes sont disponibles pour accéder à la base de données depuis le corps d'une fonction PL/Tcl :

`spi_exec ?-count` *n*`? ?-array` *name*`?` *command* `?`*loop-body*`?`

Exécute une commande SQL donnée en tant que chaîne. Une erreur dans la commande lève une erreur. Sinon, la valeur de retour de `spi_exec` est le nombre de lignes intéressées dans le processus (sélection, insertion, mise à jour ou suppression) par la commande ou zéro si la commande est une instruction utilitaire. De plus, si la commande est une instruction `SELECT`, les valeurs des données sélectionnées sont placées dans des variables Tcl décrites ci–dessous.

La valeur optionnelle `-count` indique à `spi_exec` le nombre maximum de lignes à travailler dans la commande. L'effet de ceci est comparable à l'initialisation d'une requête en tant que curseur et de dire `FETCH` *n*.

Si la commande est une instruction `SELECT`, les valeurs des colonnes de résultat sont placées dans les variables Tcl nommées d'après les colonnes. Si l'option `-array` est donnée, les valeurs de colonnes sont stockées à la place dans un tableau associatif nommé, avec les noms des colonnes utilisés comme index du tableau.

Si la commande est une instruction SELECT et qu'aucun script *loop-body* n'est donné, alors seule la première ligne de résultats est stockée dans des variables Tcl ; les lignes suivants sont ignorées. Aucun stockage n'intervient si la requête ne renvoie pas de ligne. (Ce cas est détectable avec le résultat de la fonction spi_exec.) Par exemple,

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

initialisera la variable Tcl $cnt avec le nombre de lignes dans le catalogue système pg_proc.

Si l'argument *loop-body* optionnel est donné, il existe un morceau de script Tcl qui est exécuté une fois pour chaque ligne du résultat de la requête. (*loop-body* est ignoré si la commande donnée n'est pas un SELECT.) Les valeurs des colonnes de la ligne actuelle sont stockées dans des variables Tcl avant chaque itération. Par exemple,

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

affichera un message de trace pour chaque ligne de pg_class. Cette fonctionnalité travaille de façon similaire aux autres constructions de boucles de Tcl ; en particulier, continue et break fonctionnent de la même façon à l'intérieur de *loop-body*.

Si une colonne d'un résultat de la requête est NULL, la variable cible est << dés-initialisée >> plutôt qu'initialisée.

spi_prepare *query typelist*

Prépare et sauvegarde un plan de requête pour une exécution future. Le plan sauvegardé sera conservé pour la durée de la session actuelle.

La requête peut utiliser des paramètres, c'est-à-dire des emplacements pour des valeurs à fournir lorsque le plan sera réellement exécuté. Dans la chaîne de requête, faites référence aux paramètres avec les symboles $1 ... $*n*. Si la requête utilise les paramètres, les noms des types de paramètre doivent être donnés dans une liste Tcl. (Écrivez une liste vide pour *typelist* si aucun paramètre n'est utilisé.) Actuellement, les types de paramètres doivent être identifiés par les noms de types internes affichés dans la table système pg_type ; par exemple int4 et non pas integer.

La valeur de retour de spi_prepare est l'identifiant de la requête à utiliser dans les appels suivants à spi_execp. Voir spi_execp pour un exemple.

spi_execp ?-count *n*? ?-array *name*? ?-nulls *string*? *queryid* ?*value-list*? ?*loop-body*?

Exécute une requête préparée précédemment avec spi_prepare. *queryid* est l'identifiant renvoyé par spi_prepare. Si la requête fait référence à des paramètres, une liste de valeurs (*value-list*) doit être fournie. C'est une liste Tcl des valeurs réelles des paramètres. La liste doit être de la même longueur que la liste de types de paramètres donnée précédemment lors de l'appel à spi_prepare. Oubliez-la si la requête n'a pas de paramètres.

La valeur optionnelle pour -nulls est une chaîne d'espaces et de caractères 'n' indiquant à spi_execp les paramètres nuls. Si indiqué, il doit avoir exactement la même longueur que *value-list*. Si elle est omise, toutes les valeurs de paramètres sont non NULL.

Sauf si la requête et ses paramètres sont spécifiés, spi_execp fonctionne de la même façon que spi_exec. Les options -count, -array et *loop-body* sont identiques. Du coup, la valeur du

résultat l'est aussi.

Voici un exemple d'une fonction PL/Tcl utilisant un plan préparé :

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # prépare le plan sauvegardé au premier appel
        set GD(plan) [ spi_prepare \
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;
```

Nous avons besoin des antislashs à l'intérieur de la chaîne de la requête passée à `spi_prepare` pour s'assurer que les marqueurs `$`*n* seront passés au travers de `spi_prepare` sans transformation et ne sont pas remplacés avec la substitution de variables de Tcl.

`spi_lastoid`
: Renvoie l'OID de la ligne insérée par le dernier appel à `spi_exec` ou `spi_execp`, si la commande était un `INSERT` d'une seule ligne. (Sinon, vous obtenez zéro.)

`quote` *`string`*
: Double toutes les occurrences de guillemet simple et d'antislash dans la chaîne donnée. Ceci peut être utilisé pour mettre entre guillemets des chaînes de façon sûr et pour qu'elles puissent être insérées dans des commandes SQL passées à `spi_exec` ou `spi_prepare`. Par exemple, pensez à une chaîne de commande SQL comme :

```
"SELECT '$val' AS ret"
```

où la variable Tcl `val` contient actuellement le mot `doesn't`. Ceci finirait avec la chaîne de commande :

```
SELECT 'doesn't' AS ret
```

qui pourrait causer une erreur d'analyse lors de `spi_exec` ou de `spi_prepare`. Pour fonctionner correctement, la commande soumise devrait contenir

```
SELECT 'doesn''t' AS ret
```

qui peut−être créé avec PL/Tcl en utilisant

```
"SELECT '[ quote $val ]' AS ret"
```

Un avantage de `spi_execp` est que vous n'avez pas à mettre entre guillemets des valeurs de paramètres comme ceux−ci car les paramètres ne sont jamais analysés comme faisant partie de la chaîne de la commande SQL.

`elog` *`level msg`*
: Émet une trace ou un message d'erreur. Les niveaux possibles sont `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `ERROR` et `FATAL`. `ERROR` élève une condition d'erreur ; si elle n'est pas récupérée par le code Tcl, l'erreur est propagée à la requête appelant, causant l'annulation de la transaction ou sous−transaction en cours. Ceci est en fait identique à la commande `error`. `FATAL` annule la transaction et fait que la session courante s'arrête. (Il n'existe probablement aucune raison d'utiliser ce niveau d'erreur dans les fonctions PL/Tcl mais il est fourni pour que tous les messages soient tout de

même disponibles.) Les autres niveaux génèrent seulement des messages de niveaux de priorité différent. Le fait que les messages d'un niveau de priorité particulier sont reportés au client, écrit dans les journaux du serveur, ou les deux à la fois, est contrôlé par les variables de configuration log_min_messages et client_min_messages. Voir Section 16.4 pour plus d'informations.

# 36.6. Procédures pour déclencheurs en PL/Tcl

Les procédures pour déclencheurs peuvent être écrites en PL/Tcl. PostgreSQL requiert qu'une procédure, devant être appelée en tant que déclencheur, doit être déclarée comme une fonction sans arguments et retourner une valeur de type `trigger`.

L'information du gestionnaire de déclencheur est passée au corps de la procédure avec les variables suivantes :

`$TG_name`
: Nom du déclencheur provenant de l'instruction `CREATE TRIGGER`.

`$TG_relid`
: L'identifiant objet de la table qui est à la cause du lancement du déclencheur.

`$TG_relatts`
: Une liste Tcl des noms des colonnes de la table, préfixée avec un élément de liste vide. Donc, rechercher un nom de colonne dans la liste avec la commande `lsearch` de Tcl renvoie le numéro de l'élément, en commençant à 1 pour la première colonne, de la même façon que les colonnes sont numérotées personnellement avec PostgreSQL.

`$TG_when`
: La chaîne `BEFORE` ou `AFTER` suivant le type d'appel du déclencheur.

`$TG_level`
: La chaîne `ROW` ou `STATEMENT` suivant le type d'appel du déclencheur.

`$TG_op`
: La chaîne `INSERT`, `UPDATE` ou `DELETE` suivant le type d'appel du déclencheur.

`$NEW`
: Un tableau associatif contenant les valeurs de la nouvelle ligne de la table pour les actions `INSERT` ou `UPDATE` ou vide pour `DELETE`. Le tableau est indexé par nom de colonne. Les colonnes NULL n'apparaissent pas dans le tableau.

`$OLD`
: Un tableau associatif contenant les valeurs de l'ancienne ligne de la table pour les actions `UPDATE` or `DELETE` ou vide pour `INSERT`. Le tableau est indexé par nom de colonne. Les colonnes NULL n'apparaissent pas dans le tableau.

`$args`
: Une liste Tcl des arguments de la procédure ainsi que le demande l'instruction `CREATE TRIGGER`. Ces arguments sont aussi accessibles par `$1` ... `$`*n* dans le corps de la procédure.

Le code de retour d'une procédure de déclencheur peut être faite avec une des chaînes `OK` ou `SKIP` ou une liste renvoyée par la commande Tcl `array get`. Si la valeur de retour est `OK`, l'opération (`INSERT`/`UPDATE`/`DELETE`) qui a lancé le déclencheur continuera normalement. `SKIP` indique au gestionnaire de déclencheurs de supprimer silencieusement l'opération pour cette ligne. Si une liste est renvoyée, elle indique à PL/Tcl de renvoyer la ligne modifiée au gestionnaire de déclencheurs qui l'insérera au lieu de celle données par `$NEW`. (Ceci fonctionne seulement pour `INSERT` et `UPDATE`.) Inutile de dire que tout ceci est vraiment significatif lorsque le déclencheur est `BEFORE` et `FOR EACH ROW` ; sinon le code de retour est ignoré.

Voici un petit exemple de procédure déclencheur qui force une valeur entière dans une table pour garder trace du nombre de mises à jour réalisées sur la ligne. Pour les nouvelles lignes insérées, la valeur est initialisée à 0 puis incrémentée à chaque opération de mise à jour.

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

Notez que la procédure déclencheur elle−même ne connaît pas le nom de la colonne ; c'est fourni avec les arguments du déclencheur. Ceci permet à la procédure déclencheur d'être ré−utilisée avec différentes tables.

## 36.7. Les modules et la commande `unknown`

PL/Tcl dispose du support de chargement automatique de code Tcl lorsqu'il est utilisé. Il reconnaît une table spécial, `pltcl_modules`, qui est présumée contenir les modules de code Tcl. Si cette table existe, le module `unknown` est récupéré de la table et chargé immédiatement dans l'interpréteur Tcl après création de l'interpréteur.

Alors que le module `unknown` pourrait réellement contenir tout script d'initialisation dont vous avez besoin, il définit normalement une procédure Tcl `unknown` qui est appelée lorsque Tcl ne reconnaît pas le nom de la procédure appelée. La version standard de PL/Tcl essaie de trouver un module dans `pltcl_modules` qui définira la procédure requis. Si une procédure est trouvée, elle est chargée dans l'interpréteur, puis l'exécution est permise avec l'appel original de la procédure. Une deuxième table `pltcl_modfuncs` fournit un index des fonctions et des modules qu les définissent, de façon à ce que la recherche soit rapide.

La distribution PostgreSQL inclut les scripts de support pour maintenir ces tables : `pltcl_loadmod`, `pltcl_listmod`, `pltcl_delmod`, ainsi que le source pour le module standard `unknown` dans `share/unknown.pltcl`. Ce module doit être chargeable dans chaque base de données initialement pour supporter le mécanisme de chargement automatique.

Les tables `pltcl_modules` et `pltcl_modfuncs` doivent être lisibles par tous mais il est conseillé de les laisser modifiables uniquement par le propriétaire, administrateur de la base de données.

# 36.8. Noms de procédure Tcl

Avec PostgreSQL, un nom de fonction peut être utilisé par plusieurs fonctions tant que le nombre d'arguments ou leurs types diffèrent. Néanmoins, Tcl requiert que les noms de procédure soient distincts. PL/Tcl gère ceci en faisant en sorte que les noms de procédures Tcl internes contiennent l'identifiant de l'objet de la fonction depuis la table système `pg_proc`. Du coup, les fonctions PostgreSQL avec un nom identique et des types d'arguments différents seront aussi des procédures Tcl différentes. Ceci ne concerne normalement pas le développeur PL/Tcl mais cela pourrait apparaître dans une session de débogage.

# Chapitre 37. PL/Perl – Le langage de procédures Perl

PL/Perl est un langage de procédures chargeable qui vous permet d'écrire des fonctions PostgreSQL dans le langage de programmation Perl.

Pour installer PL/Perl dans une base de données spécifique, utilisez la commande `createlang plperl` *`nom_base`*.

> **Astuce :** Si un langage est installé dans `template1`, toutes les bases de données créées ultérieurement disposeront automatiquement de ce langage.

> **Note :** Les utilisateurs des paquetages sources doivent explicitement autoriser la construction de PL/Perl pendant le processus d'installation. (Se référer à Section 14.1 pour plus d'informations.) Les utilisateurs des paquetages binaires peuvent trouver PL/Perl dans un sous−paquetage séparé.

## 37.1. Fonctions et arguments PL/Perl

Pour créer une fonction dans le langage PL/Perl, utilisez la syntaxe standard :

```
CREATE FUNCTION nomfonction
(types-arguments) RETURNS
type-retour AS $$
    # Corps de la fonction PL/Perl
$$ LANGUAGE plperl;
```

Le corps de la fonction est du code Perl normal.

La syntaxe de la commande `CREATE FUNCTION` requiert que le corps de la fonction soit écrit comme une constante de type chaîne. Il est habituellement plus agréable d'utiliser les guillemets dollar (voir Section 4.1.2.2) pour cette constante. Si vous choisissez d'utiliser la syntaxe à base de guillemets simples, vous devez faire un échappement des marques de guillemets simples (`'`) et des antislashs (`\`) utilisés dans le corps de la fonction, typiquement en les doublant (voir Section 4.1.2.1).

Les arguments et les résultats sont manipulés comme dans n'importe quel routine Perl : les arguments sont passés au tableau `@_` et une valeur de retour est indiquée par `return` ou par la dernière expression évaluée dans la fonction.

Par exemple, une fonction retournant le plus grand de deux entiers peut être définie comme suit :

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

Si une valeur NULL en SQL est passée à une fonction, cet argument apparaîtra comme << undefined >> en Perl. La fonction définie ci−dessus ne se comportera pas correctement avec des arguments NULL (en fait, tout se passera comme s'ils avaient été des zéros). Nous aurions pu ajouter `STRICT` à la définition de la fonction

pour forcer PostgreSQL à faire quelque chose de plus raisonnable : si une valeur NULL est passée en argument, la fonction ne sera pas du tout appelée mais retournera automatiquement un résultat NULL. D'une autre façon, nous aurions pu vérifier dans le corps de la fonction la présence d'arguments NULL. Par exemple, supposons que nous voulions que `perl_max` avec un argument NULL et un autre non NULL retourne une valeur non NULL plutôt qu'une valeur NULL, on aurait écrit :

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($a,$b) = @_;
    if (! defined $a) {
        if (! defined $b) { return undef; }
        return $b;
    }
    if (! defined $b) { return $a; }
    if ($a > $b) { return $a; }
    return $b;
$$ LANGUAGE plperl;
```

Comme le montre l'exemple ci−dessus, passer une valeur NULL en SQL à une fonction en PL/Perl retourne une valeur non définie. Et ceci, que la fonction soit déclarée stricte ou non.

Les arguments de type composite sont passés à la fonction en tant que références d'un tableau de découpage, les clés du tableau de découpage étant les noms des attributs du type composé. Voici un exemple :

```
CREATE TABLE employe (
    nom text,
    basesalaire integer,
    bonus integer
);

CREATE FUNCTION empcomp(employe) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalaire} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT nom, empcomp(employe.*) FROM employe;
```

Une fonction PL/Perl peut renvoyer un résultat de type composite en utilisant la même approche : renvoyer une référence à un hachage qui a les attributs requis. Par exemple,

```
CREATE TYPE testligneperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_ligne() RETURNS testligneperl AS $$
return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

Toute colonne dans le type de données déclaré du résultat qui n'est pas présente dans le hachage sera renvoyée NULL.

Les fonctions PL/Perl peuvent aussi renvoyer des ensembles de types scalaires ou composites. Pour cela, renvoyez une référence à un tableau contenant respectivement soit des scalaires soit des références à des hachages. Voici quelques exemples simples :

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
return [0..$_[0]];
```

```
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testligneperl AS $$
return [
{ f1 => 1, f2 => 'Bonjour', f3 => 'Monde' },
{ f1 => 2, f2 => 'Bonjour', f3 => 'PostgreSQL' },
{ f1 => 3, f2 => 'Bonjour', f3 => 'PL/Perl' }
];
$$  LANGUAGE plperl;

SELECT * FROM perl_set();
```

Notez qu'en faisant ceci, Perl devra construire le tableau entier en mémoire ; du coup, la technique n'est pas utilisable pour des ensembles de résultats très importants.

PL/Perl n'a actuellement pas de support complet pour les types domaine : il traite un domaine de la même façon qu'un type scalaire sous−jacent. Ceci signifie que les contraintes associées au domaine ne seront pas forcées. Ceci n'est pas un problème pour les arguments d'une fonction mais c'est hasardeux de déclarer une fonction PL/Perl renvoyant un type domaine.

## 37.2. Accès à la base de données depuis PL/Perl

L'accès à la base de données à l'intérieur de vos fonctions écrites en Perl peut se faire à partir de la fonction `spi_exec_query` décrite ci−dessous ou à partir du module expérimental `DBD::PgSPI` (aussi disponible sur un miroir du CPAN). Ce module rend accessible un descripteur de base de données conforme à DBI nommé `$pg_dbh` qui peut−être utilisé pour exécuter des requêtes en utilisant la syntaxe habituelle de DBI.

Actuellement, PL/Perl fournit deux commandes Perl supplémentaires :

`spi_exec_query (`*`query`* `[, `*`max−rows`*`])`
`spi_exec_query(` *`command`*`)`

Exécute une commande SQL. Voici un exemple d'une requête (commande `SELECT`) avec le nombre maximum optionnel de lignes

```
$rv = spi_exec_query('SELECT * FROM ma_table', 5);
```

Ceci envoie cinq lignes au maximum de la table `ma_table`. Si `ma_table` a une colonne `ma_colonne`, vous obtenez la valeur de la ligne `$i` du résultat de cette façon :

```
$foo = $rv->{rows}[$i]->{ma_colonne};
```

Le nombre total des lignes renvoyées d'une requête `SELECT` peut être accédé de cette façon :

```
$nrows = $rv->{processed}
```

Voici un exemple en utilisant un type de commande différent :

```
$query = "INSERT INTO ma_table VALUES (1, 'test')";
```

```
$rv = spi_exec_query($query);
```

Ensuite, vous pouvez accéder au statut de la commande (c'est−à−dire, `SPI_OK_INSERT`) de cette façon :

```
$res = $rv->{status};
```

Pour obtenir le nombre de lignes affectées, exécutez :

```
$nrows = $rv->{processed};
```

Voici un exemple complet :

```
CREATE TABLE test (
i int,
v varchar
);

INSERT INTO test (i, v) VALUES (1, 'première ligne');
INSERT INTO test (i, v) VALUES (2, 'deuxième ligne');
INSERT INTO test (i, v) VALUES (3, 'troisième ligne');
INSERT INTO test (i, v) VALUES (4, 'immortel');

CREATE FUNCTION test_munge() RETURNS SETOF test AS $$
my $res = [];
my $rv = spi_exec_query('select i, v from test;');
my $status = $rv->{status};
my $nrows = $rv->{processed};
foreach my $rn (0 .. $nrows - 1) {
my $row = $rv->{rows}[$rn];
$row->{i} += 200 if defined($row->{i});
$row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
push @$res, $row;
}
return $res;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

`elog(`*`level`*`, `*`msg`*`)`
: Produit un message de trace ou d'erreur. Les niveaux possibles sont `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING` et `ERROR`. `ERROR` lève une condition d'erreur ; si elle n'est pas récupérée par le code Perl l'entourant, l'erreur se propage à l'extérieur de la requête appelante, causant l'annulation de la transaction ou sous−transaction en cours. Ceci est en fait identique à la commande `die` de Perl. Les autres niveaux génèrent seulement des messages de niveaux de priorité différents. Le fait que les messages d'un niveau de priorité particulier soient rapportés au client, écrit dans les journaux du serveur, voire les deux, est contrôlé par les variables de configuration log_min_messages et client_min_messages. Voir Section 16.4 pour plus d'informations.

## 37.3. Valeurs des données dans PL/Perl

Les valeurs de l'argument fournies au code d'une fonction PL/Perl sont simplement les arguments en entrée convertis dans la forme textuelle (comme si elles avaient été affichées par une instruction `SELECT`). Par contre, une commande `return` acceptera toute chaîne dont le format en entrée est acceptable pour le type déclaré en retour. Donc, à l'intérieur de la fonction PL/Perl, toutes les valeurs sont simplement des chaînes de texte.

## 37.4. Valeurs globales dans PL/Perl

Vous pouvez utiliser le hachage global `%_SHARED` pour stocker les données, incluant les références de code, entre les appels de fonction pour la durée de vie de la session en cours.

Voici un exemple simple pour des données partagées :

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
if ($_SHARED{$_[0]} = $_[1]) {
return 'ok';
} else {
return "Ne peux pas initialiser la variable partagée $_[0] à $_[1]";
}
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Bonjour, PL/Perl ! Comment va ?');
SELECT get_var('sample');
```

Voici un exemple légèrement plus compliqué utilisant une référence de code :

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
$_SHARED{myquote} = sub {
my $arg = shift;
$arg =~ s/(['\\])/\\$1/g;
return "'$arg'";
};
$$ LANGUAGE plperl;

SELECT myfuncs(); /* initialise la fonction */

/* Set up a function that uses the quote function */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
my $text_to_quote = shift;
my $qfunc = $_SHARED{myquote};
return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

(Vous pourriez avoir remplacé le code ci–dessus avec la seule ligne `return $_SHARED{myquote}->($_[0]);` au prix d'une mauvaise lisibilité.)

# 37.5. Niveaux de confiance de PL/Perl

Normalement, PL/Perl est installé en tant que langage de programmation de << confiance >>, de nom `plperl`. Durant cette installation, certaines commandes Perl sont désactivées pour préserver la sécurité. En général, les commandes qui interagissent avec l'environnement sont restreintes. Cela inclut les commandes sur les descripteurs de fichiers, `require` et `use` (pour les modules externes). Il n'est pas possible d'accéder aux fonctions et variables internes du processus du serveur de base de données ou d'obtenir un accès au niveau du système d'exploitation avec les droits du processus serveur, tel qu'une fonction C peut le faire. Ainsi, n'importe quel utilisateur sans droits sur la base de données est autorisé à utiliser ce langage.

Voici l'exemple d'une fonction qui ne fonctionnera pas car les commandes système ne sont pas autorisées pour des raisons de sécurité :

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    open(TEMP, ">/tmp/mauvaisfichier");
    print TEMP "Je t'ai eu !\n";
    return 1;
$$ LANGUAGE plperl;
```

La création de cette fonction va réussir, mais pas son exécution.

Il est parfois souhaitable d'écrire des fonctions Perl qui ne sont pas restreintes. Par exemple, on peut souhaiter vouloir envoyer des courriers électroniques. Pour supporter ce cas de figure, PL/Perl peut aussi être installé comme un langage << douteux >> (habituellement nommé PL/PerlU ). Dans ce cas, la totalité du langage Perl est accessible. Si la commande `createlang` est utilisée pour installer le langage, le nom du langage `plperlu` sélectionnera la version douteuse de PL/Perl.

Les auteurs des fonctions PL/PerlU doivent faire attention au fait que celles–ci ne puissent être utilisées pour faire quelque chose de non désiré, car cela donnera la possibilité d'agir comme si l'on possédait les privilèges d'administrateur de la base de données. Il est à noter que le système de base de données ne permet qu'aux super–utilisateurs de créer des fonctions dans un langage douteux.

Si la fonction ci–dessus a été créée par un super–utilisateur en utilisant le langage `plperlu`, l'exécution de celle–ci réussira.

# 37.6. Déclencheurs PL/Perl

PL/Perl peut être utilisé pour écrire des fonctions pour déclencheurs. Dans une fonction déclencheur, la référence hachée `$_TD` contient des informations sur l'événement du déclencheur en cours. Les champs de la référence de hachage `$_TD` sont :

`$_TD->{new}{foo}`
: Valeur `NEW` de la colonne `foo`

`$_TD->{old}{foo}`
: Valeur `OLD` de la colonne `foo`

`$_TD->{name}`
: Nom du déclencheur appelé

`$_TD->{event}`
: Événement du déclencheur : `INSERT`, `UPDATE`, `DELETE` ou `UNKNOWN`

`$_TD->{when}`
: Quand le déclencheur a été appelé : `BEFORE` (avant), `AFTER` (après) ou `UNKNOWN` (inconnu)

`$_TD->{level}`
: Le niveau du déclencheur : `ROW` (ligne), `STATEMENT` (instruction) ou `UNKNOWN` (inconnu)

`$_TD->{relid}`
: L'OID de la table sur lequel le déclencheur a été exécuté

`$_TD->{relname}`
: Nom de la table sur lequel le déclencheur a été exécuté

`$_TD->{argc}`
: Nombre d'arguments de la fonction déclencheur

`@{$_TD->{args}}`
: Arguments de la fonction déclencheur. N'existe pas si $_TD−>{argc} vaut 0.

Les déclencheurs peuvent renvoyer un des éléments suivants :

`return;`
: Exécute l'instruction

`"SKIP"`
: N'exécute pas l'instruction

`"MODIFY"`
: Indique que la ligne `NEW` a été modifié par la fonction déclencheur

Voici un exemple d'une fonction déclencheur illustrant certains points ci−dessus : above:

```
CREATE TABLE test (
i int,
v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
return "SKIP";    # passe la commande INSERT/UPDATE
} elsif ($_TD->{new}{v} ne "immortal") {
$_TD->{new}{v} .= "(modified by trigger)";
return "MODIFY";  # modifie la ligne et exécute la commande INSERT/UPDATE
} else {
return;           # exécute la commande INSERT/UPDATE
}
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
BEFORE INSERT OR UPDATE ON test
FOR EACH ROW EXECUTE PROCEDURE valid_id();
```

## 37.7. Limitations et fonctionnalités absentes

Les fonctionnalités suivantes ne sont actuellement pas implémentées dans PL/Perl, mais peuvent faire l'objet de contributions généreuses de votre part.

- Les fonctions PL/Perl ne peuvent pas s'appeler entre elles (parce qu'elles sont considérées comme des sous−routines anonymes au sein de Perl).
- SPI n'est pas complètement implémenté.

- Avec l'implémentation actuelle, si vous récupérez ou renvoyez de très gros ensembles de données, vous devez savoir qu'ils iront tous en mémoire.

---

# Chapitre 38. PL/Python – Langage procédural Python

Le langage procédural PL/Python permet l'écriture de fonctions PostgreSQL avec le langage Python.

Pour installer PL/Python dans une base de données particulières, utilisez `createlang plpythonu` *`nom_base`*..

> **Astuce :** Si un langage est installé dans `template1`, toutes les bases nouvellement créées se verront installées ce langage automatiquement.

Depuis PostgreSQL 7.4, PL/Python est seulement disponible en tant que langage << sans confiance >> (ceci signifiant qu'il n'offre aucun moyen de restreindre ce que les utilisateurs en font). Il a donc été renommé en `plpythonu`. La variante de confiance `plpython` pourrait être de nouveau disponible dans le futur, si un nouveau mécanisme sécurisé d'exécution est développé dans Python.

> **Note :** Les utilisateurs des paquets sources doivent activer spécifiquement la construction de PL/Python lors des étapes d'installation. (Référez–vous aux instructions d'installation pour plus d'informations.) Les utilisateurs de paquets binaires pourront trouver PL/Python dans un paquet séparé.

## 38.1. Fonctions PL/Python

Les fonctions PL/Python sont déclarées de la façon habituelle, par exemple

```
CREATE FUNCTION mafonction(text) RETURNS text
    AS 'return args[0]'
    LANGUAGE plpythonu;
```

Le code Python donné comme corps de la définition de fonction est transformé en fonction Python. Par exemple, le code ci–dessus devient

```
def __plpython_procedure_mafonction_23456():
        return args[0]
```

en supposant que 23456 est l'OID affecté à la fonction par PostgreSQL.

Si vous ne fournissez pas de valeur de retour, PL/Python renvoie par défaut `None`. Le module du langage traduit le `None` de Python en la valeur NULL en SQL.

Les paramètres de fonctions avec PostgreSQL sont disponibles dans la liste globale `args`. Dans l'exemple `mafonction`, `args[0]` contient ce qui a été passé dans l'argument texte. Pour `mafonction2(text, integer)`, `args[0]` contiendrait l'argument de type `text` et `args[1]` l'argument de type `integer`.

Le dictionnaire global `SD` est disponible pour stocker des données entres les appels de fonctions. Cette variable est une donnée statique privée. Le dictionnaire global `GD` est une donnée publique, disponible pour toutes les fonctions Python à l'intérieur d'une session. À utiliser avec précaution.

Chaque fonction obtient son propre environnement d'exécution dans l'interpréteur Python, de façon à ce que les données globales et les arguments de fonction provenant de `mafonction` ne soient pas disponibles depuis `mafonction2`. L'exception concerne les données du dictionnaire `GD` comme indiqué ci−dessus.

## 38.2. Fonctions de déclencheurs

Quand une fonction est utilisé comme un déclencheur, le dictionnaire `TD` contient des valeurs relatives au déclencheur. Les lignes du déclencheur sont dans `TD["new"]` et/ou `TD["old"]` suivant l'événement ayant lancé le déclencheur. `TD["event"]` contient l'événement en tant que chaîne (`INSERT`, `UPDATE`, `DELETE` ou `UNKNOWN`). `TD["when"]` contient soit `BEFORE`, soit `AFTER` soit `UNKNOWN`. `TD["level"]` contient une valeur parmi `ROW`, `STATEMENT` et `UNKNOWN`. `TD["name"]` contient le nom du déclencheur et `TD["relid"]` l'OID de la table sur lequel le déclencheur a été activé. Si la commande `CREATE TRIGGER` incluait des arguments, ils sont disponibles dans les variables de `TD["args"][0]` à `TD["args"][(`*n*`−1)]`.

Si `TD["when"]` vaut `BEFORE`, vous pourriez renvoyer `None` ou `"OK"` à partir de la fonction Python pour indiquer que la ligne n'est pas modifiée, `"SKIP"` pour annuler l'événement ou `"MODIFY"` pour indiquer que vous avez modifié la ligne.

## 38.3. Accès à la base de données

Le module du langage PL/Python importe automatiquement un module Python appelé `plpy`. Les fonctions et constantes de ce module vous sont accessibles dans le code Python via `plpy.`*`foo`*. Actuellement, `plpy` implémente les fonctions `plpy.debug(`*`msg`*`)`, `plpy.log(`*`msg`*`)`, `plpy.info(`*`msg`*`)`, `plpy.notice(`*`msg`*`)`, `plpy.warning(`*`msg`*`)`, `plpy.error(`*`msg`*`)` et `plpy.fatal(`*`msg`*`)`. `plpy.error` et `plpy.fatal("msg")` lèvent une exception Python qui, si non attrapée, se propage à la requête appelant causant l'annulation de la transaction ou sous−transaction en cours. `raise plpy.ERROR(`*`msg`*`)` et `raise plpy.FATAL(`*`msg`*`)` sont équivalent à appeler `plpy.error` et `plpy.fatal`, respectivement. Les autres fonctions génèrent uniquement des messages de niveaux de priorité différents. Que les messages d'une priorité particulière soient reportés au client, écrit dans les journaux du serveur ou les deux, cette configuration est contrôlée par les variables log_min_messages et client_min_messages. Voir Section 16.4 pour plus d'informations.

De plus, le module `plpy` fournit deux fonctions appelées `execute` et `prepare`. Appeler `plpy.execute` avec une chaîne de requête et un argument de limite optionnel fait que la requête est lancée et que le résultat est renvoyé dans un objet résultat. L'objet résultat émule une liste ou un objet dictionnaire. L'objet résultat est accessible par le numéro de ligne et le nom de la colonne. Il a plusieurs méthodes supplémentaires : `nrows` qui renvoie le nombre de lignes retournées par la requête et `status` qui est le code de retour de `SPI_execute()`. L'objet résultat peut être modifié.

Par exemple,

```
rv = plpy.execute("SELECT * FROM ma_table", 5)
```

renvoie cinq lignes de `ma_table`. Si `ma_table` dispose d'une colonne `ma_colonne`, elle sera accessible avec

```
foo = rv[i]["ma_colonne"]
```

La seconde fonction, `plpy.prepare`, prépare le plan d'exécution pour une requête. Il est appelé avec une chaîne contenant la requête et une liste des types de paramètres si vous avez des références de paramètres dans la requête. Par exemple :

```
plan = plpy.prepare("SELECT nom FROM mes_utilisateurs WHERE prenom = $1", [
"text" ])
```

`text` est le type de la variable que vous voulez passer via `$1`. Après avoir préparé une instruction, vous utilisez la fonction `plpy.execute` pour l'exécuter :

```
rv = plpy.execute(plan, [ "nom" ], 5)
```

Le troisième argument est la limite et est optionnelle.

Lorsque vous préparez un plan en utilisant le module PL/Python, il est automatiquement sauvegardé. Lisez la documentation SPI (Chapitre 39) pour une description de ce que cela signifie. Pour faire réellement usage de ceci dans les appels de fonction, vous avez besoin d'utiliser un des dictionnaires à stockage permanent `SD` ou `GD` (voir Section 38.1). Par exemple :

```
CREATE FUNCTION utiliseplansauvegarde() RETURNS trigger AS $$
    if SD.has_key("plan"):
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # reste de la fonction
$$ LANGUAGE plpythonu;
```

# Chapitre 39. Interface de programmation serveur

L'*interface de programmation serveur* (SPI) donne aux auteurs de fonctions C la capacité de lancer des commandes SQL au sein de leurs fonctions. SPI est une série de fonctions d'interface simplifiant l'accès à l'analyseur, au planificateur, à l'optimiseur et au lanceur. SPI fait aussi de la gestion de mémoire.

> **Note :** Les langages procéduraux disponibles donnent plusieurs moyens de lancer des commandes SQL à partir de procédures. La plupart est basée à partir de SPI. Cette documentation présente donc également un intérêt pour les utilisateurs de ces langages.

Pour assurer la compréhension, nous utiliserons le terme de << fonction >> quand nous parlerons de fonctions d'interface SPI et << procédure >> pour une fonction C définie par l'utilisateur et utilisant SPI.

Notez que si une commande appelée via SPI échoue, alors le contrôle ne sera pas redonné à votre procédure. Au contraire, la transaction ou sous−transaction dans laquelle est exécutée votre procédure sera annulée. (Ceci pourrait être surprenant étant donné que les fonctions SPI ont pour la plupart des conventions documentées de renvoi d'erreur. Ces conventions s'appliquent seulement pour les erreurs détectées à l'intérieur des fonctions SPI.) Il est possible de récupérer le contrôle après une erreur en établissant votre propre sous−transaction englobant les appels SPI qui pourraient échouer. Ceci n'est actuellement pas documenté parce que les mécanismes requis sont toujours en flux.

Les fonctions SPI renvoient un résultat positif en cas de succès (soit par une valeur de retour entière, soit dans la variable globale `SPI_result` comme décrit ci−dessous). En cas d'erreur, un résultat négatif ou `NULL` sera retourné.

Les fichiers de code source qui utilisent SPI doivent inclure le fichier d'en−tête `executor/spi.h`.

## 39.1. Fonctions d'interface

**Table des matières**

SPI_saveplan −− sauvegarde un plan

# SPI_connect

## Nom

SPI_connect −− connecter une procédure au gestionnaire SPI

## Synopsis

```
int SPI_connect(void)
```

## Description

`SPI_connect` ouvre une connexion au gestionnaire SPI lors de l'appel d'une procédure. Vous devez appeler cette fonction si vous voulez lancer des commandes au travers du SPI. Certaines fonctions SPI utilitaires peuvent être appelées à partir de procédures non connectées.

Si votre procédure est déjà connectée, `SPI_connect` retournera le code d'erreur SPI_ERROR_CONNECT. Cela peut arriver si une procédure qui a appelé `SPI_connect` appelle directement une autre procédure qui appelle `SPI_connect`. Bien que des appels récursifs au gestionnaire SPI soient permis lorsqu'une commande SQL appelée au travers du SPI invoque une autre fonction qui utilise SPI, les appels directement intégrés à `SPI_connect` et `SPI_finish` sont interdits. (Mais, voir `SPI_push` et `SPI_pop`.)

## Valeur de retour

`SPI_OK_CONNECT`
: en cas de succès

`SPI_ERROR_CONNECT`
: en cas d'échec

# SPI_finish

## Nom

SPI_finish −− déconnecter une procédure du gestionnaire SPI

## Synopsis

```
int SPI_finish(void)
```

## Description

`SPI_finish` ferme une connexion existante au gestionnaire SPI. Vous devez appeler cette fonction après avoir terminé les opérations SPI souhaitées pendant l'invocation courante de votre procédure. Vous n'avez pas à vous préoccuper de ceci, sauf si vous terminez la transaction via `elog(ERROR)`. Dans ce cas, SPI terminera automatiquement.

Si `SPI_finish` est appelée sans avoir une connexion valable, elle retournera `SPI_ERROR_UNCONNECTED`. Il n'y a pas de problème fondamental avec cela ; le gestionnaire SPI n'a simplement rien à faire.

## Valeur de retour

`SPI_OK_FINISH`
: si déconnectée correctement

`SPI_ERROR_UNCONNECTED`
: si appel à partir d'une procédure non connectée

# SPI_push

## Nom

SPI_push −− pousse la pile SPI pour autoriser une utilisation récursive de SPI

## Synopsis

```
void SPI_push(void)
```

## Description

`SPI_push` devrait être appelé avant d'exécuter une autre procédure qui pourrait elle−même souhaiter utiliser SPI. Après `SPI_push`, SPI n'est plus dans un état << connecté >> et les appels de fonction SPI seront rejetés sauf si un nouveau `SPI_connect` est exécuté. Ceci nous assure une séparation propre entre l'état SPI de votre procédure et celui d'une autre procédure que vous appelez. Après le retour de cette dernière, appelez `SPI_pop` pour restaurer l'accès à votre propre état SPI.

Notez que `SPI_execute` et les fonctions relatives font automatiquement l'équivalent de `SPI_push` avant de repasser le contrôle au moteur d'exécution SQL, donc il n'est pas nécessaire de vous inquiéter de cela lors de l'utilisation de ces fonctions. Vous aurez besoin d'appeler `SPI_push` et `SPI_pop` seulement quand vous appelez directement un code arbitraire qui pourrait contenir des appels `SPI_connect`.

# SPI_pop

## Nom

SPI_pop −− récupère la pile SPI pour revenir de l'utilisation récursive de SPI

## Synopsis

```
void SPI_pop(void)
```

## Description

`SPI_pop` enlève l'environnement précédent de la pile d'appel SPI. Voir `SPI_push`.

# SPI_execute

## Nom

SPI_execute −− exécute une commande

## Synopsis

```
int SPI_execute(const char * commande, bool read_only, int
nombre)
```

## Description

`SPI_exec` lance la commande SQL spécifiée pour `nombre` lignes. Si `read_only` est `true`, la commande doit être en lecture seule et la surcharge de l'exécution est quelque peu réduit.

Cette fonction ne devrait être appelée qu'à partir d'une procédure connectée.

Si `nombre` vaut zéro, alors la commande est exécutée pour toutes les lignes auxquelles elle s'applique. Si `nombre` est plus grand que 0, alors le nombre de lignes pour lesquelles la commande sera exécutée est restreint (très semblable à une clause `LIMIT`). Par exemple,

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", 5);
```

autorisera au plus cinq lignes à être insérées dans la table.

Vous pourriez passer plusieurs commandes dans une chaîne. `SPI_execute` renvoie le résultat pour la dernière commande exécutée. La limite `nombre` s'applique à chaque commande séparément mais n'est pas appliquée aux commandes cachées par les règles.

Quand `read_only` vaut `false`, `SPI_execute` incrémente le compteur de la commande et calcule une nouvelle *image* avant d'exécuter chaque commande dans la chaîne. L'image n'est pas réellement modifiée si le niveau d'isolation de la transaction en cours est `SERIALIZABLE` mais, en mode `READ COMMITTED`, la mise à jour de l'image permet à chaque commande de voir les résultats des transactions nouvellement validées à partir des autres sessions. Ceci est essentiel pour un comportement cohérent quand les commandes modifient la base de données.

Quand `read_only` vaut `true`, `SPI_execute` ne met à jour ni l'image ni le compteur de commandes, et il autorise seulement les commandes `SELECT` dans la chaîne des commandes. Elles sont exécutées en utilisant l'image précédemment établie par la requête englobante. Ce mode d'exécution est un peu plus rapide que le mode lecture/écriture à cause de l'élimination de la surcharge par commande. Il autorise aussi directement la construction des fonctions *stable* comme les exécutions successives utiliseront toutes la même image, il n'y aura aucune modification dans les résultats.

Il n'est généralement pas conseillé de mixer les commandes en lecture seule et les commandes en lecture/écriture à l'intérieur d'une seule fonction utilisant SPI ; ceci pourrait causer un comportement portant confusion car les requêtes en mode lecture seule devraient ne pas voir les résultats de toute mise à jour de la

base de données effectuées par les requêtes en lecture/écriture.

Le nombre réel de lignes pour lesquelles la (dernière) commande a été lancée est retourné dans la variable globale `SPI_processed` (sauf si la valeur de retour de la fonction est `SPI_OK_UTILITY`). Si la valeur de retour de la fonction est `SPI_OK_SELECT`, alors vous pouvez utiliser le pointeur global `SPITupleTable *SPI_tuptable` pour accéder aux lignes de résultat.

La structure `SPITupleTable` est définie comme suit :

```
typedef struct
{
    MemoryContext tuptabcxt;    /* contexte mémoire de la table de résultat */
    uint32      alloced;        /* nombre de valeurs allouées */
    uint32      free;           /* nombre de valeurs libres */
    TupleDesc   tupdesc;        /* descripteur de rangées */
    HeapTuple  *vals;           /* rangées */
} SPITupleTable;
```

`valeurs` est un tableau de pointeurs vers des lignes. (Le nombre d'entrées valables est donné par `SPI_processed`). `tupdesc` est un descripteur de ligne que vous pouvez passer aux fonctions SPI qui traitent des lignes. `tuptabcxt`, `alloced` et `free` sont des champs internes non conçus pour être utilisés par des routines SPI appelantes.

`SPI_finish` libère tous les `SPITupleTable`s allouées pendant la procédure courante. Vous pouvez libérer une table de résultats donnée plus tôt, si vous en avez terminé avec elle, en appelant `SPI_freetuptable`.

# Arguments

`const char * commande`
: chaîne contenant la commande à exécuter

`bool read_only`
: `true` en cas d'exécution en lecture seule

`int nombre`
: nombre maximum de lignes à traiter où à retourner

# Valeur de retour

Si l'exécution de la commande a réussi, alors l'une des valeurs (positives) suivantes sera renvoyée :

`SPI_OK_SELECT`
: si un `SELECT` (mais pas `SELECT INTO`) a été lancé

`SPI_OK_SELINTO`
: si un `SELECT INTO` a été lancé

`SPI_OK_DELETE`
: si un `DELETE` a été lancé

`SPI_OK_INSERT`
: si un `INSERT` a été lancé

`SPI_OK_UPDATE`

si un UPDATE a été lancé

SPI_OK_UTILITY
si une commande utilitaire (c'est–à–dire CREATE TABLE) a été lancée

Sur une erreur, l'une des valeurs négatives suivante est renvoyée :

SPI_ERROR_ARGUMENT
si commande est NULL ou nombre est inférieur à 0

SPI_ERROR_COPY
si COPY TO stdout ou COPY FROM stdin ont été tentés

SPI_ERROR_CURSOR
si DECLARE, CLOSE ou FETCH ont été tentés

SPI_ERROR_TRANSACTION
si BEGIN, COMMIT ou ROLLBACK ont été tentés

SPI_ERROR_OPUNKNOWN
si le type de commande est inconnu (ce qui ne devrait pas arriver)

SPI_ERROR_UNCONNECTED
si appel à partir d'une procédure non connectée

# Notes

Les fonctions SPI_execute, SPI_exec, SPI_execute_plan et SPI_execp changent à la fois SPI_processed et SPI_tuptable (juste le pointeur, pas le contenu de la structure). Sauvegardez ces deux variables globales dans des variables locales de procédures si vous voulez accéder à la table des résultats de SPI_execute ou d'une fonction relative sur plusieurs appels.

# SPI_exec

## Nom

SPI_exec −− exécute une commande en lecture/écriture

## Synopsis

```
int SPI_exec(const char * commande, int nombre)
```

## Description

`SPI_exec` est identique à `SPI_execute`, mais le paramètre `read_only` de ce dernier est bloqué sur la valeur `false`.

## Arguments

`const char * commande`
: chaîne contenant la commande à exécuter

`int nombre`
: nombre maximum de lignes à traiter ou à renvoyer

## Valeur de retour

Voir `SPI_execute`.

# SPI_prepare

## Nom

SPI_prepare −− prépare un plan pour une commande sans le lancer tout de suite

## Synopsis

```
void * SPI_prepare(const char * commande, int
nargs, Oid * typesargs)
```

## Description

`SPI_prepare` crée et retourne un plan d'exécution pour la commande spécifiée mais ne lance pas la commande. Cette fonction ne peut être appelée que depuis une procédure connectée.

Lorsque la même commande ou une commande semblable doit être lancée à plusieurs reprises, il peut être intéressant de ne faire la planification qu'une seule fois. `SPI_prepare` convertit une chaîne de commande en un plan d'exécution qui peut être lancé plusieurs fois en utilisant `SPI_executeplan`.

Une commande préparée peut être généralisée en utilisant les paramètres (`$1`, `$2`, etc.) en lieu et place de ce qui serait des constantes dans une commande normale. Les valeurs actuelles des paramètres sont alors spécifiées lorsque `SPI_executeplan` est appelée. Ceci permet à la commande préparée d'être utilisée sur une plage plus grande de situations que cela ne serait possible sans paramètres.

Le plan renvoyé par `SPI_prepare` ne peut être utilisé que dans l'invocation courante de la procédure puisque `SPI_finish` libère la mémoire allouée pour le plan. Mais un plan peut être sauvegardé plus longtemps par l'utilisation de la fonction `SPI_saveplan`.

## Arguments

`const char * commande`
: chaîne contenant la commande à lancer

`int nargs`
: nombre de paramètres d'entrée (`$1`, `$2`, etc.)

`Oid * typesargs`
: pointeur vers un tableau contenant les OIDs des types de données des paramètres

## Valeurs de retour

`SPI_prepare` retourne un pointeur non nul vers un plan d'exécution. En cas d'erreur, `NULL` sera retourné et `SPI_result` sera positionnée à un des mêmes codes d'erreur utilisés par `SPI_execute`, sauf qu'il est positionné à `SPI_ERROR_ARGUMENT` si `commande` est `NULL` ou si `nargs` est inférieur à 0 ou si `nargs` est supérieur à 0 et `typesargs` est `NULL`.

# Notes

Il y a un inconvénient à utiliser les paramètres : puisque le planificateur ne connaît pas les valeurs qui seront utilisées pour les paramètres, il effectuera des choix pires que ceux qu'il prendrait pour une commande normale avec toutes les constantes visibles.

# SPI_getargcount

## Nom

SPI_getargcount −− renvoie le nombre d'arguments nécessaires au plan préparé par `SPI_prepare`

## Synopsis

```
int SPI_getargcount(void * plan)
```

## Description

`SPI_getargcount` renvoie le nombre d'arguments nécessaire pour exécuter un plan préparé par `SPI_prepare`.

## Arguments

`void * plan`
: plan d'exécution (renvoyé par `SPI_prepare`)

## Code de retour

Le nombre d'arguments attendus par le `plan` ou `SPI_ERROR_ARGUMENT` si le `plan` est `NULL`

# SPI_getargtypeid

## Nom

SPI_getargtypeid −− renvoie l'OID du type de données pour un argument du plan préparé par `SPI_prepare`

## Synopsis

```
Oid SPI_getargtypeid(void * plan, int argIndex)
```

## Description

`SPI_getargtypeid` renvoie l'OID représentant l'identifiant du type pour le `argIndex`−ième argument d'un plan préparé par `SPI_prepare`. Le premier argument se trouve à l'index zéro.

## Arguments

`void * plan`
: plan d'exécution (renvoyé par `SPI_prepare`)

`int argIndex`
: index de l'argument (à partir de zéro)

## Code de retour

L'identifiant du type de l'argument à l'index donné ou `SPI_ERROR_ARGUMENT` si le `plan` est `NULL` ou si `argIndex` est négatif ou plus petit que le nombre d'arguments déclarés pour le `plan`

# SPI_is_cursor_plan

## Nom

SPI_is_cursor_plan −− return `true` est un plan préparé par `SPI_prepare` pouvant être utilisé avec `SPI_cursor_open`

## Synopsis

```
bool SPI_is_cursor_plan(void * plan)
```

## Description

`SPI_is_cursor_plan` renvoie `true` si un plan préparé par `SPI_prepare` peut être passé comme un argument à `SPI_cursor_open` et `false` si ce n'est pas le cas. Les critères sont qu'un `plan` représente une seule commande et que cette commande est un `SELECT` sans clause `INTO`.

## Arguments

`void * plan`
: plan d'exécution (renvoyé par `SPI_prepare`)

## Return Value

`true` ou `false` pour indiquer si `plan` peut produire un curseur ou non, ou `SPI_ERROR_ARGUMENT` si le `plan` est `NULL`

# SPI_execute_plan

## Nom

SPI_execute_plan −− exécute un plan préparé par `SPI_prepare`

## Synopsis

```
int SPI_execute_plan(void * plan, Datum * valeurs, const char * nulls,
bool read_only, int nombre)
```

## Description

`SPI_execute_plan` exécute un plan préparé par `SPI_prepare`. `read_only` et `nombre` ont la même interprétation que dans `SPI_execute`.

## Arguments

`void * plan`
: plan d'exécution (retourné par `SPI_prepare`)

`Datum *valeurs`
: Un tableau des vraies valeurs des paramètres. Doit avoir la même longueur que le nombre d'arguments du plan.

`const char * nulls`
: Un tableau décrivant les paramètres nuls. Doit avoir la même longueur que le nombre d'arguments du plan. `n` indique une valeur NULL (l'entrée correspondante dans `valeurs` sera ignorée) ; un espace indique une valeur non NULL (l'entrée correspondante dans `valeurs` est valide).

: Si `nulls` est `NULL`, alors `SPI_executeplan` part du principe qu'aucun paramètre n'est nul.

`bool read_only`
: `true` pour une exécution en lecture seule

`int nombre`
: nombre maximum de lignes à traiter ou à renvoyer

## Valeur de retour

La valeur de retour est la même que pour `SPI_execute` avec les résultats d'erreurs (négatif) possibles :

`SPI_ERROR_ARGUMENT`
: si `plan` est `NULL` ou `nombre` est inférieur à 0

`SPI_ERROR_PARAM`
: si `valeurs` est `NULL` et `plan` est préparé avec des paramètres

`SPI_processed` et `SPI_tuptable` sont positionnés comme dans `SPI_execute` en cas de réussite.

## Notes

Si l'un des objets (une table, une fonction, etc.) référencés par le plan préparé est effacé pendant la session, alors le résultat de `SPI_executeplan` pour ce plan sera imprévisible.

# SPI_execp

## Nom

SPI_execp −− exécute un plan en mode lecture/écriture

## Synopsis

```
int SPI_execp(void * plan, Datum * valeurs, const char * nulls, int nombre)
```

## Description

`SPI_execp` est identique à `SPI_execute_plan`, mais le paramètre `read_only` de ce dernier vaut toujours `false`.

## Arguments

`void * plan`
: plan d'exécution (renvoyé par `SPI_prepare`)

`Datum * values`
: Un tableau des vraies valeurs de paramètre. Doit avoir la même longueur que le nombre d'arguments du plan.

`const char * nulls`
: Un tableau décrivant les paramètres NULL. Doit avoir la même longueur que le nombre d'arguments du plan. `n` indique une valeur NULL (l'entrée dans `valeurs` sera ignorée) ; un espace indique une valeur non NULL (l'entrée dans `valeurs` est valide).

  Si `nulls` est `NULL`, alors `SPI_execp` suppose qu'aucun paramètre n'est NULL.

`int nombre`
: nombre maximum de lignes à traiter ou à renvoyer

## Valeur de retour

Voir `SPI_execute_plan`.

`SPI_processed` et `SPI_tuptable` sont initialisées comme dans `SPI_execute` en cas de succès.

# SPI_cursor_open

## Nom

SPI_cursor_open −− met en place un curseur en utilisant un plan créé avec `SPI_prepare`

## Synopsis

```
Portal SPI_cursor_open(const char * nom, void * plan,
Datum * valeurs, const char * nulls,
bool read_only)
```

## Description

`SPI_cursor_open` met en place un curseur (en interne, un portail) qui lancera un plan préparé par `SPI_prepare`. Les paramètres ont la même signification que les paramètres correspondant à `SPI_execute_plan`.

Utiliser un curseur au lieu de lancer le plan directement a deux avantages. Premièrement, les lignes de résultats peuvent être récupérées un certain nombre à la fois, évitant la saturation de mémoire pour les requêtes qui retournent trop de lignes. Deuxièmement, un portail peut survivre à la procédure courante (elle peut, en fait, vivre jusqu'à la fin de la transaction courante). Renvoyer le nom du portail à l'appelant de la procédure donne un moyen de retourner une série de ligne en tant que résultat.

## Arguments

`const char * nom`
: nom pour le portail ou `NULL` pour laisser le système choisir un nom

`void * plan`
: plan d'exécution (retourné par `SPI_prepare`)

`Datum * valeurs`
: Un tableau des valeurs de paramètres actuelles. Doit avoir la même longueur que le nombre d'arguments du plan.

`const char *nulls`
: Un tableau décrivant quels paramètres sont NULL. Doit avoir la même longueur que le nombre d'arguments du plan. `n` indique une valeur NULL (l'entrée correspondante dans `values` sera ignorée) ; un espace indique une valeur non NULL (l'entrée correspondante dans `valeurs` est valide).

  Si `nulls` est `NULL`, alors `SPI_cursor_open` part du principe qu'aucun paramètre n'est nul.

`bool read_only`
: `true` pour les exécutions en lecture seule

# Valeur de retour

Pointeur vers le portail contenant le curseur ou `NULL` en cas d'erreur

# SPI_cursor_find

## Nom

SPI_cursor_find −− recherche un curseur existant par nom

## Synopsis

```
Portal SPI_cursor_find(const char * nom)
```

## Description

`SPI_cursor_find` recherche un portail par nom. Ceci est principalement utile pour résoudre un nom de curseur renvoyé en tant que texte par une autre fonction.

## Arguments

`const char * nom`
: nom du portail

## Valeur de retour

Pointeur vers le portail portant le nom spécifié ou `NULL` si aucun n'a été trouvé

# SPI_cursor_fetch

## Nom

SPI_cursor_fetch −− extrait des lignes à partir d'un curseur

## Synopsis

```
void SPI_cursor_fetch(Portal portal, bool
sensavant, int nombre)
```

## Description

`SPI_cursor_fetch` extrait des lignes à partir d'un curseur. Ceci est équivalent à la commande SQL `FETCH`.

## Arguments

`Portal portal`
: portail contenant le curseur

`bool sensavant`
: vrai pour une extraction en avant, faux pour une extraction en arrière

`int nombre`
: nombre maximum de lignes à récupérer

## Valeur de retour

`SPI_processed` et `SPI_tuptable` sont positionnés comme dans `SPI_execute` en cas de réussite.

# SPI_cursor_move

## Nom

SPI_cursor_move −− déplace un curseur

## Synopsis

```
void SPI_cursor_move(Portal portal, bool
sensavant, int nombre)
```

## Description

`SPI_cursor_move` saute un certain nombre de lignes dans un curseur. Ceci est équivalent à la commande SQL `MOVE`.

## Arguments

`Portal portal`
: portail contenant le curseur

`bool sensavant`
: vrai pour un saut en avant, faux pour un saut en arrière

`int nombre`
: nombre maximum de lignes à déplacer

# SPI_cursor_close

## Nom

SPI_cursor_close −− ferme un curseur

## Synopsis

```
void SPI_cursor_close(Portal portal)
```

## Description

`SPI_cursor_close` ferme un curseur créé précédemment et libère la mémoire du portail.

Tous les curseurs ouverts sont fermés automatiquement à la fin de la transaction. `SPI_cursor_close` n'a besoin d'être invoqué que s'il est désirable de libérer les ressources plus tôt.

## Arguments

`Portal portal`
: portail contenant le curseur

# SPI_saveplan

## Nom

SPI_saveplan −− sauvegarde un plan

## Synopsis

```
void * SPI_saveplan(void * plan)
```

## Description

`SPI_saveplan` sauvegarde un plan validé (préparé par `SPI_prepare`) dans une zone de mémoire protégée d'une éventuelle libération par `SPI_finish` et par le gestionnaire de transaction et retourne le pointeur vers le plan sauvegardé. Ceci vous donne la possibilité de réutiliser les plans préparés lors des invocations suivantes de votre procédure dans la session courante. Vous pouvez sauvegarder le pointeur retourné dans une variable locale. Vérifiez toujours si ce pointeur est `NULL` ou pas lors de la préparation d'un plan où lors de l'utilisation d'un plan déjà préparé dans `SPI_executeplan`.

## Arguments

`void * plan`
: le plan à sauvegarder

## Valeur de retour

Pointeur vers le plan sauvegardé ; `NULL` en cas d'échec. En cas d'erreur, `SPI_result` est positionnée comme suit :

`SPI_ERROR_ARGUMENT`
: si `plan` est `NULL`

`SPI_ERROR_UNCONNECTED`
: si appelé d'une procédure non connectée

## Notes

Si l'un des objets (une table, une fonction, etc.) référencés par le plan préparé est effacé pendant la session, alors le résultat de `SPI_executeplan` pour ce plan sera imprévisible.

# 39.2. Fonctions de support d'interface

**Table des matières**



Les fonctions décrites ici donnent une interface pour extraire les informations des séries de résultats renvoyés par `SPI_execute` et les autres fonctions SPI.

Toutes les fonctions décrites dans cette section peuvent être utilisées par toutes les procédures, connectées et non connectées.

# SPI_fname

## Nom

SPI_fname −− détermine le nom de colonne pour le numéro de colonne spécifiée

## Synopsis

```
char * SPI_fname(TupleDesc descligne, int
nocolonne)
```

## Description

`SPI_fname` retourne une copie du nom de colonne d'une colonne spécifiée. (Vous pouvez utiliser `pfree` pour libérer la copie du nom lorsque vous n'en avez plus besoin.)

## Arguments

`TupleDesc descligne`
: description de rangée d'entrée

`int nocolonne`
: nombre de colonne (le compte commence à 1)

## Valeur de retour

Le nom de colonne ; `NULL` si `nocolonne` est hors de portée. `SPI_result` est positionnée à `SPI_ERROR_NOATTRIBUTE` en cas d'échec.

# SPI_fnumber

## Nom

SPI_fnumber −− détermine le numéro de colonne pour le nom de colonne spécifiée

## Synopsis

```
int SPI_fnumber(TupleDesc descligne, const char *
nom_colonne)
```

## Description

`SPI_fnumber` renvoie le numéro de colonne pour la colonne portant le nom spécifié.

Si `nom_colonne` réfère à une colonne système (c'est−à−dire `oid`), alors le numéro de colonne négatif approprié sera renvoyé. L'appelant devra faire attention à tester la valeur de retour pour égalité exacte à `SPI_ERROR_NOATTRIBUTE` pour détecter une erreur ; tester le résultat pour une valeur inférieur ou égale à 0 n'est pas correcte sauf si les colonnes systèmes doivent être rejetées.

## Arguments

`TupleDesc descligne`
: description de la rangée d'entrée

`const char * nom_colonne`
: nom de colonne

## Valeur de retour

Numéro de colonne (le compte commence à 1) ou `SPI_ERROR_NOATTRIBUTE` si la colonne nommée n'est trouvée.

# SPI_getvalue

## Nom

SPI_getvalue −− renvoie la valeur de chaîne de la colonne spécifiée

## Synopsis

```
char * SPI_getvalue(HeapTuple ligne, TupleDesc
descligne, int nocolonne)
```

## Description

`SPI_getvalue` retourne la représentation chaîne de la valeur de la colonne spécifiée.

Le résultat est retourné en mémoire allouée en utilisant `palloc`. (Vous pouvez utiliser `pfree` pour libérer la mémoire lorsque vous n'en avez plus besoin.)

## Arguments

`HeapTuple ligne`
: ligne d'entrée à examiner

`TupleDesc descligne`
: description de ligne d'entrée

`int nocolonne`
: numéro de colonne (le compte commence à 1)

## Valeur de retour

Valeur de colonne ou `NULL` si la colonne est NULL, `nocolonne` est hors de portée (`SPI_result` est positionnée à `SPI_ERROR_NOATTRIBUTE`) ou aucune fonction de sortie n'est disponible (`SPI_result` est positionnée à `SPI_ERROR_NOOUTFUNC`).

# SPI_getbinval

## Nom

SPI_getbinval −− retourne la valeur binaire de la colonne spécifiée

## Synopsis

```
Datum SPI_getbinval(HeapTuple ligne, TupleDesc
descligne, int nocolonne, bool *
estNULL)
```

## Description

`SPI_getbinval` retourne la valeur de la colonne spécifiée dans le format interne (en tant que type `Datum`).

Cette fonction n'alloue pas de nouvel espace pour le datum. Dans le cas d'un type de données passé par référence, la valeur de retour sera un pointeur dans la ligne passée.

## Arguments

`HeapTuple ligne`
: ligne d'entrée à examiner

`TupleDesc descligne`
: description de la ligne d'entrée

`int nocolonne`
: numéro de colonne (le compte commence à 1)

`bool * estNULL`
: indique une valeur NULL dans la colonne

## Valeur de retour

La valeur binaire de la colonne est retournée. La variable vers laquelle pointe `estNULL` est positionnée à vrai si la colonne est NULL et sinon à faux.

`SPI_result` est positionnée à `SPI_ERROR_NOATTRIBUTE` en cas d'erreur.

# SPI_gettype

## Nom

SPI_gettype −− retourne le nom du type de donnée de la colonne spécifiée

## Synopsis

```
char * SPI_gettype(TupleDesc descligne, int
nocolonne)
```

## Description

`SPI_gettype` retourne une copie du nom du type de donnée de la colonne spécifiée. (Vous pouvez utiliser `pfree` pour libérer la copie du nom lorsque vous n'en avez plus besoin.)

## Arguments

`TupleDesc descligne`
: description de ligne d'entrée

`int nocolonne`
: numéro de colonne (le compte commence à 1)

## Valeur de retour

Le nom de type de donnée de la colonne spécifiée ou `NULL` en cas d'erreur. `SPI_result` est positionnée à `SPI_ERROR_NOATTRIBUTE` en cas d'erreur.

# SPI_gettypeid

## Nom

SPI_gettypeid −− retourne l'OID de type de donnée de la colonne spécifiée

## Synopsis

```
Oid SPI_gettypeid(TupleDesc descligne, int
nocolonne)
```

## Description

`SPI_gettypeid` retourne l'OID du type de donnée de la colonne spécifiée.

## Arguments

`TupleDesc descligne`
: description de ligne d'entrée

`int nocolonne`
: Numéro de colonne (le compte commence à 1)

## Valeur de retour

L'OID du type de donnée de la colonne spécifiée ou `InvalidOid` en cas d'erreur. En cas d'erreur, `SPI_result` est positionnée à `SPI_ERROR_NOATTRIBUTE`.

# SPI_getrelname

## Nom

SPI_getrelname −− retourne le nom de la relation spécifiée

## Synopsis

```
char * SPI_getrelname(Relation rel)
```

## Description

`SPI_getrelname` retourne une copie du nom de la relation spécifiée. (Vous pouvez utiliser `pfree` pour libérer la copie du nom lorsque vous n'en avez plus besoin.)

## Arguments

`Relation rel`
: relation d'entrée

## Valeur de retour

Le nom de la relation spécifiée.

---

# 39.3. Gestion de la mémoire

**Table des matières**



PostgreSQL alloue de la mémoire dans des *contextes mémoire*, qui donne une méthode pratique pour gérer les allocations faîtes dans plusieurs endroits qui ont besoin de vivre pour des durées différentes. Détruire un contexte libère toute la mémoire qui y était allouée. Donc, il n'est pas nécessaire de garder la trace des objets individuels pour éviter les fuites de mémoire ; à la place, seul un petit nombre de contextes doivent être gérés. `palloc` et les fonctions liées allouent de la mémoire du contexte << current >>.

`SPI_connect` crée un nouveau contexte mémoire et le rend courant. `SPI_finish` restaure le contexte mémoire précédant et détruit le contexte crée par `SPI_connect`. Ces actions garantissent que les allocations temporaires de mémoire faîtes dans votre procédure soient réclamées lors de la sortie de la procédure, évitant les fuites de mémoire.

Par contre, si votre procédure a besoin de renvoyer un objet dans de la mémoire allouée (tel que la valeur d'un type de donné passé par référence), vous ne pouvez pas allouer cette mémoire en utilisant `palloc`, au moins pas tant que vous êtes connecté à SPI. Si vous essayez, l'objet sera désalloué par `SPI_finish` et votre procédure ne fonctionnera pas de manière fiable. Pour résoudre ce problème, utilisez `SPI_palloc` pour allouer de la mémoire pour votre objet de retour. `SPI_palloc` alloue de la mémoire dans le << contexte de mémoire courant >>, c'est–à–dire, le contexte de mémoire qui était courant lorsque `SPI_connect` a été appelée, ce qui est précisément le bon contexte pour une valeur renvoyée à partir de votre procédure.

Si `SPI_palloc` est appelé pendant que la procédure n'est pas connectée à SPI, alors il agit de la même manière qu'un `palloc` normal. Avant qu'une procédure ne se connecte au gestionnaire SPI, toutes les allocations faîtes par la procédure via `palloc` ou par une fonction utilitaire SPI sont faîtes dans le contexte de mémoire courant.

Quand `SPI_connect` est appelée, le contexte privé de la procédure, qui est créée par `SPI_connect`, est nommé le contexte courant. Toute allocation faîte par `palloc`, `repalloc` ou une fonction utilitaire SPI (à part pour `SPI_copytuple`, `SPI_returntuple`, `SPI_modifytuple`, et `SPI_palloc`) sont faîtes dans ce contexte. Quand une procédure se déconnecte du gestionnaire SPI (via `SPI_finish`), le contexte courant est restauré au contexte de mémoire courant et toutes les allocations faîtes dans le contexte de mémoire de la procédure sont libérées et ne peuvent plus être utilisées.

Toutes les fonctions couvertes dans cette section peuvent être utilisées par des procédures connectées comme non connectées. Dans une procédure non connectée, elles agissent de la même façon que les fonctions serveur sous–jacentes (`palloc`, etc.).

# SPI_palloc

## Nom

SPI_palloc −− alloue de la mémoire dans le contexte de mémoire courant

## Synopsis

```
void * SPI_palloc(Size taille)
```

## Description

`SPI_palloc` alloue de la mémoire dans le contexte de mémoire courant.

## Arguments

`Size taille`
: taille en octets du stockage à allouer

## Valeur de retour

Pointeur vers le nouvel espace de stockage de la taille spécifiée

# SPI_repalloc

## Nom

SPI_repalloc −− ré−alloue de la mémoire dans le contexte de mémoire courant

## Synopsis

```
void * SPI_repalloc(void * pointeur, Size
taille)
```

## Description

`SPI_repalloc` change la taille d'un segment de mémoire alloué auparavant en utilisant `SPI_palloc`.

Cette fonction n'est plus différente du `repalloc` standard. Elle n'est gardée que pour la compatibilité du code existant.

## Arguments

`void * pointeur`
: pointeur vers l'espace de stockage à modifier

`Size taille`
: taille en octets du stockage à allouer

## Valeur de retour

Pointeur vers le nouvel espace de stockage de taille spécifiée avec le contenu copié de l'espace existant

# SPI_pfree

## Nom

SPI_pfree −− libère de la mémoire dans le contexte de mémoire courant

## Synopsis

```
void SPI_pfree(void * pointeur)
```

## Description

`SPI_pfree` libère de la mémoire allouée auparavant par `SPI_palloc` ou `SPI_repalloc`.

Cette fonction n'est plus différente du `pfree` standard. Elle n'est conservée que pour la compatibilité du code existant.

## Arguments

`void * pointeur`
: pointeur vers l'espace de stockage à libérer

# SPI_copytuple

## Nom

SPI_copytuple −− effectue une copie d'une ligne dans le contexte de mémoire courant

## Synopsis

```
HeapTuple SPI_copytuple(HeapTuple ligne)
```

## Description

`SPI_copytuple` crée une copie d'une ligne dans le contexte de mémoire courant. Ceci est normalement utilisé pour renvoyer une ligne modifiée à partir d'un déclencheur. Dans une fonction déclarée pour renvoyer un type composite, utilisez `SPI_returntuple` à la place.

## Arguments

`HeapTuple ligne`
: ligne a copier

## Valeur de retour

la ligne copiée ; `NULL` seulement si `ligne` est `NULL`

# SPI_returntuple

## Nom

SPI_returntuple −− prépare le renvoi d'une ligne en tant que Datum

## Synopsis

```
HeapTupleHeader SPI_returntuple(HeapTuple ligne, TupleDesc descligne)
```

## Description

`SPI_returntuple` crée une copie d'une ligne dans le contexte de l'exécuteur supérieur, la renvoyant sous la forme d'une ligne de type `Datum`. Le pointeur renvoyé a seulement besoin d'être converti en `Datum` via `PointerGetDatum` avant d'être renvoyé.

Notez que ceci devrait être utilisé pour les fonctions qui déclarent renvoyer des types composites. Ce n'est pas utilisé pour les déclencheurs ; utilisez après tout pour renvoyer une ligne modifiée dans un déclencheur.

## Arguments

`HeapTuple ligne`
: ligne à copier

`TupleDesc descligne`
: descripteur pour la ligne (passez le même descripteur chaque fois pour un cache plus efficace)

## Valeur de retour

`HeapTupleHeader` pointant vers la ligne copiée ; `NULL` seulement si `row` ou `rowdesc` est `NULL`

# SPI_modifytuple

## Nom

SPI_modifytuple −− crée une ligne en remplaçant les champs sélectionnés d'une ligne donnée

## Synopsis

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple
ligne, ncols,
nocolonne, Datum * valeurs, const
char * nulls)
```

## Description

`SPI_modifytuple` crée une nouvelle ligne en retirant les nouvelles valeurs pour les colonnes sélectionnées et en copiant les colonnes de la ligne d'origine à d'autres positions. La ligne d'entrée n'est pas modifiée.

## Arguments

`Relation rel`
: Utilisé seulement en tant que source du descripteur de ligne pour la ligne. (Passez une relation plutôt qu'un descripteur de ligne est une erreur.)

`HeapTuple ligne`
: rangée à modifier

`int ncols`
: nombre de numéros de colonnes dans le tableau `nocolonne`

`int * nocolonne`
: tableau des numéros des colonnes à modifier (le numéro des colonnes commence à 1)

`Datum * valeurs`
: nouvelles valeurs pour les colonnes spécifiées

`const char * nulls`
: quelles nouvelles valeurs sont NULL, si elles existent (voir `SPI_executeplan` pour le format)

## Valeur de retour

nouvelle ligne avec modifications, allouée dans le contexte de mémoire courant ; `NULL` seulement si `ligne` est `NULL`

En cas d'erreur, `SPI_result` est positionnée comme suit :

`SPI_ERROR_ARGUMENT`

si `rel` est `NULL` ou si `ligne` est `NULL` ou si `ncols` est inférieur ou égal à 0 ou si `nocolonne` est `NULL` ou si `valeurs` est `NULL`.

`SPI_ERROR_NOATTRIBUTE`

si `nocolonne` contient un numéro de colonne invalide (inférieur ou égal à 0 ou supérieur au numéro de colonne dans `ligne`)

# SPI_freetuple

## Nom

SPI_freetuple −− libère une ligne allouée dans le contexte de mémoire courant

## Synopsis

```
void SPI_freetuple(HeapTuple ligne)
```

## Description

`SPI_freetuple` libère une rangée allouée auparavant dans le contexte de mémoire courant.

Cette fonction n'est plus différente du standard `heap_freetuple`. Elle est gardé juste pour la compatibilité du code existant.

## Arguments

`HeapTuple ligne`
: rangée à libérer

# SPI_freetuptable

## Nom

SPI_freetuptable −− libère une série de lignes créée par `SPI_execute` ou une fonction semblable

## Synopsis

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

## Description

`SPI_freetuptable` libère une série de lignes créée auparavant par une fonction d'exécution de commandes SPI, tel que `SPI_execute`. Par conséquent, cette fonction est souvent appelée avec la variable globale `SPI_tupletable` comme argument.

Cette fonction est utile si une procédure SPI a besoin d'exécuter de multiples commandes et ne veut pas garder les résultats de commandes précédentes en mémoire jusqu'à sa fin. Notez que toute série de lignes non libérées seront libérées quand même lors de `SPI_finish`.

## Arguments

`SPITupleTable * tuptable`
: pointeur vers la série de lignes à libérer

# SPI_freeplan

## Nom

SPI_freeplan −− libère un plan sauvegardé auparavant

## Synopsis

```
int SPI_freeplan(void *plan)
```

## Description

`SPI_freeplan` libère un plan de commandes d'exécution retourné auparavant par `SPI_prepare` ou sauvegardé par `SPI_saveplan`.

## Arguments

`void * plan`
: pointeur vers le plan à libérer

## Valeur de retour

`SPI_ERROR_ARGUMENT` si `plan` est `NULL`.

---

# 39.4. Visibilité des modifications de données

Les règles suivantes gouvernent la visibilité des modifications de données dans les fonctions qui utilisent SPI (ou tout autre fonction C) :

- Pendant l'exécution de la commande SQL, toutes modification de données faite par la commande sont invisibles à la commande. Par exemple, dans la commande

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  les lignes insérées sont invisibles à la partie `SELECT`.
- Les modifications effectuées par une commande C sont visibles par toutes les commandes qui sont lancées après C, peu importe qu'elles soient lancées à l'intérieur de C (pendant l'exécution de C) ou après que C soit terminée.
- Les commandes exécutées via SPI à l'intérieur d'une fonction appelée par une commande SQL (soit une fonction ordinaire soit un déclencheur) suivent une des règles ci−dessus suivant le commutateur lecture/écriture passé à SPI. Les commandes exécutées en mode lecture seule suivent la première règle : elles ne peuvent pas voir les modifications de la commande appelante. Les commandes exécutées en mode lecture/écriture suivent la deuxième règle : elles peuvent voir toutes les

modifications réalisées jusqu'à maintenant.

- Tous les langages standards de procédures initialisent le mode lecture/écriture suivant l'attribut de volatibilité de la fonction. Les commandes des fonctions `STABLE` et `IMMUTABLE` sont réalisées en mode lecture seule alors que les fonctions `VOLATILE` sont réalisées en mode lecture/écriture. Alors que les auteurs de fonctions C sont capables de violer cette convention, il est peu probable que cela soit une bonne idée de le faire.

La section suivante contient un exemple qui illustre l'application de ces règles.

## 39.5. Exemples

Cette section contient un exemple très simple d'utilisation de SPI. La procédure `execq` prend une commande SQL comme premier argument et un compteur de lignes comme second, exécute la commande en utilisant `SPI_exec` et renvoie le nombre de lignes qui ont été traitées par la commande. Vous trouverez des exemples plus complexes pour SPI dans l'arborescence source dans `src/test/regress/regress.c` et dans `contrib/spi`.

```
#include "executor/spi.h"

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
    char *command;
    int ret;
    int proc;

    /* Convertir l'objet texte donné en chaîne C */
    command = DatumGetCString(DirectFunctionCall1(textout,
                                                  PointerGetDatum(sql)));

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;
    /*
     * Si ceci est un SELECT et que des lignes ont été récupérées,
     * alors les lignes sont affichées via elog(INFO).
     */
    if (ret == SPI_OK_SELECT && SPI_processed > 0)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i, j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen (buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog (INFO, "EXECQ: %s", buf);
        }
```

```
    }

    SPI_finish();
    pfree(command);

    return (proc);
}
```

(Cette fonction utilisera la convention d'appel version 0 pour rendre l'exemple plus simple à comprendre. Dans des applications réelles, vous devriez utiliser la nouvelle interface version 1.)

Voici comment déclarer la fonction après l'avoir compilée en une bibliothèque partagée :

```
CREATE FUNCTION execq(text, integer) RETURNS integer
    AS 'filename'
    LANGUAGE C;
```

Voici une session d'exemple :

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 167631 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- inséré par execq
INFO:  EXECQ:  1    -- retourné par execq et inséré par l'INSERT précédant

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2    -- 0 + 2, une seule ligne insérée - comme spécifié

 execq
-------
     3              -- 10 est la valeur max seulement, 3 est le nombre réel de rangées
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 167712 1
=> SELECT * FROM a;
 x
---
 1                  -- aucune rangée dans a (0) + 1
```

```
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  0
INSERT 167713 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- il y a une rangée dans a + 1
(2 rows)

-- Ceci montre la règle de visibilité de modifications de données :

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 rangées * 1 (x dans la première rangée)
 6                  -- 3 rangées (2 + 1 juste insérée) * 2 (x dans la deuxième rangée)
(4 rows)               ^^^^^^
                       rangées visible à execq() dans des invocations différentes
```

# VI. Référence

Les points abordés dans la référence sont supposés fournir d'une manière concise un résumé précis, complet et formel sur leurs sujets respectifs. Des informations complémentaires sur l'utilisation de PostgreSQL sont présentées dans d'autres parties de ce livre sous la forme de descriptions, de tutoriels ou d'exemples. Vous pourrez vous reporter à la liste de références croisées que vous pourrez trouver sur chaque page de référence.

Les entrées de références sont également disponibles sur les pages << man >> traditionnelles.

**Table des matières**

CREATE RULE −− définit une nouvelle règle de réécriture
CREATE SCHEMA −− définit un nouveau schéma
CREATE SEQUENCE −− définit un nouveau générateur de séquence
CREATE TABLE −− définit une nouvelle table
CREATE TABLE AS −− définit une nouvelle table à partir des résultats d'une requête
CREATE TABLESPACE −− définit un nouvel espace logique
CREATE TRIGGER −− définit un nouveau déclencheur
CREATE TYPE −− définit un nouveau type de donnée
CREATE USER −− définit un nouveau compte utilisateur dans la base de données
CREATE VIEW −− définit une nouvelle vue
DEALLOCATE −− désalloue une instruction préparée
DECLARE −− définit un curseur
DELETE −− supprime les lignes d'une table
DROP AGGREGATE −− supprime une fonction d'agrégat
DROP CAST −− supprime un transtypage
DROP CONVERSION −− supprime une conversion
DROP DATABASE −− supprime une base de données
DROP DOMAIN −− supprime un domaine
DROP FUNCTION −− supprime une fonction
DROP GROUP −− supprime un groupe d'utilisateurs
DROP INDEX −− supprime un index
DROP LANGUAGE −− supprime un langage procédural
DROP OPERATOR −− supprime un opérateur
DROP OPERATOR CLASS −− supprime une classe d'opérateur
DROP RULE −− supprime une règle de réécriture
DROP SCHEMA −− supprime un schéma
DROP SEQUENCE −− supprime une séquence
DROP TABLE −− supprime une table
DROP TABLESPACE −− supprime un espace logique
DROP TRIGGER −− supprime un déclencheur
DROP TYPE −− supprime un type de données
DROP USER −− supprime un compte utilisateur de bases de données
DROP VIEW −− supprime une vue
END −− valide la transaction en cours
EXECUTE −− exécute une instruction préparée
EXPLAIN −− affiche le plan d'exécution d'une instruction
FETCH −− récupère des lignes d'une requête en utilisant un curseur
GRANT −− définit les droits d'accès
INSERT −− insère de nouvelles lignes dans une table
LISTEN −− écoute une notification
LOAD −− charge ou décharge une bibliothèque partagée
LOCK −− verrouille une table
MOVE −− positionne un curseur
NOTIFY −− génère une notification
PREPARE −− prépare une instruction pour exécution
REINDEX −− reconstruit les index
RELEASE SAVEPOINT −− détruit un point de sauvegarde défini précédemment
RESET −− remet un paramètre d'exécution à sa valeur par défaut
REVOKE −− supprime les droits d'accès
ROLLBACK −− annule la transaction en cours
ROLLBACK TO SAVEPOINT −− annule les instructions depuis un point de sauvegarde

# I. Commandes SQL

Cette partie contient les informations de référence pour les commandes SQL reconnues par PostgreSQL. Généralement, on désigne par << SQL >> le langage ; toute information sur la structure et la compatibilité standard de chaque commande peut être retrouvée respectivement sur les pages de références.

**Table des matières**

CREATE TABLE AS −− définit une nouvelle table à partir des résultats d'une requête
CREATE TABLESPACE −− définit un nouvel espace logique
CREATE TRIGGER −− définit un nouveau déclencheur
CREATE TYPE −− définit un nouveau type de donnée
CREATE USER −− définit un nouveau compte utilisateur dans la base de données
CREATE VIEW −− définit une nouvelle vue
DEALLOCATE −− désalloue une instruction préparée
DECLARE −− définit un curseur
DELETE −− supprime les lignes d'une table
DROP AGGREGATE −− supprime une fonction d'agrégat
DROP CAST −− supprime un transtypage
DROP CONVERSION −− supprime une conversion
DROP DATABASE −− supprime une base de données
DROP DOMAIN −− supprime un domaine
DROP FUNCTION −− supprime une fonction
DROP GROUP −− supprime un groupe d'utilisateurs
DROP INDEX −− supprime un index
DROP LANGUAGE −− supprime un langage procédural
DROP OPERATOR −− supprime un opérateur
DROP OPERATOR CLASS −− supprime une classe d'opérateur
DROP RULE −− supprime une règle de réécriture
DROP SCHEMA −− supprime un schéma
DROP SEQUENCE −− supprime une séquence
DROP TABLE −− supprime une table
DROP TABLESPACE −− supprime un espace logique
DROP TRIGGER −− supprime un déclencheur
DROP TYPE −− supprime un type de données
DROP USER −− supprime un compte utilisateur de bases de données
DROP VIEW −− supprime une vue
END −− valide la transaction en cours
EXECUTE −− exécute une instruction préparée
EXPLAIN −− affiche le plan d'exécution d'une instruction
FETCH −− récupère des lignes d'une requête en utilisant un curseur
GRANT −− définit les droits d'accès
INSERT −− insère de nouvelles lignes dans une table
LISTEN −− écoute une notification
LOAD −− charge ou décharge une bibliothèque partagée
LOCK −− verrouille une table
MOVE −− positionne un curseur
NOTIFY −− génère une notification
PREPARE −− prépare une instruction pour exécution
REINDEX −− reconstruit les index
RELEASE SAVEPOINT −− détruit un point de sauvegarde défini précédemment
RESET −− remet un paramètre d'exécution à sa valeur par défaut
REVOKE −− supprime les droits d'accès
ROLLBACK −− annule la transaction en cours
ROLLBACK TO SAVEPOINT −− annule les instructions depuis un point de sauvegarde
SAVEPOINT −− définit un nouveau point de sauvegarde à l'intérieur de la transaction en cours
SELECT −− récupère des lignes d'une table ou d'une vue
SELECT INTO −− définit une nouvelle table à partir des résultats d'une requête
SET −− change un paramètre d'exécution

SET CONSTRAINTS −− initialise le mode de vérification de contrainte de la transaction en cours
SET SESSION AUTHORIZATION −− Initialise l'identifiant de l'utilisateur de la session et l'identifiant de l'utilisateur courant de la session courante
SET TRANSACTION −− initialise les caractéristiques de la transaction actuelle
SHOW −− affiche la valeur d'un paramètre d'exécution
START TRANSACTION −− débute un bloc de transaction
TRUNCATE −− vide une table
UNLISTEN −− arrête l'écoute d'une notification
UPDATE −− met à jour les lignes d'une table
VACUUM −− récupère l'espace inutilisé et, optionnellement, analyse une base

# ABORT

## Nom

ABORT −− Interrompt la transaction en cours

## Synopsis

```
ABORT [ WORK | TRANSACTION ]
```

## Description

`ABORT` annule la transaction en cours et entraine l'annulation de toutes les mises à jour effectuées par la transaction. Cette commande a un comportement identique à la commande SQL *ROLLBACK*, elle est présente uniquement pour des raisons historiques.

## Paramètres

`WORK`
`TRANSACTION`
: Mots−clé optionnels. Ils n'ont aucun effet.

## Notes

Utilisez *COMMIT* pour terminer avec succès une transaction.

Lancer `ABORT` en dehors d'une transaction ne cause aucun dégât, mais provoquera un message d'avertissement.

## Exemples

Pour annuler toutes les modifications:

```
ABORT;
```

## Compatibilité

Cette commande est une extension PostgreSQL présente pour des raisons historiques. `ROLLBACK` est la commande équivalente du standard SQL.

# Voir aussi

*BEGIN*, *COMMIT*, *ROLLBACK*

# ALTER AGGREGATE

## Nom

ALTER AGGREGATE −− change la définition d'une fonction d'agrégat

## Synopsis

```
ALTER AGGREGATE nom ( type
) RENAME TO nouveau nom
ALTER AGGREGATE nom (type
) OWNER TO nouveau propriétaire
```

## Description

`ALTER AGGREGATE` change la définition d'une fonction d'agrégat.

## Paramètres

*nom*
: Le nom d'une fonction d'agrégat existante.

*type*
: L'argument de donnée type d'une fonction d'agrégat ou `*` si la fonction accepte divers données type.

*nouveau nom*
: Le nouveau nom de la fonction d'agrégat.

*nouveau propriétaire*
: Le nouveau propriétaire de la fonction d'agrégat. Vous devez être un superutilisateur pour modifier le propriétaire de l'agrégat.

## Exemples

Pour renommer la fonction d'agrégat `mamoyenne` de type `integer` en `ma_moyenne` :

```
ALTER AGGREGATE mamoyenne(integer) RENAME TO ma_moyenne;
```

Pour changer le propriétaire de la fonction d'agrégat `mamoyenne` de type `integer` par `joe` :

```
ALTER AGGREGATE mamoyenne(integer) OWNER TO joe;
```

## Compatibilité

Il n'y a pas de relation `ALTER AGGREGATE` dans le standard SQL.

# Voir aussi

*CREATE AGGREGATE*, *DROP AGGREGATE*

# ALTER CONVERSION

## Nom

ALTER CONVERSION −− change la définition d'une conversion

## Synopsis

```
ALTER CONVERSION nom RENAME TO nouveau nom
ALTER CONVERSION nom OWNER TO nouveau propriéaire
```

## Description

`ALTER CONVERSION` change la définition d'une conversion.

## Paramètres

*nom*
: Le nom d'une conversion existante.

*nouveau nom*
: Le nouveau nom de la conversion.

*nouveau propriétaire*
: Le nouveau propriétaire de la conversion. Pour modifier le propriétaire d'une conversion, vous devez être un superutilisateur.

## Exemples

Pour renommer la conversion `iso_8859_1_to_utf_8` en `latin1_to_unicode` :

```
ALTER CONVERSION iso_8859_1_to_utf_8 RENAME TO latin1_to_unicode;
```

Pour modifier le propriétaire de la conversion `iso_8859_1_to_utf_8` par `joe` :

```
ALTER CONVERSION iso_8859_1_to_utf_8 OWNER TO joe;
```

## Compatibilité

Il n'y a pas d'instruction `ALTER CONVERSION` dans le standard SQL.

# Voir aussi

*CREATE CONVERSION*, *DROP CONVERSION*

# ALTER DATABASE

## Nom

ALTER DATABASE −− modifie une base de données

## Synopsis

```
ALTER DATABASE nom SET
paramètre { TO | = } {
valeur | DEFAULT }
ALTER DATABASE nom RESET
paramètre
ALTER DATABASE nom RENAME TO
nouveau nom
ALTER DATABASE nom OWNER TO
nouveau_propriétaire
```

## Description

`ALTER DATABASE` modifie les attributs d'une base de données.

Les deux premières formes modifient la session par défaut de la variable de configuration de lancement d'une base PostgreSQL. Chaque fois qu'une nouvelle session est démarrée ultérieurement dans cette base, la valeur spécifiée devient la valeur de session par défaut. La valeur par défaut de la base prend le pas sur la configuration présente dans le fichier `postgresql.conf` ou sur celle qui a été reçue de la ligne de commande du `postmaster`. Seul le propriétaire de la base ou le superutilisateur peut changer les valeurs par défaut de la session d'une base. Certaines variables ne peuvent pas être configurées de cette façon ou peuvent seulement être configurées par un superutilisateur.

La troisième forme modifie le nom de la base. Seul le propriétaire ou le superutilisateur peut renommer une base ; le propriétaire doit aussi posséder le droit `CREATEDB`. La base utilisée ne peut pas être renommée. (Connectez−vous à une base différente si vous voulez faire ça).

La quatrième forme change le propriétaire de la base de données. Seul un superutilisateur peut modifier le propriétaire de la base de données.

## Paramètres

*nom*
: Le nom d'une base dont les attributs sont à modifier.

*paramètre*
*valeur*
: Configure la valeur par défaut de la session sur cette base de donnée pour ce paramètre avec cette valeur. Si *valeur* est `DEFAULT` ou, de façon équivalente, `RESET` est utilisé, la variable de configuration de la base est supprimée et la configuration par défaut du système sera récupérée lors

des nouvelles sessions. Utilisez `RESET ALL` pour rafraîchir toutes les configurations spécifiques à cette base de données.

Voir *SET* et Section 16.4 pour plus d'informations sur les paramètres de nom et de valeur admis.

*`nouveau nom`*
Le nouveau nom d'une base.

*`nouveau_propriétaire`*
Le nouveau propriétaire de la base de données.

## Notes

Il est possible de lier une session par défaut à un utilisateur plutôt qu'à une base ; voir *ALTER USER*. Les configurations spécifiques à l'utilisateur prennent le pas sur celles spécifiques à la base s'il y a conflit.

## Exemples

Pour désactiver les parcours d'index par défaut de la base `test` :

```
ALTER DATABASE test SET enable_indexscan TO off;
```

## Compatibilité

La relation `ALTER DATABASE` est une extension PostgreSQL.

## Voir aussi

*CREATE DATABASE*, *DROP DATABASE*, *SET*

# ALTER DOMAIN

## Nom

ALTER DOMAIN −− change la définition d'un domaine

## Synopsis

```
ALTER DOMAIN nom
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN nom
    { SET | DROP } NOT NULL
ALTER DOMAIN nom
    ADD contrainte de domaine
ALTER DOMAIN nom
    DROP CONSTRAINT nom de contrainte [ RESTRICT | CASCADE ]
ALTER DOMAIN nom
    OWNER TO nouveau propriétaire
```

## Description

`ALTER DOMAIN` change la définition d'un domaine existant. Il existe sous plusieurs sous−formes :

SET/DROP DEFAULT
: Ces formes placent ou suppriment la valeur par défaut d'un domaine. Notez que ces valeurs par défaut s'appliquent seulement aux commandes `INSERT` subséquentes ; elles n'affectent pas les lignes d'une table utilisant dèja le domaine.

SET/DROP NOT NULL
: Ces formes modifient un domaine marqué pour permettre les valeurs NULL ou rejettent les valeurs NULL. Vous pouvez faire seulement `SET NOT NULL` quand les colonnes utilisant le domaine contiennent des valeurs non nulles.

ADD *`contrainte de domaine`*
: Cette forme ajoute une nouvelle contrainte à un domaine utilisant la même syntaxe que ***CREATE DOMAIN***. Ceci fonctionnera seulement si toutes les colonnes utilisant le domaine satisfont à la nouvelle contrainte.

DROP CONSTRAINT
: Cette forme supprime les contraintes sur un domaine.

OWNER
: Cette forme change le propriétaire du domaine.

Vous devez être le propriétaire du domaine pour utiliser `ALTER DOMAIN` ; sauf pour `ALTER DOMAIN OWNER`, lequel peut seulement être exécuté par le super−utilisateur.

# Paramètres

*nom*
: Le nom d'un domaine existant à modifier.

*contrainte de domaine*
: Nouvelle contrainte de domaine pour le domaine.

*nom de contrainte*
: Nom d'une contrainte existante à supprimer.

CASCADE
: Supprime automatiquement les objets qui dépendent de la contrainte.

RESTRICT
: Refuse de supprimer la contrainte s'il y a divers objets dépendants. C'est le comportement par défaut.

*nouveau propriétaire*
: Le nom utilisateur du nouveau propriétaire du domaine.

# Exemples

Pour ajouter une contrainte `NOT NULL` à un domaine :

```
ALTER DOMAIN codezip SET NOT NULL;
```

Pour supprimer une contrainte `NOT NULL` d'un domaine :

```
ALTER DOMAIN codezip DROP NOT NULL;
```

Pour ajouter une contrainte de contrôle à un domaine :

```
ALTER DOMAIN codezip ADD CONSTRAINT verif_zip CHECK (char_length(VALUE) = 5);
```

Pour supprimer une contrainte de contrôle d'un domaine :

```
ALTER DOMAIN codezip DROP CONSTRAINT verif_zip;
```

# Compatibilité

La relation `ALTER DOMAIN` est compatible avec SQL:1999, sauf pour la variante `OWNER`, laquelle est une extension PostgreSQL.

# ALTER FUNCTION

## Nom

ALTER FUNCTION −− change la définition d'une fonction

## Synopsis

```
ALTER FUNCTION nom ( [ type [, ...] ] ) RENAME TO nouveau
nom
ALTER FUNCTION nom ( [ type [, ...] ] ) OWNER TO
nouveau propriétaire
```

## Description

`ALTER FUNCTION` change la définition d'une fonction.

## Paramètres

*nom*
: Le nom d'une fonction existante.

*type*
: Le type de donnée d'un argument de la fonction.

*nouveau nom*
: Le nouveau nom de la fonction.

*nouveau propriétaire*
: Le nouveau propriétaire de la fonction. Pour changer le propriétaire d'une fonction, vous devez être un superutilisateur. Notez que, si la fonction est marquée, elle sera ensuite exécutée en tant que ce nouveau propriétaire.

## Exemples

Pour renommer la fonction `sqrt` pour le type `integer` en `square_root` :

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

Pour changer le propriétaire de la fonction `sqrt` pour le type `integer` en `joe` :

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

# Compatibilité

Il existe une relation `ALTER FUNCTION` en SQL standard, mais il ne fournit pas l'option de renomage de la fonction ainsi que celle de changement de propriétaire.

# Voir aussi

*CREATE FUNCTION*, *DROP FUNCTION*

# ALTER GROUP

## Nom

ALTER GROUP −− ajoute ou supprime des utilisateurs d'un groupe

## Synopsis

```
ALTER GROUP nom groupe ADD USER nom utilisateur [, ... ]
ALTER GROUP nom groupe DROP USER nom utilisateur [, ... ]

ALTER GROUP nom groupe RENAME TO nouveau nom
```

## Description

`ALTER GROUP` modifie les utilisateurs d'un groupe.

Les deux premières formes ajoutent ou suppriment des utilisateurs d'un groupe. Seul le super−utilisateur peut utiliser cette commande.

La troisième forme change le nom du groupe. Seul le super−utilisateur peut renommer des groupes.

## Paramètres

*nom groupe*
: Le nom du groupe à modifier.

*nom utilisateur*
: Les utilisateurs qui seront ajoutés ou supprimés du groupe. Les utilisateurs doivent exister ; `ALTER GROUP` ne crée pas et ne supprime pas des utilisateurs.

*nouveau nom*
: Le nouveau nom du groupe.

## Exemples

Ajouter des utilisateurs à un groupe :

```
ALTER GROUP staff ADD USER karl, john;
```

Supprimer des utilisateurs d'un groupe :

```
ALTER GROUP workers DROP USER beth;
```

# Compatibilité

Il n'existe pas de relation `ALTER GROUP` en SQL standard. Le concept de rôle est similaire.

# Voir aussi

*CREATE GROUP*, *DROP GROUP*

# ALTER INDEX

## Nom

ALTER INDEX −− modifie la définition d'un index

## Synopsis

```
ALTER INDEX nom
    action [, ... ]
ALTER INDEX nom
    RENAME TO nouveau_nom

où action fait partie de :

    OWNER TO nouveau_propriétaire
    SET TABLESPACE nom_espacelogique
```

## Description

`ALTER INDEX` modifie la définition d'un index existant. Il existe plusieurs sous−formes :

`OWNER`
: Cette forme modifie le propriétaire d'un index par l'utilisateur spécifié. Ceci peut seulement être fait par un superutilisateur.

`SET TABLESPACE`
: Cette forme change l'espace logique de l'index par l'espace logique spécifié et déplace le(s) fichier(s) de données associé(s) avec l'index pour le nouvel espace logique. Voir aussi *CREATE TABLESPACE*.

`RENAME`
: La forme `RENAME` modifie le nom de l'index. Cela n'a aucun effet sur les données stockées.

Toutes les actions sauf `RENAME` peuvent être combinées dans une liste de plusieurs modifications à appliquer en parallèle.

## Paramètres

*nom*
: Le nom de l'index à modifier (pouvant être qualifié par le nom du schéma).

*nouveau_nom*
: Le nouveau nom de l'index.

*nouveau_propriétaire*
: Le nom du nouveau propriétaire de l'index.

*nom_espacelogique*
: Le nom de l'espace logique dans lequel doit être déplacé l'index.

# Notes

Ces opérations sont aussi possibles en utilisant ***ALTER TABLE***. `ALTER INDEX` est en fait simplement un alias pour les formes d'`ALTER TABLE` s'appliquant aux index.

Modifier toute partie d'un index du catalogue système n'est pas autorisé.

# Exemples

Pour renommer un index existant :

```
ALTER INDEX distributeurs RENAME TO fournisseurs;
```

Pour déplacer un index dans un autre espace logique :

```
ALTER INDEX distributeurs SET TABLESPACE espacelogiquerapide;
```

# Compatibilité

`ALTER INDEX` est une extension PostgreSQL.

# ALTER LANGUAGE

## Nom

ALTER LANGUAGE −− change la définition d'un langage procédural

## Synopsis

```
ALTER LANGUAGE nom RENAME TO nouveau nom
```

## Description

`ALTER LANGUAGE` change la définition d'un langage. La seule fonctionnalité est de renommer la langage. Seul le super−utilisateur peut renommer des langages.

## Paramètres

*nom*
: Nom d'un langage

*nouveau nom*
: Le nouveau nom du langage

## Compatibilité

Il n'existe pas de relation `ALTER LANGUAGE` en SQL standard.

## Voir aussi

*CREATE LANGUAGE*, *DROP LANGUAGE*

# ALTER OPERATOR

## Nom

ALTER OPERATOR −− modifie la définition d'un opérateur

## Synopsis

```
ALTER OPERATOR name ( { typegauche | NONE } , { typedroit | NONE } ) OWNER TO nouveau_propriétair
```

## Description

`ALTER OPERATOR` modifie la définition d'un opérateur. La seule fonctionnalité disponible est de changer le propriétaire d'un opérateur.

## Paramètres

*nom*
: Le nom d'un opérateur existant (pouvant être qualifié par le nom du schéma).

*typegauche*
: Le type de données de l'opérande gauche de l'opérateur ; écrivez `NONE` si l'opérateur n'a pas d'opérande gauche.

*typedroit*
: Le type de données de l'opérande droit de l'opérateur ; écrivez `NONE` si l'opérateur n'a pas d'opérande droit.

*nouveau_propriétaire*
: Le nouveau propriétaire d'un opérateur. Vous devez être superutilisateur pour modifier le propriétaire d'un opérateur.

## Exemples

Modifier le propriétaire d'un opérateur personnalisé `a @@ b` pour le type `text` :

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

## Compatibilité

Il n'existe pas d'instructions `ALTER OPERATOR` dans le standard SQL.

# Voir aussi

*CREATE OPERATOR*, *DROP OPERATOR*

# ALTER OPERATOR CLASS

## Nom

ALTER OPERATOR CLASS −− change la définition d'un opérateur de classe

## Synopsis

```
ALTER OPERATOR CLASS nom USING méthode_indexage RENAME TO nouveau nom
ALTER OPERATOR CLASS nom USING méthode_indexage OWNER TO nouveau propriétaire
```

## Description

`ALTER OPERATOR CLASS` change la définition d'un opérateur de classe.

## Paramètres

*nom*
: Le nom d'un opérateur de classe existant.

*index_method*
: Le nom de la méthode d'indexage pour lequel est fait l'opérateur de classe.

*nouveau nom*
: Le nouveau nom de l'opérateur de classe.

*nouveau propriétaire*
: Le nouveau propriétaire de la classe d'opérateur. Vous devez être un superutilisateur pour changer le propriétaire d'une classe d'opérateur.

## Compatibilité

Il n'existe pas d'instruction `ALTER OPERATOR CLASS` dans le standard SQL.

## Voir aussi

*CREATE OPERATOR CLASS*, *DROP OPERATOR CLASS*

# ALTER SCHEMA

## Nom

ALTER SCHEMA −− change la définition d'un schéma

## Synopsis

```
ALTER SCHEMA nom RENAME TO nouveau nom
ALTER SCHEMA nom OWNER TO nouveau propriétaire
```

## Description

`ALTER SCHEMA` change la définition d'un schéma. Pour renommer un schéma, vous devez être propriétaire du schéma et avoir les droits `CREATE` sur la base. Pour changer le propriétaire d'un schéma, vous devez être un superutilisateur.

## Paramètres

*nom*
: Nom d'un schéma existant

*nouveau nom*
: Nouveau nom du schéma. Le nouveau nom ne peut pas commencer par `pg_` car de tels noms sont réservés aux schémas système.

*newowner*
: Le nouveau propriétaire du schéma.

## Compatibilité

Il n'existe pas de relation `ALTER SCHEMA` en SQL standard.

## Voir aussi

*CREATE SCHEMA*, *DROP SCHEMA*

# ALTER SEQUENCE

## Nom

ALTER SEQUENCE −− modifie la définition d'un générateur de séquence

## Synopsis

```
ALTER SEQUENCE nom [ INCREMENT [ BY
] increment ]
    [ MINVALUE valeurmin | NO
MINVALUE ] [ MAXVALUE valeurmax |
NO MAXVALUE ]
    [ RESTART [ WITH ] debut ] [
CACHE cache ] [ [ NO ] CYCLE ]
```

## Description

`ALTER SEQUENCE` modifie les paramètres d'un générateur de séquence. Tout paramètre non précisé dans la commande `ALTER SEQUENCE` reste à sa valeur précédente.

## Paramètres

*nom*
: Le nom d'une séquence à modifier (pouvant être qualifié avec le nom du schéma).

*increment*
: La clause `INCREMENT BY` *increment* est en option. Une valeur positive créera une séquence ascendante, une valeur négative une séquence descendante. Si non spécifié, la valeur de l'ancien incrément reste.

*valeurmin*
`NO MINVALUE`
: La clause `MINVALUE` *valeurmin* en option détermine la valeur minimale qu'une séquence peut générer. Si `NO MINVALUE` est spécifié, La valeur par défaut, respectivement, de 1 et de $-2^{63}-1$ pour les séquences ascendantes et descendantes sera utilisée. SI aucune option n'est spécifiée, la valeur minimale courante reste.

*valeurmax*
`NO MAXVALUE`
: La clause `MAXVALUE` *valeurmax* en option détermine la valeur maximale pour la séquence. Si `NO MAXVALUE` est spécifié, les valeurs par défaut $2^{63}-1$ et $-1$ pour, respectivement, les séquences ascendantes et descendantes seront utilisées. Si aucune option n'est spécifiée, la valeur maximale courante reste.

*debut*
: La clause `RESTART WITH` *start* en option modifie la valeur actuelle de la séquence.

*cache*

La clause CACHE *cache* active la préallocation des numéros de séquences et leur stockage en mémoire pour un accès plus rapide. La valeur minimale est de 1 (seule une valeur sera générée à la fois, donc pas de cache). Si non spécifiée, l'ancienne valeur reste.

CYCLE

Le mot clé CYCLE en option pourrait être utilisé pour autoriser le cycle de la séquence lorsque *valeurmax* ou *valeurmin* a été atteint par, respectivement, une séquence ascendante ou descendante. Si la limite est atteinte, le prochain numéro généré sera, respectivement, *valeurmin* ou *valeurmax*.

NO CYCLE

Si le mot clé NO CYCLE en option est spécifié, tout appel à nextval une fois que la séquence a atteint sa valeur max renverra une erreur. Si ni CYCLE ni NO CYCLE ne sont spécifiés, l'ancien comportement persiste.

# Exemples

Relancer une séquence appelée serie, à 105 :

```
ALTER SEQUENCE serie RESTART WITH 105;
```

# Notes

Pour éviter de bloquer des transactions concurrentes lors de la demande de numéro de la même séquence, ALTER SEQUENCE n'est jamais annulé ; les modifications prennent effet immédiatement et ne sont pas réversibles.

ALTER SEQUENCE n'affectera pas immédiatement les résultats de nextval sur les serveurs autres que celui qui a lancé la commande, s'ils ont préalloué (caché) des valeurs de la séquence. Ils utiliseront toutes les valeurs en cache avant de s'apercevoir des modifications sur les paramètres de la séquence. Le serveur qui a lancé la commande le saura immédiatement.

# Compatibilité

ALTER SEQUENCE est conforme au standard SQL:2003.

# ALTER TABLE

## Nom

ALTER TABLE −− change la définition d'une table

## Synopsis

```
ALTER TABLE [ ONLY ] nom [ * ]
    action [, ... ]
ALTER TABLE [ ONLY ] nom [ * ]
    RENAME [ COLUMN ] colonne TO nouvelle_colonne
ALTER TABLE nom
    RENAME TO nouveau_nom

où action fait partie de :

    ADD [ COLUMN ] colonne
type [ contrainte_colonne [ ... ] ]
    DROP [ COLUMN ] colonne [
RESTRICT | CASCADE ]
    ALTER [ COLUMN ] colonne TYPE
type [ USING expression ]
    ALTER [ COLUMN ] colonne SET
DEFAULT expression
    ALTER [ COLUMN ] colonne DROP
DEFAULT
    ALTER [ COLUMN ] colonne { SET
| DROP } NOT NULL
    ALTER [ COLUMN ] colonne SET
STATISTICS entier
    ALTER [ COLUMN ] colonne SET
STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ADD contrainte_table
    DROP CONSTRAINT nom_contrainte
[ RESTRICT | CASCADE ]
    CLUSTER ON nom_index
    SET WITHOUT CLUSTER
    SET WITHOUT OIDS
    OWNER TO nouveau_proprietaire
    SET TABLESPACE nom_espacelogique
```

## Description

`ALTER TABLE` change la définition d'une table existante. Il existe plusieurs variantes :

`ADD COLUMN`
: Cette variante ajoute une nouvelle colonne à la table en utilisant la même syntaxe que *CREATE TABLE*.

`DROP COLUMN`
: Cette variante supprime une colonne d'une table. Les index et les contraintes de table référençant cette colonne sont automatiquement supprimés. Il faut utiliser l'option `CASCADE` si certains objets hors de

la table dépendent de cette colonne, comme par exemple des références de clés étrangères ou des vues.

`ALTER COLUMN TYPE`

Cette variante modifie le type d'une colonne de la table. Les index et les contraintes de table simples impliquant la colonne seront automatiquement convertis pour utiliser le nouveau type de la colonne en réanalysant l'expression fournie au départ. La clause `USING` optionnelle spécifie comment calculer la nouvelle valeur de la colonne à partir de l'ancienne ; en cas d'omission, la conversion par défaut est identique à une affectation de l'ancien type vers le nouveau. Une clause `USING` doit être fournie s'il n'existe pas de conversion implicite de l'ancien vers le nouveau type.

`SET/DROP DEFAULT`

Ces variantes ajoutent ou enlèvent des valeurs par défaut pour une colonne. Ces valeurs par défaut ne s'appliquent qu'aux prochaines commandes `INSERT`. Elles ne modifient pas les lignes déjà présentes dans la table. Des valeurs par défaut peuvent aussi être créées pour les vues. Dans ce cas, elles sont ajoutées aux commandes `INSERT` de la vue avant que la règle `ON INSERT` de la vue ne soit appliquée.

`SET/DROP NOT NULL`

Ces variantes changent le fait que la colonne indique autoriser les valeurs NULL ou non. `SET NOT NULL` ne peut être utilisé que si la colonne ne contient pas de valeurs NULL.

`SET STATISTICS`

Cette variante permet de modifier l'objectif de collecte de statistiques par colonne pour les opérations d'analyse (*ANALYZE*) à venir. L'objectif prend une valeur entre 0 et 1000. Le mettre à −1 pour utiliser l'objectif de statistiques par défaut du système (default_statistics_target). Pour plus d'informations sur l'utilisation des statistiques par le planificateur de requêtes de PostgreSQL, référez−vous à Section 13.2.

`SET STORAGE`

Ces variantes changent le mode de stockage pour une colonne. Cela permet de contrôler si cette colonne est gardée en ligne dans la table ou bien externalisée dans une table supplémentaire, et si les données doivent être compressées ou non. `PLAIN` doit être utilisé pour les valeurs de longueur fixe, comme les `integer` et est en ligne non compressé. `MAIN` est pour les données en ligne compressibles. `EXTERNAL` est pour les données externes non compressées. `EXTENDED` est pour les données externes compressées. `EXTENDED` est la valeur par défaut pour la plupart des types qui supportent les autres stockages que `PLAIN`. Utiliser `EXTERNAL` rendra les opérations d'extraction de sous−chaînes `text` et `bytea` plus rapides mais utilisera plus d'espace de stockage. Notez que `SET STORAGE` ne modifie rien dans la table, il configure la stratégie à poursuivre lors des prochaines mises à jour de tables. Voir Section 49.2 pour plus d'informations.

`ADD` *`contrainte_table`*

Cette variante ajoute une nouvelle contrainte à une table en utilisant la même syntaxe que *CREATE TABLE*.

`DROP CONSTRAINT`

Cette forme supprime les contraintes d'une table. Actuellement, les contraintes des tables n'ont pas besoin d'avoir des noms uniques, donc il pourrait y avoir plus d'une correspondance de contraintes pour le nom spécifié. Toutes les contraintes correspondantes seront supprimées.

`CLUSTER`

Cette variante sélectionne l'index par défaut pour les prochaines opérations *CLUSTER*. Il ne ré−exécute pas la commande CLUSTER.

`SET WITHOUT CLUSTER`

Cette variante supprime la spécification d'index *CLUSTER* la plus récemment utilisée à partir de cette table. Ceci affecte les prochaines opérations CLUSTER qui ne spécifient pas d'index.

`SET WITHOUT OIDS`

Cette variante supprime la colonne système `oid` de la table. Ceci est strictement équivalent à `DROP COLUMN oid RESTRICT`, sauf qu'il ne se plaindra pas s'il existe déjà une colonne `oid`.

Notez qu'il n'existe pas de variante de ALTER TABLE qui autoriserait la restauration des OID d'une table, une fois ceux−ci supprimés.

OWNER
: Cette variante change le propriétaire d'une table, d'un index, d'une séquence, ou d'une vue. Le nouveau propriétaire est celui passé en paramètre.

SET TABLESPACE
: Cette forme modifie l'espace logique de la table par l'espace logique spécifié et déplace le(s) fichier(s) de données associé(s) avec la table vers le nouvel espace logique. Les index de la table, s'il y en a, ne sont pas déplacés ; mais ils peuvent l'être séparément avec des commandes SET TABLESPACE supplémentaires. Voir aussi *CREATE TABLESPACE*.

RENAME
: La forme RENAME modifie le nom d'une table (ou d'un index, séquence ou vue) ou le nom d'une colonne individuelle d'une table. Elle n'a aucun effet sur la donnée stockée.

Toutes les actions à l'exception de RENAME peuvent être combinées dans une liste de plusieurs altérations à appliquer en parallèle. Par exemple, il est possible d'ajouter plusieurs colonnes et/ou modifier le type de plusieurs colonnes en une seule commande. Ceci est particulièrement utile avec les grosses tables car seule une passe sur la table sera nécessaire.

Il faut être propriétaire de la table pour utiliser ALTER TABLE, sauf pour ALTER TABLE OWNER, qui ne peut être utilisée que par un super utilisateur.

# Paramètres

*nom*
: Le nom (éventuellement précisé par un schéma) d'une table existante, que l'on veut modifier. Si ONLY est indiqué, cette table seulement est modifiée. Si ONLY est absent, alors la table et toutes ses tables filles (s'il y en a) sont modifiées. * peut être ajouté au nom de la table pour indiquer que ses tables descendantes doivent être modifiées. Dans la version courante, c'est le comportement par défaut. Dans les versions antérieures à la 7.1, ONLY était le comportement par défaut. Le comportement par défaut peut être modifié en changeant le paramètre de configuration sql_inheritance.

*colonne*
: Nom d'une colonne existante ou nouvelle.

*nouvelle_colonne*
: Nouveau nom d'une colonne existante.

*nouveau_nom*
: Nouveau nom de la table.

*type*
: Type de donnée de la nouvelle colonne, ou nouveau type de données d'une colonne existante.

*contraintedetable*
: Nouvelle contrainte de table pour la table.

*nomdecontrainte*
: Nom d'une contrainte existante à supprimer

CASCADE
: Supprime automatiquement les objets dépendant de la colonne ou contrainte supprimée (par exemple, les vues référençant la colonne).

RESTRICT
: Refuse de supprimer la colonne ou la contrainte si des objets en dépendent. Ceci est le comportement par défaut.

*nomindex*
: Nom de l'index sur lequel la table doit être réorganisée en cluster.

*nouveau_propriétaire*
: Le nom du nouveau propriétaire de la table.

*nom_espacelogique*
: Le nom de l'espace logique où sera déplacé la table.

# Notes

Le mot clé `COLUMN` n'est pas nécessaire. Il peut être omis.

Quand une colonne est ajoutée avec `ADD COLUMN`, toutes les lignes existantes de cette table sont initialisées avec la valeur par défaut de la colonne (NULL si aucune clause `DEFAULT` n'a été définie).

Ajouter une colonne avec une valeur par défaut différente de NULL ou modifier le type d'une colonne existante requiert que la table entière soit ré−écrite. Ceci pourrait prendre un temps considérable pour une grande table ; et cela demandera temporairement le double d'espace disque.

Ajouter une contrainte `CHECK` ou `NOT NULL` requiert de parcourir la table pour vérifier que les lignes existantes respectent cette contrainte.

La raison principale pour fournir l'option de spécification de changements multiples en une seule commande `ALTER TABLE` est que les parcours ou ré−écritures multiples de table peuvent être combinés en une seule passe sur la table.

La forme `DROP COLUMN` ne supprime pas physiquement la colonne, mais la rend simplement invisible au SQL. Par la suite, les ordres d'insertion et de mise à jour sur cette table stockeront une valeur NULL pour la colonne. Du coup, supprimer une colonne ne réduit pas immédiatement la taille de la table sur le disque car l'espace occupé par la colonne n'est pas récupéré. Cet espace sera récupéré petit à petit, au fur et à mesure des mises à jour des lignes de la table.

Le fait qu'`ALTER TYPE` requiert la ré−écriture de la table entière est quelque fois un avantage car le processus de ré−écriture élimine tout espace mort dans la table. Par exemple, pour réclamer immédiatement la place occupée par une colonne supprimée, la façon la plus rapide est

```
ALTER TABLE table ALTER COLUMN toutecolonne TYPE touttype;
```

où `toutecolonne` est toute colonne de table restante et `touttype` est le même type que ce que possède déjà la colonne. Cela résulte en des modifications sémantiquement visibles dans la table mais la commande force la ré−écriture, qui oublie toute donnée inutile.

L'option `USING` d'`ALTER TYPE` peut en fait spécifier toute expression impliquant les anciennes valeurs de la ligne ; c'est−à−dire qu'il peut référencer les autres colonnes ainsi que celle en cours de conversion. Ceci permet des conversions très générales grâce à la syntaxe `ALTER TYPE`. À cause de cette flexibilité, l'expression `USING` n'est pas appliquée à la valeur par défaut de la colonne (si elle en a) ; le résultat pourrait ne pas être une expression constante requise pour une valeur par défaut. Cela signifie que, quand il n'existe pas de conversion implicite ou d'affectation de l'ancien au nouveau type, `ALTER TYPE` pourrait échouer dans sa conversion de la valeur par défaut bien que la clause `USING` soit spécifiée. Dans de tels cas, supprimez la valeur par défaut avec `DROP DEFAULT`, exécutez le `ALTER TYPE` et enfin utilisez `SET DEFAULT` pour

ajouter une nouvelle valeur par défaut convenable. Des considérations similaires s'appliquent aux index et contraintes impliquant la colonne.

Si une table a des tables descendantes, il n'est pas possible d'ajouter, de renommer ou de modifier le type d'une colonne dans la table parent sans le faire aussi pour ses descendantes. Donc, la commande `ALTER TABLE ONLY` est rejetée. Ceci assure que les descendantes ont toujours des colonnes correspondant à celles de la table parente.

Un appel récursif à `DROP COLUMN` supprimera une colonne d'une table descendante si et seulement si la table descendante n'hérite pas de cette colonne d'une autre table et n'a jamais eu de définition indépendante de la colonne. Une suppression de colonne non récursive (c'est à dire une commande `ALTER TABLE ONLY ... DROP COLUMN`) ne supprime jamais les colonnes descendantes mais les marque comme définies de manière indépendante, plutôt qu'héritées.

On ne peut pas changer quoi que ce soit dans une table du catalogue système.

Voir la commande *CREATE TABLE* pour avoir une description plus complète des paramètres valides. Chapitre 5 donne plus d'informations sur l'héritage.

# Exemples

Pour ajouter une colonne de type `varchar` à une table :

```
ALTER TABLE distributeurs ADD COLUMN adresse varchar(30);
```

Pour supprimer une colonne d'une table :

```
ALTER TABLE distributeurs DROP COLUMN adresse RESTRICT;
```

Pour modifier les types de deux colonnes existantes en une opération :

```
ALTER TABLE distributeurs
    ALTER COLUMN adresse TYPE varchar(80),
    ALTER COLUMN nom TYPE varchar(100);
```

Pour modifier une colonne de type entier contenant une date/heure UNIX en `timestamp with time zone` avec une clause `USING` :

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

Pour renommer une colonne existante :

```
ALTER TABLE distributeurs RENAME COLUMN adresse TO city;
```

Pour renommer une table existante :

```
ALTER TABLE distributeurs RENAME TO suppliers;
```

Pour ajouter une contrainte NOT NULL à une colonne :

```
ALTER TABLE distributeurs ALTER COLUMN rue SET NOT NULL;
```

Pour supprimer une contrainte NOT NULL d'une colonne :

```
ALTER TABLE distributeurs ALTER COLUMN rue DROP NOT NULL;
```

Pour ajouter une contrainte de vérification sur une table :

```
ALTER TABLE distributeurs ADD CONSTRAINT verif_zip CHECK (char_length(zipcode) = 5);
```

Pour supprimer une contrainte de vérification d'une table et de toutes ses tables filles :

```
ALTER TABLE distributeurs DROP CONSTRAINT verif_zip;
```

Pour ajouter une contrainte de clé étrangère à une table :

```
ALTER TABLE distributeurs ADD CONSTRAINT distfk FOREIGN KEY (adresse) REFERENCES adresses (adress
```

Pour ajouter une contrainte unique (multicolonnes) à une table :

```
ALTER TABLE distributeurs ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

Pour ajouter une clé primaire nommée automatiquement à une table. Remarque : une table ne peut avoir qu'une seule clé primaire dans toute sa vie.

```
ALTER TABLE distributeurs ADD PRIMARY KEY (dist_id);
```

Pour déplacer une table dans un espace logique différent :

```
ALTER TABLE distributeurs SET TABLESPACE espacelogiquerapide;
```

# Compatibilité

Les formes `ADD`, `DROP` et `SET DEFAULT` se conforment au standard SQL. Les autres formes sont des extensions PostgreSQL au standard SQL. De plus, la capacité à spécifier plus d'une manipulation en une seule commande `ALTER TABLE` est une extension.

`ALTER TABLE DROP COLUMN` peut être utilisé pour supprimer la seule colonne d'une table, laissant une table sans colonne. C'est une extension de SQL, qui ne permet pas les tables sans colonne.

# ALTER TABLESPACE

## Nom

ALTER TABLESPACE −− modifie la définition d'un espace logique

## Synopsis

```
ALTER TABLESPACE nom RENAME TO nouveau_nom
ALTER TABLESPACE nom OWNER TO nouveau_propriétaire
```

## Description

`ALTER TABLESPACE` change la définition d'un espace logique.

## Paramètres

*nom*
: Le nom d'un espace logique existant.

*nouveau_nom*
: Le nouveau nom de l'espace logique. Le nouveau nom ne peut pas commencer avec `pg_` car ces noms sont réservés pour les espaces logiques système.

*nouveau_propriétaire*
: Le nouveau propriétaire de l'espace logique. Vous devez être superutilisateur pour modifier le propriétaire d'un espace logique.

## Exemples

Renommer l'espace logique `espace_index` par `raid_rapide` :

```
ALTER TABLESPACE espace_index RENAME TO raid_rapide;
```

Modifier le propriétaire de l'espace logique `espace_index` :

```
ALTER TABLESPACE espace_index OWNER TO mary;
```

## Compatibilité

Il n'existe pas d'instruction `ALTER TABLESPACE` dans le standard SQL.

# Voir aussi

*CREATE TABLESPACE*, *DROP TABLESPACE*

# ALTER TRIGGER

## Nom

ALTER TRIGGER −− change la définition d'un déclencheur

## Synopsis

```
ALTER TRIGGER nom ON table RENAME TO nouveau nom
```

## Description

`ALTER TRIGGER` modifie les propriétés d'un déclencheur existant. La clause `RENAME` change le nom d'un déclencheur donné sans modifier la définition du déclencheur.

Vous devez être propriétaire de la table sur laquelle le déclencheur agit pour avoir la permission de modifier ses propriétés.

## Paramètres

*nom*
: Le nom d'un déclencheur existant à modifier.

*table*
: Nom de la table sur laquelle le déclencheur agit.

*nouveau nom*
: Nouveau nom du déclencheur.

## Exemples

Pour renommer un déclencheur existant:

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

## Compatibilité

`ALTER TRIGGER` est une extension PostgreSQL du standard SQL.

# ALTER TYPE

## Nom

ALTER TYPE -- modifie la définition d'un type

## Synopsis

```
ALTER TYPE nom OWNER TO
nouveau_nom
```

## Description

`ALTER TYPE` modifie la définition d'un type existant. La seule fonctionnalité actuellement disponible est de modifier le propriétaire d'un type.

## Paramètres

*nom*
: Le nom du type à modifier (pouvant être qualifié par le nom du schéma).

*nouveau_propriétaire*
: Le nom du nouveau propriétaire du type. Vous devez être un superutilisateur pour modifier le propriétaire d'un type.

## Exemples

Pour modifier le propriétaire du type `email` défini par l'utilisateur par `joe` :

```
ALTER TYPE email OWNER TO joe;
```

## Compatibilité

Il n'existe aucune instruction `ALTER TYPE` dans le standard SQL.

# ALTER USER

## Nom

ALTER USER −− modifie le compte d'un utilisateur de la base de données

## Synopsis

```
ALTER USER nom [ [ WITH ] option [ ... ] ]

où option peut être :

    CREATEDB | NOCREATEDB
    | CREATEUSER | NOCREATEUSER
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'mot_de_passe'
    | VALID UNTIL 'dateheure'

ALTER USER nom RENAME TO nouveaunom

ALTER USER nom SET parametre { TO | = } { valeur | DEFAULT }
ALTER USER nom RESET parametre
```

## Description

`ALTER USER` change les attributs d'un compte utilisateur de PostgreSQL. Les attributs non mentionnés dans la commande gardent leur valeur précédente.

La première variante de cette commande listée dans les synopsis change certains droits par utilisateur et les paramètres d'authentification. (Voir plus bas pour les détails.) Les super−utilisateurs des bases de données peuvent changer tous ces privilèges pour tout utilisateur. Les utilisateurs ordinaires peuvent seulement changer leur mot de passe.

La deuxième variante change le nom de l'utilisateur. Seul un super− utilisateur peut renommer des comptes utilisateurs. L'utilisateur de la session actuelle ne peut être renommé. (Connectez−vous avec un autre utilisateur si vous avez besoin de le faire.) Comme les mots de passe cryptés avec `MD5` utilisent le nom de l'utilisateur en utilisant un élément cryptographique, renommer un utilisateur efface leur mot de passe `MD5`.

La troisième et la quatrième variante changent les paramètres de session par défaut pour une variable de configuration donnée. À chaque fois que l'utilisateur débute une nouvelle session, les valeurs spécifiées deviennent les valeurs par défaut de la session, quels que soient les paramètres présents dans `postgresql.conf` ou ceux reçus sur la ligne de commande par le `postmaster`. Les utilisateurs ordinaires peuvent changer les valeurs par défaut de leur propre session. Les super−utilisateurs peuvent changer les valeurs de session par défaut de tout le monde. Certaines variables ne peuvent pas être configurées de cette façon ou peuvent seulement être configurées par un superutilisateur.

# Paramètres

*nom*
: Le nom de l'utilisateur dont les attributs doivent être modifiés.

CREATEDB
NOCREATEDB
: Ces clauses définissent la capacité d'un utilisateur à créer des bases de données. Si CREATEDB est spécifié, l'utilisateur aura le droit de créer ses propres bases de données. Utiliser NOCREATEDB interdira à l'utilisateur de créer des bases de données. (Si l'utilisateur est aussi un superutilisateur, alors ce paramètre n'aura aucun effet réel.)

CREATEUSER
NOCREATEUSER
: Ces clauses déterminent si un utilisateur sera autorisé à créer de nouveaux utilisateurs lui−même. CREATEUSER fera aussi de l'utilisateur un superutilisateur, qui dépassera toutes les restrictions d'accès.

*mot_de_passe*
: Le nouveau mot de passe à utiliser pour ce compte.

ENCRYPTED
UNENCRYPTED
: Ces mots clés contrôlent si le mot de passe est stocké crypté dans pg_shadow. (Voir *CREATE USER* pour plus d'informations sur ce choix.)

*dateheure*
: La date (et optionnellement l'heure) à laquelle le mot de passe de l'utilisateur expire. Pour indiquer que le mot de passe n'expire jamais, utilisez le littéral 'infinity'.

*nouveaunom*
: Le nouveau nom de l'utilisateur.

*paramètre*
*valeur*
: Remet la valeur de ce paramètre de configuration de session à sa valeur par défaut. Si *valeur* vaut DEFAULT ou RESET (les deux sont équivalents), alors la valeur spécifique à l'utilisateur qui était enregistrée est supprimée, donc l'utilisateur héritera des valeurs générales du système dans ses nouvelles sessions. Utilisez RESET ALL pour supprimer toutes les valeurs enregistrées spécifiques à l'utilisateur.

  Voir *SET* et Section 16.4 pour plus d'informations sur les valeurs possibles pour les paramètres et leurs valeurs.

# Notes

Utilisez *CREATE USER* pour ajouter de nouveaux utilisateurs, et *DROP USER* pour supprimer un utilisateur.

ALTER USER ne permet pas de changer les groupes d'un utilisateur. Utilisez *ALTER GROUP* pour cela.

La clause VALID UNTIL définit une date d'expiration pour un mot de passe seulement, pas pour le compte utilisateur lui−même. En particulier, la date d'expiration n'est pas requise lors de la connexion en utilisant une méthode d'authentification par mot de passe.

Il est aussi possible de lier les valeurs par défaut d'une session sur une base de données spécifique plutôt que sur un utilisateur ; voir *ALTER DATABASE*. Les paramétrages spécifiques de l'utilisateur surchargent ceux de

la base de données s'il y a un conflit.

# Exemples

Changer le mot de passe d'un utilisateur :

```
ALTER USER davide WITH PASSWORD 'hu8jmn3';
```

Changer la date de fin de validité du mot de passe d'un utilisateur :

```
ALTER USER manuel VALID UNTIL 'Jan 31 2030';
```

Changer la date de fin de validité d'un utilisateur, en indiquant que son autorisation doit expirer à midi le 4 mai 2005 pour le fuseau horaire UTC+1 :

```
ALTER USER chris VALID UNTIL 'May 4 12:00:00 2005 +1';
```

Rendre un mot de passe valide indéfiniment :

```
ALTER USER fred VALID UNTIL 'infinity';
```

Donner à un utilisateur le droit de créer de nouvelles bases de données et de nouveaux utilisateurs :

```
ALTER USER miriam CREATEUSER CREATEDB;
```

# Compatibilité

La commande `ALTER USER` est une extension de PostgreSQL. En effet, le standard SQL laisse la façon de définir les utilisateurs au choix du SGBD.

# Voir aussi

*CREATE USER*, *DROP USER*, *SET*

# ANALYZE

## Nom

ANALYZE −− récupère des statistiques sur une base de données

## Synopsis

```
ANALYZE [ VERBOSE ] [ table [ (colonne [, ...] ) ] ]
```

## Description

`ANALYZE` collecte des statistiques sur le contenu des tables de la base de données et stocke les résultats dans la table système `pg_statistic`. L'optimiseur de requêtes les utilise pour déterminer les plans d'exécution les plus efficaces.

Sans paramètre, `ANALYZE` examine chaque table dans la base de données en cours. Avec un paramètre, `ANALYZE` examine seulement cette table. Il est possible de donner une liste des noms de colonnes, auquel cas seules les statistiques concernant ces colonnes sont collectées.

## Paramètres

`VERBOSE`
: Active l'affichage de messages de progression.

*`table`*
: Le nom (éventuellement précédé du nom du schéma) d'une table spécifique à analyser. Par défaut, il s'agit de toutes les tables dans la base de données en cours.

*`column`*
: Le nom d'une colonne spécifique à analyser. Par défaut, toutes les colonnes.

## Sorties

Quand `VERBOSE` est spécifié, `ANALYZE` émet des messages de progression pour indiquer quelle table est en cours de traitement. Sont aussi affichées différentes statistiques sur les tables.

## Notes

Lancer périodiquement `ANALYZE` est une bonne idée, tout comme le lancer après de grosses modifications sur le contenu d'une table. Des statistiques à jour aideront l'optimiseur à choisir le plan de requête le plus approprié et amélioreront du coup la rapidité du traitement des requêtes. Une stratégie habituelle est de lancer *VACUUM* et `ANALYZE` une fois par jour au moment où le serveur est le moins sollicité.

Contrairement à VACUUM FULL, ANALYZE requiert seulement un verrou en lecture sur la table cible, donc il peut être lancé en parallèle à d'autres activités sur la table.

Les statistiques récupérées par ANALYZE incluent habituellement une liste des quelques valeurs les plus communes dans chaque colonne et un histogramme affichant une distribution approximative des données dans chaque colonne. L'un ou l'autre pourraient être omis si ANALYZE les trouve inintéressants (par exemple, dans une colonne à clé unique, il n'y a pas de valeurs communes) ou si le type de données de la colonne ne supporte pas les opérateurs appropriés. Il y a plus d'informations sur les statistiques dans le Chapitre 21.

Pour les grosses tables, ANALYZE prend aléatoirement plusieurs lignes de la table, au hasard, plutôt que d'examiner chaque ligne. Ceci permet à des tables très larges d'être examinées dans une petite période de temps. Notez, néanmoins, que les statistiques sont seulement approximatives et qu'elles changeront légèrement à chaque fois qu'ANALYZE est lancé, même si le contenu réel de la table n'a pas changé. Ceci pourrait résulter en de petites modifications dans les coûts estimés de l'optimiseur affichés par EXPLAIN. Dans de rare situations, ce non−déterminisme causera le choix d'un plan de requête différent par l'optimiseur entre deux lancements d'ANALYZE. Pour éviter ceci, augmentez le nombre de statistiques récupérées par ANALYZE, comme décrit ci−dessous.

L'étendue des analyses est contrôlable globalement en ajustant la variable de configuration default_statistics_target ou colonne par colonne en initialisant la cible des statistiques par colonne avec ALTER TABLE ... ALTER COLUMN ... SET STATISTICS (voir *ALTER TABLE*). Cette valeur cible initialise le nombre maximum d'entrées dans la liste des valeurs les plus communes et le nombre maximum de points dans l'histogramme. La valeur cible par défaut est de 10 mais ceci peut être ajusté en prenant compte la précision des estimations de l'optimiseur, le temps pris par ANALYZE et l'espace total occupé dans pg_statistic. En particulier, initialiser la cible des statistiques à zéro désactive la récupération de statistiques pour cette colonne. Il pourrait être utile de le faire pour les colonnes qui ne sont jamais utilisées dans les clauses WHERE, GROUP BY ou ORDER BY de requêtes car l'optimiseur n'utilisera pas les statistiques pour ces colonnes.

La plus grande cible de statistiques de toutes les colonnes en cours d'analyse détermine le nombre de lignes testées pour préparer les statistiques de la table. Augmenter cette cible cause une augmentation proportionnelle du temps et de l'espace nécessaires à exécuter ANALYZE.

# Compatibilité

Il n'existe pas d'instruction ANALYZE dans le standard SQL.

# BEGIN

## Nom

BEGIN −− débute un bloc de transaction

## Synopsis

```
BEGIN [ WORK | TRANSACTION ] [ mode_transaction [, ...] ]

où mode_transaction fait partie
de :

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
```

## Description

`BEGIN` initie un bloc de transaction, c'est−à−dire que toutes les instructions après la commande `BEGIN` seront exécutées dans une seule transaction jusqu'à ce qu'un *COMMIT* ou *ROLLBACK* explicite soit exécuté. Par défaut (sans `BEGIN`), PostgreSQL exécute des transactions en mode << autocommit >>, c'est−à−dire que chaque instruction est exécutée dans sa propre transaction et une validation (commit) est traitée implicitement à la fin de l'instruction (si l'exécution a réussi, sinon une annulation est exécutée).

Les instructions sont exécutées plus rapidement dans un bloc de transaction parce que la séquence début/validation de transaction demande une activité significative du CPU et du disque. L'exécution de plusieurs instructions dans une transaction est aussi utile pour s'assurer d'une cohérence lors de la réalisation de certains modifications liées : les autres sessions ne seront pas capables de voir les états intermédiaires tant que toutes les mises à jour n'auront pas été réalisées.

Si le niveau d'isolation ou le mode lecture/écriture est spécifié, la nouvelle transaction a ces caractéristiques, comme si *SET TRANSACTION* a été exécutée.

## Paramètres

`WORK`
`TRANSACTION`
: Mots clés optionnels. Ils n'ont pas d'effet.

Référez−vous à *SET TRANSACTION* pour des informations sur la signification des autres paramètres de cette instruction.

# Notes

*START TRANSACTION* a la même fonctionnalité que `BEGIN`.

Utilisez *COMMIT* ou *ROLLBACK* pour terminer un bloc de transaction.

Lancer `BEGIN` en étant déjà dans un bloc de transaction provoquera l'apparition d'un message d'avertissement. L'état de la transaction n'en sera pas affecté. Pour intégrer des transactions à l'intérieur d'un bloc de transaction, utilisez les points de sauvegarde (voir *SAVEPOINT*).

Pour des raisons de compatibilité descendante, les virgules entre chaque *`mode_transaction`* peuvent être omis.

# Exemples

Pour commencer un bloc de transaction :

```
BEGIN;
```

# Compatibilité

`BEGIN` est une extension du langage PostgreSQL. C'est équivalent à la commande *START TRANSACTION* du standard SQL, qui donne des informations de compatibilité supplémentaires.

Les autres systèmes de bases de données relationnels peuvent offrir une fonctionnalité autocommit en supplément.

Incidemment, le mot clé `BEGIN` est utilisé dans un but différent en SQL embarqué. Faites attention à la sémantique de la transaction lors du portage d'applications de bases de données.

# Voir aussi

*COMMIT*, *ROLLBACK*, *START TRANSACTION*, *SAVEPOINT*

# CHECKPOINT

## Nom

CHECKPOINT −− force un point de vérification du journal des transactions

## Synopsis

```
CHECKPOINT
```

## Description

Les WAL (*Write−Ahead Log*, journaux des transactions) placent un point de vérification dans le journal des transactions à intervalle régulier. (Pour ajuster cet intervalle, voir les options de configuration à l'exécution checkpoint_segments et checkpoint_timeout.) La commande `CHECKPOINT` force un point de vérification immédiat, sans attendre un point de vérification planifié.

Un point de vérification est un point dans la séquence du journal des transactions pour lequel tous les fichiers de données ont été mis à jour pour refléter l'information des journaux. Tous les fichiers de données sont écrits sur le disque. Référez−vous à Chapitre 25 pour plus d'informations sur le système WAL.

Seuls les superutilisateurs peuvent appeler `CHECKPOINT`. Cette commande n'est pas utilisée lors d'opérations normales.

## Compatibilité

La commande `CHECKPOINT` est une extension du langage PostgreSQL.

# CLOSE

## Nom

CLOSE −− ferme un curseur

## Synopsis

```
CLOSE nom
```

## Description

`CLOSE` libère les ressources associées à un curseur ouvert. Une fois le curseur fermé, aucune opération n'est autorisée sur celui−ci. Un curseur doit être fermé lorsqu'il n'est plus nécessaire.

Tout curseur ouvert non détenable est fermé implicitement lorsqu'une transaction est terminée avec `COMMIT` ou `ROLLBACK`. Un curseur détenable est implicitement fermé si la transaction qui l'a créé est annulée via `ROLLBACK`. Si cette transaction est validée (avec succès), le curseur détenable reste ouvert jusqu'à ce qu'une commande `CLOSE` explicite soit lancée ou jusqu'à la déconnexion du client.

## Paramètres

*name*
: Le nom du curseur ouvert à fermer.

## Notes

PostgreSQL n'a pas d'instruction explicite d'ouverture (`OPEN`) de curseur ; un curseur est considéré ouvert à sa déclaration. Utilisez l'instruction ***DECLARE*** pour déclarer un curseur.

## Exemples

Fermer le curseur `liahona` :

```
CLOSE liahona;
```

## Compatibilité

`CLOSE` est totalement conforme avec le standard SQL.

# Voir aussi

*DECLARE*, *FETCH*, *MOVE*

# CLUSTER

## Nom

CLUSTER −− réorganise une table suivant un index

## Synopsis

```
CLUSTER nomindex ON nomtable
CLUSTER nomtable
CLUSTER
```

## Description

`CLUSTER` demande à PostgreSQL de réorganiser (regrouper) la table spécifiée par *`nomtable`* en se basant sur l'index spécifié par *`nomindex`*. L'index doit déjà avoir été défini sur *`nomtable`*.

Quand une table est réorganisée, elle est physiquement réordonnée en se basant sur les informations de l'index. Ce groupement est une opération unique : quand une table est mise à jour après coup, les modifications ne tiennent pas compte de la réorganisation. C'est−à−dire qu'aucune tentative n'est réalisée pour stocker les nouvelles données ou les données mises à jour suivant l'ordre de l'index. Si vous le souhaitez, vous pouvez grouper périodiquement en lançant de nouveau la commande.

Quand une table est groupée, PostgreSQL se rappelle de l'index avec lequel elle a été réorganisée. La forme `CLUSTER` *`nomtable`* groupe de nouveau la table avec le même index qu'auparavant.

`CLUSTER` sans aucun paramètre groupe toutes les tables de la base de données courante et dont l'utilisateur est propriétaire, ou toutes les tables s'il s'agit d'un superutilisateur. (Les tables qui n'ont jamais été groupées sont ignorées.) Cette forme de `CLUSTER` ne peut pas être appelée à partir d'une transaction ou d'une fonction.

Quand une table est en cours de groupement, un verrou `ACCESS EXCLUSIVE` est acquis. Ceci empêche toute opération sur la table (à la fois en lecture et en écriture) tant que la commande `CLUSTER` n'est pas terminée.

## Paramètres

*`nomindex`*
: Le nom d'un index.

*`nomtable`*
: Le nom d'une table (pouvant être qualifié du nom du schéma).

# Notes

Dans les cas où vous accédez aux lignes d'une table de façon aléatoire, et une par une, l'ordre réel des données dans la table n'est pas important. Néanmoins, si vous avez tendance à accéder à certaines données plus qu'à d'autres et qu'il existe un index qui les groupe ensemble, vous bénéficierez de l'utilisation de CLUSTER. Si vous demandez un ensemble de valeurs indexées à partir d'une table, ou une seule valeur indexée correspondant à plusieurs lignes, CLUSTER vous aidera car une fois que l'index a identifié la page principale pour la première ligne correspondante, toutes les autres lignes correspondantes sont déjà probablement sur la même page et vous économisez du coup des accès disque et accélérez ainsi la requête.

Lors de l'opération de réorganisation, une copie temporaire de la table est créée pour contenir les données de la table dans l'ordre de l'index. Des copies temporaires de chaque index de la table sont aussi créées. Du coup, vous avez besoin d'un espace libre sur le disque au moins égal à la somme de la taille de la table et des tailles des index.

Comme CLUSTER se rappelle des informations de réorganisation, vous pouvez réorganiser les tables que vous voulez la première fois manuellement et planifier régulièrement une réorganisation de la même façon que vous faites un VACUUM de façon à ce que les tables soient périodiquement groupées de nouveau.

Comme le planificateur enregistre les statistiques des enregistrements suivant l'ordre des tables, il est conseillé de lancer *ANALYZE* sur la table réorganisée. Sinon, le planificateur pourrait faire de mauvais choix pour les plans de requêtes.

Il existe une autre façon de grouper les données. La commande CLUSTER réordonne la table originale en utilisant l'ordre de l'index que vous spécifiez. Ceci peut être lent sur les grandes tables parce que les lignes sont récupérées dans l'ordre de l'index et que si la table n'est pas ordonnée, les entrées sont dans des pages aléatoires, donc il y a une page disque lue pour chaque ligne déplacée. (PostgreSQL a un cache mais la totalité d'une grande table ne tiendra pas dans le cache.) L'autre moyen de réorganiser une table est d'utiliser

```
CREATE TABLE nouvelletable AS
    SELECT listecolonnes FROM table ORDER BY listecolonnes;
```

qui utilise le code de tri de PostgreSQL dans la clause ORDER BY pour créer l'ordre désiré ; ceci est généralement bien plus rapide qu'un parcours d'index pour des données non triées. Vous pouvez alors supprimer l'ancienne table, utiliser ALTER TABLE ... RENAME pour renommer *nouvelletable* par l'ancien nom et recréer les index de la table. Néanmoins, cette approche ne préserve pas les OID, les contraintes, les relations de clés étrangères, les droits donnés et d'autres propriétés de la table — tous ces éléments devront être recréés manuellement.

# Exemples

Grouper la table employes sur la base de son index emp_ind :

```
CLUSTER emp_ind ON emp;
```

Grouper la relation employes en utilisant le même index qu'auparavant :

```
CLUSTER emp;
```

Grouper toutes les tables de la base de données qui ont déjà été groupées :

```
CLUSTER;
```

## Compatibilité

Il n'existe pas d'instruction `CLUSTER` dans le standard SQL.

## Voir aussi

*clusterdb*

# COMMENT

## Nom

COMMENT −− définit ou modifie le commentaire sur un objet

## Synopsis

```
COMMENT ON
{
  TABLE nom_objet |
  COLUMN nom_table.nom_colonne |
  AGGREGATE nom_agrégat
(type_agrégat) |
  CAST (typesource AS
typecible) |
  CONSTRAINT nom_contrainte ON
nom_table |
  CONVERSION nom_objet |
  DATABASE nom_objet |
  DOMAIN nom_objet |
  FUNCTION nom_fonction
(type_arg1, type_arg2, ...) |
  INDEX nom_objet |
  LARGE OBJECT oid_large_objet |
  OPERATOR op (type_operande1, type_operande2) |
  OPERATOR CLASS nom_objet USING
méthode_indexage |
  RULE nom_role ON nom_table |
  SCHEMA nom_objet |
  SEQUENCE nom_objet |
  TRIGGER nom_declencheur ON
nom_table |
  TYPE nom_objet |
  VIEW nom_objet
} IS 'texte'
```

## Description

`COMMENT` stocke un commentaire sur un objet de base de données.

Pour modifier un commentaire, lancez une nouvelle commande `COMMENT` pour le même objet. Seule une chaîne de commentaire est stockée pour chaque objet. Pour supprimer un commentaire, écrivez `NULL` à la place de la chaîne de texte. Les commentaires sont automatiquement supprimés lorsque l'objet est supprimé.

Les commentaires peuvent être facilement récupérés avec les commandes de psql `\dd`, `\d+` et `\l+`. Les autres interfaces utilisateur permettant de récupérer les commentaires peuvent être construites au−dessus des même fonctions intégrées que celles utilisées par psql, nommément `obj_description` et `col_description`. (Voir Tableau 9−43.)

# Paramètres

*nom_objet*
*nom_table.nom_colonne*
*nom_agregat*
*nom_contrainte*
*nom_fonction*
*op*
*nom_regle*
*nom_declencheur*
: Le nom de l'objet à commenter. Les noms des tables, agrégats, domaines, fonctions, index, opérateurs, classes d'opérateurs, séquences, types et vues pourraient être qualifiés du nom du schéma.

*type_agregat*
: Le type de données de l'argument de la fonction d'agrégat ou `*` si la fonction accepte tout type de données.

*large_object_oid*
: L'OID de l'objet large.

PROCEDURAL
: Ceci est du bruit.

*sourcetype*
: Le nom du type de données source de la conversion.

*targettype*
: Le nom du type de données cible de la conversion.

*texte*
: Le nouveau commentaire, écrit comme une chaîne littérale ; ou `NULL` pour supprimer le commentaire.

# Notes

Un commentaire pour une base de données peut seulement être créé dans cette base de données et ne sera visible que de cette base de données, pas des autres.

Il n'existe pas de mécanisme de sécurité pour les commentaires : tout utilisateur connecté à une base de données peut voir tous les commentaires des objets dans la base de données (bien que seuls les superutilisateurs peuvent modifier les commentaires sur des objets qu'ils ne possèdent pas). Du coup, ne placez pas d'informations critiques en terme de sécurité dans vos commentaires.

# Exemples

Attacher un commentaire sur la table `matable` :

```
COMMENT ON TABLE matable IS 'Ceci est ma table.';
```

Supprimez–le de nouveau :

```
COMMENT ON TABLE matable IS NULL;
```

Quelques exemples supplémentaires :

```
COMMENT ON AGGREGATE mon_agregat (double precision) IS 'Calcule une variance type';
COMMENT ON CAST (text AS int4) IS 'Autorise les conversions de text vers int4';
COMMENT ON COLUMN ma_table.ma_colonne IS 'Numéro employé';
COMMENT ON CONVERSION ma_conv IS 'Conversion vers Unicode';
COMMENT ON DATABASE ma_database IS 'Base de données de développement';
COMMENT ON DOMAIN mon_domaine IS 'Domaine des adresses email';
COMMENT ON FUNCTION ma_fonction (timestamp) IS 'Renvoie en chiffres romain';
COMMENT ON INDEX mon_index IS 'Renforce l'unicité de l'ID de l'employé';
COMMENT ON LANGUAGE plpython IS 'Support de Python pour les procedures
stockées';
COMMENT ON LARGE OBJECT 346344 IS 'Document de planification';
COMMENT ON OPERATOR ^ (text, text) IS 'Traite l\'intersection de deux textes';
COMMENT ON OPERATOR ^ (NONE, text) IS 'Opérateur de préfix sur un texte';
COMMENT ON OPERATOR CLASS int4ops USING btree IS 'Opérateurs d'entiers sur
quatre octets pour les index btrees';
COMMENT ON RULE ma_regle ON my_table IS 'Trace les mises à jour des
enregistrements d\'employé';
COMMENT ON SCHEMA mon_schema IS 'Données du département';
COMMENT ON SEQUENCE ma_sequence IS 'Utilisé pour générer des clés primaires';
COMMENT ON TABLE mon_schema.ma_table IS 'Informations sur les employés';
COMMENT ON TRIGGER mon_declencheur ON my_table IS 'Utilisé pour RI';
COMMENT ON TYPE complex IS 'Type de données pour les nombres complexes';
COMMENT ON VIEW ma_vue IS 'Vue des coûts départementaux';
```

# Compatibilité

Il n'existe pas de commande `COMMENT` dans le standard SQL.

# COMMIT

## Nom

COMMIT −− valide la transaction en cours

## Synopsis

```
COMMIT [ WORK | TRANSACTION ]
```

## Description

`COMMIT` valide la transaction en cours. Toutes les modifications réalisées par la transaction deviennent visibles aux autres et sont garanties pour être durables si un arrêt brutal survient.

## Paramètres

`WORK`
`TRANSACTION`
: Mots clés optionnels et sans effet.

## Notes

Utilisez *ROLLBACK* pour annuler une transaction.

Lancer `COMMIT` à l'extérieur d'une transaction ne fait aucun mal mais provoque l'affiche d'un message d'avertissement.

## Exemples

Pour valider la transaction en cours et rendre toutes les modifications permanentes :

```
COMMIT;
```

## Compatibilité

Le standard SQL spécifie seulement les deux formes `COMMIT` et `COMMIT WORK`. Sinon, cette commande est totalement conforme.

# Voir aussi

*BEGIN*, *ROLLBACK*

# COPY

## Nom

COPY −− copie des données entre un fichier et une table

## Synopsis

```
COPY nomtable [ ( colonne [, ...] ) ]
    FROM { 'nomfichier' | STDIN }
    [ [ WITH ]
        [ BINARY ]
        [ OIDS ]
        [ DELIMITER [ AS ] 'délimiteur' ]
        [ NULL [ AS ] 'chaîne NULL' ]
        [ CSV [ QUOTE [ AS ] 'guillemet' ]
        [ ESCAPE [ AS ] 'échappement' ]
        [ FORCE NOT NULL colonne [, ...] ] ]

COPY nomtable [ ( colonne [, ...] ) ]
    TO { 'nomfichier' | STDOUT }
    [ [ WITH ]
        [ BINARY ]
        [ OIDS ]
        [ DELIMITER [ AS ] 'délimiteur' ]
        [ NULL [ AS ] 'chaîne NULL' ]
        [ CSV [ QUOTE [ AS ] 'guillemet' ]
        [ ESCAPE [ AS ] 'échappement' ]
        [ FORCE QUOTE colonne [, ...] ] ]
```

## Description

`COPY` déplace des données entre les tables de PostgreSQL et les fichiers du système de fichiers standard. `COPY TO` copie le contenu d'une table *dans* un fichier alors que `COPY FROM` copie des données *à partir* d'un fichier dans une table (ajoutant les données à ce qui se trouve déjà dans la table).

Si une liste de colonnes est spécifiée, `COPY` copiera seulement les données des colonnes spécifiées vers ou à partir du fichier. S'il y a des colonnes dans la table qui ne se trouvent pas dans la liste de colonnes, `COPY FROM` insérera les valeurs par défaut de ces colonnes.

`COPY` avec un nom de fichier indique au serveur PostgreSQL de lire directement à partir d'un fichier ou d'écrire dans un fichier. Le fichier doit être accessible du serveur et le nom doit être spécifié à partir du point de vue du serveur. Lorsque `STDIN` ou `STDOUT` est indiqué, les données sont transmises via la connexion entre le client et le serveur.

## Paramètres

*nomtable*
: Le nom d'une table existante (pouvant être qualifié avec le nom du schéma).

*colonne*
: Une liste optionnelle de colonnes à copier. Si aucune liste n'est donnée, toutes les colonnes seront utilisées.

*nomfichier*
: Le chemin absolu du fichier en entrée ou en sortie.

STDIN
: Spécifie que l'entrée provient de l'application cliente.

STDOUT
: Spécifie que la sortie va vers l'application cliente.

BINARY
: Fait que toutes les données sont stockées ou lues au format binaire plutôt qu'en texte. Vous ne pouvez pas spécifier les options DELIMITER, NULL ou CSV en mode binaire.

OIDS
: Demande la copie des OID pour chaque ligne. (Une erreur est levée si OIDS est spécifié pour une table qui ne possède pas d'OID.)

*délimiteur*
: Le caractère simple qui sépare les colonnes pour chaque ligne d'un fichier. La valeur par défaut est le caractère de tabulation en mode texte et une virgule en mode CSV.

*chaîne NULL*
: La chaîne qui représente une valeur NULL. Par défaut, elle vaut \N (antislash–N) en mode texte et une valeur vide sans guillemets en mode CSV. Vous pourriez préférer une chaîne vide par exemple y compris en mode texte pour les cas où vous ne voulez pas distinguer les valeurs NULL des chaînes vides.

  > **Note :** En utilisant COPY FROM, tout élément de données correspondant à cette chaîne sera stocké comme valeur NULL, donc vous devriez vous assurer que vous utilisez la même chaîne que celle que vous avez utilisé avec COPY TO.

CSV
: Sélectionne le mode CSV (valeurs séparées par des virgules).

*quote*
: Spécifie le caractère guillemet dans le mode CSV. Par défaut, il s'agit du guillemet double.

*escape*
: Spécifie le caractère qui devrait apparaître avant un caractère de donnée QUOTE dans le mode CSV. Par défaut, il s'agit de la valeur QUOTE (habituellement un double guillemet).

FORCE QUOTE
: Dans le mode CSV de COPY TO, force l'utilisation des guillemets pour toutes les valeurs différentes de NULL dans chaque colonne spécifiée. La sortie NULL n'est jamais entre guillemets.

FORCE NOT NULL
: Dans le mode CSV de COPY FROM, traite chaque colonne spécifiée comme si elle était entre guillemets et, du coup, différente d'une valeur NULL. Pour la chaîne NULL par défaut du mode CSV (''), ceci fait que les valeurs manquantes soient comprises comme des chaînes de longueur nulle.

# Notes

COPY peut seulement être utilisé avec des tables réelles, pas avec des vues.

Le mot clé BINARY fait que toutes les données sont stockées/lues dans le format binaire plutôt qu'en texte. Il est un peu plus rapide que le mode texte standard mais un fichier binaire est moins portable au travers des architectures machine et des versions de PostgreSQL.

Vous devez avoir le droit SELECT sur la table dont les valeurs sont lues par `COPY TO` et le droit INSERT sur la table dont les valeurs sont insérées par `COPY FROM`.

Les fichiers nommés dans une commande `COPY` sont lus ou écrits directement par le serveur, et non pas par l'application cliente. Du coup, ils doivent résider ou être accessible par la machine du serveur de bases de données. Ils doivent être accessibles et lisibles ou modifiables par l'utilisateur PostgreSQL (l'identifiant de l'utilisateur qui a exécuté le serveur), et non pas par le client. `COPY` nommant un fichier est seulement autorisé pour les superutilisateurs de la base de données car il autorise la lecture ou l'écriture de tout fichier auquel a accès le serveur.

Ne confondez pas `COPY` avec l'instruction `\copy` de psql. `\copy` appelle `COPY FROM STDIN` ou `COPY TO STDOUT`, puis récupère/stocke la donnée dans un fichier accessible au client psql. Du coup, l'accès au fichier et les droits d'accès dépendent du client plutôt que du serveur quand `\copy` est utilisé.

Il est recommandé que le nom du fichier utilisé dans `COPY` soit toujours utilisé avec un chemin absolu. Ceci est renforcé par le serveur dans le cas d'un `COPY TO` mais, pour les `COPY FROM`, vous devez avoir l'option de lire à partir d'un fichier spécifié par un chemin relatif. Le chemin sera interprété de façon relative au répertoire de travail du processus serveur (quelque part en−dessous du répertoire de données), pas dans le répertoire de travail du client.

`COPY FROM` appellera tout déclencheur et vérifiera les contraintes sur la table de destination. Néanmoins, il n'appellera pas les règles.

L'entrée et la sortie de `COPY` sont affectées par `DateStyle`. Pour s'assurer de la portabilité vers les autres installations de PostgreSQL qui pourraient utiliser des paramétrages `DateStyle` différents de ceux par défaut, `DateStyle` devrait être configuré en `ISO` avant d'utiliser `COPY TO`.

`COPY` stoppe l'opération à la première erreur. Ceci ne devrait apporter aucun problème dans le cas d'un `COPY TO` mais la table cible aura déjà reçu des lignes précédentes dans un `COPY FROM`. Ces lignes ne seront pas visibles ou accessibles mais elle occuperont toujours de l'espace disque. Ceci pourrait totaliser une perte considérable d'espace disque si l'échec arrivait lors d'une grande opération de copie. Vous pourriez souhaiter appeler `VACUUM` pour récupérer l'espace perdu.

# Formats de fichiers

## Format texte

Quand `COPY` est utilisé sans les options `BINARY` ou `CSV`, les données lues ou écrites sont dans un fichier avec une ligne de données par ligne de la table. Les colonnes d'une ligne sont séparées par le caractère délimiteur. Les valeurs des colonnes sont des chaînes générées par la fonction de sortie ou acceptables par la fonction d'entrée de chaque type de données d'attribut. La chaîne NULL spécifiée est utilisée à la place des colonnes NULL. `COPY FROM` lèvera une erreur si une des lignes du fichier en entrée contient plus ou moins de colonnes qu'attendues. Si `OIDS` est spécifié, l'OID est lu ou écrit dans la première colonne, précédant ainsi les colonnes de données de l'utilisateur.

La fin des données peut être représentée par une simple ligne contenant juste un antislash et un point (`\.`). Un marqueur de fin de données n'est pas nécessaire lors de la lecture d'un fichier car la fin du fichier sert bien il est seulement nécessaire lors de la copie de données vers ou à partir d'une application cliente en utilisant le

protocole client pre−3.0.

Les caractères antislash (\) pourraient être utilisés dans les données COPY pour mettre entre guillemets les données de caractères qui pourraient sinon être prises comme délimiteurs de ligne ou de colonne. En particulier, les caractères suivants *doivent* être précédés par un antislash s'ils apparaissent comme faisant partie de la valeur d'une colonne : antislash lui−même, nouvelle ligne, retour chariot et caractère délimiteur actuel.

La chaîne NULL spécifiée est envoyée par COPY TO sans ajout d'antislash ; au contraire, COPY FROM fait correspondre l'entrée avec la chaîne NULL avant de supprimer les antislash. Du coup, une chaîne NULL telle que \N ne peut pas être confondu avec la valeur de donnée réelle \N (qui serait représentée par \\N).

Les séquences spéciales suivantes sont reconnues par COPY FROM :

| Séquence | Représenté |
|---|---|
| \b | Suppression (ASCII 8) |
| \f | Retour chariot (ASCII 12) |
| \n | Nouvelle ligne (ASCII 10) |
| \r | Retour chariot (ASCII 13) |
| \t | Tabulation (ASCII 9) |
| \v | Tabulation verticale (ASCII 11) |
| *chiffres* | Un antislash suivi par un à trois chiffres en octal spécifie le caractère ayant ce code numérique |

Actuellement, COPY TO n'émettra jamais une séquence octale mais utilisera les autres séquences listées ci−dessus pour les caractères de contrôle.

Tout autre caractère avec un antislash se représentera lui−même. Néanmoins, attention à l'ajout non nécessaire d'antislash car cela pourrait produire accidentellement une correspondance de chaîne avec le marqueur de fin de données (\.) ou la chaîne NULL (\N par défaut). Ces chaînes seront reconnues avant tout traitement des antislashs.

Il est fortement recommandé que les applications générant des données COPY convertissent les nouvelles lignes et retours chariot avec les séquences \n et \r, respectivement. À présent, il est possible de représenter un retour chariot par un antislash et un retour chariot, et de représenter une nouvelle ligne par un antislash et une nouvelle ligne. Néanmoins, ces représentations pourraient ne pas être acceptées dans les prochaines versions. Elles sont aussi hautement vulnérables à la corruption si le fichier COPY est transféré via différentes machines (par exemple, à partir d'un Unix vers un Windows ou vice versa).

COPY TO terminera chaque ligne avec une nouvelle ligne style Unix (<< \n >>). Les serveurs fonctionnant avec Microsoft Windows terminent la ligne avec un retour chariot/nouvelle ligne (<< \r\n >>) mais seulement pour COPY dans un fichier serveur ; pour des raisons de cohérence au niveau des plateformes, COPY TO STDOUT envoie toujours << \n >> quelque soit la plateforme du serveur. COPY FROM peut gérer des fin de lignes avec une nouvelle ligne, un retour chariot ou un retour chariot suivi d'une nouvelle ligne. Pour réduire les risques d'erreurs dûes à une nouvelle ligne ou un retour chariot sans antislash qui étaient des données, COPY FROM se plaindra si les fins de ligne dans l'entrée ne sont pas identiques.

## Format CSV

Ce format est utilisé pour importer et exporter des valeurs en les séparant avec des virgules (CSV, acronyme de *Comma Separated Value), format utilisé par un grand nombre de programmes comme les tableurs. Au lieu de l'échappement utilisé par le mode texte standard de PostgreSQL, il produit et reconnaît le mécanisme d'échappement habituel de CSV.*

Les valeurs dans chaque enregistrement sont séparées par le caractère DELIMITER. Si la valeur contient ce caractère, le caractère QUOTE, la chaîne NULL, un retour chariot ou un retour à la ligne, alors la valeur complète est préfixée et suffixée avec le caractère QUOTE et toute occurrence du caractère QUOTE ou du caractère ESCAPE est précédée par le caractère d'échappement. Vous pouvez aussi utiliser FORCE QUOTE pour forcer les guillemets en affichant des valeurs différentes de NULL dans des colonnes spécifiques.

Le format CSV ne dispose pas d'une façon standard de distinguer une valeur NULL d'une chaîne vide. La commande COPY de PostgreSQL gère ceci avec les guillemets. Un NULL est affiché par la chaîne NULL et n'est jamais entre guillemets. Du coup, en utilisant les paramètres par défaut, NULL est écrit comme une chaîne vide sans guillemets alors qu'une chaîne vide est écrit avec des guillemets doubles (""). La lecture des valeurs suit des règles similaires. Vous pouvez utiliser FORCE NOT NULL pour empêcher des comparaisons d'entrées NULL pour des colonnes spécifiques.

> **Note :** Le mode CSV reconnaîtra et produira des fichiers CSV avec des valeurs entre guillemets contenant des retours chariot et des retours à la ligne. Du coup, les fichiers ne sont pas strictement une ligne par ligne de la table comme les fichiers en mode texte. Néanmoins, PostgreSQL rejètera les entrées COPY si un champ contient des séquences de fin de ligne ne correspondant pas à la convention utilisée dans le fichier CSV lui−même. Il est généralement plus sûr d'importer des données contenant des caractères de fin de ligne en utilisant les formats texte ou binaire plutôt que CSV.

> **Note :** Beaucoup de programmes produisent des fichiers CSV étranges et pervers, donc le format du fichier est plus une convention qu'un standard. Du coup, vous pourriez rencontrer certains fichiers que vous ne pouvez pas importer en utilisant ce mécanisme et COPY pourrait produire des fichiers que les programmes ne pourraient pas traiter.

## Format binaire

Le format de fichier utilisé pour COPY BINARY a été modifié dans PostgreSQL 7.4. Le nouveau format consiste en un fichier d'en−tête, zéro ou plusieurs lignes contenant la donnée de la ligne et une queue de fichier. Les en−têtes et les données sont maintenant dans l'ordre d'octets du réseau.

### En−tête du fichier

Le fichier d'en−tête consiste en 15 octets de champs fixes, suivis par une aire d'extension de l'en−tête de longueur variable. Les champs fixes sont :

Signature
: séquence de 11 octets PGCOPY\n\377\r\n\0 — notez que l'octet zéro est une partie requise de la signature. (La signature est conçue pour permettre une identification aisée des fichiers qui ont été déteriorés par un transfert qui ne s'est pas fait sur huit bits. Cette signature sera modifiée

par des filtres de traduction de fin de ligne, suppression des octets zéro, suppression des bits de poids forts ou modification de parité.)

Champs de commutateurs
: masque entier de 32 bits dénotant les aspects importants du format de fichier. Les bits sont numérotés de 0 (LSB) à 31 (MSB). Notez que ce champ est stocké dans l'ordre des octets du réseau (l'octet le plus significatif en premier), comme le sont tous les champs d'entier utilisés dans le format de fichier. Les bits 16 à 31 sont réservés pour dénoter les problèmes critiques de format de fichier ; un lecteur devrait annuler l'opération s'il trouve un bit inattendu dans cet ensemble. Les bits 0 à 15 sont réservés pour signaler des problèmes de compatibilité de formats ; un lecteur devrait simplement ignorer les bits inattendus dans cet ensemble. Actuellement, seul un bit est défini et le reste doit être à zéro :

  Bit 16
  : si 1, les OID sont inclus dans la donnée ; si 0, non

Longueur de l'aire d'extension de l'en−tête
: Entier sur 32 bits, longueur en octets sur le reste de l'en−tête, ne s'incluant pas lui−même. Actuellement, c'est zéro et la première ligne suit immédiatement. Les modifications futures du format pourraient permettre la présence de données supplémentaires dans l'en−tête. Un lecteur devrait passer silencieusement toute donnée dans l'extension de l'en−tête s'il ne sait pas quoi faire avec.

L'aire d'extension de l'en−tête devrait contenir une séquence de morceaux qui s'identifient eux−même. Le champ des commutateurs n'a pas pour but d'indiquer aux lecteurs ce qui se trouve dans l'aire d'extension. La conception spécifique du contenu de l'extension de l'en−tête est laissée à une prochaine version.

Cette conception permet des ajouts d'en−têtes compatibles (ajout de morceaux d'extension d'en−tête, ou initialisation des options de faible poids) et modifications non compatibles (initialisation des options de poids pour signaler des modifications, et ajout des données de support dans l'aire d'extension si nécessaire).

### Lignes

Chaque ligne commence avec un compteur sur 16 bits du nombre de champs dans la ligne. (Actuellement, toutes les lignes dans une table auront le même compte mais ceci pourrait ne pas être toujours vrai.) Ensuite, répété pour chaque champ de la ligne, il y a un mot de 32 bits suivi par autant d'octets que de données. (Le mot clé n'inclut pas sa propre longueur et peut donc valoir zéro.) Comme cas spécial, −1 indique une valeur de champ NULL. Aucun octet de valeur ne suit dans le cas NULL.

Il n'y a pas d'ajout pour l'alignement ou toute autre donnée supplémentaire entre les champs.

Actuellement, toutes les valeurs dans un fichier `COPY BINARY` sont supposées être dans un format binaire (format code un). Une future extension pourrait ajouter un champ d'en−tête qui autorise la spécification des codes de format par colonne.

Pour déterminer le format binaire approprié pour la donnée réelle, vous devriez consulter le source de PostgreSQL, en particulier les fonctions `*send` et `*recv` pour chaque type de données de la colonne (typiquement ces fonctions se trouvent dans le répertoire `src/backend/utils/adt/` des sources).

Si les OID sont inclus dans le fichier, le champ OID est immédiatement suivi du compteur de champ. C'est un champ normal sauf qu'il n'est pas inclus dans le champ compteur. En particulier, il a une longueur d'un mot — ceci permettra la gestion d'OID à quatre octets plutôt que huit sans trop de peine et permettra l'affichage des OID comme NULL si cela se révèle intéressant.

### Queue du fichier

La fin du fichier consiste en un entier sur 16 bits contenant −1. Ceci est facilement distingué du compteur de champ d'une ligne.

Un lecteur devrait rapporter une erreur si un mot de comptage de champ est soit −1 soit le nombre attendu de colonnes. Ceci fournit une vérification supplémentaire sur quelque chose pouvant être hors synchronisation avec les données.

## Exemples

L'exemple suivant copie une table vers le client en utilisant la barre verticale (`|`) comme délimiteur de champ :

```
COPY pays TO STDOUT WITH DELIMITER '|';
```

Pour copier des données de la table `pays` vers un fichier :

```
COPY pays FROM '/usr1/proj/bray/sql/pays_donnees';
```

Voici un exemple de données convenable pour une table provenant de `STDIN` :

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

Notez que l'espace blanc sur chaque ligne est en fait un caractère de tabulation.

Ce qui suit représente les même données, au format binaire. La donnée est affichée après filtrage à travers l'outil Unix `od -c`. La table a trois colonnes ; la première est de type `integer`. Toutes les lignes ont une valeur NULL sur la troisième colonne.

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

## Compatibilité

Il n'existe pas d'instruction `COPY` dans le standard SQL.

La syntaxe suivante était utilisée avant PostgreSQL version 7.3 et est toujours supportée :

```
COPY [ BINARY ] nomtable [ WITH OIDS ]
```

```
    FROM { 'nomfichier' | STDIN }
    [ [USING] DELIMITERS 'délimiteur' ]
    [ WITH NULL AS 'chaîne NULL' ]

COPY [ BINARY ] nomtable [ WITH OIDS ]
    TO { 'nomfichier' | STDOUT }
    [ [USING] DELIMITERS 'délimiteur' ]
    [ WITH NULL AS 'chaîne NULL' ]
```

# CREATE AGGREGATE

## Nom

CREATE AGGREGATE −− définit une nouvelle fonction d'agrégat

## Synopsis

```
CREATE AGGREGATE nom (
    BASETYPE = type_donnée_entrée,
    SFUNC = sfonc,
    STYPE = type_donnée_état
    [ , FINALFUNC = ffonc ]
    [ , INITCOND = condition_initiale ]
)
```

## Description

`CREATE AGGREGATE` définit une nouvelle fonction d'agrégat. Quelques fonctions d'agrégat basiques et largement utilisées sont fournies dans la distribution standard ; elles sont documentées dans le Section 9.15>. Si une d'entre elles définit de nouveaux types ou a besoin d'une fonction d'agrégat non fournie, alors `CREATE AGGREGATE` peut être utilisé pour fournir les fonctionnalités désirées.

Si un nom de schéma est donné (par exemple, `CREATE AGGREGATE monschema.monagg ...`), alors la fonction d'agrégat est créée dans le schéma spécifié. Sinon, elle est créée dans le schéma courant.

Une fonction d'agrégat est identifiée par son nom et son type de données en entrée. Deux agrégats dans le même schéma peuvent avoir le même nom s'ils opèrent sur des types différents en entrée. Le nom et le type de données en entrée d'un agrégat doivent aussi être distincts du nom et du type de données de toutes les fonctions ordinaires du même schéma.

Une fonction d'agrégat est réalisée à partir d'une ou deux fonctions ordinaires : une fonction de transition d'état *`sfonc`*, et une fonction de traitement final optionnelle *`ffonc`*. Elles sont utilisées ainsi :

```
sfonc( état-interne, nouvel-élément-données ) ---> prochain-état-interne
ffonc( état-interne ) ---> valeur-agrégat
```

PostgreSQL crée une variable temporaire de type *`stype`* pour contenir l'état interne courant de l'agrégat. À chaque élément de données en entrée, la fonction de transition d'état est appelée pour calculer une nouvelle valeur de l'état interne. Une fois que toutes les données sont traitées, la fonction finale est appelée une fois pour calculer la valeur de retour de l'agrégat. S'il n'existe pas de fonction finale, alors la valeur d'état final est retournée ainsi.

Une fonction d'agrégat peut fournir une condition initiale, c'est−à−dire une valeur initiale pour la valeur de l'état interne. Ceci est spécifié et stocké dans la base de données comme une colonne de type `text` mais doit être une représentation externe valide d'une constante du type de donnée de la valeur de l'état. Si elle n'est pas fournie, la valeur de l'état commence avec NULL.

Si la fonction de transition de l'état est déclarée << strict >>, alors elle ne peut pas être appelée avec des entrées NULL. Avec une telle fonction de transition, l'exécution d'agrégat se comporte ainsi. Les valeurs des entrées NULL sont ignorées (la fonction n'est pas appelée et la valeur de l'état précédent est conservée). Si la valeur de l'état initial est NULL, alors la première valeur en entrée non NULL remplace la valeur de l'état et la fonction de transition est appelée en commençant avec la seconde valeur en entrée non NULL. Ceci est pratique pour implémenter les agrégats comme `max`. Notez que ce comportement est seulement disponible quand *`type_donnée_état`* est identique à *`type_donnée_entrée`*. Lorsque ces types sont différents, vous devez fournir une condition initiale non NULL ou utiliser une fonction de transition non stricte.

Si la fonction de transition d'état n'est pas stricte, alors elle sera appelée sans condition à chaque valeur en entrée et devra gérer les entrées NULL et les valeurs de transition NULL. Ceci permet à l'auteur de l'agrégat d'avoir le contrôle complet sur la gestion des valeurs NULL par l'agrégat.

Si la fonction finale est déclarée << strict >>, alors elle ne sera pas appelée quand la valeur d'état finale est NULL ; à la place, un résultat NULL sera retourné automatiquement. (Bien sûr, c'est simplement le comportement normal de fonctions strictes.) Dans tous les cas, la fonction finale a l'option de renvoyer une valeur NULL. Par exemple, la fonction finale pour `avg` renvoie NULL lorsqu'elle n'a aucune lignes en entrée.

# Paramètres

*`nom`*
: Le nom de la fonction d'agrégat à créer (pouvant être qualifié avec le nom du schéma).

*`type_donnée_entrée`*
: Le type de données en entrée sur lequel opère la fonction d'agrégat. Elle peut être spécifiée comme `"ANY"` pour un agrégat qui n'examine pas les valeurs en entrée (un exemple est `count(*)`).

*`sfonc`*
: Le nom de la fonction de transition de l'état à appeler pour chaque valeur en entrée. C'est normalement une fonction à deux arguments, le premier étant de type *`type_donnée_état`* et le second de type *`type_donnée_entrée`*. Autrement, pour un agrégat qui n'examine pas les valeurs en entrée, la fonction prend un seul argument de type *`type_donnée_état`*. Dans chaque cas, la fonction doit renvoyer une valeur de type *`type_donnée_état`*. Cette fonction prend la valeur de l'état en cours et l'élément de donnée en cours et renvoie la prochaine valeur d'état.

*`type_donnée_état`*
: Le type de donnée pour la valeur d'état de l'agrégat.

*`ffonc`*
: Le nom de la fonction finale à appeler pour traiter le résultat de l'agrégat une fois que toutes les données en entrée aient été parcourues. La fonction prend un seul argument de type *`type_donnée_état`*. Le type de retour de l'agrégat de la fonction est défini comme le type de retour de cette fonction. Si *`ffonc`* n'est pas spécifiée, alors la valeur d'état finale est utilisée comme résultat de l'agrégat et le type de retour est *`type_donnée_état`*.

*`condition_initiale`*
: La configuration initiale pour la valeur de l'état. Elle doit être une constante de type chaîne de caractères dans la forme acceptée par le type de données *`type_donnée_état`*. Si non spécifié, la valeur d'état commence à NULL.

Les paramètres de `CREATE AGGREGATE` peuvent être écrit dans n'importe quel ordre, pas uniquement dans l'ordre illustré ci−dessus.

# Exemples

Voir Section 31.10.

# Compatibilité

`CREATE AGGREGATE` est une extension du langage PostgreSQL. Le standard SQL ne fournit pas de fonctions d'agrégat définies par l'utilisateur.

# Voir aussi

*ALTER AGGREGATE*, *DROP AGGREGATE*

# CREATE CAST

## Nom

CREATE CAST −− définit une nouvelle conversion

## Synopsis

```
CREATE CAST (typesource AS typecible)
    WITH FUNCTION nomfonction (argtype)
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (typesource AS typecible)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

## Description

`CREATE CAST` définit une nouvelle conversion. Une conversion spécifie la façon de réaliser une conversion entre deux types de données. Par exemple,

```
SELECT CAST(42 AS text);
```

convertit la constante entière 42 dans le type `text` en appelant une fonction précédemment spécifiée, dans ce cas `text(int4)`. (Si aucune conversion convenable n'a été définie, la conversion échoue.)

Deux types pourraient être *compatibles binairement*, ce qui signifie qu'ils peuvent être convertis dans l'autre type << librement >> sans appeler de fonction. Ceci requiert que les valeurs correspondantes utilisent la même représentation interne. En fait, les types `text` et `varchar` sont compatibles binairement.

Par défaut, une conversion peut être appelée par une demande explicite. Voici des constructions explicites : `CAST(`*`x`* `AS` *`nomtype`*`)` ou *`x`*`::`*`nomtype`*.

Si la conversion est marquée `AS ASSIGNMENT`, alors elle peut être appelée implicitement lors de l'affectation d'une valeur à une colonne du type de données cible. Par exemple, en supposant que `foo.f1` est une colonne de type `text`, alors

```
INSERT INTO foo (f1) VALUES (42);
```

sera autorisé si la conversion du type `integer` vers le type `text` est indiquée `AS ASSIGNMENT`, sinon cela sera interdit. (Nous utilisons généralement le terme de *conversion d'affectation* pour décrire ce type de conversion.)

Si la conversion est marquée `AS IMPLICIT`, alors elle peut être appelée implicitement dans tout contexte, que ce soit une affectation ou en interne dans une expression. Par exemple, comme `||` prend deux opérandes `text`,

```
SELECT 'L\'heure est ' || now();
```

sera autorisé seulement si la conversion du type `timestamp` vers le type `text` est marquée `AS IMPLICIT`. Sinon, il sera nécessaire d'écrire explicitement la conversion, par exemple

```
SELECT 'L\'heure est ' || CAST(now() AS text);
```

(Nous utilisons généralement le terme de *conversion implicite* pour décrire ce type de conversion.)

Il est conseillé d'être conservateur sur le marquage des conversions comme implicites. Une surabondance de chemins de conversions implicites peut faire en sorte que PostgreSQL effectue des choix surprenant suite à l'interprétations des commandes ou soit complètement incapable de résoudre les commandes parce qu'il existe plusieurs interprétations possibles. Une bonne règle à suivre est de réaliser une conversion implicite appelable seulement pour les transformations préservant l'information entre les types dans la même catégorie générale. Par exemple, la conversion entre `int2` et `int4` peut être raisonnablement implicite mais celle entre `float8` et `int4` devraient être probablement uniquement sur affectation. Les conversions entre catégorie, tels que de `text` vers `int4`, sont bien préférables en mode explicite seul.

Pour être capable de créer une conversion, vous devez être le propriétaire du type source ou destination. Pour créer une conversion compatible binairement, vous devez être superutilisateur. (Cette restriction est faite parce qu'une conversion compatible binairement erronée peut facilement causer un arrêt brutal du serveur.)

# Paramètres

*typesource*
: Le nom du type de données source dans la conversion.

*typecible*
: Le nom du type de données cible dans la conversion.

*nomfonction*(*type_argument*)
: La fonction utilisée pour effectuer la conversion. Le nom de la fonction pourrait être qualifié du nom du schéma. Si ce n'est pas le cas, la fonction sera recherchée dans le chemin des schémas. Le type de données résultant de la fonction doit correspondre au type cible de la conversion. Ses arguments sont discutés ci–dessous.

`WITHOUT FUNCTION`
: Indique que le type source et le type cible sont compatibles binairement, donc aucune fonction n'est requise pour effectuer la conversion.

`AS ASSIGNMENT`
: Indique que la conversion pourrait être appelée implicitement dans les contextes d'affectation.

`AS IMPLICIT`
: Indique que la conversion pourrait être appelée implicitement dans tout contexte.

Les fonctions de conversion peuvent avoir de un à trois arguments. Le premier argument doit avoir le même type que celui de la source de la conversion. Le deuxième argument, si fourni, doit être de type `integer`. Il correspond au modificateur de type associé au type de destination, ou il vaut `-1` si il n'y en a pas. Le troisième argument, si fourni, doit être de type `boolean`. Il vaut `true` si la conversion est explicite, `false` dans le cas contraire. Bizarrement, les spécifications SQL demandent différents comportements pour les conversions explicites et implicites dans certains cas. Ce paramètre est fourni pour des fonctions qui doivent implémenter de tel cas. Il n'est pas recommandé que vous conceviez vos propres types de données qui entreraient dans ce cas de figure.

Généralement, une conversion doit avoir une source et une destination de type différent. Cependant, il est

permis de déclarer une conversion avec un type source identique au type destination si la fonction de conversion a plus d'un argument. C'est utilisé pour représenter des fonctions de restrictions sur une longueur d'un type spécifique. La fonction nommée est utilisée pour restreindre la valeur d'un type suivant la valeur du modificateur de type donnée par le second paramètre (depuis que la grammaire actuelle permet seulement a certains types d'avoir des modificateurs de type, cette fonctionnalité n'est d'aucun intérêt pour les types sources définis par l'utilisateur, néanmoins nous le signalons).

Quand une conversion a un type source et un type destination différent et que la fonction a plus d'un argument, alors la conversion d'un type vers un autre et la restriction sur la longeur du type destination sont faits en une seule etape. Quand aucune entrée n'est disponible, alors la restriction du type qui utilise un modificateur de type implique deux étapes, une pour la conversion entre les types de données et la seconde pour appliquer le modificateur.

# Notes

Utilisez *DROP CAST* pour supprimer les conversions définies par l'utilisateur.

Rappelez−vous que si vous souhaitez être capable de convertir les types dans les deux sens, vous devez déclarer explicitement les deux sens.

Avant PostgreSQL 7.3, chaque fonction qui avait le même nom qu'un type de données, envoyait ce type de données et prenait un argument d'un autre type était automatiquement détectée comme une fonction de conversion. Ceci a été abandonné lors de l'introduction des schémas et pour être capable de représenter des conversions compatibles binairement dans les catalogues système. Les fonctions de conversion intégrées suivent toujours le même schéma de nommage mais elle doivent maintenant être données comme conversion dans le catalogue système `pg_cast`.

Bien que cela ne soit pas requis, il est recommandé que vous continuez à suivre l'ancienne convention de nommage des fonctions de conversion après le type de données de destination. Beaucoup d'utilisateurs sont habitués à convertir des types de données en utilisant un style de notation de fonction, c'est−à−dire *nom_type*(*x*). En fait, cette notation n'est ni plus ni moins un appel à une fonction de conversion. Ce n'est pas forcément géré comme une conversion. Si vos fonctions de conversions ne sont pas nommées de tel façon a supporter cette convention alors vous aurez des utilisateurs surpris. Depuis que PostgreSQL permet de surcharger le même nom de fonction avec différents types d'argument, il n'y a aucune difficulté pour avoir plusieurs fonctions de conversion avec des types différents qui utilise le même nom de type destination.

> **Note :** Il y a un petit mensonge dans le paragraphe précédent : il existe un cas dans lequel `pg_cast` sera utilisé pour résoudre le sens d'un appel de fonction évidente. Si un appel de fonction *nom*(*x*) coincide avec aucune fonction existante, alors *nom*(*x*) est le nom du type de donnée et `pg_cast` indique une conversion binairement compatible depuis le type de *x*, alors l'appel sera interprété comme une conversion explicite. Cette exception est faite de telle façon à ce que les conversions compatibles puissent etre invoquées en utilisant une syntaxe fonctionnel, même si elles n'ont aucune fonction (de conversion).

# Exemples

Pour créer une conversion du type `text` vers le type `int4` en utilisant la fonction `int4(text)` :

```
CREATE CAST (text AS int4) WITH FUNCTION int4(text);
```

(Cette conversion est déjà prédéfinie dans le système.)

## Compatibilité

La commande `CREATE CAST` est conforme à SQL:1999 sauf que SQL:1999 ne parle pas des types compatibles binairement ou d'arguments supplémentaires pour les fonctions d'implémentation. `AS IMPLICIT` est aussi une extension PostgreSQL.

## Voir aussi

*CREATE FUNCTION*, *CREATE TYPE*, *DROP CAST*

# CREATE CONSTRAINT TRIGGER

## Nom

CREATE CONSTRAINT TRIGGER −− définit un nouveau déclencheur contrainte

## Synopsis

```
CREATE CONSTRAINT TRIGGER nom
    AFTER événements ON table contrainte attributs
    FOR EACH ROW EXECUTE PROCEDURE fonction ( args )
```

## Description

`CREATE CONSTRAINT TRIGGER` est utilisé à l'intérieur de `CREATE TABLE`/`ALTER TABLE` et par pg_dump pour créer les déclencheurs spéciaux pour l'intégrité référentielle. Elle n'a pas pour but une utilisation générale.

## Paramètres

*nom*
: Le nom du déclencheur en contrainte.

*événements*
: Les catégories d'événements pour lesquelles ce déclencheur sera lancé.

*table*
: Le nom (qualifié ou non du nom du schéma) de la table dans laquelle a lieu les événements du déclencheur.

*contrainte*
: Spécification d'une contrainte actuelle.

*attributs*
: Les attributs de la contrainte.

*fonction*(*args*)
: La fonction à appeler faisant parti du traitement du déclencheur.

# CREATE CONVERSION

## Nom

CREATE CONVERSION −− définir une nouvelle conversion de codage

## Synopsis

```
CREATE [DEFAULT] CONVERSION nom
    FOR codage_source TO codage_dest FROM fonction
```

## Description

`CREATE CONVERSION` définit une nouvelle conversion entre les codages d'ensembles de caractères. Les noms de conversion pourront être utilisés dans la fonction `convert` pour spécifier une conversion de codage particulière. De plus, les conversions marquées `DEFAULT` peuvent être utilisées pour une conversion de codage automatique entre le client et le serveur. Pour y arriver, deux conversions, du codage A à B *et* du codage B à A, doivent être définies.

Pour être capable de créer conversion, vous devez avoir le droit `EXECUTE` sur la fonction et le droit `CREATE` sur le schéma de destination.

## Paramètres

`DEFAULT`
: La clause `DEFAULT` indique que cette conversion est celle par défaut pour un codage de cette source vers cette destination. Il ne doit y avoir qu'un codage par défaut dans un schéma pour une paire de codage.

*`nom`*
: Le nom de la conversion. Ce nom pourrait être qualifié avec le nom du schéma. Si ce n'est pas le cas, la conversion est définie dans le schéma actuel. Le nom de la conversion doit être unique dans un schéma.

*`codage_source`*
: Le nom du codage source.

*`codage_dest`*
: Le nom du codage destination.

*`fonction`*
: La fonction utilisée pour réaliser la conversion. Le nom de la fonction pourrait être qualifié avec le nom du schéma. Si ce n'est pas le cas, la fonction sera cherchée dans le chemin.

  La fonction doit avoir la signature suivante :

```
conv_proc(
    integer,  -- ID codage source
    integer,  -- ID codage destination
    cstring,  -- chaîne source (chaîne C terminée par un caractère nul)
```

```
        internal, -- destination (remplie avec une chaîne C terminée par un caractère nul)
        integer   -- longueur de la chaîne source
    ) RETURNS void;
```

# Notes

Utilisez `DROP CONVERSION` pour supprimer une conversion définie par l'utilisateur.

Les droits requis pour créer une conversion pourraient être modifiées dans une version ultérieure.

# Exemples

Pour créer une conversion du codage `UNICODE` vers le codage `LATIN1` en utilisant `mafonc` :

```
CREATE CONVERSION maconv FOR 'UNICODE' TO 'LATIN1' FROM mafonc;
```

# Compatibilité

`CREATE CONVERSION` est une extension PostgreSQL. Il n'existe pas d'instruction `CREATE CONVERSION` dans le standard SQL.

# Voir aussi

*ALTER CONVERSION*, *CREATE FUNCTION*, *DROP CONVERSION*

# CREATE DATABASE

## Nom

CREATE DATABASE −− crée une nouvelle base de données

## Synopsis

```
CREATE DATABASE nom
    [ [ WITH ] [ OWNER [=] propriétaire ]
           [ TEMPLATE [=] modèle
]
           [ ENCODING [=] codage
]
           [ TABLESPACE [=] espacelogique ] ]
```

## Description

`CREATE DATABASE` crée une nouvelle base de données PostgreSQL.

Pour créer une base de données, vous devez être super−utilisateur ou avoir le droit spécial `CREATEDB`. Regardez ***CREATE USER***.

Normalement, le créateur devient le propriétaire de la nouvelle base. Les superutilisateurs peuvent créer des bases dont le propriétaire sera un autre utilisateur en utilisant la clause `OWNER`. Ils peuvent même créer des bases qui appartiendront à des utilisateurs qui n'ont aucun droits spéciaux. Les utilisateurs qui ne sont pas superutilisateurs mais qui possèdent le privilège `CREATEDB` peuvent seulement créer des bases de données dont ils seront les propriétaires.

Par défaut, la nouvelle base de données sera créée en clonant la base système standard `template1`. Un modèle différent peut être spécifié en écrivant `TEMPLATE` *nom*. En particulier, en écrivant `TEMPLATE template0`, vous pouvez créer une base de données vierge contenant seulement les objets standards pré−définis par votre version de PostgreSQL. Ceci est utile si vous souhaitez éviter de copier les objets de l'installation locale qui ont pû être ajoutés à `template1`.

## Paramètres

*nom*
: Le nom de la base de données à créer.

*propriétaire*
: Un nom d'utilisateur de la base qui sera le propriétaire de la nouvelle base de données ou `DEFAULT` pour utiliser l'option par défaut (c'est−à−dire utiliser le nom de l'utilisateur qui exécute la commande).

*modèle*
: Le nom du modèle depuis lequel créer la nouvelle base de données ou `DEFAULT` pour utiliser le modèle par défaut (`template1`).

*encodage*

Jeu d'encodage à utiliser par la nouvelle base de données. Spécifiez une chaîne (par exemple `'SQL_ASCII'`), un nombre d'encodage de type entier ou `DEFAULT` pour utiliser l'encodage par défaut. Les ensembles de caractères supportés par le serveur PostgreSQL sont décrits dans Section 20.2.1.

*espacelogique*

Le nom de l'espace logique qui sera associé avec la nouvelle base de données ou `DEFAULT` pour utiliser l'espace logique de la base de données modèle. Cet espace logique sera l'espace logique par défaut utilisé pour les objets créés dans cette base de données. Voir *CREATE TABLESPACE* pour plus d'informations.

Les paramètres optionnels peuvent être écrits dans n'importe quel ordre, pas seulement l'ordre illustré au−dessus.

# Notes

`CREATE DATABASE` ne peut pas être exécuté à l'intérieur d'un bloc de transactions.

Les erreurs suivants la ligne << could not initialize database directory >> (ne peut pas initialiser le répertoire de la base de données) sont le plus souvent dûes à des droits insuffisants sur le répertoire de données, à un disque plein ou à un autre problème du système de fichiers.

Utilisez l'instruction *DROP DATABASE* pour supprimer la base de données.

Le programme *createdb* est un emballage autour de cette commande. Il est fourni pour la convenance.

Bien qu'il soit possible de copier une base de données autre que `template1` en spécifiant son nom comme modèle, ceci n'est pas (encore) prévu comme un service << `COPY DATABASE` >> d'usage général. Nous recommandons que les bases de données utilisées comme modèles soient en mode lecture seule. Regardez la Section 18.3 pour plus d'informations.

# Exemples

Pour créer une nouvelle base de données :

```
CREATE DATABASE lusiadas;
```

Pour créer une base de données `ventes` possédée par l'utilisateur `app_ventes` avec l'espace logique `espace_ventes` par défaut :

```
CREATE DATABASE ventes OWNER app_ventes TABLESPACE espace_ventes;
```

Pour créer une base de données `musique` qui supporte l'ensemble de caractères ISO−8859−1 :

```
CREATE DATABASE musique ENCODING 'LATIN1';
```

# Compatibilité

Il n'existe pas d'instruction `CREATE DATABASE` dans le standard SQL. Les bases de données sont équivalentes aux catalogues, dont la création est définie lors de l'implémentation.

# CREATE DOMAIN

## Nom

CREATE DOMAIN −− définit un nouveau domaine

## Synopsis

```
CREATE DOMAIN nom [AS] type_donnee
    [ DEFAULT expression ]
    [ contrainte [ ... ] ]

où contrainte est :

[ CONSTRAINT nom_contrainte ]
{ NOT NULL | NULL | CHECK (expression) }
```

## Description

`CREATE DOMAIN` crée un nouveau domaine de données. L'utilisateur définissant un domaine devient son propriétaire.

Si un nom de schéma est donné (par exemple, `CREATE DOMAIN monschema.mondomaine ...`), alors le domaine est créé dans le schéma spécifié. Sinon, il est créé dans le schéma actuel. Le nom du domaine doit être unique parmi les types et domaines existant dans son schéma.

Les domaines sont utiles pour faire une abstraction des champs communs entre deux tables en un seul emplacement. Par exemple, une colonne d'adresse de courrier électronique utilisé dans plusieurs tables, toutes avec les mêmes propriétés. Définissez un domaine et utilisez−le plutôt que de configurer les contraintes de chaque table individuellement.

## Paramètres

*nom*
: Le nom d'un domaine à créer (pouvant être qualifié du nom du schéma).

*type_donnees*
: Le type de données sous−jacent du domaine. Il peut contenir des spécifications de tableau.

`DEFAULT` *expression*
: La clause `DEFAULT` définit une valeur par défaut pour les colonnes d'un type de données domaine. La valeur correspond à toute expression de variable (mais les sous−requêtes ne sont pas autorisées). Le type de données de l'expression par défaut doit correspondre au type de données du domaine. Si la valeur par défaut n'est pas indiquée, alors il s'agit de la valeur NULL.

  L'expression par défaut sera utilisée dans toute opération d'insertion qui ne spécifierait pas de valeur pour cette colonne. Si une valeur par défaut est définie pour une colonne particulière, elle surcharge toute valeur par défaut associée avec le domaine. En retour, la valeur par défaut surcharge tout valeur

par défaut associée avec le type de données sous−jacent.

`CONSTRAINT` *`nom_contrainte`*
: Un nom optionnel pour une contrainte. Si non spécifié, le système en génère un.

`NOT NULL`
: Les valeurs de ce domaine n'ont pas le droit d'être NULL.

`NULL`
: Les valeurs de ce domaine peuvent être NULL. C'est la valeur par défaut.

  Cette clause a seulement pour but la compatibilité avec les bases de données SQL non standard. Son utilisation n'est pas encouragée dans les nouvelles applications.

`CHECK (`*`expression`*`)`
: Les clauses `CHECK` spécifient des contraintes d'intégrité ou des tests que les valeurs du domaine doivent satisfaire. Chaque contrainte doit être une expression produisant un résultat booléen. Elle devrait utiliser le nom `VALUE` pour se référer à la valeur en cours de tests.

  Actuellement, les expressions `CHECK` ne peuvent ni contenir de sous−requêtes ni se référer à des variables autres que `VALUE`.

# Exemples

Cet exemple crée le type de données `code_postal_us`, puis l'utilise dans la définition d'une table. Un test d'expression rationnelle est utilisé pour vérifier que la valeur ressemble à un code postal US valide.

```
CREATE DOMAIN code_postal_us AS TEXT
CHECK(
   VALUE ~ '^\\d{5}$'
OR VALUE ~ '^\\d{5}-\\d{4}$'
);

CREATE TABLE courrier_us (
  id_adresse SERIAL NOT NULL PRIMARY KEY
, rue1 TEXT NOT NULL
, rue2 TEXT
, rue3 TEXT
, ville TEXT NOT NULL
, code_postal code_postal_us NOT NULL
);
```

# Compatibilité

La commande `CREATE DOMAIN` est conforme au standard SQL.

# Voir aussi

*ALTER DOMAIN*, *DROP DOMAIN*

# CREATE FUNCTION

## Nom

CREATE FUNCTION −− définit une nouvelle fonction

## Synopsis

```
CREATE [ OR REPLACE ] FUNCTION nom
( [ [ nomarg ] typearg [, ...] ] )
    RETURNS type_ret
  { LANGUAGE nomlang
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    | AS 'definition'
    | AS 'fichier_obj', 'symbole_lien'
  } ...
    [ WITH ( attribut [, ...] ) ]
```

## Description

`CREATE FUNCTION` définit une nouvelle fonction. `CREATE OR REPLACE FUNCTION` créera une nouvelle fonction ou remplacera une fonction existante.

Si un nom de schéma est inclus, alors la fonction est créée dans le schéma spécifié. Sinon, elle est créée dans le schéma courant. Le nom de la nouvelle fonction ne doit pas correspondre à une autre fonction existante avec les mêmes types d'argument dans le même schéma. Néanmoins, les fonctions de types d'arguments différents pourraient partager un nom (ceci est appelé le *surchargement*).

Pour mettre à jour la définition d'une fonction existante, utilisez `CREATE OR REPLACE FUNCTION`. Il n'est pas possible de changer le nom ou les types d'argument d'une fonction de cette façon (si vous avez essayé, vous devrez seulement créer une nouvelle fonction distincte). De même, `CREATE OR REPLACE FUNCTION` ne vous laissera pas modifier le type en retour d'une fonction existante. Pour cela, vous devez supprimer et recréer la fonction.

Si vous supprimez, puis recréez une fonction, la nouvelle fonction n'est pas la même entité que l'ancienne ; vous devrez supprimer les règles, vues, déclencheurs, etc. qui référençaient l'ancienne fonction. Utilisez `CREATE OR REPLACE FUNCTION` pour modifier la définition d'une fonction sans casser d'objets qui se réfèrent à la fonction.

L'utilisateur qui crée la fonction devient le propriétaire de la fonction.

## Paramètres

*nom*
: Le nom de la fonction à créer (pouvant être qualifié du nom du schéma).

*nomarg*
: Le nom d'un argument. Quelques langages (actuellement seulement PL/pgSQL) vous laissent utiliser le nom dans le corps de la fonction. Pour les autres langages, le nom de l'argument est une documentation supplémentaire.

*argtype*
: Le(s) type(s) de données des arguments de la fonction (pouvant être qualifié par le nom du schéma), s'il y en a. Les types des arguments peuvent être de base, composite ou de domaines, ou pourraient aussi référencer le type d'une colonne.

  Suivant le langage d'implémentation, il pourrait aussi être autorisé de spécifier des << pseudotypes >> plutôt que des `cstring`. Les pseudotypes indiquent que le type d'argument réel est soit non complètement spécifié, soit en dehors de l'ensemble des types de données ordinaires SQL.

  Le type d'une colonne est utilisé en écrivant *`nomtable.nomcolonne`*`%TYPE`. Utiliser cette fonctionnalité peut quelque fois aider à rendre une fonction dépendante des modifications de la définition d'une table.

*typeret*
: Le type de données en retour (pouvant être qualifié du nom du schéma). Le type de retour pourrait être un type de base, complexe ou un domaine, ou pourrait être spécifié pour référencer le type d'une colonne existante. Suivant le langage d'implémentation, il pourrait aussi être autorisé de spécifier un << pseudotype >> tel que `cstring`.

  Le modificateur `SETOF` indique que la fonction renverra un ensemble d'éléments plutôt qu'un seul élément.

  Le type d'une colonne est référencé en écrivant *`nomtable.nomcolonne`*`%TYPE`.

*nomlang*
: Le nom du langage dans laquelle la fonction est implémentée. Pourrait être `SQL`, `C`, `internal` ou le nom d'un langage de procédures défini par l'utilisateur. Pour une compatibilité descendante, le nom peut être englobé avec des guillemets simples.

`IMMUTABLE`
`STABLE`
`VOLATILE`
: Ces attributs informent le système s'il est sain de remplacer plusieurs évaluations de la fonction avec une seule évaluation pour une optimisation en exécution. Au plus un choix devra être donné. Si aucun n'apparaît, `VOLATILE` est la valeur par défaut.

  `IMMUTABLE` indique que la fonction renvoie toujours le même résultat si elle reçoit les mêmes valeurs en argument ; c'est−à−dire qu'elle n'effectue pas de recherches dans la base de données ou, autrement, qu'elle utilise l'information non présente directement dans la liste d'arguments. Si cette option est donnée, tout appel de la fonction avec des arguments constants peut être immédiatement remplacé par la valeur de la fonction.

  `STABLE` indique qu'à l'intérieur d'un seul parcours de la table, la fonction renverra le même résultat pour les mêmes valeurs d'argument, mais son résultat pourrait varier au travers des instructions SQL. Ceci est la sélection appropriée pour les fonctions dont les résultats dépendent des recherches en base de données, des variables paramètres (tel que la zone horaire en cours), etc. Notez aussi que la famille de fonctions `current_timestamp` est qualifiée de stable car leur valeur ne change pas à l'intérieur d'une transaction.

`VOLATILE` indique que la valeur de la fonction peut changer même avec un seul parcours de table, donc aucune optimisation ne peut être réalisée. Relativement peu de fonctions de bases de données sont volatiles dans ce sens ; quelques exemples sont `random()`, `currval()`, `timeofday()`. Notez que toute fonction qui a des effets de bord doit être classée comme volatile, même si son résultat est assez prévisible pour empêcher l'optimisation des appels ; un exemple est `setval()`.

Pour des détails supplémenaires, voir Section 31.6.

`CALLED ON NULL INPUT`
`RETURNS NULL ON NULL INPUT`
`STRICT`

`CALLED ON NULL INPUT` (la valeur par défaut) indique que la fonction sera appelée normalement quand certains de ses arguments sont NULL. C'est alors de la responsabilité de l'auteur de la fonction de vérifier les valeurs NULL si nécessaire et de répondre en conséquence.

`RETURNS NULL ON NULL INPUT` ou `STRICT` indiquent que la fonction renvoie toujours NULL si un de ces arguments est NULL. Si ce paramètre est spécifié, la fonction n'est pas exécutée quand il y a des arguments NULL ; à la place, un résultat NULL est automatiquement renvoyé.

`[EXTERNAL] SECURITY INVOKER`
`[EXTERNAL] SECURITY DEFINER`

`SECURITY INVOKER` indique que la fonction doit être exécutée avec les droits de l'utilisateur qui l'appelle. C'est la valeur par défaut. `SECURITY DEFINER` spécifie que la fonction doit être exécutée avec les droits de l'utilisateur qui l'a créé.

Le mot clé `EXTERNAL` est présent pour la conformité SQL mais est optionnelle car, contrairement à SQL, cette fonctionnalité ne s'applique qu'aux fonctions externes.

*`definition`*

Une constante de type chaîne définissant la fonction ; la signification dépend du langage. Cela pourrait être un nom de fonction interne, le chemin vers un fichier objet, une commande SQL ou un texte dans un langage de procédures.

*`fichier_obj, symbole_lien`*

Cette forme de clause `AS` est utilisée pour les fonctions en langage C chargeables dynamiquement quand le nom de la fonction dans le code source C n'est pas le même que celui de la fonction C. La chaîne *`fichier_obj`* est le nom du fichier contenant l'objet chargeable dynamiquement et *`symbole_lien`* est le symbole de lien de la fonction, c'est−à−dire le nom de la fonction dans le code source C. Si ce lien est omis, il est supposé être le même que le nom de la fonction en cours de définition.

*`attribut`*

La façon historique de spécifier les morceaux optionnels d'informations sur la fonction. Les attributs suivants pourraient apparaître ici :

`isStrict`
: Équivalent à `STRICT` ou `RETURNS NULL ON NULL INPUT`

`isCachable`
: `isCachable` est un équivalent obsolète de `IMMUTABLE` ; il est toujours accepté pour des raisons de compatibilité ascendante.

Les noms d'attribut ne sont pas sensibles à la casse.

# Notes

Référez−vous à Section 31.3 pour plus d'informations sur l'écriture de fonctions.

La syntaxe complète de type SQL est autorisée pour les arguments en entrée et pour la valeur de sortie. Néanmoins, quelques détails de spécification de type (c'est−à−dire le champ précision pour le type `numeric`) sont de la responsabilité de l'implémentation de la fonction sous−jacente et sont silencieusement avalés (c'est−à−dire non reconnus ou forcés) par la commande `CREATE FUNCTION`.

PostgreSQL autorise le *surchargement* de fonctions ; c'est−à−dire que le même nom peut être utilisé pour plusieurs fonctions différentes si tant est qu'elles ont des types d'arguments distincts. Néanmoins, les noms C de toutes les fonctions doivent être différents, donc vous devez donner des noms différentes aux fonctions C suchargées (par exemple, utilisez les types d'argument comme morceaux des noms des fonctions C).

Lors d'appels répétés à `CREATE FUNCTION` et se référant au même fichier objet, le fichier est chargé une seule fois. Pour décharger et recharger le fichier (peut−être pendant le développement), utilisez la commande *LOAD*.

Utilisez *DROP FUNCTION* pour supprimer les fonctions définies par l'utilisateur.

Il est souvent utile d'utiliser les guillemets dollar (voir Section 4.1.2.2) pour écrire la chaîne de définition d'une fonction, plutôt que la syntaxe habituelle à guillemets simples. Sans guillemets dollar, tout guillemet simple et tout antislash devrait être échappé en les doublant dans la définition de la fonction.

Pour être capable de définir une fonction, l'utilisateur doit avoir le droit `USAGE` sur le langage.

# Exemples

Voici un exemple trivial pour vous aider à commencer. Pour plus d'informations et d'exemples, voir Section 31.3.

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

Incrémente un entier, en utilisant le nom de l'argument, dans PL/pgSQL :

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

# Compatibilité

Une commande `CREATE FUNCTION` est définie en SQL:1999 et ultérieur. La version PostgreSQL est similaire mais pas entièrement compatible. Les attributs ne sont pas portables, pas plus que les différents

langages disponibles.

## Voir aussi

*ALTER FUNCTION*, *DROP FUNCTION*, *GRANT*, *LOAD*, *REVOKE*, *createlang*

# CREATE GROUP

## Nom

CREATE GROUP −− définit un nouveau groupe d'utilisateurs

## Synopsis

```
CREATE GROUP nom [ [ WITH ]
option [ ... ] ]

où option peut être :

     SYSID gid
   | USER  nomutilisateur [, ...]
```

## Description

`CREATE GROUP` créera un nouveau groupe d'utilisateurs. Vous devez être le superutilisateur de la base de données pour utiliser cette commande.

Notez que les utilisateurs comme les groupes sont créés au niveau du groupe de bases de données et sont donc valides pour toutes les bases de données du groupe.

Utilisez *ALTER GROUP* pour modifier l'appartenance d'un groupe et *DROP GROUP* pour en supprimer un.

## Paramètres

*nom*
: Le nom du groupe.

*gid*
: La clause `SYSID` peut être utilisée pour choisir l'identifiant de groupe PostgreSQL pour le nouveau groupe. Ceci n'est normalement pas nécessaire mais pourrait être utile si vous avez besoin d'un groupe référencé dans la définition des droits d'un objet.

  Si ce n'est pas spécifié, l'identifiant de groupe ayant la plus grande valeur plus un, avec un minimum de 100, sera utilisé par défaut.

*username*
: Une liste d'utilisateurs à inclure dans le groupe. Les utilisateurs doivent déjà exister.

## Exemples

Créer un groupe vide :

```
CREATE GROUP staff;
```

Créer un groupe avec des membres :

```
CREATE GROUP marketing WITH USER jonathan, david;
```

# Compatibilité

Il n'existe pas d'instruction `CREATE GROUP` dans le standard SQL. Les rôles sont le concept similaire aux groupes.

# Voir aussi

*ALTER GROUP*, *DROP GROUP*

# CREATE INDEX

## Nom

CREATE INDEX −− définit un nouvel index

## Synopsis

```
CREATE [ UNIQUE ] INDEX nom ON table [ USING méthode ]
    ( { colonne | ( expression ) } [ classeop ] [, ...] )
    [ TABLESPACE espacelogique ]
    [ WHERE prédicat ]
```

## Description

`CREATE INDEX` construit un index *`nom_index`* sur la table spécifiée. Les index sont principalement utilisés pour améliorer les performances de la base de données (bien qu'une utilisation inappropriée résultera en des performances moindres).

Les champs clé pour l'index sont spécifiés par les noms des colonnes ou comme des expressions écrites entre parenthèses. Plusieurs champs peuvent être spécifiés si la méthode d'indexage supporte les index multi−colonnes.

Un champ index peut être une expression calculée à partir des valeurs d'une ou plusieurs colonnes d'une ligne d'une table. Cette fonctionnalité peut être utilisée pour obtenir un accès rapide aux données basées sur quelques transformations des données basiques. Par exemple, un index calculé sur `upper(col)` permettra l'utilisation d'un index par `WHERE upper(col) = 'JIM'`.

PostgreSQL fournit les méthodes d'indexage B−tree, R−tree, hash et GiST. La méthode d'indexage B−tree est une implémentation des arbres balancées à haute concurrence de Lehman−Yao. La méthode d'indexage R−tree implémente les R−tree en utilisant l'algorithme divisé quadratique de Guttman. La méthode d'indexage hash est une implémentation du découpage linaire de Litwin. Les utilisateurs peuvent aussi définir leur propre méthode d'indexage mais ceci est particulièrement compliqué.

Lorsque la clause `WHERE` est présente, un *index partiel* est créé. Un index partiel est un index contenant des entrées pour seulement une portion d'une table, habituellement une portion qui est plus utile pour l'indexage que le reste de la table. Par exemple, si vous disposez d'une table contenant des ordres facturés et non facturés où les ordres non facturés prennent une petite fraction du total de la table et qu'il s'agit en plus d'une section fréquemment utilisée, vous pouvez améliorer les performances en créant un index sur cette portion. Une autre application possible est d'utiliser `WHERE` avec `UNIQUE` pour renforcer l'unicité sur un sous−ensemble d'une table. Voir Section 11.7 pour plus de discussions.

L'expression utilisée dans la clause `WHERE` pourrait référencer seulement les colonnes de la table sous−jacente mais il peut utiliser toute les colonnes, pas seulement celles en cours d'indexage. Actuellement, les sous−requêtes et les expressions d'agrégats sont aussi interdites dans `WHERE`. Les mêmes restrictions s'appliquent aux champs d'index qui sont des expressions.

Toutes les fonctions et opérateurs utilisés dans la définition d'index doivent être << immutable >>, c'est−à−dire que leur résultat doit uniquement dépendre de leurs arguments et jamais d'une influence externe (telle que le contenu d'une autre table ou l'heure actuelle). Cette restriction permet de s'assurer que le comportement de l'index est bien défini. Pour utiliser une fonction définie par l'utilisateur dans une expression d'index ou dans une clause `WHERE`, rappelez−vous de marquer la fonction comme immutable lorsque vous la créez.

# Paramètres

`UNIQUE`
: Fait que le système vérifie les valeurs dupliquées dans la table à la création de l'index (si des données existent déjà) et à chaque fois qu'une donnée est ajoutée. Les tentatives d'insertion ou de mises à jour qui résulteraient en des entrées dupliquées généreront une erreur.

*nom*
: Le nom de l'index à créer. Aucun nom de schéma ne peut être inclus ici ; l'index est toujours créé dans le même schéma que sa table parent.

*table*
: Le nom de la table à indexer (pouvant être qualifié du nom du schéma).

*méthode*
: Le nom de la méthode à utiliser pour l'index. Les choix sont `btree`, `hash`, `rtree` et `gist`. La méthode par défaut est `btree`.

*colonne*
: Le nom d'une colonne de la table.

*expression*
: Une expression basée sur une ou plusieurs colonnes de la table. L'expression doit habituellement être écrite en l'entourant de parenthèses, comme le montre la syntaxe. Néanmoins, les parenthèses peuvent être oubliées si l'expression est de la forme d'un appel de fonction.

*classeop*
: Le nom d'une classe d'opérateur. Voir ci−dessous pour les détails.

*espacelogique*
: L'espace logique dans lequel créer l'index. Si non spécifié, default_tablespace est utilisé ou l'espace logique par défaut de la base de données si `default_tablespace` est une chaîne vide.

*prédicat*
: L'expression de contrainte pour un index partiel.

# Notes

Voir Chapitre 11 pour des informations sur le moment où les index sont utilisés, quand ils ne le sont pas et dans quelles situations particulières ils peuvent être utiles.

Actuellement, seules les méthodes d'indexage B−tree et GiST supportent les index multi−colonnes. Jusqu'à 32 champs peuvent être spécifiés par défaut. (Cette limite peut être modifiée lors de la construction de PostgreSQL.) Seul B−tree supporte actuellement les index uniques.

Une *classe d'opérateur* peut être spécifiée pour chaque colonne d'un index. La classe d'opérateur identifie les opérateurs à utiliser par l'index pour cette colonne. Par exemple, un index B−tree sur des entiers sur quatre octets utiliserait la classe `int4_ops` ; cette classe d'opérateur inclut des fonctions de comparaison pour les entiers sur quatre octets. En pratique, la classe d'opérateur par défaut pour le type de données de la colonne est

généralement suffisant. Le point principal pour avoir des classes d'opérateur est que pour certains types de données, il pourrait y avoir plus d'un ordonnancement significatif. Par exemple, nous pourrions vouloir trier un type de données << nombre complexe >> soit par sa valeur absolue soit par sa partie réelle. Nous pourrions le faire en définissant deux classes d'opérateur pour le type de données, puis en sélectionnant la bonne classe lors de la création d'un index. Plus d'informations sur les classes d'opérateurs sont disponibles dans Section 11.6 et dans Section 31.14.

Utilisez *DROP INDEX* pour supprimer un index.

Les index ne sont pas utilisés pour les clauses `IS NULL` par défaut. La meilleure façon d'utiliser des index dans de tels cas est de créer un index partiel en utilisant un prédicat `IS NULL`.

# Exemples

Pour créer un index B−tree sur la colonne `titre` dans la table `films` :

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

Pour créer un index sur la colonne `code` de la table `films` et donner à l'index l'emplacement de l'espace logique `espaceindex` :

```
CREATE INDEX code_idx ON films(code) TABLESPACE espaceindex;
```

# Compatibilité

`CREATE INDEX` est une extension du langage PostgreSQL. Les index n'existent pas dans le standard SQL.

# Voir aussi

*ALTER INDEX*, *DROP INDEX*

# CREATE LANGUAGE

## Nom

CREATE LANGUAGE −− définit un nouveau langage de procédures

## Synopsis

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE nom
    HANDLER gestionnaire_appel [
VALIDATOR fonction_validation ]
```

## Description

En utilisant `CREATE LANGUAGE`, un utilisateur PostgreSQL peut enregistrer un nouveau langage de procédure sur une base de données PostgreSQL. En conséquence, les fonctions et les procédures de déclencheurs peuvent être définies dans ce nouveau langage. L'utilisateur doit avoir les droits de superutilisateur de PostgreSQL pour enregistrer un nouveau langage.

`CREATE LANGUAGE` associe réellement le nom du langage avec un gestionnaire d'appels qui est responsable de l'exécution des fonctions écrites dans le langage. Référez−vous à Chapitre 34 pour plus d'informations sur les gestionnaires d'appels.

Notez que les langages de procédures sont locaux au niveau des bases de données individuelles. Pour rendre disponible un langage dans toutes les bases de données par défaut, il devrait être installé dans la base de données `template1`.

## Paramètres

`TRUSTED`
: `TRUSTED` spécifie que le gestionnaire d'appels du langage est sûr, c'est−à−dire qu'il n'offre pas de fonctions surpassant les restrictions d'accès aux utilisateurs. Si ce mot clé est omis à l'enregistrement de ce langage, seuls les utilisateurs disposant du droit superutilisateur de PostgreSQL peuvent utiliser ce langage pour créer de nouvelles fonctions.

`PROCEDURAL`
: Aucune influence.

*`nom`*
: Le nom du nouveau langage de procédures. Le nom du langage n'est pas sensible à la casse. Le nom doit être unique parmi les langages de la base de données.

  Pour une compatibilité descendante, le nom doit être entouré par des guillemets simples.

`HANDLER` *`gestionnaire_appel`*
: *`gestionnaire_appels`* est le nom d'une fonction précédemment enregistrée qui sera appelée pour exécuter les fonctions du langage de procédures. Le gestionnaire d'appels pour un langage de procédures doit être écrit dans un langage compilé comme le C avec la convention d'appel version 1 et

enregistré sur PostgreSQL comme une fonction ne prenant aucun argument et retournant le type `language_handler`, un type spécifiquement utilisé pour identifier la fonction comme gestionnaire d'appels.

`VALIDATOR` *`fonction_validation`*

*`fonction_validation`* est le nom d'une fonction précédemment enregistrée qui sera appelée lorsqu'une nouveau fonction sera créée avec ce langage, pour valider la nouvelle fonction. Si aucune fonction de validation n'est spécifiée, alors une nouvelle fonction ne sera pas vérifiée à sa création. La fonction de validation prend un argument de type `oid`, qui sera l'OID de la fonction à créer, et renverra typiquement `void`.

Une fonction de validation inspecterait typiquement le corps de la fonction pour s'assurer de la justesse syntaxique mais il regarderait aussi d'autres propriétés de la fonction, par exemple si le langage ne peut pas gérer certains types d'argument. Pour signaler une erreur, la fonction de validation devrait utiliser la fonction `ereport()`. La valeur de retour de la fonction est ignorée.

# Notes

Cette commande ne devrait normalement pas être exécutée directement par les utilisateurs. Pour les langages de procédure fournis dans la distribution PostgreSQL, le programme createlang devrait être utilisé, qui installera aussi le bon gestionnaire d'appels. (`createlang` appellera `CREATE LANGUAGE` en interne.)

Dans les versions de PostgreSQL antérieures à la 7.3, il était nécessaire de déclarer les fonctions du gestionnaire comme renvoyant le type `opaque`, plutôt que `language_handler`. Pour supporter le chargement des anciens fichiers de sauvegarde, `CREATE LANGUAGE` acceptera une fonction déclarée renvoyant le type `opaque`, mais affichera un message d'avertissement et modifié le type renvoyé par la fonction en `language_handler`.

Utilisez la commande *CREATE FUNCTION* pour créer une nouvelle fonction.

Utilisez *DROP LANGUAGE*, ou encore mieux le programme droplang, pour supprimer les langages de procédures.

Le catalogue système `pg_language` (voir Section 41.18) enregistre des informations sur les langages actuellement installés. De plus, `createlang` a une option pour lister les langages installés.

Pour être capable d'utiliser un langage de procédures, un utilisateur doit avoir le droit `USAGE`. Le programme `createlang` donne automatiquement les droits à toute personne si le langage est « de confiance ».

# Exemples

Les deux commandes suivantes exécutées en séquence enregistreront un nouveau langage de procédures et le gestionnaire d'appels associé.

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

## Compatibilité

`CREATE LANGUAGE` est un extension de PostgreSQL.

## Voir aussi

*ALTER LANGUAGE*, *CREATE FUNCTION*, *DROP LANGUAGE*, *GRANT*, *REVOKE*, *createlang*, *droplang*

# CREATE OPERATOR

## Nom

CREATE OPERATOR −− définit un nouvel opérateur

## Synopsis

```
CREATE OPERATOR nom (
    PROCEDURE = nomfonc
    [, LEFTARG = typegauche ]
    [, RIGHTARG = typedroit ]
    [, COMMUTATOR = op_com ]
    [, NEGATOR = op_neg ]
    [, RESTRICT = proc_res ]
    [, JOIN = proc_join ]
    [, HASHES ] [, MERGES ]
    [, SORT1 = op_tri_gauche ]
    [, SORT2 = op_tri_droit ]
    [, LTCMP = op_inf ]
    [, GTCMP = op_sup ]
)
```

## Description

`CREATE OPERATOR` définit un nouvel opérateur, *nom*. L'utilisateur qui définit un opérateur en devient son propriétaire. Si un nom de schéma est donné, alors l'opérateur est créé dans le schéma spécifié. Sinon, il est créé dans le schéma courant.

Le nom de l'opérateur est une séquence d'au moins `NAMEDATALEN`−1 (63 par défaut) caractères de la liste suivante :

+ − * / < > = ~ ! @ # % ^ & | ` ?

Il existe quelques restrictions dans le choix du nom :

- −− et /* ne peuvent pas apparaître n'importe où dans le nom d'un opérateur car ils sont pris pour le début d'un commentaire.
- Un nom d'opérateur multicaractères ne peut pas finir avec + ou − sauf si le nom contient aussi au moins un de ces caractères :

  ~ ! @ # % ^ & | ` ?

  Par exemple, @− est un nom d'opérateur autorisé mais *− ne l'est pas. Cette restriction permet à PostgreSQL d'analyser les commandes compatibles SQL sans nécessiter d'espaces entre les jetons.

L'opérateur != correspond à <> en entrée, donc ces deux noms sont toujours équivalents.

Au moins un de LEFTARG et RIGHTARG doit être défini. Pour les opérateurs binaires, les deux doivent être définis. Pour les opérateurs unaires droits, seul LEFTARG devrait être défini alors que pour les opérateurs unaires gauches seul RIGHTARG devrait être défini.

La procédure *nomfonc* doit être été précédemment définie en utilisant CREATE FUNCTION et doit accepter le bon nombre d'arguments (soit un ou deux) des types indiqués.

Les autres clauses spécifient des clauses optionnelles d'optimisation d'opérateur. Leur signification est détaillée dans Section 31.13.

# Paramètres

*nom*
: Le nom de l'opérateur à définir. Voir ci−dessus pour les caractères autorisés. Le nom peut être qualifié avec le nom du schéma, par exemple CREATE OPERATOR monschema.+ (...). Sinon, l'opérateur est créé dans le schéma courant. Deux opérateurs dans le même schéma peuvent avoir le même nom s'ils opérent sur des types de données différents. Ceci est appelé le *surchargement*.

*nomfonc*
: La fonction utilisée pour implémenter cet opérateur.

*typegauche*
: Le type de données de l'opérande gauche de l'opérateur, s'il existe. Cette option sera omise pour un opérateur unaire gauche.

*typedroit*
: Le type de données de l'opérande droit de l'opérateur, s'il existe. Cette option sera omise pour un opérateur unaire droit.

*op_com*
: Le commutateur de cet opérateur.

*op_neg*
: La négation de cet opérateur.

*proc_res*
: La fonction d'estimation de la sélectivité restreinte pour cet opérateur.

*proc_join*
: La fonction d'estimation de la sélectivité de jointure pour cet opérateur.

HASHES
: Indique que cet opérateur peut supporter une jointure de découpage.

MERGES
: Indique que cet opérateur peut supporter une jointure d'assemblage.

*op_tri_gauche*
: Si cet opérateur peut supporter une jointure d'assemblage, l'opérateur inférieur qui trie le type de données à gauche de cet opérateur.

*op_tri_droit*
: Si cet opérateur peut supporter une jointure d'assemblage, l'opérateur supérieur qui trie le type de données à droite de cet opérateur.

*op_inf*
: Si cet opérateur peut supporter une jointure d'assemblage, l'opérateur inférieur qui compare les types de données de cet opérateur.

*op_sup*
: Si cet opérateur peut supporter une jointure d'assemblage, l'opérateur supérieur qui compare les types de données de cet opérateur.

Pour donner un nom d'opérateur qualifié d'un schéma dans *op_com* ou les autres arguments optionnels, utilisez la syntaxe OPERATOR() par exemple

```
COMMUTATOR = OPERATOR(mon_schema.===) ,
```

# Notes

Référez−vous à Section 31.12 pour plus d'informations.

Utilisez *DROP OPERATOR* pour supprimer les opérateurs définis par l'utilisateur sur une base de données. Utilisez *ALTER OPERATOR* pour modifier les opérateurs dans une base de données.

# Exemples

La commande suivante définit un nouvel opérateur, << area−equality >>, pour le type de données box :

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES,
    SORT1 = <<<,
    SORT2 = <<<
    -- Since sort operators were given, MERGES is implied.
    -- LTCMP and GTCMP are assumed to be < and > respectively
);
```

# Compatibilité

CREATE OPERATOR est une extension PostgreSQL. Il n'existe pas d'opérateurs définis par l'utilisateur dans le standard SQL.

# Voir aussi

*ALTER OPERATOR*, *CREATE OPERATOR CLASS*, *DROP OPERATOR*

# CREATE OPERATOR CLASS

## Nom

CREATE OPERATOR CLASS −− définit une nouvelle classe d'opérateur

## Synopsis

```
CREATE OPERATOR CLASS nom [ DEFAULT ] FOR TYPE type_données USING méthode_index AS
  {  OPERATOR numéro_stratégie nom_opérateur [ ( op_type, op_type ) ] [ RECHECK ]
   | FUNCTION numéro_support nom_fonction ( type_argument [, ...] )
   | STORAGE type_stockage
  } [, ... ]
```

## Description

`CREATE OPERATOR CLASS` crée une nouvelle classe d'opérateur. Une classe d'opérateur définit comment un certain type de données peut être utilisé avec un index. La classe d'opérateur spécifie que certains opérateurs joueront des rôles particuliers ou des << stratégies >> pour ce type de données et cette méthode d'indexage. La classe d'opérateur spécifie aussi les procédures de support pour être utilisée par la méthode d'indexage quand la classe d'opérateur est sélectionnée pour une colonne d'index. Tous les opérateurs et fonctions utilisés par une classe d'opérateur doivent être définis avant la création de la classe d'opérateur.

Si un nom de schéma est donné, alors la classe d'opérateur est créée dans le schéma spécifié. Sinon, elle est créée dans le schéma courant. Deux classes d'opérateur peuvent avoir le même nom seulement s'ils sont pour des méthodes d'indexage différentes.

L'utilisateur qui définit une classe d'opérateur devient son propriétaire. Actuellement, le créateur doit être un superutilisateur. (Cette restriction est faite parce qu'une définition erronée d'une classe d'opérateur pourrait gêner le serveur, voire causer un arrêt brutal de celui−ci.)

Actuellement, `CREATE OPERATOR CLASS` ne vérifie pas si la définition de la classe d'opérateur inclut tous les opérateurs et fonctions requis par la méthode d'indexage. C'est de la responsabilité de l'utilisateur de définir une classe d'opérateur valide.

Référez−vous au Section 31.14 pour plus d'informations.

## Paramètres

*nom*
: Le nom de la classe d'opérateur à créer. Ce nom pourrait être qualifié avec le nom du schéma.

`DEFAULT`
: Si présent, la classe d'opérateur deviendra la classe d'opérateur par défaut pour ce type de données. Au plus une classe d'opérateur peut être la classe par défaut pour un type de données et une méthode d'indexage spécifiques.

*type_données*

Le type de données de la colonne pour cette classe d'opérateur.

*méthode_index*
Le nom de la méthode d'indexage pour cette classe d'opérateur.

*numéro_stratégie*
Le numéro de stratégie de la méthode d'indexage pour un opérateur associé avec la classe d'opérateur.

*nom_opérateur*
Le nom (pouvant être qualifié du nom du schéma) d'un opérateur associé avec la classe d'opérateur.

*op_type*
Le type de données de l'opérande d'un opérateur ou NONE pour signifier un opérateur unaire. Les types de données de l'opérande pourraient être omis dans le cas normal où ils sont identiques au type de données de la classe d'opérateur.

RECHECK
Si présent, l'index est << à perte >> pour cet opérateur. Du coup, les lignes récupérées en utilisant l'index doivent être de nouveau vérifiées pour s'assurer qu'elles satisfont réellement la clause de qualification impliquant cet opérateur.

*numéro_support*
Le numéro de procédure du support pour cette méthode d'indexage pour une fonction associée avec la classe d'opérateur.

*nom_fonction*
Le nom (pouvant être qualifié avec le nom du schéma) d'une fonction qui est la procédure de support pour la méthode d'indexage sur la classe d'opérateur.

*types_argument*
Le(s) type(s) de données des paramètres de la fonction.

*type_stockage*
Le type de données réellement stocké dans l'index. Normalement, c'est le même que le type de données de la colonne mais certaines méthodes d'indexage (uniquement GiST au moment de l'écriture de ce document) autorisent une différence. La clause STORAGE doit être omise sauf si la méthode d'indexage autorise l'utilisation d'un type différent.

Les clauses OPERATOR, FUNCTION et STORAGE pourraient apparaître dans n'importe quel ordre.

# Notes

Les opérators ne devraient pas être définis par des fonctions SQL. Une fonction SQL a des chances d'être intégrée dans la requête appelante, ce qui empêchera l'optimiseur de reconnaître que la requête correspond à un index.

# Exemples

La commande issue de l'exemple suivant définit une classe d'opérateur d'indexage GiST pour le type de données _int4 (tableau de int4). Voir contrib/intarray/ pour l'exemple complet.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       =       RECHECK,
        OPERATOR        7       @,
        OPERATOR        8       ~,
        OPERATOR        20      @@ (_int4, query_int),
```

```
FUNCTION        1       g_int_consistent (internal, _int4, int4),
FUNCTION        2       g_int_union (bytea, internal),
FUNCTION        3       g_int_compress (internal),
FUNCTION        4       g_int_decompress (internal),
FUNCTION        5       g_int_penalty (internal, internal, internal),
FUNCTION        6       g_int_picksplit (internal, internal),
FUNCTION        7       g_int_same (_int4, _int4, internal);
```

# Compatibilité

`CREATE OPERATOR CLASS` est une extension PostgreSQL. Il n'existe pas d'instruction `CREATE OPERATOR CLASS` dans le standard SQL.

# Voir aussi

*ALTER OPERATOR CLASS*, *DROP OPERATOR CLASS*

# CREATE RULE

## Nom

CREATE RULE −− définit une nouvelle règle de réécriture

## Synopsis

```
CREATE [ OR REPLACE ] RULE nom AS
ON événement
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | commande | ( commande ; commande ... ) }
```

## Description

`CREATE RULE` définit une nouvelle règle s'appliquant à une table ou à une vue. `CREATE OR REPLACE RULE` soit créera une nouvelle règle soit remplacera une règle existante du même nom pour la même table.

Le système de règles de PostgreSQL autorise la définition d'actions alternatives à réaliser sur les insertions, mises à jour ou suppressions dans les tables de la base de données. Directement, une règle fait que des commandes supplémentaires soient exécutées lorsqu'une commande donnée est exécutée sur une table donnée. Autrement dit, une règle `INSTEAD` peut remplacer une commande donnée par un autre ou faire qu'une commande ne soit pas exécutée du tout. Les règles sont aussi utilisées pour implémenter les vues de tables. Il est important de réaliser qu'une règle est réellement un mécanisme de transformation de commandes. La transformation survient avant le début de l'exécution de la commande. Si vous voulez réellement une opération qui s'exécute indépendamment pour chaque ligne physique, vous voulez probablement utiliser un déclencheur et non pas une règle. Plus d'informations sur le système des règles se trouvent dans Chapitre 33.

Actuellement, les règles `ON SELECT` doivent être des règles `INSTEAD` sans condition et doivent avoir des actions consistant en une simple commande `SELECT`. Du coup, une règle `ON SELECT` transforme réellement la table en une vue dont le contenu visible est les lignes renvoyées par la commande `SELECT` de la règle plutôt que les lignes qui sont stockées dans la table (si elles existent). Il est mieux considéré d'écrire une commande `CREATE VIEW` plutôt que de créer une vraie table et de définir une règle `ON SELECT` sur elle.

Vous pouvez créer l'illusion d'une vue pouvant être mise à jour en définissant les règles `ON INSERT`, `ON UPDATE` et `ON DELETE` (ou tout sous−ensemble de celles suffisantes à vos buts) pour remplacer les actions de mises à jour sur la vue par des mises à jours appropriées sur les autres tables.

Voici une astuce si vous utilisez les règles conditionnelles pour les mises à jour de vues : elle *doit* être une règle `INSTEAD` sans condition pour chaque action que vous souhaitez autoriser sur la vue. Si la règle est conditionnelle ou si elle n'est pas `INSTEAD`, alors le système rejettera toutes tentatives d'exécution de l'action de mise à jour parce qu'il pense qu'il pourrait finir par essayer de réaliser l'action sur la table de la vue dans certains cas. Si vous souhaitez gérer tous les cas utiles dans les règles conditionnelles, ajoutez une règle `DO INSTEAD NOTHING` sans condition pour vous assurer que le système comprend qu'il ne sera jamais appelé pour modifier la table. Puis, rendez les règles conditionnelles en non `INSTEAD` ; dans les cas où cela s'applique, elles ajoutent à l'action `INSTEAD NOTHING` par défaut.

# Paramètres

*nom*
: Le nom d'une règle à créer. Elle doit être distincte du nom de toute autre règle sur la même table. Les règles multiples sur la même table et le même type d'événement sont appliquées dans l'ordre alphabétique des noms.

*événement*
: L'événement est un de ceux–ci : `SELECT`, `INSERT`, `UPDATE` ou `DELETE`.

*table*
: Le nom (pouvant être qualifié par le nom du schéma) de la table ou de la vue où s'applique la règle.

*condition*
: Toute expression SQL conditionnelle (renvoyant le type `boolean`). L'expression de condition ne peut pas référer à une table autre que `NEW` et `OLD` et ne peut pas contenir de fonction d'agrégat.

`INSTEAD`
: `INSTEAD` indique que les commandes devraient être exécutées *à la place de* la commande originale.

`ALSO`
: `ALSO` indique que les commandes devraient être exécutées *en plus de* la commande originale.

  Si ni `ALSO` ni `INSTEAD` ne sont spécifiés, `ALSO` est la valeur par défaut.

*commande*
: La où les commandes réalisant l'action de la règle. Les commandes valides sont `SELECT`, `INSERT`, `UPDATE`, `DELETE` ou `NOTIFY`.

À l'intérieur d'une *condition* et d'une *commande*, les noms de tables spéciales `NEW` et `OLD` pourraient être utilisés pour se référer aux valeurs de la table référencée. `NEW` est valide dans les règles `ON INSERT` et `ON UPDATE` pour référencer la nouvelle ligne en cours d'insertion ou de mise à jour. `OLD` est valide dans les règles `ON UPDATE` et `ON DELETE` pour référencer la ligne existante en cours de modification ou de suppression.

# Notes

Vous devez avoir le droit `RULE` sur une table pour être autorisé à définir une règle sur elle.

Il est très important de faire attention aux règles circulaires. Par exemple, bien que chacune des deux définitions de règles est acceptée par PostgreSQL, la commande `SELECT` fera que PostgreSQL rapportera une erreur parce que la requête sera en boucle trop longtemps :

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
        SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
        SELECT * FROM t1;

SELECT * FROM t1;
```

Actuellement, si l'action d'une règle contient une commande `NOTIFY`, cette commande sera exécutée sans condition, c'est–à–dire que le `NOTIFY` sera envoyé même si aucune lignes de la règle ne s'applique. Par

exemple, dans

```
CREATE RULE notify_me AS ON UPDATE TO matable DO ALSO NOTIFY matable;

UPDATE matable SET name = 'foo' WHERE id = 42;
```

un événement `NOTIFY` sera lancé durant un `UPDATE`, qu'il y ait ou non des lignes satisfaisant la condition `id = 42`. L'implémentation de cette restriction pourrait être corrigée dans les prochaines versions.

# Compatibilité

`CREATE RULE` est une extension du langage de PostgreSQL ainsi que tout le système de réécriture de requêtes.

# CREATE SCHEMA

## Nom

CREATE SCHEMA −− définit un nouveau schéma

## Synopsis

```
CREATE SCHEMA nom_schéma [ AUTHORIZATION nom_utilisateur ] [ élément_schéma [ ... ] ]
CREATE SCHEMA AUTHORIZATION nom_utilisateur [ élément_schéma [ ... ] ]
```

## Description

`CREATE SCHEMA` crée un nouveau schéma dans la base de données en cours. Le nom du schéma doit être distinct du nom des différents schémas existants dans la base de données en cours.

Un schéma est essentiellement un espace logique : il contient des objets nommés (tables, types de données, fonctions et opérateurs) dont les noms pourraient être ceux d'autres objets existants dans d'autres schémas. Les noms d'objets sont accessible soit en << qualifiant >> leur noms avec le nom du schéma en préfixe ou en configurant un chemin de recherche qui inclut le(s) schéma(s) désiré(s). Une commande `CREATE` spécifiant un objet non qualifié crée l'objet dans le schéma actuel (celui qui se trouve au début du chemin de rechercher, qui peut être déterminé avec la fonction `current_schema`).

De façon optionnelle, `CREATE SCHEMA` peut inclure des sous−commandes pour créer des objets à l'intérieur du nouveau schéma. Les sous−commandes sont traitées essentiellement comme des commandes séparées lancées après la création du schéma, sauf que, si la clause `AUTHORIZATION` est utilisée, tous les objets créés seront possédés par cet utilisateur.

## Paramètres

*`nom_schéma`*
: Le nom d'un schéma à créer. Si il est oublié, le nom de l'utilisateur est utilisé comme nom de schéma. Le nom ne peut pas commencer avec `pg_` car de tels noms sont réservés aux schémas du système.

*`nom_utilisateur`*
: Le nom de l'utilisateur qui possédera le schéma. Si oublié, récupère par défaut l'utilisateur exécutant la commande. Seuls les superutilisateurs peuvent créer des schémas possédés par d'autres utilisateurs qu'eux−mêmes.

*`élément_schéma`*
: Une instruction SQL définissant un objet à créer à l'intérieur du schéma. Actuellement, seules `CREATE TABLE`, `CREATE VIEW`, `CREATE SEQUENCE`, `CREATE TRIGGER` et `GRANT` sont acceptées comme clauses à l'intérieur de `CREATE SCHEMA`. D'autres types d'objets peuvent être créés dans des commandes séparées une fois le schéma créé.

# Notes

Pour créer un schéma, l'utilisateur appelant doit avoir le droit `CREATE` sur la base de données actuelle. (Bien sûr, les superutilisateurs n'ont pas besoin de cette vérification.)

# Exemples

Créer un schéma :

```
CREATE SCHEMA mon_schema;
```

Créer un schéma pour l'utilisateur `joe` ; le schéma sera aussi nommé `joe` :

```
CREATE SCHEMA AUTHORIZATION joe;
```

Créer un schéma et lui ajouter une table et une vue :

```
CREATE SCHEMA hollywood
    CREATE TABLE films (titre text, sortie date, recompenses text[])
    CREATE VIEW gagnants AS
        SELECT titre, sortie FROM films WHERE recompenses IS NOT NULL;
```

Notez que les sous−commandes individuelles ne se terminent pas avec des points−virgules.

Ce qui suit est une façon équivalente d'accomplir la même chose :

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (titre text, sortie date, recompenses text[]);
CREATE VIEW hollywood.gagnants AS
    SELECT titre, sortie FROM hollywood.films WHERE recompenses IS NOT NULL;
```

# Compatibilité

Le standard SQL autorise une clause `DEFAULT CHARACTER SET` dans `CREATE SCHEMA`, ainsi que plus de types de sous−commandes qui ce qui est actuellement accepté par PostgreSQL.

Le standard SQL spécifie que les sous−commandes dans `CREATE SCHEMA` peuvent apparaître dans n'importe quel ordre. L'implémentation actuelle de PostgreSQL ne gère pas tous les cas de références dans les sous−commandes ; il pourrait être nécessaire de réordonner les sous−commandes pour éviter certaines références.

Suivant le standard SQL, le propriétaire d'un schéma possède toujours des objets. PostgreSQL permet aux schémas de contenir des objets possédés par d'autres utilisateurs que le propriétaire du schéma. Ceci peut arriver seulement si le propriétaire du schéma donne le privilège `CREATE` de son schéma à d'autres personnes.

# Voir aussi

*ALTER SCHEMA*, *DROP SCHEMA*

# CREATE SEQUENCE

## Nom

CREATE SEQUENCE −− définit un nouveau générateur de séquence

## Synopsis

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE nom [ INCREMENT [ BY ] incrément ]
    [ MINVALUE valeurmin | NO MINVALUE ]
    [ MAXVALUE valeurmax | NO MAXVALUE ]
    [ START [ WITH ] début ]
    [ CACHE cache ]
    [ [ NO ] CYCLE ]
```

## Description

`CREATE SEQUENCE` crée un nouveau générateur du numéro de séquence. Ceci implique la création et l'initialisation d'une nouvelle table à une seule ligne avec le nom *nom*. Le générateur appartiendra à l'utilisateur exécutant la commande.

Si un nom de schéma est donné, alors la séquence est créée dans le schéma spécifié. Sinon, elle est créée dans le schéma courant. Des séquences temporaires existent dans un schéma spécial, donc un nom de schéma pourrait ne pas être donné lors de la création d'une séquence temporaire. Le nom de séquence doit être distinct du nom de tout autre séquence, table, index ou vue dans le même schéma.

Après la création d'une séquence, vous utilisez les fonctions `nextval`, `currval` et `setval` pour opérer sur la séquence. Ces fonctions sont documentées dans Section 9.12.

Bien que vous ne pouvez pas mettre à jour directement une séquence, vous pouvez utiliser une requête comme

```
SELECT * FROM nom;
```

pour examiner les paramètres et l'état courant d'une séquence. En particulier, le champ `last_value` de la séquence affiche la dernière valeur allouée par une session. (Bien sûr, cette valeur pourrait être obsolète au moment où elle est appelée si d'autres sessions lancent des appels de `nextval`.)

## Paramètres

`TEMPORARY` ou `TEMP`
: Si spécifiée, l'objet de séquence est créé seulement pour cette session et est automatiquement supprimé lors de la sortie de la session. Les séquences existantes permanentes avec le même nom ne sont pas visibles (dans cette session) alors que la séquence temporaire existe, sauf si elles sont référencées avec les noms qualifiés du schéma.

*nom*
: Le nom (pouvant être qualifié avec le nom du schéma) de la séquence à créer.

*incrément*
: La clause optionnelle `INCREMENT BY` *incrément* spécifie la valeur ajoutée à la valeur de la séquence courante pour créer une nouvelle valeur. Une valeur positive créera une séquence ascendante, une négative en créera une descendante. La valeur par défaut est 1.

*valeurmin*
`NO MINVALUE`
: La clause optionnelle `MINVALUE` *valeurmin* détermine la valeur minimale qu'une séquence peut générer. Si cette clause n'est pas fournie ou si `NO MINVALUE` est spécifié, alors les valeurs par défaut seront utilisées. Les valeurs par défaut sont 1 et $-2^{63}-1$ pour les séquences respectivement ascendantes et descendantes.

*valeurmax*
`NO MAXVALUE`
: La clause optionnelle `MAXVALUE` *valeurmax* détermine la valeur maximale pour la séquence. Si cette clause n'est pas fournie ou si `NO MAXVALUE` est specifié, alors les valeurs par défaut seront utilisées. Les valeurs par défaut sont $2^{63}-1$ et $-1$ pour les séquences respectivement ascendantes et descendantes.

*début*
: La clause optionnelle `START WITH` *début* autorise la séquence à commencer quelque part. La valeur de début par défaut est *valeurmin* pour les séquences ascendantes et *valeurmax* pour les séquences descendantes.

*cache*
: La clause optionnelle `CACHE` *cache* spécifie comment les numéros de séquence doivent être préalloués et stockés en mémoire pour un accès plus rapide. La valeur minimale est 1 (seule une valeur peut être générée à un moment, c'est–à–dire pas de cache), et c'est aussi la valeur par défaut.

`CYCLE`
`NO CYCLE`
: L'option `CYCLE` autorise la séquence à recommencer au début lorsque *valeurmax* ou *valeurmin* a été atteinte par une séquence respectivement ascendante ou descendante. Si la limite est atteinte, le prochain nombre généré sera respectivement *valeurmin* ou *valeurmax*.

  Si `NO CYCLE` est spécifié, tout appel à `nextval` après que la séquence est atteint la valeur minimale renverra une erreur. Si ni `CYCLE` ni `NO CYCLE` ne sont spécifiés, `NO CYCLE` est la valeur par défaut.

# Notes

Utilisez `DROP SEQUENCE` pour supprimer une séquence.

Les séquences sont basées sur l'arithmétique `bigint`, donc l'échelle ne peut pas excéder l'échelle d'un entier sur huit octets (–9223372036854775808 à 9223372036854775807). Sur certaines vieilles plateformes, il pourrait ne pas y avoir de support du compilateur pour les entiers sur huit octets, auquel cas les séquences utilisent l'arithmétique des entiers (type `integer`) habituels (allant de –2147483648 à +2147483647).

Des résultats inattendus pourraient être obtenus si un paramétrage de *cache* plus grand que un est utilisé pour un objet séquence qui sera utilisé en concurrence par plusieurs séquences. Chaque session allouera du cache et des valeurs de séquences successives lors d'un accès à l'objet séquence et augmentera la valeur de `last_value` en concordance. Ensuite, les utilisations prochaines de *cache*–1 dans cette session renverront simplement les valeurs préallouées sans toucher à l'objet séquence. Donc, tout nombre alloué mais non utilisé dans une session sera perdu lorsque la session se termine, causant des << trous >> dans la séquence.

De plus, bien que de nombreuses sessions sont garanties pour allouer des valeurs de séquences distinctes, les valeurs pourraient être générées en dehors de la séquence lorsque toutes les sessions sont considérées. Par exemple, avec un paramétrage du *cache* à 10, la session A pourrait réserver des valeurs 1..10 et renvoyer nextval=1, alors la session B pourrait réserver des valeurs 11..20 et renvoyer nextval=11 avant que la session A ait généré nextval=2. Du coup, avec un paramétrage de *cache* de un, il est sain d'assumer que les valeurs nextval sont générées séquentiellement ; avec un paramétrage *cache* plus grand que un, vous devriez seulement assumer que les valeurs nextval sont tous distinctes, et non pas qu'elles soient générées purement séquentiellement. De plus, last_value reflètera la dernière valeur réservée par toute session, qu'il ne soit ou non pas encore renvoyé par nextval.

Une autre considération est qu'un setval exécuté sur une telle séquence ne sera pas notifiée pour les autres sessions jusqu'à ce qu'ils aient utilisé des valeurs préallouées qu'ils ont caché.

# Exemples

Créer une séquence ascendante appelée serie, commençant à 101 :

```
CREATE SEQUENCE serie START 101;
```

Sélectionner le prochain numéro à partir de cette séquence :

```
SELECT nextval('serie');

 nextval
---------
     114
```

Utiliser cette séquence dans une commande INSERT :

```
INSERT INTO distributors VALUES (nextval('serie'), 'nothing');
```

Mettre à jour la valeur de la séquence après un COPY FROM :

```
BEGIN;
COPY distributeurs FROM 'fichier_entrees';
SELECT setval('serie', max(id)) FROM distributeurs;
END;
```

# Compatibilité

CREATE SEQUENCE est spécifiée en SQL:2003. PostgreSQL se conforme au standard avec les exceptions suivantes :

- L'expression AS <type_données> n'est pas supportée.
- Obtenir la prochaine valeur est possible en utilisant la fonction nextval() au lieu de l'expression NEXT VALUE FOR du standard.

# CREATE TABLE

## Nom

CREATE TABLE −− définit une nouvelle table

## Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nom_table (
  { nom_colonne type_données [ DEFAULT
default_expr ] [ contrainte_colonne [ ... ] ]
    | contrainte_table
    | LIKE table_parent [ { INCLUDING | EXCLUDING }
DEFAULTS ] } [, ... ]
)
[ INHERITS ( table_parent [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE espacelogique ]

où contrainte_colonne
est :

[ CONSTRAINT nom_contrainte ]
{ NOT NULL | NULL | UNIQUE [ USING INDEX TABLESPACE espacelogique ] |
PRIMARY KEY [ USING INDEX TABLESPACE espacelogique ] |
  CHECK (expression) |
  REFERENCES table_reference [ ( colonne_reference ) ] [ MATCH FULL
| MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

et contrainte_table est :

[ CONSTRAINT nom_contrainte ]
{ UNIQUE ( nom_colonne [, ... ] ) [ USING INDEX TABLESPACE espacelogique ] |
  PRIMARY KEY ( nom_colonne [, ...
  ] ) [ USING INDEX TABLESPACE espacelogique ] |
  CHECK ( expression ) |
  FOREIGN KEY ( nom_colonne [, ...
] ) REFERENCES table_reference [ (
colonne_reference [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

## Description

`CREATE TABLE` créera une nouvelle table initialement vide dans la base de données courante. La table sera la propriété de l'utilisateur qui a lançé cette commande.

Si un nom de schéma est donné (par exemple, `CREATE TABLE monschema.matable ...`), alors la table est créée dans le schéma spécifié. Sinon, il est créé dans le schéma actuel. Les tables temporaires existent dans un schéma spécial, donc un nom de schéma pourrait ne pas être donné lors de la création d'une

table temporaire. Le nom de la table doit être distinct des noms des autres tables, séquences, index ou vues dans le même schéma.

`CREATE TABLE` crée aussi automatiquement un type de données qui représente le type composé correspondant à une ligne de la table. Du coup, les tables ne peuvent pas avoir le même nom que tout type de données du même schéma.

Les clauses de contrainte optionnelle spécifient les contraintes (ou tests) que les nouvelles lignes ou les lignes mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Une contrainte est un objet SQL qui aide à définir l'ensemble de valeurs valides de plusieurs façons.

Il existe deux façons de définir des contraintes : les contraintes de table et celles des colonnes. Une contrainte de colonne est définie pour faire partie d'une définition de la colonne. Une définition de la contrainte des tables n'est pas liée à une colonne particulière et elle comprend plus d'une colonne. Chaque contrainte de colonne peut aussi être écrite comme une contrainte de table ; une colonne de contrainte est seulement un outil de notation à utiliser lorsque la contrainte affecte seulement une colonne.

# Paramètres

`TEMPORARY` ou `TEMP`
: Si spécifié, la table est créée comme une table temporaire. Les tables temporaires sont automatiquement supprimées à la fin d'une session ou, optionnellement, à la fin de la transaction en cours (voir `ON COMMIT` ci−dessous). Les tables permanentes existantes avec le même nom ne sont pas visibles dans la session en cours alors que la table temporaire existe sauf si elles sont référencées avec les noms qualifiés du schéma. Tous les index créés sur une table temporaire sont aussi automatiquement temporaires.

  Optionellement, `GLOBAL` ou `LOCAL` peuvent être écrit avant `TEMPORARY` ou `TEMP`. Ceci ne fait pas de différence dans PostgreSQL, mais voir *Compatibilité*.

*`nom_table`*
: Le nom (peut−être qualifié par le nom du schéma) de la table à créer.

*`nom_colonne`*
: Le nom d'une colonne à créer dans la nouvelle table.

*`type_données`*
: Le type de données de la colonne. Ceci pourrait inclure des spécificateurs de tableaux. Pour plus d'informations sur les types de données supportés par PostgreSQL, référez−vous à Chapitre 8.

`DEFAULT` *`default_expr`*
: La clause `DEFAULT` affecte une valeur par défaut pour la colonne dont la définition apparaît à l'intérieur. La valeur est toute expression libre de variable (les sous−requêtes et références croisées aux autres colonnes dans la table en cours ne sont pas autorisées). Le type de données de l'expression par défaut doit correspondre au type de données de la colonne.

  L'expression par défaut sera utilisée dans les opérations d'insertion qui ne spécifient pas une valeur pour la colonne. S'il n'y a pas de valeur par défaut pour une colonne, alors la valeur par défaut est NULL.

`INHERITS ( table_parent [, ... ] )`
: La clause optionnelle `INHERITS` spécifie une liste des tables à partir desquelles la nouvelle table hérite automatiquement de toutes les colonnes.

L'utilisation d'INHERITS crée une relation persistante entre la nouvelle table enfant et sa table parent. Les modifications du schéma au(x) parent(s) se propagent normalement aussi aux enfants et, par défaut, les données de la table enfant sont inclus dans les parcours de(s) parent(s).

Si le même nom de colonne existe dans plus d'une table parente, une erreur est rapportée sauf si les types de données des colonnes correspondent à chacune des tables parentes. S'il n'y a aucun conflit, alors les colonnes dupliquées sont assemblées pour former une seule colonne dans la nouvelle table. Si la liste de noms de colonnes de la nouvelle table contient un nom de colonne qui est aussi héritée, le type de données doit correspondre aux colonnes héritées et les définitions de la colonne sont assemblées en une seule. Néanmoins, les déclarations des colonnes héritées et nouvelles du même nom ont besoin de ne pas spécifier des contraintes identiques : toutes les contraintes fournies par toute déclaration sont assemblées et sont toutes appliquées à la nouvelle table. Si la nouvelle table spécifie explicitement une valeur par défaut pour la colonne, cette valeur surcharge toute valeur par défaut des déclarations héritées pour la colonne.Sinon, tout parent spécifiant des valeurs par défaut pour la colonne doit spécifier la même valeur par défaut. Sinon une erreur sera rapportée.

LIKE *table_parent* [ { INCLUDING | EXCLUDING } DEFAULTS ]

La clause LIKE spécifie une table à partir de laquelle la nouvelle table copie automatiquement tous les noms de colonnes, leur types de données et les contraintes non NULL.

Contrairement à INHERITS, la nouvelle table et la table héritée sont complètement découplées après la fin de la création. Les modifications sur la table originale ne sont pas appliquées à la nouvelle table et il n'est pas possible d'inclure les données de la nouvelle table dans des parcours de l'ancienne table.

Les expressions par défaut pour les définitions des colonnes héritées seront seulement copiées si INCLUDING DEFAULTS est spécifié. Le comportement par défaut est d'exclure les expressions par défaut, ce qui a pour résultat d'avoir des valeurs par défaut NULL pour toutes les colonnes de la nouvelle table.

WITH OIDS

WITHOUT OIDS

Cette clause optionnelle spécifie si les lignes de la nouvelle table devraient avoir des OID (identifiants d'objets) qui leur sont affectés. Si ni WITH OIDS ni WITHOUT OIDS ne sont spécifiés, la valeur par défaut dépend du paramètre de configuration default_with_oids. (Si la nouvelle table hérite d'autres tables possédant des OID, alors WITH OIDS est forcé même si la commande indique WITHOUT OIDS.)

Si WITHOUT OIDS est spécifié ou implicite, la nouvelle table ne stocke pas les OID aucun OID ne sera affecté pour une ligne insérée dans cette table. Ceci est généralement considéré comme intéressant pour les grosses tables car il réduit la consommation d'OID et, du coup, annule pour cette table le problème du retour à zéro du compteur d'OID. Une fois que le compteur est revenu à zéro, l'unicité des OID ne peut plus être garantie, ce qui réduit considérablement leur utilité. De plus, exclure les OID d'une table réduit aussi l'espace requis pour stocker la table sur disque de quatre octets par ligne de la table (sur la plupart des machines), améliorant ainsi leur performance.

Pour supprimer les OID d'une table après qu'elle ait été créée, utilisez *ALTER TABLE*.

CONSTRAINT *nom_contrainte*

Un nom optionnel pour une contrainte de colonne ou de table. S'il n'est pas spécifié, le système génère un nom.

NOT NULL

La colonne n'est pas autorisée à contenir des valeurs NULL.

NULL

La colonne est autorisée pour contenir des valeurs NULL. Ceci est la valeur par défaut.

Cette clause est seulement fournie pour la compatibilité avec les bases de données SQL non standards. Son utilisation n'est pas encouragée dans les nouvelles applications.

`UNIQUE` (contrainte_colonne)

`UNIQUE ( nom_colonne [, ... ] )` (contrainte table)

La contrainte `UNIQUE` spécifie qu'un groupe d'une ou plusieurs colonnes d'une table pourrait seulement contenir des valeurs uniques. Le comportement de la contrainte de table unique est le même que pour les contraintes de colonnes avec la capacité supplémentaire de diviser les colonnes multiples.

Dans le but d'une contrainte unique, les valeurs NULL ne sont pas considérées égales.

Chaque contrainte de table unique doit nommer un ensemble de colonnes qui est différent de l'ensemble des colonnes nommées par toute autre contrainte unique ou de clé primaire définie pour la table. (Sinon, cela pourrait être juste la même contrainte donnée deux fois.)

`PRIMARY KEY` (contrainte colonne)

`PRIMARY KEY ( nom_colonne [, ... ] )` (contrainte table)

La contrainte de clé primaire spécifie qu'une ou plusieurs colonnes d'une table pourraient contenir seulement des valeurs uniques, non NULL. Techniquement, `PRIMARY KEY` est simplement une combinaison de `UNIQUE` et `NOT NULL`, mais identifier un ensemble de colonnes comme clé primaire fournit aussi des métadonnées sur le concept du schéma, car une clé primaire implique que d'autres tables pourraient se lier à cet ensemble de colonnes comme un unique identifiant pour les lignes.

Seule une clé primaire peut être spécifiée pour une table, s'il s'agit d'une contrainte de colonne ou de table.

La contrainte de clé primaire devrait nommer un ensemble de colonnes qui est différent des autres ensembles de colonnes nommés par une contrainte unique définie pour la même table.

`CHECK (expression)`

La clause `CHECK` spécifie une expression produisant un résultat booléen que les nouvelles lignes ou que les lignes mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Les expressions évaluant à TRUE ou UNKNOWN réussissent. Si une ligne d'une opération d'insertion ou de mise à jour produit un résultat FALSE, une exception est levée et l'insertion ou la mise à jour ne modifie pas la base de données. Une contrainte de vérification spécifiée comme une contrainte de colonne devrait seulement référencer la valeur de la colonne alors qu'une expression apparaissant dans une contrainte de table pourrait référencer plusieurs colonnes.

Actuellement, les expressions `CHECK` ne peuvent ni contenir des sous−requêtes ni se référer à des variables autres que les colonnes de la ligne actuelle.

`REFERENCES table_reference [ ( colonne_reference ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ]` (contrainte de colonne)

`FOREIGN KEY ( colonne [, ... ] ) REFERENCES table_reference [ ( colonne_reference [, ... ] ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ]` (contrainte de colonne)

Ces clauses spécifient une contrainte de clé étrangère, ce qui requiert qu'un groupe d'une ou plusieurs colonnes de la nouvelle table doit seulement contenir des valeurs correspondant aux valeurs dans le(s) colonne(s) référencée(s) de quelques lignes de la table référencée. Si *colonne_reference* est omis, la clé primaire de la *table_reference* est utilisée. Les colonnes référencées doivent être les colonnes d'une contrainte unique ou de clé primaire dans la table référencée.

Une valeur insérée dans les colonnes référencées est comparée aux valeurs de la table référencée et des colonnes référencées en utilisant le type correspondant donné. Il existe trois types de correspondance : MATCH FULL, MATCH PARTIAL et MATCH SIMPLE, qui est aussi la valeur par défaut. MATCH FULL n'autorisera pas une colonne d'une clé étrangère composée de plusieurs colonnes pour être NULL sauf si les colonnes de clés étrangères sont nulles. MATCH SIMPLE autorise quelques colonnes de clé étrangère pour être NULL alors que les autres parties de la clé étrangère ne sont pas nulles. MATCH PARTIAL n'est pas encore implémenté.

En plus, lorsque les données des colonnes référencées sont modifiées, certaines actions sont réalisées sur les données dans les colonnes de cette table. La clause ON DELETE spécifie l'action à réaliser lorsqu'une ligne référencée de la table référencée est en cours de suppression. De la même façon, la clause ON UPDATE spécifie l'action à réaliser lorsqu'une colonne référencée dans la table référencée est en cours de mise à jour pour une nouvelle valeur. Si la ligne est mise à jour mais la colonne référencée n'est pas réellement modifiée, aucune action n'est réalisée. Les actions référencées autres que la vérification de contrainte NO ACTION ne peuvent pas être déferrées même si la contrainte est déclarée comme déferrable. Il existe les actions possibles suivantes pour chaque clause :

NO ACTION
: Produit une erreur indiquant que la suppression ou la mise à jour créerait une violation de la contrainte de clé étrangère. Si la contrainte est déferrée, cette erreur sera produite au moment de la vérification de la contrainte s'il existe toujours des lignes de référence. Ceci est l'action par défaut.

RESTRICT
: Produit une erreur indiquant que la suppression ou mise à jour créera une violation de la contrainte de clé étrangère. Ceci est identique à NO ACTION sauf que la vérification n'est pas déferrable.

CASCADE
: Supprime toute ligne référençant la ligne supprimée ou met à jour la valeur de la colonne référencée avec la nouvelle valeur de la colonne référencée, respectivement.

SET NULL
: Initialise la colonne de référence à NULL.

SET DEFAULT
: Initialise la colonne de référence à sa valeur par défaut.

Si les colonnes référencées sont modifiées fréquemment, il pourrait être conseillé d'ajouter un index vers la colonne de clé étrangère de façon à ce que les actions référentielles associées avec la colonne de clé étrangère puissent être réalisées avec efficacité.

DEFERRABLE
NOT DEFERRABLE
: Ceci contrôle si la contrainte peut être déferrée. Une contrainte qui n'est pas déferrable sera vérifiée immédiatement après chaque commande. La vérification des contraintes qui sont déferrables pourraient attendre la fin de la transaction (en utilisant la commande *SET CONSTRAINTS*). NOT DEFERRABLE est la valeur par défaut. Seulement des contraintes de clé étrangère acceptent réellement cette clause. Tous les autres types de contraintes ne sont pas déferrables.

INITIALLY IMMEDIATE
INITIALLY DEFERRED
: Si une contrainte est déferrable, cette clause spécifie le temps par défaut pour vérifier la contrainte. Si la contrainte est INITIALLY IMMEDIATE, elle est vérifiée après chaque instruction. Si la contrainte est INITIALLY DEFERRED, elle est vérifiée seulement à la fin de la transaction. Le moment de vérification de la contrainte peut être modifié avec la commande *SET CONSTRAINTS*.

ON COMMIT

Le comportement des tables temporaires à la fin d'un bloc de transaction peut se contrôler en utilisant `ON COMMIT`. Les trois options sont

`PRESERVE ROWS`
: Aucune action n'est prise à la fin des transactions. Ceci est le comportement par défaut.

`DELETE ROWS`
: Toutes les lignes dans la table temporaire seront détruites à la fin de chaque bloc de transaction. En fait, un TRUNCATE automatique est réalisé à chaque validation.

`DROP`
: La table temporaire sera supprimée à la fin du bloc de transaction.

`TABLESPACE` *`espacelogique`*
: L'*`espacelogique`* est le nom de l'espace logique dans lequel est créée la nouvelle table. Si elle n'est pas spécifiée, default_tablespace est utilisée ou l'espace logique par défaut de la base de données si `default_tablespace` est une chaîne vide.

`USING INDEX TABLESPACE` *`espacelogique`*
: Cette clause permet la sélection de l'espace logique dans lequel les index associés à une contrainte `UNIQUE` ou `PRIMARY KEY` seront créés. Si elle n'est pas spécifiée, default_tablespace est utilisée ou l'espace logique par défaut de la base de données si `default_tablespace` est une chaîne vide.

# Notes

Utiliser les OID dans les nouvelles applications n'est pas recommandé : si possible, utilisez de préférence un `SERIAL` ou un autre générateur de séquence comme clé primaire de la table. Néanmoins, si votre application utilise les OID pour identifier des lignes spécifiques d'une table, il est recommandé de créer une contrainte unique sur la colonne `oid` de cette table pour s'assurer que les OID de la table identifieront les lignes réellement de façon unique même après une remise à zéro du compteur. Évitez d'assumer que les OID sont uniques pour les différentes tables ; si vous avez besoin d'un identifiant unique sur la base de données, utilisez une combinaison de `tableoid` et de l'OID de la ligne dans ce but.

> **Astuce :** L'utilisation de `WITHOUT OIDS` n'est pas recommandée pour les tables sans clé primaire car, sans soit un OID soit une clé de données unique, il est difficile d'identifier des lignes spécifiques.

PostgreSQL crée automatiquement un index pour chaque contrainte unique et pour chaque contrainte de clé étrangère pour renforcer l'unicité. Du coup, il n'est pas nécessaire de créer un index spécifiquement pour les colonnes de clés primaires. (Voir *CREATE INDEX* pour plus d'informations.)

Les contraintes uniques et les clés primaires ne sont pas héritées dans l'implémentation actuelle. Ceci rend la combinaison de l'héritage et des contraintes uniques assez disfonctionnelle.

Une table ne peut pas avoir plus de 1600 colonnes (en pratique, la limite réelle est plus basse à cause de contraintes sur la longueur des lignes).

# Exemples

Créez une table `films` et une table `distributeurs` :

```
CREATE TABLE films (
```

```
    code        char(5) CONSTRAINT premierecle PRIMARY KEY,
    titre       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    genre       varchar(10),
    duree       interval hour to minute
);

CREATE TABLE distributeurs (
     did    integer PRIMARY KEY DEFAULT nextval('serial'),
     nom    varchar(40) NOT NULL CHECK (nom <> '')
);
```

Crée une table avec un tableau à deux dimensions :

```
CREATE TABLE array_int (
    vecteur  int[][]
);
```

Définir une contrainte unique de table pour la table `films`. Les contraintes uniques de table peuvent être définies sur une ou plusieurs colonnes de la table.

```
CREATE TABLE films (
    code        char(5),
    titre       varchar(40),
    did         integer,
    date_prod   date,
    genre       varchar(10),
    duree       interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Définir une contrainte de colonne de vérification :

```
CREATE TABLE distributeurs (
    did     integer CHECK (did > 100),
    nom    varchar(40)
);
```

Définir une contrainte de table de vérification :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40)
    CONSTRAINT con1 CHECK (did > 100 AND nom <> '')
);
```

Définir une contrainte de clé primaire sur la table `films`. Les contraintes de clé primaire peuvent être définies sur une ou plusieurs colonnes de la table.

```
CREATE TABLE films (
    code        char(5),
    titre       varchar(40),
    did         integer,
    date_prod   date,
    genre       varchar(10),
    duree       interval hour to minute,
    CONSTRAINT code_titre PRIMARY KEY(code,titre)
```

```
);
```

Définir une contrainte de clé primaire pour la table `distributeurs`. Les deux exemples suivants sont équivalents, le premier utilisant la syntaxe de contrainte de la table, le second la syntaxe de contrainte de la colonne.

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributeurs (
    did     integer PRIMARY KEY,
    nom     varchar(40)
);
```

Ceci affecte une valeur par défaut pour la colonne `nom`, arrange la valeur par défaut de la colonne `did` pour être générée en sélectionnant la prochaine valeur d'un objet séquence et fait que la valeur par défaut de `modtime` soit le moment où la ligne est insérée.

```
CREATE TABLE distributeurs (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributeurs_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

Définir deux contraintes de colonnes `NOT NULL` sur la table `distributeurs`, dont une se voit donner explicitement un nom :

```
CREATE TABLE distributeurs (
    did     integer CONSTRAINT no_null NOT NULL,
    nom     varchar(40) NOT NULL
);
```

Définit une contrainte unique pour la colonne `nom` :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40) UNIQUE
);
```

Ce qui se trouve ci–dessus est équivalent à ce qui suit, spécifié comme une contrainte de table :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40),
    UNIQUE(nom)
);
```

Créer une table `cinemas` dans l'espace logique `diskvol1` :

```
CREATE TABLE cinemas (
    id serial,
    nom text,
    emplacement text
) TABLESPACE diskvol1;
```

# Compatibilité

La commande `CREATE TABLE` se conforme à SQL−92 et à un sous−ensemble de SQL:1999, avec les exceptions indiquées ci−dessous.

## Tables temporaires

Bien que la syntaxe de `CREATE TEMPORARY TABLE` ressemble à celle du SQL standard, l'effet n'est pas le même. Dans le standard, les tables temporaires sont définies seulement une fois et existent automatiquement (en commençant avec un contenu vide) dans chaque session qui en a besoin. À la place, PostgreSQL requiert que chaque session lance sa propre commande `CREATE TEMPORARY TABLE` pour chaque table temporaire à utiliser. Ceci permet à différentes sessions d'utiliser le même nom de table temporaire dans des buts différents alors que l'approche du standard contraint toutes les instances d'un nom de table temporaire donné pour avoir la même structure de table.

La définition du standard pour le comportement des tables temporaires est largement ignorée. Le comportement de PostgreSQL sur ce point est similaire à celui de nombreuses autres bases de données SQL.

La distinction du standard entre tables temporaires globales et locales n'est pas dans PostgreSQL car cette distinction dépend du concept de modules, que PostgreSQL ne possède pas. Pour le bien de la compatibilité, PostgreSQL acceptera les mots clés `GLOBAL` et `LOCAL` dans la déclaration d'une table temporaire mais cela n'aura aucun effet.

La clause `ON COMMIT` pour les tables temporaires ressemble aussi au standard SQL mais a quelques différences. Si la clause `ON COMMIT` est omise, SQL spécifie que le comportement par défaut est `ON COMMIT DELETE ROWS`. Néanmoins, le comportement par défaut dans PostgreSQL est `ON COMMIT PRESERVE ROWS`. L'option `ON COMMIT DROP` n'existe pas en SQL.

## Contraintes de vérification de colonnes

Le standard SQL dit que les contraintes de vérification `CHECK` de colonne pourraient seulement référencer la colonne à laquelle elles s'appliquent ; seulement les contraintes de tables `CHECK` pourraient se référencer à de nombreuses colonnes. PostgreSQL ne force pas cette restriction ; il traite de la même façon les contraintes de vérifications des colonnes et des tables.

## Contrainte `NULL`

La << contrainte >> `NULL` (réellement une non−contrainte) est une extension PostgreSQL au standard SQL qui est inclus pour des raisons de compatibilité avec quelques autres systèmes de bases de données (et pour la symétrie avec la contrainte `NOT NULL`). Comme ceci est la valeur par défaut de cette colonnes, sa présence est un simple bruit.

### Héritage

Plusieurs héritages via la clause `INHERITS` est une extension du langage PostgreSQL. SQL:1999 (et non pas SQL−92) définit un héritage simple en utilisant une syntaxe et des sémantiques différentes. L'héritage style SQL:1999 n'est pas encore supporté par PostgreSQL.

### Object ID

Le concept PostgreSQL des OID n'est pas standard.

### Tables à zéro colonne

PostgreSQL autorise la création de tables sans colonnes (par exemple, `CREATE TABLE foo();`). Ceci est une extension du standard SQL, qui ne le permet pas. Les tables sans colonnes ne sont pas très utiles mais les désactiver pourrait apporter quelques cas bizarres spéciaux pour `ALTER TABLE DROP COLUMN`, donc il semble plus propre d'ignorer la restriction de cette spécification.

### Espaces logiques

Le concept PostgreSQL d'espaces logiques n'est pas celui du standard. Du coup, les clauses `TABLESPACE` et `USING INDEX TABLESPACE` sont des extensions.

## Voir aussi

*ALTER TABLE*, *DROP TABLE*, *CREATE TABLESPACE*

# CREATE TABLE AS

## Nom

CREATE TABLE AS −− définit une nouvelle table à partir des résultats d'une requête

## Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE
nom_table [
(nom_colonne [, ...] ) ]
    AS requête [ [ WITH | WITHOUT ] OIDS ]
```

## Description

`CREATE TABLE AS` crée une table et la remplit avec des données récupérées par une commande `SELECT` ou un `EXECUTE` qui lance une commande préparée `SELECT`. Les colonnes de table ont les noms et les types de données associés avec les colonnes en sortie du `SELECT` (sauf que vous pouvez surcharger les noms de colonne en donnant une liste explicite des nouveaux noms de colonnes).

`CREATE TABLE AS` a une certaine ressemblance pour créer une vue mais elle est réellement assez différente : elle crée une nouvelle table et évalue la requête juste une fois pour remplir la nouvelle table initialement. La nouvelle table ne tracera pas les changements suivants pour les tables source de la requête. À l'opposé, une vue ré−évalue son instruction `SELECT` à chaque fois qu'elle est appelée.

## Paramètres

`GLOBAL` or `LOCAL`
: Ignoré. Indiqué pour la compatibilité. Référez−vous à *CREATE TABLE* pour des détails.

`TEMPORARY` ou `TEMP`
: Si spécifié, la table est créée comme une table temporaire. Référez−vous à *CREATE TABLE* pour plus de détails.

*`nom_table`*
: Le nom de la table à créer (pouvant être qualifié avec le nom du schéma).

*`nom_colonne`*
: Le nom d'une colonne dans une nouvelle table. Si les noms de colonnes ne sont pas fournis, ils sont pris des noms de colonnes en sortie de la requête. Si la table est créée à partir d'une commande `EXECUTE`, une liste de noms de colonnes peut ne pas être spécifiée.

`WITH OIDS`
`WITHOUT OIDS`
: Cette clause optionnelle spécifie si la table créée par `CREATE TABLE AS` devrait inclure OIDs. Si aucune forme de cette clause n'est spécifiée, la valeur du paramètre de configuration default_with_oids est utilisée.

*`requête`*
: Une instruction de requête (c'est−à−dire une commande `SELECT` ou une commande `EXECUTE` qui

exécute une commande SELECT préparée). Référez−vous à *SELECT* ou *EXECUTE*, respectivement pour une description de la syntaxe autorisée.

# Notes

Cette commande est équivalente fonctionnellement à *SELECT INTO* mais elle est préférée car il y a moins de risque de confusion avec les autres utilisations de la syntaxe SELECT ... INTO. De plus, CREATE TABLE AS offre un ensemble de fonctionnalités plus important que celles proposées par SELECT INTO.

Avant PostgreSQL 8.0, CREATE TABLE AS incluait toujours les OIDs dans la table qu'elle créait. À partir de PostgresSQL 8.0, la commande CREATE TABLE AS autorise l'utilisateur à spécifier explicitement les OID à inclure. Si la présence de OID n'est pas spécifiée explicitement, la variable de configuration default_with_oids est utilisée. Alors que la valeur par défaut de cette variable est true, la valeur par défaut pourrait être modifiée dans le futur. Du coup, les applications nécessitant des OID dans la table créée par CREATE TABLE AS devraient indiquer explicitement WITH OIDS pour s'assurer de la compatibilité avec les prochaines versions de PostgreSQL.

# Exemples

Créer une nouvelle table films_recent consistant des entrées récentes de la table films :

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

# Compatibilité

CREATE TABLE AS est spécifiée par le standard SQL:2003. Il existe quelques petites différences entre la définition de la commande dans SQL:2003 et son implémentation dans PostgreSQL :

- Le standard requiert des parenthèses autour de la clause de la sous−requête ; dans PostgreSQL, ces parenthèses sont optionnsselles.
- Le standard définit une clause ON COMMIT ; elle n'est pas encore implémentée par PostgreSQL.
- Le standard définit une clause WITH DATA ; elle n'est pas encore implémentée par PostgreSQL.

# Voir aussi

*CREATE TABLE*, *EXECUTE*, *SELECT*, *SELECT INTO*

# CREATE TABLESPACE

## Nom

CREATE TABLESPACE −− définit un nouvel espace logique

## Synopsis

```
CREATE TABLESPACE nomespacelogique
[ OWNER nom_utilisateur ]
LOCATION 'répertoire'
```

## Description

`CREATE TABLESPACE` enregistre un nouvel espace logique pour le groupe de bases de données. Le nom de l'espace logique doit être distinct du nom de tout autre espace logique du groupe.

Un espace logique permet aux superutilisateurs de définir un autre emplacement sur le système de fichiers où les fichiers de données contenant des objets de la base de données (comme les tables et les index) pourront résider.

Un utilisateur disposant des droits appropriés peut passer *`nomespacelogique`* à `CREATE DATABASE`, `CREATE TABLE`, `CREATE INDEX` ou `ADD CONSTRAINT` pour que les fichiers de données de ces objets soient stockés à l'intérieur de l'espace logique spécifié.

## Paramètres

*`nomespacelogique`*
: Le nom d'un espace logique à créer. Le nom ne peut pas commencer avec `pg_` car de tels noms sont réservés pour les espaces logiques système.

*`nomutilisateur`*
: Le nom de l'utilisateur, propriétaire de l'espace logique. En cas d'omission, il s'agit de l'utilisateur ayant exécuté la commande. Seuls les superutilisateurs peuvent créer des espaces logiques mais ils peuvent donner la propriété des espaces logiques à des utilisateurs standards.

*`répertoire`*
: Le répertoire qui sera utilisé pour l'espace logique. Le répertoire doit être vide et doit être possédé par l'utilisateur système PostgreSQL. Le répertoire doit être spécifié par un chemin absolu.

## Notes

Les espaces logiques sont seulement supportés sur les systèmes supportant les liens symboliques.

# Exemples

Créer un espace logique `espace_base` sur `/data/dbs` :

```
CREATE TABLESPACE espace_base LOCATION '/data/dbs';
```

Créer un espace logique `espace_index` sur `/data/indexes` et donner le propriété à l'utilisatrice `genevieve` :

```
CREATE TABLESPACE espace_index OWNER genevieve LOCATION '/data/indexes';
```

# Compatibilité

`CREATE TABLESPACE` est une extension PostgreSQL.

# Voir aussi

*CREATE DATABASE*, *CREATE TABLE*, *CREATE INDEX*, *DROP TABLESPACE*, *ALTER TABLESPACE*

# CREATE TRIGGER

## Nom

CREATE TRIGGER −− définit un nouveau déclencheur

## Synopsis

```
CREATE TRIGGER nom { BEFORE | AFTER } { evenement [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    EXECUTE PROCEDURE nomfonc ( arguments )
```

## Description

`CREATE TRIGGER` crée un nouveau déclencheur. Le déclencheur sera associé avec la table spécifiée et exécutera la fonction spécifiée *`nomfonc`* lorsque certains événements surviennent.

Le déclencheur peut être spécifié pour être lancé soit avant que l'opération ne soit tentée sur une ligne (avant la vérification des contraintes et que l'`INSERT`, l'`UPDATE` ou le `DELETE` ne soit tenté) ou une fois que l'opération s'est terminée (après la vérification des contraintes et que l'`INSERT`, l'`UPDATE` ou le `DELETE` ne soit terminé). Si le déclencheur est lancé avant l'événement, le déclencheur pourrait annuler l'opération pour la ligne courante ou modifier la ligne en cours d'insertion (seulement pour les opérations `INSERT` et `UPDATE`). Si le déclencheur se lance après l'événement, toute modification, incluant la dernière insertion, mise à jour ou suppression, sont << visibles >> par le déclencheur.

Un déclencheur qui est marqué `FOR EACH ROW` est appelé pour chaque ligne que l'opération modifie. Par exemple, un `DELETE` affectant dix lignes causera le lancement de tout déclencheur `ON DELETE` sur la relation cible dix fois séparément, une fois pour chaque ligne supprimée. Au contraire, un déclencheur marqué `FOR EACH STATEMENT` s'exécute seulement une fois pour toute opération donnée, quelque soit le nombre de lignes qu'il modifie (en particulier, une opération qui ne modifie aucune ligne, résultera toujours en l'exécution des déclencheurs `FOR EACH STATEMENT` applicables).

Si plusieurs déclencheurs du même genre sont définis pour le même événement, ils sont déclenchés suivant l'ordre alphabétique de leur nom.

`SELECT` ne modifie aucune ligne donc vous ne pouvez pas créer de déclencheurs `SELECT`. Les règles et vues sont plus appropriées dans de tels cas.

Référez−vous à Chapitre 32 pour plus d'informations sur les déclencheurs.

## Paramètres

*`nom`*

Le nom à donner au nouveau déclencheur. Il doit être distinct du nom de tout autre déclencheur pour la même table.

`BEFORE`
`AFTER`
: Détermine si la fonction est appelée avant ou après l'événement.

*`événement`*
: Fait partie de `INSERT`, `UPDATE` ou `DELETE` ; ceci spécifie l'événement qui lance le déclencheur. Plusieurs événements peuvent être donnés en utilisant `OR`.

*`table`*
: Le nom (pouvant être qualifié du nom du schéma) de la table à laquelle est rattaché le déclencheur.

`FOR EACH ROW`
`FOR EACH STATEMENT`
: Ceci spécifie si la procédure du déclencheur doit être lancée une fois pour chaque ligne affectée par l'événement du déclencheur ou simplement une fois par instruction SQL. Si rien n'est indiqué, `FOR EACH STATEMENT` est la valeur par défaut.

*`nomfonc`*
: Une fonction fournie par l'utilisateur, déclarée comme ne prenant aucun argument et renvoyant le type `trigger`, qui est exécutée lorsque le déclencheur est lancé.

*`arguments`*
: Une liste optionnelle d'arguments séparés par des virgules à fournir à la fonction lors le déclencheur est exécuté. Les arguments sont des constantes littérales de chaînes. Des noms simples et des constantes numériques peuvent être écrits ici aussi mais ils seront tous convertis en chaîne. Merci de vérifier la description du langage d'implémentation de la fonction déclencheur sur la façon dont les arguments du déclencheur sont accessibles à l'intérieur d'une fonction ; cela pourrait être différents des arguments de fonctions standards.

# Notes

Pour créer un déclencheur sur une table, l'utilisateur doit posséder le droit `TRIGGER` sur la table.

Dans les versions de PostgreSQL antérieures à la 7.3, il était nécessaire de déclarer que les fonctions déclencheur renvoyaient un type `opaque`, plutôt que `trigger`. Pour supporter le chargement des anciens fichiers de sauvegarde, `CREATE TRIGGER` acceptera qu'une fonction déclare une valeur de retour de type `opaque` mais il affichera un message d'avertissement et modifiera le type de retour déclaré de la fonction par `trigger`.

Utilisez ***DROP TRIGGER*** pour supprimer un déclencheur.

# Exemples

Section 32.4 contient un exemple complet.

# Compatibilité

L'instruction `CREATE TRIGGER` de PostgreSQL implémente un sous−ensemble du standard SQL:1999. (Il n'existe rien sur les déclencheurs dans SQL−92.) Les fonctionnalités manquantes sont :

- SQL:1999 permet aux déclencheurs de se lancer dans des mises à jour de colonnes spécifiques (c'est–à–dire `AFTER UPDATE OF col1, col2`).
- SQL:1999 vous permet de définir des alias pour les lignes << old >> et << new >> ou pour les tables à utiliser dans la définition des actions déclenchées (c'est–à–dire `CREATE TRIGGER ... ON nomtable REFERENCING OLD ROW AS unnom NEW ROW AS unautrenom ...`). Comme PostgreSQL autorise l'écriture de procédures pour déclencheurs dans tout langage défini par l'utilisateur, l'accès aux données est géré d'une façon spécifique à chaque langage.
- PostgreSQL permet seulement l'exécution d'une fonction définie par l'utilisateur pour l'action déclenchée. SQL:1999 permet l'exécution d'un certain nombres d'autres commandes SQL, telles que `CREATE TABLE` comme action déclenchée. Cette limitation n'est pas difficile à contourner en créant une fonction définie par l'utilisateur et qui exécute les commandes désirées.

SQL:1999 spécifie que les déclencheurs multiples devraient être lancés dans l'ordre de leur création. PostgreSQL utilise l'ordre alphabétique de leur nom, ce qui a été jugé plus agréable à utiliser.

La capacité à spécifier plusieurs actions pour un simple déclencheur en utilisant `OR` est une extension PostgreSQL du standard SQL.

# Voir aussi

*CREATE FUNCTION*, *ALTER TRIGGER*, *DROP TRIGGER*

# CREATE TYPE

## Nom

CREATE TYPE −− définit un nouveau type de donnée

## Synopsis

```
CREATE TYPE nom AS
    ( nom_attribut type_donnée [, ... ] )

CREATE TYPE nom (
    INPUT = fonction_entrée,
    OUTPUT = fonction_sortie
    [ , RECEIVE = fonction_réception ]
    [ , SEND = fonction_envoi ]
    [ , ANALYZE = fonction_analyze ]
    [ , INTERNALLENGTH = { longueurinterne | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignement ]
    [ , STORAGE = stockage ]
    [ , DEFAULT = défaut ]
    [ , ELEMENT = élément ]
    [ , DELIMITER = délimiteur ]
)
```

## Description

`CREATE TYPE` enregistre un nouveau type de données à utiliser dans la base de données actuelle. L'utilisateur qui définit un type devient son propriétaire.

Si un nom de schéma est donné, alors le type est créé dans le schéma spécifié. Sinon, il est créé dans le schéma courant. Le nom du type doit être distinct du nom des types ou domaines existants dans le même schéma. (Comme les tables ont des types de données associés, le nom du type doit aussi être distinct du nom de toute table existante dans le même schéma.)

### Types composés

La première forme `CREATE TYPE` crée un type composite. Le type composé est spécifié par une liste de noms d'attributs et de types de données. Ceci est essentiellement le même que le type row d'une table mais utiliser `CREATE TYPE` évite le besoin de créer une table réelle quand tout ce qui est souhaité est la définition d'un type. Un type composite autonome est utile comme type d'argument ou de retour d'une fonction.

### Types de base

La seconde forme `CREATE TYPE` crée un nouveau type de base (type scalaire). Les paramètres pourraient apparaître dans n'importe quel ordre, pas seulement celui illustré ci−dessus, et la plupart sont optionnels. Vous

devez enregistrer au moins deux fonctions (en utilisant `CREATE FUNCTION`) avant de définir le type. Les fonctions de support *`fonction_entrée`* et *`fonction_sortie`* sont requises alors que les fonctions *`fonction_réception`*, *`fonction_envoi`* et *`fonction_analyze`* sont optionnelles. Généralement, ces fonctions doivent être codées en C ou dans un autre langage de bas niveau.

La *`fonction_entrée`* convertit la représentation textuelle externe du type avec la représentation interne utilisée par les opérateurs et fonctions définis pour le type. *`fonction_sortie`* réalise la transformation inverse. La fonction en entrée pourrait être déclarée comme prenant un argument de type `cstring` ou comme prenant trois arguments de types `cstring`, `oid`, `integer`. Le premier argument est le texte en entrée en tant que chaîne C, le second argument est l'OID du type d'élément dans le cas où il s'agit d'un type tableau (ou le propre OID du type pour un type composite) et le troisième est le `typmod` de la colonne destination, s'il est connu (−1 sinon). La fonction en entrée devrait renvoyer une valeur du type de données lui−même. La fonction en sortie pourrait être déclarée comme prenant un argument du nouveau type de données ou comme prenant deux arguments dont le second est de type `oid`. Le second argument est de nouveau l'OID du type d'élément de tableau pour les types tableau ou l'OID du type pour les types composites. La fonction de sortie devrait renvoyer le type `cstring`.

La *`fonction_réception`* optionnelle convertit la représentation binaire externe du type vers la représentation interne. Si cette fonction n'est pas fournie, le type ne peut pas participer à l'entrée binaire. La représentation binaire devrait être choisie pour être peu coûteuse sur la conversion vers la forme interne tout en étant raisonnablement portable. (Par exemple, les types de données standard entiers utilisent l'ordre réseau des octets comme représentation binaire externe alors que la représentation interne est dans l'ordre natif des octets de la machine.) La fonction de réception devrait réaliser des vérifications adéquates pour s'assurer que la valeur est valide. La fonction de réception pourrait être déclarée comme prenant un argument de type `internal` ou deux arguments de types `internal` et `oid`. Elle doit renvoyer une valeur de type de données lui−même. (Le premier argument est un pointeur vers un tampon `StringInfo` contenant la chaîne d'octets reçus ; le second argument optionnel est soit l'OID du type d'élément dans le cas où il s'agit d'un tableau soit le propre OID du type pour un type composite.) De façon similaire, la *`fonction_envoi`* optionnelle convertit à partir de la représentation interne vers la représentation binaire externe. Si cette fonction n'est pas fournie, le type ne peut pas participer dans la sortie binaire. La fonction d'envoi pourrait être déclarée comme prenant un argument du nouveau type de données ou comme prenant deux arguments dont le second est de type `oid`. Le second argument est encore l'OID du type d'élément du tableau pour les types tableau ou l'OID du type pour les types composites. La fonction d'envoi doit renvoyer le type `bytea`.

La *`fonction_analyse`* en option réalise des récupérations de statistiques spécifiques au type pour les colonnes de ce type de données. Par défaut, `ANALYZE` tentera de récupérer des statistiques en utilisant les opérateurs d'<< égalité >> et d'<< infériorité >> du type, s'il existe une classe d'opérateur par défaut pour le B−tree de ce type. Pour les types non scalaires, ce comportement risque d'être moins convenable, donc il peut être surchargé en indiquant une fonction d'analyse personnalisée. La fonction d'analyse doit être déclarée comme prenant un seul argument de type `internal` et renvoyer un résultat de type `boolean`. L'API détaillée pour les fonctions d'analyses apparaît dans `src/include/commands/vacuum.h`.

Maintenant, vous devriez vous demander comment les fonctions d'entrée et de sortie peuvent être déclarées pour avoir des résultats ou des arguments du nouveau type avant que le nouveau type soit créé. La réponse est que la fonction en entrée doit être créée en premier, puis la fonction de sortie (et les fonctions d'entrées/sorties binaires si souhaitées) et enfin le type de données. PostgreSQL verra tout d'abord le nom du nouveau type de données comme type de retour de la fonction en entrée. Il créera un type << shell >>, qui est une simple entrée d'emplacement dans le catalogue système et lie la définition de fonction en entrée au type de shell. De façon similaire, les autres fonctions seront liées au (maintenant déjà existant) type de shell. Enfin, `CREATE TYPE` remplace l'entrée de shell avec une définition complète du type et le nouveau type peut être utilisé.

Alors que les détails de la représentation interne du nouveau type sont seulement connus des fonctions d'entrées/sorties et des autres fonctions que vous créez pour travailler avec le type, il existe plusieurs propriétés de représentation interne devant être déclarées à PostgreSQL. En premier lieu se trouve *longueurinterne*. Les types de données de base peuvent avoir une longueur fixe auquel cas *longueurinterne* est un entier positif, ou une longueur variable, indiquée en initialisant *longueurinterne* à VARIABLE. (En interne, ceci est représenté en initialisant typlen à −1.) La représentation interne de tous les types de longueur variable doit commencer avec un entier de quatre octets donnant la longueur totale de cette valeur du type.

Le drapeau optionnel PASSEDBYVALUE indique que les valeurs de ce type de données sont passées par valeur plutôt que par référence. Vous pourriez ne pas passer par les types de valeurs dont la représentation interne est plus importante que la taille du type Datum (quatre octets sur la plupart des machines, huit sur quelques−unes).

Le paramètre *alignement* spécifie l'alignement de stockage requis pour le type de données. Les valeurs permises ont un alignement sur les limites de 1, 2, 4 ou 8 octets. Notez que les types de longueurs variables doivent avoir un alignement d'au moins quatre octets car ils contiennent nécessairement un int4 comme premier composant.

Le paramètre *stockage* permet la sélection de stratégies de stockage pour des types de données de longueur variable. (Seul plain est autorisé pour les types de longueur fixe.) plain spécifie que les données du type seront toujours stockées en ligne et non compressées. extended spécifie que le système essaiera tout d'abord de compresser une valeur longue de données et déplacera la valeur en dehors de la ligne de la table principale si elle est trop longue. external permet à la valeur d'être déplacée hors de la table principale mais le système ne tentera pas de la compresser. main autorise la compression mais décourage le déplacement de la valeur en dehors de la table principale. (Les éléments de données avec cette stratégie de stockage pourraient toujours être déplacés hors de la table principale s'il n'y a pas d'autres moyens d'y placer une ligne mais ils seront conservés de préférence dans la table principale sur les éléments extended et external.)

Une valeur par défaut pourrait être spécifiée dans le cas où l'utilisateur souhaite que les colonnes du type de données aient pour valeur par défaut une valeur autre que NULL. Spécifiez la valeur par défaut avec le mot clé DEFAULT. (Une telle valeur par défaut peut être surchargée par une clause DEFAULT explicite attachée dans une colonne particulière.)

Pour indiquer qu'un type est un tableau, spécifiez le type des éléments du tableau en utilisant le mot clé ELEMENT. Par exemple, pour définir un tableau d'entiers de quatre octets (int4), spécifiez ELEMENT = int4. Plus de détails sur les types tableau apparaissent ci−dessous.

Pour indiquer le délimiteur à utiliser entre les valeurs dans la représentation externe des tableaux de ce type, *délimiteur* peut être configuré à un caractère particulier. Le délimiteur par défaut est la virgule (,). Notez que le délimiteur est associé avec le type d'élément de tableau, pas le type tableau lui−même.

## Types de tableaux

À chaque fois qu'un type de données de base défini par un utilisateur est créé, PostgreSQL crée automatiquement un type tableau associé dont le nom consiste du nom du type de base précédé d'un tiret base. L'analyseur comprend cette convention de nommage et traduit les requêtes pour les colonnes de type foo[] en requêtes pour le type _foo. Le type tableau créé implicitement est de longueur variable et utilise les

fonctions entrée et sortie intégrée `array_in` et `array_out`.

Vous pourriez raisonnablement vous demander pourquoi il existe une option `ELEMENT` si le système fabrique automatiquement le bon type tableau. Le seul cas où il est utile d'utiliser `ELEMENT` est lorsque vous réalisez un type de longueur fixe qui est en interne un tableau d'un nombre identique d'éléments et que vous souhaitez autoriser ces éléments à accéder directement aux indices, en plus de toute autre opération que vous pensiez fournir pour le type dans sa globalité. Par exemple, le type `name` permet à ses éléments `char` constituants d'être accessible de cette façon. Un type `point` en 2−D pourrait autoriser les deux numéros de composant à être accédé ainsi : `point[0]` et `point[1]`. Notez que cette fonctionnalité fonctionne uniquement avec les types de longueur fixe dont la forme interne est exactement une séquence de champs de longueur variable. Un type longueur de variable à indice doit avoir la représentation interne généralisée utilisée par `array_in` et `array_out`. Pour des raisons historiques (c'est−à−dire qu'il est clairement mauvais parce qu'il est trop tard pour changer) s'abonner à des types de longueur variable commence à partir de zéro plutôt que de un pour les tableaux de longueur variable.

# Paramètres

*nom*
: Le nom d'un type à créer (pouvant être qualifié par le nom du schéma).

*nom_attribut*
: Le nom d'un attribut (colonne) pour le type composé.

*type_données*
: Le nom d'un type de données existant qui deviendra une colonne du type composé.

*fonction_entrée*
: Le nom d'une fonction qui convertit les données à partir de la forme textuelle externe du type en sa forme interne.

*fonction_sortie*
: Le nom d'une fonction qui convertit les données à partir de la forme interne du type dans sa forme textuelle externe.

*fonction_réception*
: Le nom d'une fonction qui convertit la forme binaire externe du type dans sa forme interne.

*fonction_envoi*
: Le nom d'une fonction qui convertit les données à partir de la forme interne du type dans sa forme binaire externe.

*analyze_function*
: Le nom d'une fonction qui réalise des analyses statistiques pour le type de données.

*longueurinterne*
: Une constante numérique qui spécifie la longueur en octets de la représentation interne du nouveau type. La supposition par défaut est que la longueur est variable.

*alignement*
: Le besoin d'alignement du stockage du type de données. Si spécifié, il doit être parmi `char`, `int2`, `int4` ou `double` ; par défaut, il s'agit d'`int4`.

*stockage*
: La stratégie de stockage pour le type de données. Si spécifiée, elle doit faire partie de `plain`, `external`, `extended` ou `main` ; la valeur par défaut est `plain`.

*défaut*
: La valeur par défaut pour le type de données. Si elle est omise, elle est NULL.

*élément*
: Le type en cours de création est un tableau ; ceci spécifie le type des éléments du tableau.

*délimiteur*
: Le caractère délimiteur à utiliser entre les valeurs des tableaux de ce type.

# Notes

Les noms de types définis par l'utilisateur ne peuvent pas commencer avec un tiret bas (_) et ont une longueur maximale de 62 caractères (ou en général `NAMEDATALEN` – 2, plutôt que `NAMEDATALEN` – 1 caractères autorisés pour les autres noms). Les noms de types commençant avec un tiret bas sont réservés par les noms de type tableau créés en interne.

Dans les versions de PostgreSQL antérieures à la 7.3, il était coutumier d'éviter de créer un type shell en remplaçant les références des fonctions au nom du type avec le pseudotype `opaque`. Les arguments `cstring` et les résultats doivent aussi être déclarés comme `opaque` avant la 7.3. Pour supporter le chargement d'anciens fichiers de sauvegarde, `CREATE TYPE` acceptera les fonctions déclarées avec `opaque` mais il affichera un message d'avertissement et modifiera la déclaration de la fonction pour utiliser les bons types.

# Exemples

Cet exemple crée un type composite et l'utilise dans la définition d'une fonction :

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

Cet exemple crée le type de données de base `box`, puis l'utilise dans une définition de table :

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = ma_fonction_entree_box,
    OUTPUT = ma_fonction_sortie_box
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

Si la structure interne de `box` était un tableau de quatre éléments `float4`, nous pourrions faire

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = ma_fonction_entree_box,
    OUTPUT = ma_fonction_sortie_box,
    ELEMENT = float4
);
```

ce qui permettrait un accès aux nombres composant la valeur d'une boîte avec les indices. Sinon, le type se comporte de la même façon qu'avant.

Cet exemple crée un objet large et l'utilise dans une définition de table :

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

Vous trouverez plus d'exemples, intégrant des fonctions intéressantes en entrée et en sortie, dans Section 31.11.

# Compatibilité

Cette commande `CREATE TYPE` est une extension PostgreSQL. Il existe une instruction `CREATE TYPE` dans SQL:1999 et ses versions ultérieures qui est plutôt différente dans le détail.

# Voir aussi

*CREATE FUNCTION*, *DROP TYPE*, *ALTER TYPE*

# CREATE USER

## Nom

CREATE USER −− définit un nouveau compte utilisateur dans la base de données

## Synopsis

```
CREATE USER nom [ [ WITH ]
option [ ... ] ]

where option can be:

      SYSID uid
    | CREATEDB | NOCREATEDB
    | CREATEUSER | NOCREATEUSER
    | IN GROUP nomgroupe [, ...]
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'mot_de_passe'
    | VALID UNTIL 'temps_absolu'
```

## Description

`CREATE USER` ajoute un nouvel utilisateur dans le groupe de bases de données PostgreSQL. Référez−vous à Chapitre 17 et Chapitre 19 pour des informations sur la gestion des utilisateurs et l'authentification. Vous devez être le superutilisateur de la base de données pour utiliser cette commande.

## Paramètres

*nom*
: Le nom du nouvel utilisateur.

*uid*
: La clause `SYSID` peut être utilisée pour choisir l'identifiant du nouvel utilisateur PostgreSQL. Ceci n'est normalement pas nécessaire mais pourrait être utile si vous avez besoin de recréer le propriétaire d'un objet orphelin.

  Si ceci n'est pas spécifié, la valeur la plus haute des identifiants déjà créés plus un (avec un minimum de 100) sera utilisée par défaut.

`CREATEDB`
`NOCREATEDB`
: Ces clauses définissent la capacité de l'utilisateur pour créer des bases de données. Si `CREATEDB` est spécifié, l'utilisateur en cours de définition sera autorisé à créer ses propres bases de données. Utiliser `NOCREATEDB` interdira la capacité de créer des bases de données à cet utilisateur. Si cette clause est omise, `NOCREATEDB` est utilisé par défaut.

`CREATEUSER`
`NOCREATEUSER`
: Ces clauses déterminent si un utilisateur aura la permission de créer des nouveaux utilisateurs lui−même. Cette option fait aussi de l'utilisateur un superutilisateur qui peut dépasser toutes les

restrictions d'accès. Oublier cette clause initialisera la valeur de cet attribut à NOCREATEUSER.

*nomgroupe*

Un nom de groupe dans lequel insérer l'utilisateur. Plusieurs noms de groupe peuvent être cités.

*mot_de_passe*

Initialise le mot de passe de l'utilisateur. Si vous ne pensez pas utiliser l'authentification par mot de passe, vous pouvez oublier cette option, mais l'utilisateur ne sera pas capable ensuite de se connecter si vous décidez de basculer à une authentification par mot de passe. Le mot de passe peut être initialisé ou changé plus tard en utilisant *ALTER USER*.

ENCRYPTED

UNENCRYPTED

Ces mots clés contrôlent si le mot de passe doit être stocké crypté dans les catalogues système. (Si aucun des deux n'est spécifié, le comportement par défaut est déterminé par le paramètre de configuration PASSWORD_ENCRYPTION.) Si la chaîne de mot de passe fournie est déjà dans le format crypté MD5, alors il est stocké directement, que ENCRYPTED ou UNENCRYPTED soit founi ou non (car le système ne peut pas décrypter la chaîne de mot de passe cryptée). Ceci permet le rechargement des mots de passe cryptés lors de la sauvegarde/restauration.

Notez que les anciens clients pourraient ne pas disposer des mécanismes d'authentification MD5 nécessaires pour travailler avec des mots de passe stockés en cryptés.

*temps_absolu*

La clause VALID UNTIL initialise le temps absolu après lequel le mot de passe de l'utilisateur n'est plus valide. Si cette clause est omise, le mot de passe sera valide en permanence.

# Notes

Utilisez *ALTER USER* pour changer les attributs d'un utilisateur, et *DROP USER* pour le supprimer. Utilisez *ALTER GROUP* pour ajouter l'utilisateur aux groupes ou supprimer cet utilisateur de ces groupes.

PostgreSQL inclut un programme *createuser* qui a les mêmes fonctionnalités que CREATE USER (en fait, il appelle cette commande) mais qui peut être lancé à partir du shell.

La clause VALID UNTIL définit une date d'expiration pour un mot de passe seulement, pas pour le compte utilisateur en lui−même. En particulier, la date d'expiration n'est pas obligatoire lors d'une connexion avec une méthode d'authentification non basée sur un mot de passe.

# Exemples

Créez un utilisateur sans mot de passe :

```
CREATE USER jonathan;
```

Créez un utilisateur avec un mot de passe :

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

Créez un utilisateur avec un mot de passe qui est valide jusqu'à la fin 2004. Après une seconde dans 2005, le mot de passe n'est plus valide.

```
CREATE USER miriam WITH PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

Créez un compte où l'utilisateur peut créer des bases de données :

```
CREATE USER manuel WITH PASSWORD 'jw8s0F4' CREATEDB;
```

# Compatibilité

L'instruction `CREATE USER` est une extension PostgreSQL. Le standard SQL laisse la définition des utilisateurs à l'implémentation.

# Voir aussi

*ALTER USER*, *DROP USER*, createuser

# CREATE VIEW

## Nom

CREATE VIEW −− définit une nouvelle vue

## Synopsis

```
CREATE [ OR REPLACE ] VIEW nom [ ( nom_colonne [, ...] ) ] AS requête
```

## Description

`CREATE VIEW` définit une vue d'une requête. La vue n'est pas matérialisée physiquement. Au lieu de cela, la requête est lancée chaque fois qu'une vue est référencée dans une requête.

`CREATE OR REPLACE VIEW` est similaire mais elle est remplacée si une vue du même nom existe déjà. Vous pouvez seulement remplacer une vue avec une nouvelle requête qui génère un ensemble de colonnes identiques (c'est−à−dire les mêmes noms de colonnes et les mêmes types de données).

Si un nom de schéma est donné (par exemple `CREATE VIEW monschema.mavue ...`), alors la vue est créée dans le schéma donné. Dans les autres cas, elle est créée dans le schéma courant. Le nom de la vue doit être différent du nom des autres vues, tables, séquences ou index du même schéma.

## Paramètres

*nom*
: Le nom de la vue à créer (qualifié ou non du nom du schéma).

*nom de colonne*
: Une liste optionnelle de noms à utiliser pour les colonnes de la vue. Si elle n'est pas donnée, le nom des colonnes est déduit de la requête.

*requête*
: Une requête (c'est−à−dire une instruction `SELECT`) qui fournit les colonnes et les lignes de la vue.

  Référez−vous à l'instruction ***SELECT*** pour plus d'informations sur les requêtes valides.

## Notes

Actuellement, les vues sont en lecture seule : le système n'autorise pas une insertion, une mise à jour ou une suppression sur la vue. Vous pouvez obtenir les effets d'une vue pouvant être mise à jour en créant des règles qui ré−écrivent les insertions, etc. sur la vue en actions appropriées sur les autres tables. Pour plus d'informations, regardez ***CREATE RULE***.

Utilisez l'instruction `DROP VIEW` pour supprimer les vues.

Faites attention à ce que les noms et les types des colonnes de la vue correspondent bien à ce que vous souhaitez. Par exemple,

```
CREATE VIEW vista AS SELECT 'Hello World';
```

est doublement une mauvaise forme : le nom de la colonne est par défaut `?column?` et le type de données de la colonne est par défaut `unknown`. Si vous voulez une chaîne de caractères dans le résultat de la vue, utilisez quelque chose comme

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

L'accès aux tables référencées dans la vue est déterminé par les droits du propriétaire de la vue. Cependant, les fonctions appelées dans la vue sont traitées comme si elles avaient été appelées directement par la requête utilisant la vue. Par conséquent, l'utilisateur de la vue doit avoir les droits sur toutes les fonctions utilisées par la vue.

# Exemples

Créer une vue sur tous les films de comédie :

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE genre = 'Comédie';
```

# Compatibilité

Le standard SQL spécifie quelques possibilités supplémentaires pour l'instruction `CREATE VIEW` :

```
CREATE VIEW nom [ ( nom_colonne [, ...] ) ]
    AS requête
    [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

Les clauses optionnelles de la commande SQL complète sont :

`CHECK OPTION`
: Cette option est à utiliser avec les vues pouvant être mises à jour. Toutes les commandes `INSERT` et `UPDATE` sur la vue sont contrôlées pour s'assurer que les données sont conformes à la condition définissant la vue. (C'est–à–dire que les nouvelles données seraient visibles à travers la vue). Si cela n'est pas le cas, la mise à jour est rejetée.

`LOCAL`
: Contrôle l'intégrité sur la vue.

`CASCADE`
: Contrôle l'intégrité sur cette vue et sur toutes les vues dépendantes. `CASCADE` est utilisé si ni `CASCADE` ni `LOCAL` ne sont spécifiés.

`CREATE OR REPLACE VIEW` est une extension du langage de PostgreSQL.

# Voir aussi

*DROP VIEW*

# DEALLOCATE

## Nom

DEALLOCATE −− désalloue une instruction préparée

## Synopsis

```
DEALLOCATE [ PREPARE ] nom_plan
```

## Description

`DEALLOCATE` est utilisé pour désallouer une instruction SQL préparée précédemment. Si vous ne désallouez pas explicitement une instruction préparée, elle est désallouée à la fin de la session.

Pour plus d'informations sur les instructions préparées, voir *PREPARE*.

## Paramètres

`PREPARE`
: Ce mot clé est ignoré.

*`nom_plan`*
: Le nom de l'instruction préparée à désallouer.

## Compatibilité

Le standard SQL inclut une instruction `DEALLOCATE` mais elle est seulement utilisée pour le SQL imbriqué.

## Voir aussi

*EXECUTE*, *PREPARE*

# DECLARE

## Nom

DECLARE −− définit un curseur

## Synopsis

```
DECLARE nom [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR requête
    [ FOR { READ ONLY | UPDATE [ OF colonnes [, ...] ] } ]
```

## Description

`DECLARE` permet à un utilisateur de créer des curseurs qui peuvent être utilisés pour récupérer un petit nombre de lignes à la fois à partir d'une requête plus importante. Les curseurs peuvent renvoyer des données soit au format texte soit au format binaire en utilisant ***FETCH***.

Les curseurs normaux renvoient des données au format texte, le même qu'un `SELECT` produirait. Quand des données sont stockées nativement dans le format binaire, le système doit faire une conversion pour produire le format texte. Une fois que l'information revient au format texte, l'application cliente pourrait avoir besoin de la convertir en un format binaire pour la manipuler. De plus, des données au format texte sont souvent plus volumineuse qu'au format binaire. Les curseurs binaires renvoient les données dans une représentation binaire qui pourrait être plus facilement manipulée. Néanmoins, si vous avez l'intention d'afficher les données en texte, le récupérer au format texte vous épargne quelques efforts du côté client.

Par exemple, si une requête renvoie une valeur de "un" à partir d'une colonne entier, vous obtenez une chaîne de `1` avec un curseur par défaut alors qu'avec un curseur binaire, vous obtenez un champ de quatre octets contenant la représentation interne de la valeur (dans l'ordre d'octets big−endian).

Les curseurs binaires doivent être utilisés avec attention. Beaucoup d'applications, dont psql, ne sont pas préparées pour gérer des curseurs binaires et attendent leurs données au format texte.

> **Note :** Quand l'application cliente utilise le protocole de << requête étendue >> pour lancer une commande `FETCH`, le message du protocole Bind indique si la donnée doit être récupérée au format texte ou binaire. Ce choix surcharge la façon dont le curseur est défini. Le concept de curseur binaire en tant que tel est du coup obsolète lors de l'utilisation du protocole de requête étendue — tout curseur peut être traité soit en texte soit en binaire.

## Paramètres

*nom*
: Le nom du curseur à créer.

`BINARY`
: Fait que le curseur renvoie des données au format binaire plutôt qu'au format texte.

`INSENSITIVE`
: Indique que les données récupérées à partir du curseur ne doivent pas être affectées par les mises à jour des tables sous−jacentes au curseur tant que celui−ci existe. Dans PostgreSQL, tous les curseurs sont insensibles ; cet mot clé n'a actuellement aucun effet et est présent pour la compatibilité avec le standard SQL.

`SCROLL`
`NO SCROLL`
: `SCROLL` spécifie que le curseur peut être utilisé pour récupérer des lignes de façon non séquentielle (c'est−à−dire en arrière). Suivant la complexité du plan d'exécution de la requête, spécifier `SCROLL` pourrait induire des pertes de performance sur le temps d'exécution de la requête. `NO SCROLL` spécifie que le curseur ne peut pas être utilisé pour récupérer des lignes d'une façon non séquentielle.

`WITH HOLD`
`WITHOUT HOLD`
: `WITH HOLD` spécifie que le curseur peut continuer à être utilisé après que la transaction qui l'a créé ait été validée avec succès. `WITHOUT HOLD` spécifie que le curseur ne peut être utilisé en dehors de la transaction qui l'a créé. Si ni `WITHOUT HOLD` ni `WITH HOLD` n'est spécifié, `WITHOUT HOLD` est la valeur par défaut.

*`requête`*
: Une commande `SELECT` qui fournit les lignes à renvoyer par le curseur. Référez−vous à ***SELECT*** pour plus d'informations sur les requêtes valides.

`FOR READ ONLY`
`FOR UPDATE`
: `FOR READ ONLY` indique que le curseur sera utilisé en mode lecture seule. `FOR UPDATE` indique que le curseur sera utilisé pour mettre à jour des tables. Comme les mises à jour de curseur ne sont pas supportées actuellement dans PostgreSQL, spécifier `FOR UPDATE` cause un message d'erreur et spécifier `FOR READ ONLY` n'a pas d'effet.

*`colonnes`*
: Colonne(s) à mettre à jour par le curseur. Comme les mises à jour de curseur ne sont actuellement pas supportées dans PostgreSQL, la clause `FOR UPDATE` provoque un message d'erreur.

Les mots clés `BINARY`, `INSENSITIVE` et `SCROLL` peuvent apparaître dans n'importe quel ordre.

# Notes

Sauf si `WITH HOLD` est spécifié, le curseur créé par cette commande peut seulement être utilisée à l'intérieur d'une transaction. Du coup, `DECLARE` sans `WITH HOLD` est inutile à l'extérieur d'un bloc de transaction : le curseur survivrait seulement jusqu'à la fin de l'instruction. Du coup, PostgreSQL rapporte une erreur si cette commande est utilisée en dehors d'un bloc de transaction. Utilisez ***BEGIN***, ***COMMIT*** et ***ROLLBACK*** pour définir un bloc de transaction.

Si `WITH HOLD` est spécifié et que la transaction créant le curseur valide avec succès, le curseur peut continuer à être accédé par les transactions suivantes de la même session. (Mais si la transaction l'ayant créé est annulée, le curseur est supprimé.) Un curseur créé avec `WITH HOLD` est fermé quand une commande `CLOSE` explicite est lancée sur lui ou quand la session se termine. Dans l'implémentation actuelle, les lignes représentées par un curseur `WITH HOLD` sont copiées dans un fichier temporaire ou dans une partie de la mémoire pour qu'elles restent disponibles pour les transactions suivantes.

L'option `SCROLL` doit normalement être spécifiée lors de la création d'un curseur qui sera utilisé avec des récupérations inverses. Ceci est requis par le standard SQL. Néanmoins, pour la compatibilité avec les

anciennes versions, PostgreSQL autorise les récupérations en arrière sans `SCROLL`, si le plan de requête du curseur est assez simple pour qu'aucune surcharge ne soit nécessaire pour le supporter. Néanmoins, les développeurs d'applications ne doivent compter sur l'utilisation de la recherche en arrière à partir de curseurs qui n'ont pas été créés avec `SCROLL`. Si `NO SCROLL` est spécifié, alors les recherches inverses sont interdites dans tous les cas.

Le standard SQL ne mentionne les curseurs que dans le SQL embarqué. Le serveur PostgreSQL n'implémente pas l'instruction `OPEN` pour les curseurs ; un curseur est considéré ouvert à sa déclaration. Néanmoins, ECPG, le préprocesseur SQL embarqué de PostgreSQL, supporte les conventions des curseurs du SQL standard, dont celles utilisant les instructions `DECLARE` et `OPEN`.

# Exemples

Pour déclarer un curseur :

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

Voir *FETCH* pour plus d'exemples sur l'utilisation des curseurs.

# Compatibilité

Le standard SQL autorise les curseurs uniquement dans le SQL embarqué et dans des modules. PostgreSQL permet aux curseurs d'être utilisés interactivement.

Le standard SQL autorise les curseurs à mettre à jour les données d'une table. Tous les curseurs PostgreSQL sont en lecture seule.

Les curseurs binaires sont une extension PostgreSQL.

# Voir aussi

*CLOSE*, *FETCH*, *MOVE*

# DELETE

## Nom

DELETE −− supprime les lignes d'une table

## Synopsis

```
DELETE FROM [ ONLY ] table [ WHERE condition ]
```

## Description

`DELETE` supprime les lignes qui satisfont la clause `WHERE` pour la table spécifiée. Si la clause `WHERE` est absente, toutes les lignes de la table sont supprimées. Le résultat est une table valide mais vide.

> **Astuce :** *TRUNCATE* est une extension PostgreSQL fournissant un mécanisme plus rapide pour supprimer toutes les lignes d'une table.

Par défaut, `DELETE` supprime les lignes de la table spécifiée et de ses sous−tables. Si vous souhaitez supprimer uniquement les lignes de la table mentionnée, vous devez utiliser la clause `ONLY`.

Vous devez avoir le droit `DELETE` sur la table pour en supprimer des lignes ainsi que le droit `SELECT` pour toute table dont les valeurs sont lues dans la *`condition`*.

## Paramètres

*`table`*
: Le nom d'une table existante (qualifié ou non du nom du schéma).

*`condition`*
: Une expression de valeur renvoyant une valeur de type `boolean` déterminant les lignes à supprimer.

## Sorties

En cas de succès, une commande `DELETE` renvoie une information de la forme

```
DELETE nombre
```

Le *`nombre`* correspond au nombre de lignes supprimées. Si *`nombre`* vaut 0, c'est qu'aucune ligne ne correspond à *`condition`* (ceci n'est pas considéré comme une erreur).

# Notes

PostgreSQL vous laisse référencer des colonnes d'autres tables dans la condition `WHERE`. Par exemple, pour supprimer tous les films produits par un producteur donné

```
DELETE FROM films
  WHERE id_producteur = producteurs.id AND producteurs.nom = 'foo';
```

En gros, une jointure s'établit entre `films` et `producteurs` dont toutes les lignes jointes sont supprimées. Cette syntaxe n'est pas standard. Une façon plus standard de le faire est d'utiliser une sous−selection comme :

```
DELETE FROM films
  WHERE id_producteur IN (SELECT id FROM producteur WHERE nom = 'foo');
```

Dans certains cas, la jointure est plus facile à écrire ou plus rapide à exécuter que la sous−sélection. Une objection à la jointure est qu'il n'y a pas une liste explicite des tables à utiliser, ce qui rend ce moyen plus difficile à mettre en œuvre ; de plus, cette méthode ne permet pas les auto jointures.

# Exemples

Supprime tous les films à part les films musicaux :

```
DELETE FROM films WHERE genre <> 'Musique';
```

Efface toutes les lignes de la table `films` :

```
DELETE FROM films;
```

# Compatibilité

Cette commande est conforme au standard SQL, à l'exception de la possibilité de référencer d'autres tables dans la clause `WHERE`, qui est une extension de PostgreSQL.

# DROP AGGREGATE

## Nom

DROP AGGREGATE −− supprime une fonction d'agrégat

## Synopsis

```
DROP AGGREGATE nom ( type ) [ CASCADE | RESTRICT ]
```

## Description

`DROP AGGREGATE` supprime une fonction d'agrégat existante. Pour exécuter cette commande, l'utilisateur courant doit être le propriétaire de la fonction d'agrégat.

## Paramètres

*nom*
: Le nom (qualifié ou non d'un nom de schéma) d'une fonction d'agrégat existante.

*type*
: Le type de données en argument de la fonction d'agrégat ou `*` si la fonction accepte tout type de données.

`CASCADE`
: Supprime automatiquement les objets dépendants de la fonction d'agrégat.

`RESTRICT`
: Refuse de supprimer la fonction d'agrégat si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Pour supprimer la fonction d'agrégat `mamoyenne` pour le type `integer` :

```
DROP AGGREGATE mamoyenne(integer);
```

## Compatibilité

Il n'existe pas d'instruction `DROP AGGREGATE` dans le standard SQL.

## Voir aussi

*ALTER AGGREGATE*, *CREATE AGGREGATE*

# DROP CAST

## Nom

DROP CAST −− supprime un transtypage

## Synopsis

```
DROP CAST (typesource AS typecible) [ CASCADE | RESTRICT ]
```

## Description

`DROP CAST` supprime un transtypage (conversion entre deux types de données) défini précédemment.

Pour avoir le droit de supprimer un transtypage, vous devez être le propriétaire du type de données source ou cible. Ce sont les mêmes droits que ceux requis pour la création d'un transtypage.

## Paramètres

*typesource*
: Le nom du type de données source du transtypage.

*typecible*
: Le nom du type de données cible du transtypage

CASCADE
RESTRICT
: Ces mots clés n'ont pas d'effet car il n'existe pas de dépendances dans les transtypages.

## Exemples

Pour supprimer le transtypage du type `text` en un type `int` :

```
DROP CAST (text AS int);
```

## Compatibilité

La commande `DROP CAST` est conforme au standard SQL.

## Voir aussi

*CREATE CAST*

# DROP CONVERSION

## Nom

DROP CONVERSION −− supprime une conversion

## Synopsis

```
DROP CONVERSION nom [ CASCADE | RESTRICT ]
```

## Description

`DROP CONVERSION` supprime une conversion précédemment définie. Pour avoir le droit de supprimer une conversion, vous devez en être le propriétaire.

## Paramètres

*nom*
: Le nom de la conversion. Il pourrait être qualifié du nom du schéma.

`CASCADE`
`RESTRICT`
: Ces mots clés n'ont pas d'effet car il n'existe pas de dépendances sur les conversions.

## Exemples

Pour supprimer la conversion nommée `mon_nom` :

```
DROP CONVERSION mon_nom;
```

## Compatibilité

Il n'existe pas d'instruction `DROP CONVERSION` dans le standard SQL.

## Voir aussi

*ALTER CONVERSION*, *CREATE CONVERSION*

# DROP DATABASE

## Nom

DROP DATABASE −− supprime une base de données

## Synopsis

```
DROP DATABASE nom
```

## Description

`DROP DATABASE` supprime une base de données. Elle supprime les entrées du catalogue pour la base de données et supprime le répertoire contenant les données. Elle ne peut être exécutée que par le propriétaire de la base de données. De plus, elle ne peut être exécutée si quelqu'un est connecté sur la base de données cible, vous y compris. (Connectez−vous à `template1` ou à toute autre base de données pour lancer cette commande.)

`DROP DATABASE` ne peut pas être annulé. À utiliser avec précaution !

## Paramètres

*name*
: Le nom de la base de données à supprimer.

## Notes

`DROP DATABASE` ne peut pas être exécuté à l'intérieur d'un bloc de transaction.

Cette commande ne peut pas être exécutée en étant connecté à la base de données cible. Du coup, il pourrait être plus facile d'utiliser le programme *dropdb* à la place, qui est un emballage pour cette commande.

## Compatibilité

Il n'existe pas d'instruction `DROP DATABASE` dans le standard SQL.

## Voir aussi

*CREATE DATABASE*

# DROP DOMAIN

## Nom

DROP DOMAIN −− supprime un domaine

## Synopsis

```
DROP DOMAIN nom [, ...]  [ CASCADE | RESTRICT ]
```

## Description

`DROP DOMAIN` supprime un domaine. Seul le propriétaire d'un domaine peut le supprimer.

## Paramètres

*name*
: Le nom (éventuellement qualifié du nom du schéma) d'un domaine existant.

`CASCADE`
: Supprime automatiquement les objets dépendants de ce domaine (tels que les colonnes de table).

`RESTRICT`
: Refuse de supprimer le domaine si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Pour supprimer le domaine `boite` :

```
DROP DOMAIN boite;
```

## Compatibilité

Cette commande est conforme au standard SQL.

## Voir aussi

*CREATE DOMAIN*

# DROP FUNCTION

## Nom

DROP FUNCTION −− supprime une fonction

## Synopsis

```
DROP FUNCTION nom ( [ type [, ...] ] ) [ CASCADE | RESTRICT ]
```

## Description

`DROP FUNCTION` supprime la définition d'une fonction existante. Pour exécuter cette commande, l'utilisateur doit être le propriétaire de la fonction. Les types d'argument de la fonction doivent être spécifiés car plusieurs fonctions différentes peuvent exister avec le même nom mais avec une liste d'arguments différents.

## Paramètres

*name*
: Le nom (qualifié ou non du nom du schéma) d'une fonction existante.

*type*
: Le type de données d'un argument de la fonction.

`CASCADE`
: Supprime automatiquement les objets dépendants de la fonction (tels que les opérateurs ou les déclencheurs).

`RESTRICT`
: Refuse de supprimer la fonction si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Cette commande supprime la fonction de calcul de la racine carrée :

```
DROP FUNCTION sqrt(integer);
```

## Compatibilité

Une instruction `DROP FUNCTION` est définie dans le standard SQL mais elle n'est pas compatible avec cette commande.

# Voir aussi

*CREATE FUNCTION*, *ALTER FUNCTION*

# DROP GROUP

## Nom

DROP GROUP −− supprime un groupe d'utilisateurs

## Synopsis

```
DROP GROUP nom
```

## Description

`DROP GROUP` supprime le groupe spécifié. Les utilisateurs du groupe ne sont pas supprimés.

## Paramètres

*nom*
: Le nom d'un groupe existant.

## Notes

Il est déconseillé de supprimer un groupe possédant des droits sur des objets. Actuellement, cela n'est pas vérifié mais il est probable que les prochaines versions de PostgreSQL vérifieront ce problème.

## Exemples

Pour supprimer un groupe :

```
DROP GROUP staff;
```

## Compatibilité

Il n'existe pas d'instruction `DROP GROUP` dans le standard SQL.

## Voir aussi

*ALTER GROUP*, *CREATE GROUP*

# DROP INDEX

## Nom

DROP INDEX −− supprime un index

## Synopsis

```
DROP INDEX nom [, ...] [ CASCADE | RESTRICT ]
```

## Description

`DROP INDEX` supprime un index existant du système de bases de données. Pour exécuter cette commande, vous devez être le propriétaire de l'index.

## Paramètres

*nom*
: Le nom (pouvant être qualifié du nom du schéma) de l'index à supprimer.

`CASCADE`
: Supprime automatiquement les objets dépendants de l'index.

`RESTRICT`
: Refuse de supprimer l'index si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Cette commande supprime l'index `title_idx`:

```
DROP INDEX title_idx;
```

## Compatibilité

`DROP INDEX` est une extension de PostgreSQL. Il n'est pas fait mention des index dans le standard SQL.

## Voir aussi

*CREATE INDEX*

# DROP LANGUAGE

## Nom

DROP LANGUAGE −− supprime un langage procédural

## Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE nom [ CASCADE | RESTRICT ]
```

## Description

`DROP LANGUAGE` supprime la définition du langage procédural enregistré précédemment sous le nom *`nom`*.

## Paramètres

*`name`*
: Le nom d'un langage procédural existant. Pour une compatibilité ascendante, le nom peut être entouré de guillemets simples.

`CASCADE`
: Supprime automatiquement les objets dépendants du langage (comme les fonctions du langage).

`RESTRICT`
: Refuse de supprimer le langage si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Cette commande supprime le langage procédural `plexemple` :

```
DROP LANGUAGE plexemple;
```

## Compatibilité

Il n'existe pas d'instruction `DROP LANGUAGE` dans le standard SQL.

## Voir aussi

*ALTER LANGUAGE*, *CREATE LANGUAGE*, droplang

# DROP OPERATOR

## Nom

DROP OPERATOR −− supprime un opérateur

## Synopsis

```
DROP OPERATOR nom ( { typegauche | NONE } , { typedroit | NONE } ) [ CASCADE | RESTRICT ]
```

## Description

`DROP OPERATOR` supprime un opérateur existant du système de bases de données. Pour exécuter cette commande, vous devez être le propriétaire de l'opérateur.

## Paramètres

*nom*
: Le nom d'un opérateur existant (parfois qualifié avec le nom du schéma).

*typegauche*
: Le type de données de l'opérande gauche de l'opérateur ; écrivez `NONE` si l'opérateur n'a pas d'opérande gauche.

*typedroit*
: Le type de données de l'opérande droit de l'opérateur ; écrivez `NONE` si l'opérateur n'a pas d'opérande droit.

`CASCADE`
: Supprime automatiquement les objets dépendants de l'opérateur.

`RESTRICT`
: Refuse de supprimer l'opérateur si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Supprime l'opérateur puissance `a^b` pour le type `integer` :

```
DROP OPERATOR ^ (integer, integer);
```

Supprime l'opérateur de complément binaire `~b` pour le type `bit` :

```
DROP OPERATOR ~ (none, bit);
```

Supprime l'opérateur unaire factorielle `x!` pour le type `bigint` :

```
DROP OPERATOR ! (bigint, none);
```

## Compatibilité

Il n'existe pas d'instruction `DROP OPERATOR` dans le standard SQL.

## Voir aussi

*CREATE OPERATOR*, *ALTER OPERATOR*

# DROP OPERATOR CLASS

## Nom

DROP OPERATOR CLASS −− supprime une classe d'opérateur

## Synopsis

```
DROP OPERATOR CLASS nom USING méthode_index [ CASCADE | RESTRICT ]
```

## Description

`DROP OPERATOR CLASS` supprime une classe d'opérateur existante. Pour exécuter cette commande, vous devez être le propriétaire de la classe d'opérateur.

## Paramètres

*name*
: Le nom (éventuellement qualifié du nom du schéma) d'une classe d'opérateur existante.

*méthode_index*
: Le nom de la méthode d'accès à l'index associée à la classe d'opérateur.

`CASCADE`
: Supprime automatiquement les objets dépendants de cette classe.

`RESTRICT`
: Refuse de supprimer cette classe d'opérateur si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Supprime la classe d'opérateur `widget_ops` des index de type arbres balancés (B−tree) :

```
DROP OPERATOR CLASS widget_ops USING btree;
```

Cette commande échoue s'il n'y a pas d'index utilisant la classe d'opérateur. Ajoutez `CASCADE` pour supprimer ces index avec la classe d'opérateur.

## Compatibilité

Il n'existe pas d'instruction `DROP OPERATOR CLASS` dans le standard SQL.

# Voir aussi

*ALTER OPERATOR CLASS*, *CREATE OPERATOR CLASS*

# DROP RULE

## Nom

DROP RULE −− supprime une règle de réécriture

## Synopsis

```
DROP RULE nom ON relation [ CASCADE | RESTRICT ]
```

## Description

`DROP RULE` supprime une règle de réécriture.

## Paramètres

*`nom`*
: Le nom de la règle à supprimer.

*`relation`*
: Le nom (qualifié ou non du nom du schéma) de la table ou vue sur laquelle s'applique la règle.

`CASCADE`
: Supprime automatiquement les objets dépendants de la règle.

`RESTRICT`
: Refuse de supprimer la règle si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Pour supprimer la règle de réécriture `nouvellerègle` :

```
DROP RULE nouvellerègle ON matable;
```

## Compatibilité

Il n'existe pas d'instruction `DROP RULE` dans le standard SQL.

## Voir aussi

*CREATE RULE*

# DROP SCHEMA

## Nom

DROP SCHEMA −− supprime un schéma

## Synopsis

```
DROP SCHEMA nom [, ...] [ CASCADE | RESTRICT ]
```

## Description

`DROP SCHEMA` supprime des schémas de la base de données.

Un schéma peut seulement être supprimé par son propriétaire ou par un super utilisateur. Notez que le propriétaire peut supprimer un schéma et tous les objets qu'il contient même s'il ne possède pas certains objets contenus dans ce schéma.

## Paramètres

*nom*
: Le nom du schéma.

`CASCADE`
: Supprime automatiquement les objets (tables, fonctions, etc.) contenus dans le schéma.

`RESTRICT`
: Refuse de supprimer le schéma s'il contient des objets. Ceci est la valeur par défaut.

## Exemples

Pour supprimer le schéma `mestrucs` de la base de données avec tout ce qu'il contient :

```
DROP SCHEMA mestrucs CASCADE;
```

## Compatibilité

`DROP SCHEMA` est totalement compatible avec le standard SQL, si ce n'est que le standard n'autorise pas la suppression de plusieurs schémas par commande.

# Voir aussi

*ALTER SCHEMA*, *CREATE SCHEMA*

# DROP SEQUENCE

## Nom

DROP SEQUENCE −− supprime une séquence

## Synopsis

```
DROP SEQUENCE nom [, ...] [ CASCADE | RESTRICT ]
```

## Description

`DROP SEQUENCE` supprime les générateurs de nombres séquentiels.

## Paramètres

*nom*
: Le nom de la séquence (qualifié ou non du nom d'un schéma).

`CASCADE`
: Supprime automatiquement les objets qui dépendent de la séquence.

`RESTRICT`
: Refuse de supprimer la séquence si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Pour supprimer la séquence `serie` :

```
DROP SEQUENCE serie;
```

## Compatibilité

`DROP SEQUENCE` se conforme à SQL:2003, sauf que le standard n'autorise pas la suppression de plusieurs séquences par commande.

## Voir aussi

*CREATE SEQUENCE*

# DROP TABLE

## Nom

DROP TABLE −− supprime une table

## Synopsis

```
DROP TABLE nom [, ...] [ CASCADE | RESTRICT ]
```

## Description

`DROP TABLE` supprime des tables de la base de données. Seul son propriétaire peut détruire une table. Pour vider une table de ses lignes, sans détruire la table, utilisez `DELETE`.

`DROP TABLE` supprime tout index, règle, déclencheur et contrainte existant sur la table cible. Néanmoins, pour supprimer une table référencée par une vue ou par une contrainte de clé étrangère sur une autre table, `CASCADE` doit être ajouté. (`CASCADE` supprime complètement une vue dépendante mais dans le cas de la clé étrangère, il ne supprime que la contrainte, pas l'autre table.)

## Paramètres

*nom*
: Le nom de la table à supprimer (éventuellement qualifié par le nom d'un schéma).

`CASCADE`
: Supprime automatiquement les objets dépendants de la table (comme les vues).

`RESTRICT`
: Refuse de supprimer la table si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Pour supprimer deux tables, `films` et `distributeurs` :

```
DROP TABLE films, distributeurs;
```

## Compatibilité

Cette commande est conforme au standard SQL, si ce n'est que le standard ne permet pas de supprimer plusieurs tables avec une seule commande.

# Voir aussi

*ALTER TABLE*, *CREATE TABLE*

# DROP TABLESPACE

## Nom

DROP TABLESPACE −− supprime un espace logique

## Synopsis

```
DROP TABLESPACE nomespacelogique
```

## Description

`DROP TABLESPACE` supprime un espace logique du système.

Un espace logique peut seulement être supprimé par son propriétaire ou par un superutilisateur. L'espace logique doit être vide de tout objet de la base de données avant d'être supprimé. Il est possible que des objets d'autres bases de données résident encore dans l'espace logique même si aucun objet de la base de données courante n'utilise l'espace logique.

## Paramètres

*nomespacelogique*
: Le nom d'un espace logique.

## Exemples

Pour supprimer l'espace logique `mes_affaires` du système :

```
DROP TABLESPACE mes_affaires;
```

## Compatibilité

`DROP TABLESPACE` est une extension PostgreSQL.

## Voir aussi

*CREATE TABLESPACE*, *ALTER TABLESPACE*

# DROP TRIGGER

## Nom

DROP TRIGGER −− supprime un déclencheur

## Synopsis

```
DROP TRIGGER nom ON table [ CASCADE | RESTRICT ]
```

## Description

`DROP TRIGGER` supprime la définition d'un déclencheur existant. Pour exécuter cette commande, l'utilisateur courant doit être le propriétaire de la table pour laquelle le déclencheur est défini.

## Paramètres

*nom*
: Le nom du déclencheur à supprimer.

*table*
: Le nom de la table (pouvant être qualifié par le nom d'un schéma) sur laquelle le déclencheur est défini.

`CASCADE`
: Supprime automatiquement les objets qui dépendent du déclencheur.

`RESTRICT`
: Refuse de supprimer le déclencheur si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Détruit le déclencheur `si_dist_existe` de la table `films` :

```
DROP TRIGGER si_dist_existe ON films;
```

## Compatibilité

L'instruction `DROP TRIGGER` de PostgreSQL est incompatible avec le standard SQL. Dans le standard SQL, les noms de déclencheurs ne se définissent pas par rapport tables, donc la commande est simplement `DROP TRIGGER` *nom*.

# Voir aussi

*CREATE TRIGGER*

# DROP TYPE

## Nom

DROP TYPE −− supprime un type de données

## Synopsis

```
DROP TYPE nom [, ...] [ CASCADE | RESTRICT ]
```

## Description

`DROP TYPE` supprime tout type de données défini par l'utilisateur. Seul le propriétaire d'un type peut le supprimer.

## Paramètres

*nom*
: Le nom du type de données (peut−être qualifié d'un nom de schéma) à supprimer.

`CASCADE`
: Supprime automatiquement les objets dépendants du type (tels que les colonnes de table, les fonctions, les opérateurs).

`RESTRICT`
: Refuse de supprimer le type si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Pour supprimer le type de données `boite` :

```
DROP TYPE boite;
```

## Compatibilité

Cette commande est similaire à la commande correspondante du standard SQL mais notez que la commande `CREATE TYPE` et les mécanismes d'extension du type de données de PostgreSQL diffèrent du standard SQL.

## Voir aussi

*CREATE TYPE*, *ALTER TYPE*

# DROP USER

## Nom

DROP USER −− supprime un compte utilisateur de bases de données

## Synopsis

```
DROP USER nom
```

## Description

`DROP USER` supprime l'utilisateur spécifié. Elle ne supprime pas les tables, vues ou autres objets possédés par l'utilisateur. Si l'utilisateur possède une base de données, une erreur est levée.

## Paramètres

*nom*
: Le nom de l'utilisateur à supprimer.

## Notes

PostgreSQL inclut un programme ***dropuser*** qui dispose des mêmes fonctionnalités que cette commande (en fait, il appelle cette commande) mais peut être lancé depuis la ligne de commandes système.

Pour supprimer un utilisateur possédant une base de données, commencez par supprimer la base de données ou modifiez son propriétaire.

Il n'est pas conseillé de supprimer un utilisateur qui soit possède des objets de la base soit à des droits sur ces objets. Actuellement, ceci est vrai principalement pour les propriétaires de bases de données mais il est probable que les prochaines versions de PostgreSQL vérifieront d'autres cas.

## Exemples

Pour supprimer un compte utilisateur :

```
DROP USER jonathan;
```

# Compatibilité

L'instruction `DROP USER` est une extension PostgreSQL. Le standard SQL laisse la définition des utilisateurs à l'implémentation.

# Voir aussi

*ALTER USER*, *CREATE USER*

# DROP VIEW

## Nom

DROP VIEW −− supprime une vue

## Synopsis

```
DROP VIEW nom [, ...] [ CASCADE | RESTRICT ]
```

## Description

`DROP VIEW` supprime une vue existante. Pour exécuter cette commande, vous devez être le propriétaire de la vue.

## Paramètres

*nom*
: Le nom de la vue à supprimer (qualifié ou non du nom du schéma).

`CASCADE`
: Supprime automatiquement les objets qui dépendent de la vue (comme par exemple d'autres vues).

`RESTRICT`
: Refuse de supprimer la vue si un objet en dépend. Ceci est la valeur par défaut.

## Exemples

Cette commande supprime la vue appelée `genre` :

```
DROP VIEW genre;
```

## Compatibilité

Cette commande est conforme au standard SQL à l'exception du fait que le standard n'autorise que la suppression d'une seule vue par commande.

## Voir aussi

*CREATE VIEW*

# END

## Nom

END −− valide la transaction en cours

## Synopsis

```
END [ WORK | TRANSACTION ]
```

## Description

`END` valide la transaction en cours. Toutes les modifications réalisées lors de la transaction deviennent visibles pour les autres utilisateurs et il est garanti que les données ne seront pas perdues si un arrêt brutal survient. Cette commande est une extension PostgreSQL équivalente à *COMMIT*.

## Paramètres

`WORK`
`TRANSACTION`
: Mots clés optionnels. Ils n'ont pas d'effet.

## Notes

Utilisez *ROLLBACK* pour annuler une transaction.

Lancer `END` à l'extérieur d'une transaction ne cause aucun mal mais provoque un message d'avertissement.

## Exemples

Pour valider la transaction en cours et rendre toutes les modifications permanentes :

```
END;
```

## Compatibilité

`END` est une extension PostgreSQL fournissant une fonctionnalité équivalente à *COMMIT*, qui est spécifié dans le standard SQL.

# Voir aussi

*BEGIN*, *COMMIT*, *ROLLBACK*

# EXECUTE

## Nom

EXECUTE −− exécute une instruction préparée

## Synopsis

```
EXECUTE nom_plan [ (paramètre [, ...] ) ]
```

## Description

`EXECUTE` est utilisé pour exécuter une instruction préparée précédemment. Comme les instructions préparées existent seulement pour la durée d'une session, l'instruction préparée doit avoir été créée par une instruction `PREPARE` exécutée plus tôt dans la session en cours.

Si l'instruction `PREPARE` qui a créé l'instruction a spécifié des paramètres, un ensemble compatible de paramètres doit être passé à l'instruction `EXECUTE`, sinon une erreur est levée. Notez que, contrairement aux fonctions, les instructions préparées ne sont pas surchargées suivant leur type ou le nombre de leurs paramètres ; le nom d'une instruction préparée doit être unique à l'intérieur d'une session.

Pour plus d'informations sur la création et sur l'utilisation des instructions préparées, voir ***PREPARE***.

## Paramètres

*`nom_plan`*
: Le nom de l'instruction préparée à exécuter.

*`paramètre`*
: La valeur réelle du paramètre d'une instruction préparée. Ce doit être une expression ramenant une valeur dont le type est compatible avec le type de données spécifié pour la position de ce paramètre dans la commande `PREPARE` qui a créé l'instruction préparée.

## Sorties

La sortie renvoyée par la commande `EXECUTE` est celle de l'instruction préparée, et non pas celle d'`EXECUTE`.

## Exemples

Des exemples sont donnés dans la section ***Exemples*** de la documentation de ***PREPARE***.

# Compatibilité

Le standard SQL inclut une instruction `EXECUTE` mais est seulement utilisé dans le SQL embarqué. Cette version de l'instruction `EXECUTE` utilise aussi une syntaxe un peu différente.

# Voir aussi

*DEALLOCATE*, *PREPARE*

# EXPLAIN

## Nom

EXPLAIN −− affiche le plan d'exécution d'une instruction

## Synopsis

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] instruction
```

## Description

Cette commande affiche le plan d'exécution que l'optimiseur PostgreSQL génère pour l'instruction fournie. Le plan d'exécution affiche comment la (les) table(s) référencée(s) par l'instruction seront parcourue(s) — parcours séquentiel, parcours d'index, etc. — et si plusieurs tables sont référencées, quels algorithmes de jointure seront utilisés pour amener les lignes requises à partir de chaque table en entrée.

La partie la plus critique de l'affichage est le coût d'exécution estimé de l'instruction, qui est l'impression que l'optimiseur a sur la durée que prendra l'exécution de l'instruction (mesuré en unité de récupération de pages disque). En fait, deux nombres sont affichés : le temps d'exécution avant que la première ligne ne soit renvoyée et le temps total pour renvoyer toutes les lignes. Pour la plupart des requêtes, le temps total est ce qui importe mais dans des contextes tels qu'une sous−requête dans `EXISTS`, l'optimiseur choisit le plus petit temps de lancement plutôt que le plus petit temps total (car, de toute façon, l'exécuteur s'arrête après avoir récupéré une ligne). De même, si vous limitez le nombre de lignes à renvoyer avec une clause `LIMIT`, l'optimiseur fait une interpolation appropriée pour estimer le plan le moins cher.

L'option `ANALYZE` fait que l'instruction est réellement exécutée, pas uniquement planifiée. Le temps total passé sur chaque nœud du plan (en millisecondes) et le nombre total de lignes renvoyées sont ajoutés à l'affichage. Ceci est utile pour voir si les estimations du planificateur sont proches de la réalité.

> **Important :** Gardez en tête que l'instruction est réellement exécutée quand `ANALYZE` est utilisé. Bien que `EXPLAIN` annule tout affichage qu'un `SELECT` renverrait, les autres effets immédiats de l'instruction ont lieu. Si vous souhaitez utiliser `EXPLAIN ANALYZE` sur une instruction `INSERT`, `UPDATE`, `DELETE` ou `EXECUTE` sans que la commande n'affecte vos données, utilisez cette approche :
>
> ```
> BEGIN;
> EXPLAIN ANALYZE ...;
> ROLLBACK;
> ```

## Paramètres

`ANALYZE`
: Exécute la commande et affiche les temps d'exécution réels.

`VERBOSE`

Affiche la représentation interne complète du plan plutôt qu'un simple résumé. Utiliser cette option est seulement utile pour un débogage. Le formatage de la sortie de VERBOSE peut être modifié avec le paramètre de configuration explain_pretty_print.

*instruction*

Toute instruction SELECT, INSERT, UPDATE, DELETE, EXECUTE ou DECLARE dont vous souhaitez voir le plan d'exécution.

# Notes

La documentation sur l'utilisation que l'optimiseur fait des informations de coût est assez limitée dans PostgreSQL. Référez−vous à Section 13.1 pour plus d'informations.

Pour permettre au planificateur de requêtes de PostgreSQL de prendre des décisions raisonnables lors de l'optimisation de requêtes, l'instruction ANALYZE doit être exécutée pour enregistrer les statistiques sur la distribution des données à l'intérieur de la table. Si vous n'avez pas fait ceci (ou si la distribution statistique des données dans la table a changé significativement depuis la dernière exécution d'ANALYZE), les coûts estimés ne sont pas conformes aux réelles propriétés de la requête et, par conséquence, un plan de requête médiocre pourrait être choisi.

Avant PostgreSQL 7.3, le plan était émis sous la forme d'un message NOTICE. Maintenant, il apparaît comme le résultat d'une requête (formaté comme une table composée d'une seule colonne de type texte).

# Exemples

Pour afficher le plan d'une simple requête sur une table avec une seule colonne de type integer et 10000 lignes :

```
EXPLAIN SELECT * FROM foo;

                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

S'il existe un index et que nous utilisons une requête avec un condition WHERE indexable, EXPLAIN pourrait afficher un plan différent :

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

Et voici un exemple d'un plan de requête pour une requête utilisant une fonction d'agrégat :

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                            QUERY PLAN
-------------------------------------------------------------------
```

```
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

Voici un exemple d'utilisation de `EXPLAIN EXECUTE` pour afficher le plan d'exécution d'une requête préparée :

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=39.53..39.53 rows=1 width=8) (actual time=0.661..0.672 rows=7 loops=1)
   ->  Index Scan using test_pkey on test  (cost=0.00..32.97 rows=1311 width=8) (actual time=0.05
         Index Cond: ((id > $1) AND (id < $2))
 Total runtime: 0.851 ms
(4 rows)
```

Bien sûr, les nombres réellement affichés dépendent du contenu réel des tables impliquées. Notez aussi que les nombres, et même la stratégie de la requête sélectionnée, pourrait varier entre les versions de PostgreSQL à cause des améliorations du planificateur. De plus, la commande `ANALYZE` utilise une distribution aléatoire pour estimer les statistiques des données ; du coup, il est possible que les estimations de coût changent après un lancement d'`ANALYZE`, même si la distribution réelle des données n'a pas changé dans la table.

# Compatibilité

Il n'existe pas d'instruction `EXPLAIN` définie dans le standard SQL.

# Voir aussi

*ANALYZE*

# FETCH

## Nom

FETCH −− récupère des lignes d'une requête en utilisant un curseur

## Synopsis

```
FETCH [ direction { FROM | IN } ]
nomcurseur

où direction peut être vide ou
faire partie de :

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE nombre
    RELATIVE nombre
    nombre
    ALL
    FORWARD
    FORWARD nombre
    FORWARD ALL
    BACKWARD
    BACKWARD nombre
    BACKWARD ALL
```

## Description

`FETCH` récupère des lignes en utilisant un curseur déjà ouvert.

Un curseur a une position associée qui est utilisée par `FETCH`. La position du curseur peut être avant la première ligne du résultat d'une requête, une ligne particulière du résultat ou après la dernière ligne du résultat. Une fois créé, un curseur est positionné sur la ligne la plus récemment récupérée. Si `FETCH` arrive à la fin des lignes disponibles, alors le curseur est positionné après la dernière ligne ou avant la première ligne au cas où la récupération se fait en marche arrière. `FETCH ALL` ou `FETCH BACKWARD ALL` laisse toujours le curseur positionné après la dernière ligne ou avant la première ligne.

Les formes `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE`, `RELATIVE` récupèrent une seule ligne après avoir déplacé le curseur de façon appropriée. Si cette ligne n'existe pas, un résultat vide est renvoyé et le curseur est positionné à avant la première ligne ou après la dernière ligne suivant le sens de la progression.

Les formes utilisant `FORWARD` et `BACKWARD` récupèrent le nombre de lignes indiqué en se déplaçant en avant ou en arrière, laissant le curseur positionné sur la dernière ligne renvoyée (ou après/avant toutes les lignes si *`nombre`* dépasse le nombre de lignes disponibles).

`RELATIVE 0`, `FORWARD 0` et `BACKWARD 0` demandent la récupération de la ligne actuelle sans déplacer le curseur, c'est−à−dire une nouvelle récupération de la ligne récemment récupérée. La commande réussit sauf

si le curseur est positionné avant la première ligne ou après la dernière ligne ; auquel cas, aucune ligne n'est renvoyée.

# Paramètres

*direction*

*direction* définit la direction de la récupération et le nombre de lignes à récupérer. Elle peut prendre les valeurs suivantes :

NEXT
: Récupère la prochaine ligne. Ceci est la valeur par défaut si *direction* est omis.

PRIOR
: Récupère la ligne précédente.

FIRST
: Récupère la première ligne de la requête (identique à ABSOLUTE 1).

LAST
: Récupère la dernière ligne de la requête (identique à ABSOLUTE -1).

ABSOLUTE *nombre*
: Récupère la *nombre*–ième ligne de la requête ou la abs(*nombre*) –ième ligne à partir de la fin si *nombre* est négatif. La position avant la première ligne ou après la dernière si *nombre* est en dehors des bornes ; en particulier, ABSOLUTE 0 se positionne avant la première ligne.

RELATIVE *nombre*
: Récupère la *nombre*–ième ligne ou la abs(*nombre*) –ième ligne précédente si *nombre* est négatif. RELATIVE 0 récupère de nouveau la ligne actuelle si elle existe.

*nombre*
: Récupère les *nombre* lignes suivantes (identique à FORWARD *nombre*).

ALL
: Récupère toutes les lignes restantes (identique à FORWARD ALL).

FORWARD
: Récupère la ligne suivante (identique à NEXT).

FORWARD *nombre*
: Récupère les *nombre* lignes suivantes. FORWARD 0 récupère de nouveau la ligne actuelle.

FORWARD ALL
: Récupère toutes les lignes restantes.

BACKWARD
: Récupère la ligne précédente (identique à PRIOR).

BACKWARD *nombre*
: Récupère les *nombre* lignes précédentes (parcours inverse). BACKWARD 0 récupère de nouveau la ligne actuelle.

BACKWARD ALL
: Récupère toutes les lignes précédentes (parcours inverse).

*nombre*

*nombre* est une constante de type entier pouvant être signée, déterminant l'emplacement ou le nombre de lignes à récupérer. Pour les cas FORWARD et BACKWARD, spécifier une valeur négative pour *nombre* est équivalent à modifier le sens de FORWARD et BACKWARD.

*nomcurseur*

Le nom d'un curseur ouvert.

# Sorties

En cas de succès, une commande FETCH renvoie une balise de commande de la forme

```
FETCH nombre
```

Le *nombre* est le nombre de lignes récupérées (zéro étant possible). Notez que dans psql, la balise de commande n'est pas réellement affichée car psql affiche à la place les lignes récupérées.

# Notes

Le curseur doit être déclaré avec l'option SCROLL si vous avez l'intention d'utiliser une variante de FETCH autre que FETCH NEXT ou FETCH FORWARD avec un nombre positif. Pour les requêtes simples, PostgreSQL autorise les parcours inverses à partir de curseurs non déclarés avec SCROLL, mais ce comportement n'est pas sûr. Si le curseur est déclaré avec NO SCROLL, aucun parcours inverse n'est autorisé.

Les récupérations ABSOLUTE ne sont pas plus rapides que de naviguer vers la ligne désirée avec un déplacement relatif : de toute façon, l'implémentation sous−jacente doit parcourir toutes les lignes intermédiaires. Les récupérations absolues négatives sont même pires : la requête doit être lue jusqu'à la fin pour trouver la dernière ligne relue en sens inverse. Néanmoins, remonter vers le début de la requête (comme avec FETCH ABSOLUTE 0) est rapide.

Mettre à jour des données via un curseur n'est pas supporté actuellement par PostgreSQL.

*DECLARE* est utilisé pour définir un curseur. Utilisez *MOVE* pour modifier la position du curseur sans récupérer les données.

# Exemples

L'exemple suivant parcourt une table en utilisant un curseur.

```
BEGIN WORK;

-- Initialise un curseur :
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- Récupère les 5 premières lignes du curseur liahona :
FETCH FORWARD 5 FROM liahona;

 code  |          titre           | did | date_prod  |   genre   |  longueur
-------+--------------------------+-----+------------+-----------+-----------
 BL101 | The Third Man            | 101 | 1949-12-23 | Drama     | 01:44
 BL102 | The African Queen        | 101 | 1951-08-11 | Romantic  | 01:43
 JL201 | Une Femme est une Femme  | 102 | 1961-03-12 | Romantic  | 01:25
 P_301 | Vertigo                  | 103 | 1958-11-14 | Action    | 02:08
 P_302 | Becket                   | 103 | 1964-02-03 | Drama     | 02:28

-- Récupère la ligne précédente :
FETCH PRIOR FROM liahona;

 code  |  titre  | did | date_prod  |  genre |  longueur
```

```
-------+---------+-----+------------+--------+-----------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- Ferme le curseur et termine la transaction:
CLOSE liahona;
COMMIT WORK;
```

# Compatibilité

Le standard SQL définit `FETCH` comme étant à utiliser uniquement dans le SQL embarqué. La variante de `FETCH` décrite ici renvoie les données comme s'il s'agissait du résultat d'un `SELECT` plutôt que de le placer dans des variables hôtes. A part ce point, `FETCH` est totalement compatible avec le standard SQL.

Les formes de `FETCH` impliquant `FORWARD` et `BACKWARD`, ainsi que les formes `FETCH` *`nombre`* et `FETCH ALL`, dans lesquelles `FORWARD` est implicite, sont des extensions de PostgreSQL.

Le standard SQL autorise seulement `FROM` précédant le nom du curseur ; La possibilité d'utiliser `IN` est une extension.

# Voir aussi

*CLOSE*, *DECLARE*, *MOVE*

# GRANT

## Nom

GRANT –– définit les droits d'accès

## Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] nomtable [, ...]
    TO { nomutilisateur | GROUP
nomgroupe | PUBLIC } [, ...] [ WITH
GRANT OPTION ]

GRANT { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE nombase [, ...]
    TO { nomutilisateur | GROUP
nomgroupe | PUBLIC } [, ...] [ WITH
GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION nomfonction
([type, ...]) [, ...]
    TO { nomutilisateur | GROUP
nomgroupe | PUBLIC } [, ...] [ WITH
GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE nomlangage [, ...]
    TO { nomutilisateur | GROUP
nomgroupe | PUBLIC } [, ...] [ WITH
GRANT OPTION ]

GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA nomschéma [, ...]
    TO { nomutilisateur | GROUP
nomgroupe | PUBLIC } [, ...] [ WITH
GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE nomespacelogique [, ...]
    TO { nomutilisateur | GROUP
nomgroupe | PUBLIC } [, ...] [ WITH
GRANT OPTION ]
```

## Description

La commande `GRANT` donne des droits spécifiques sur un objet (table, vue, séquence, base de données, fonction, langage de procédure, schéma ou espace logique) à un ou plusieurs utilisateurs ou groupes d'utilisateurs. Ces droits sont ajoutés à ceux déjà obtenus.

Le mot clé PUBLIC indique que les droits sont donnés à tous les utilisateurs, y compris ceux qui seront créés ultérieurement. PUBLIC pourrait être vu comme un groupe implicitement défini qui inclut en permanence tous les utilisateurs. Un utilisateur particulier dispose de la somme des droits qui lui sont acquis, des droits de tout groupe dont il est membre et des droits donnés à PUBLIC.

Si WITH GRANT OPTION est spécifié, celui qui reçoit ce droit peut le donner à d'autres. Sans option GRANT, l'utilisateur ne peut pas le faire. Actuellement, les options de droits peuvent seulement être données à des utilisateurs individuels, pas à des groupes ou à PUBLIC.

Il n'y a pas besoin d'accorder des droits au propriétaire d'un objet (habituellement l'utilisateur qui l'a créé) car le propriétaire a tous les droits par défaut. (Le propriétaire pourrait, néanmoins, choisir de révoquer certains de ses propres droits.) Le droit de supprimer un objet ou de modifier sa définition n'est pas décrit par un droit donnable ; il est inhérent au propriétaire et ne peut être ni donné ni enlevé. Le propriétaire a aussi toutes les options GRANT sur cet objet.

Suivant le type de l'objet, PUBLIC peut avoir certains droits initiaux par défaut. La valeur par défaut est aucun accès public aux tables, schémas et espaces logiques ; droit de création de table TEMP pour les bases de données ; droit EXECUTE pour les fonctions ; et droit USAGE pour les langages. Le propriétaire de l'objet peut bien sûr choisir de révoquer ces droits. (Pour un maximum de sécurité, lancez REVOKE dans la même transaction que la création de l'objet ; ainsi, il n'y aura pas de laps temps pendant lequel un autre utilisateur pourrait utiliser l'objet.)

Les droits possibles sont :

SELECT
: Autorise *SELECT* parmi toutes les colonnes de la table, vue ou séquence spécifiée. Autorise aussi l'utilisation de *COPY* TO. Pour les séquences, ce droit permet aussi l'utilisation de la fonction currval.

INSERT
: Autorise *INSERT* d'une nouvelle ligne dans la table spécifiée. Autorise aussi *COPY* FROM.

UPDATE
: Autorise *UPDATE* de toute colonne de la table spécifiée. SELECT ... FOR UPDATE requiert aussi ce droit (en plus du droit SELECT). Pour les séquences, ce droit autorise l'utilisation des fonctions nextval et setval.

DELETE
: Autorise *DELETE* d'une ligne pour la table spécifiée.

RULE
: Autorise la création d'une règle sur la table/vue. (Voir l'instruction *CREATE RULE*.)

REFERENCES
: Pour créer une contrainte de clé étrangère, il est nécessaire d'avoir ce droit sur la table de référence et sur celle référencée.

TRIGGER
: Autorise la création d'un déclencheur sur la table spécifiée. (Voir l'instruction *CREATE TRIGGER*.)

CREATE
: Pour les bases de données, autorise la création de nouveaux schémas à l'intérieur de la base de données.

  Pour les schémas, autorise la création de nouveaux objets dans le schéma. Pour renommer un objet existant, vous devez être le propriétaire de l'objet *et* avoir ce droit pour le schéma qui le contient.

Pour les espaces logiques (tablespaces), autorise la création de tables et d'index à l'intérieur de l'espace logique et autorise la création de bases de données ayant cet espace logique par défaut. (Notez que révoquer ce privilège ne modifie pas l'emplacement des objets existants.)

TEMPORARY
TEMP
: Autorise la création de tables temporaires lors de l'utilisation de la base de données.

EXECUTE
: Autorise l'utilisation de la fonction spécifiée et l'utilisation de tout opérateur qui est implémenté avec cette fonction. C'est le seul type de droit qui est applicable aux fonctions. (Cette syntaxe fonctionne aussi pour les fonctions d'agrégat.)

USAGE
: Pour les langages de procédures, autorise l'utilisation du langage spécifié pour la création de fonctions dans ce langage. Ceci est le seul type de droit applicable aux langages de procédures.

  Pour les schémas, autorise l'accès aux objets contenus dans le schéma spécifié (en supposant que les besoins de droits des objets propres sont aussi respectés). Essentiellement, ceci permet à celui ayant reçu le droit de << rechercher >> les objets contenus dans ce schéma.

ALL PRIVILEGES
: Donne tous les droits disponibles en une fois. Le mot clé `PRIVILEGES` est optionnel dans PostgreSQL bien qu'il soit requis par le SQL strict.

Les droits requis par d'autres commandes sont listés sur la page référencée de la commande respective.

# Notes

La commande *REVOKE* est utilisée pour retirer les droits d'accès.

Quand un utilisateur, non propriétaire d'un objet, essaie de donner les droits `GRANT` de cet objet, la commande échoue si l'utilisateur n'a aucun droit sur l'objet. Si l'utilisateur dispose de droits, la commande continue mais ne donne que les droits pour lesquels l'utilisateur dispose de l'option GRANT. Les formes `GRANT ALL PRIVILEGES` affichent un message d'avertissement si aucune option GRANT n'est détenue. (En principe, ces instructions s'appliquent aussi au propriétaire de l'objet mais, comme ce dernier est toujours traité comme détenant les options GRANT, le cas n'arrive jamais.)

Il faut noter que les superutilisateurs de la base de données peuvent accéder à tous les objets sans paramétrage de droits sur les objets. Ceci est comparable aux droits de `root` sur un système Unix. Comme avec `root`, il est déconseillé d'opérer en tant que superutilisateur sauf lorsque cela est absolument nécessaire.

Si un superutilisateur choisit de lancer une commande `GRANT` ou `REVOKE`, la commande est exécutée comme si elle avait été lancée par le propriétaire de l'objet affecté. En particulier, les droits donnés via une telle commande semblent avoir été donnés par le propriétaire de l'objet.

Actuellement, PostgreSQL ne permet pas d'attribuer ou de révoquer des droits sur des colonnes individuelles d'une table. Un contournement possible est de créer une vue ayant les colonnes désirées et donner les droits GRANT sur cette vue.

Utilisez la commande `\z` de psql pour obtenir des informations sur les droits existants, par exemple :

```
=> \z matable
```

```
                        Access privileges for database "lusitania"
 Schema |  Name   | Type  |                     Access privileges
--------+---------+-------+-----------------------------------------------------
-------
 public | matable | table | {miriam=arwdRxt/miriam,=r/miriam,"group
todos=arw/miriam"}
(1 row)
```

Les entrées affichées par `\z` sont interprétées ainsi :

```
            =xxxx -- droits donnés à PUBLIC
          uname=xxxx -- droits donnés à un utilisateur
   group gname=xxxx -- droits donnés à un groupe

                r -- SELECT ("lecture")
                w -- UPDATE ("écriture")
                a -- INSERT ("ajout")
                d -- DELETE
                R -- RULE
                x -- REFERENCES
                t -- TRIGGER
                X -- EXECUTE
                U -- USAGE
                C -- CREATE
                T -- TEMPORARY
           arwdRxt -- ALL PRIVILEGES (pour les tables)
                * -- option de droit pour le droit précédant

              /yyyy -- utilisateur qui a donné ce droit
```

L'affichage de l'exemple ci–dessus pourrait être vue par l'utilisatrice `miriam` après la création de la table `matable` et le lancement de

```
GRANT SELECT ON matable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON matable TO GROUP todos;
```

Si la colonne << Access privileges >> est vide pour un objet donné, cela signifie que l'objet a les droits par défaut (c'est–à–dire que la colonne des droits est NULL). Les droits par défaut incluent toujours les droits pour le propriétaire et peuvent inclure quelques droits pour `PUBLIC` suivant le type d'objet comme expliqué ci–dessus. Le premier `GRANT` ou `REVOKE` sur un objet instancie les droits par défaut (produisant, par exemple, `{=,miriam=arwdRxt/miriam}`) puis les modifie par la requête spécifiée.

Notez que les options GRANT implicites du propriétaire ne sont pas indiquées dans l'affichage des droits d'accès. Une `*` apparaît seulement lorsque des options GRANT ont été explicitement données à quelqu'un.

# Exemples

Donner le droit d'insertion à tous les utilisateurs sur la table `films` :

```
GRANT INSERT ON films TO PUBLIC;
```

Donner tous les droits à l'utilisateur `manuel` pour la vue `genres` :

```
GRANT ALL PRIVILEGES ON genres TO manuel;
```

Notez que, bien que la commande ci−dessus donne bien tous les droits si elle est exécutée par un superutilisateur ou par le propriétaire de `genres`, si elle est exécutée par quelqu'un d'autre, elle ne donne les droits qu'en accord avec ce que possède cet utilisateur.

# Compatibilité

Suivant le standard SQL, le mot clé `PRIVILEGES` dans `ALL PRIVILEGES` est requis. Le standard SQL ne supporte pas l'initialisation des droits de plus d'un objet par commande.

PostgreSQL autorise un propriétaire d'objet à révoquer ses propres droits ordinaires : par exemple, le propriétaire d'un objet peut le placer en lecture seule pour lui−même en révoquant ses propres droits INSERT, UPDATE et DELETE. Ceci n'est pas possible d'après le standard SQL. La raison en est que PostgreSQL traite les droits du propriétaire comme ayant été donnés par le propriétaire ; du coup, il peut aussi les révoquer. Dans le standard SQL, les droits du propriétaires sont donnés par une entité << _SYSTEM >>. Sans être << _SYSTEM >>, le propriétaire ne peut pas révoquer ces droits.

Le standard SQL autorise l'initialisation de droits pour des colonnes spécifiques à l'intérieur d'une table :

```
GRANT droits
    ON table [ ( colonne [, ...] ) ] [, ...]
    TO { PUBLIC | nomutilisateur [,
...] } [ WITH GRANT OPTION ]
```

Le standard SQL fournit un droit `USAGE` sur d'autres types d'objets : ensembles de caractères, collations, traductions, domaines.

Le droit `RULE` et les droits sur les bases de données, les espaces logiques, langages, schémas et séquences sont des extensions de PostgreSQL.

# Voir aussi

*REVOKE*

# INSERT

## Nom

INSERT −− insère de nouvelles lignes dans une table

## Synopsis

```
INSERT INTO table [ ( colonne [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) | requête }
```

## Description

`INSERT` insère de nouvelles lignes dans une table. Il est possible d'insérer une ligne spécifiée par des valeurs ou plusieurs lignes comme résultat d'une requête.

Les noms des colonnes cibles peuvent être donnés dans un ordre quelconque. Si aucune liste de noms de colonnes n'est donnée, la valeur par défaut est toutes les colonnes de la table dans leur ordre de déclaration ; ou les *N* premiers noms de colonnes si seulement *N* valeurs de colonnes sont fournies dans la clause `VALUES` ou dans la *requête*. Les valeurs fournies par la clause `VALUES` ou par la *requête* sont associées à la liste explicite ou implicite des colonnes de gauche à droite.

Chaque colonne absente dans la liste de colonnes explicite ou implicite est remplie avec sa valeur par défaut déclarée s'il y en a une ou avec NULL s'il n'en a pas.

Si l'expression pour chaque colonne n'est pas du bon type de données, une conversion automatique de type est tentée.

Vous devez avoir le droit `INSERT` sur une table pour pouvoir y insérer des lignes. Si vous utilisez la clause *requête* pour insérer des lignes à partir d'une requête, vous avez aussi besoin d'avoir un droit `SELECT` sur toute table utilisée dans la requête.

## Paramètres

*table*
: Le nom d'une table existante (pouvant être qualifié du nom du schéma).

*colonne*
: Le nom d'une colonne dans *table*. Le nom de la colonne peut être qualifié avec un nom de sous−champ ou un indice de tableau, si nécessaire. (N'insérer que certains champs d'une colonne composite laisse les autres champs à NULL.)

`DEFAULT VALUES`
: Toutes les colonnes auront leurs valeurs par défaut.

*expression*
: Une expression ou une valeur à affecter à la *colonne* correspondante.

`DEFAULT`

La *colonne* correspondante sera remplie avec sa valeur par défaut.

*requête*

Une requête (instruction SELECT) fournissant les lignes à insérer. Référez–vous à l'instruction SELECT pour une description de la syntaxe.

# Sorties

En cas de succès, une commande INSERT renvoie un code de la forme

```
INSERT oid
 nombre
```

Le *nombre* correspond au nombre de lignes insérées. Si *nombre* vaut exactement un et que la table cible contient des OID, alors *oid* est l'OID affecté à la ligne insérée. Sinon, *oid* vaut zéro.

# Exemples

Insérer une seule ligne dans la table films :

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comédie', '82 minutes');
```

Dans cet exemple, la colonne longueur est omise et, du coup, elle prend la valeur par défaut :

```
INSERT INTO films (code, titre, did, date_prod, genre)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drame');
```

Cet exemple utilise la clause DEFAULT pour les colonnes date plutôt que de spécifier une valeur :

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comédie', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drame');
```

Pour insérer une ligne consistant uniquement en des valeurs par défaut :

```
INSERT INTO films DEFAULT VALUES;
```

Cet exemple insère quelques lignes dans la table films à partir de la table tmp_films avec le même emplacement des colonnes :

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

Cet exemple insére dans des colonnes de type tableau :

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
-- (these commands create the same board)
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1,'{{"","",""},{"","",""},{"","",""}}');
INSERT INTO tictactoe (game, board)
    VALUES (2,'{{,,},{,,},{,,}}');
```

# Compatibilité

`INSERT` est totalement compatible avec le standard SQL. Le cas où une liste de noms de colonnes est omise, mais où toutes les colonnes ne sont pas remplies à partir des valeurs de la clause `VALUES` ou à partir de la *`requête`*, est interdit par le standard.

Les limitations possibles de la clause *`requête`* sont documentées sous ***SELECT***.

# LISTEN

## Nom

LISTEN −− écoute une notification

## Synopsis

```
LISTEN nom
```

## Description

`LISTEN` enregistre la session courante comme étant en attente d'une condition de notification *nom*. Si la session courante est déjà enregistrée comme étant en écoute de cette condition de notification, rien n'est fait.

Quand la commande `NOTIFY` *nom* est appelée, soit par cette session soit par une autre connectée à la même base de données, toutes les sessions en cours d'écoute de cette condition de notification sont notifiées et chacune pourra notifier en retour son application cliente connectée. Voir la discussion de `NOTIFY` pour plus d'informations.

Une condition de notification donnée peut être effacée pour une session avec la commande `UNLISTEN`. Les enregistrements d'écoute d'une session sont automatiquement effacés lorsque la session se termine.

La méthode qu'une application client doit utiliser pour détecter des événements de notification dépend de l'interface de programmation PostgreSQL qu'elle utilise. Avec la bibliothèque libpq, l'application exécute `LISTEN` comme une commande SQL ordinaire, puis doit appeler périodiquement la fonction `PQnotifies` pour savoir quels événements de notification ont été reçus. Les autres interfaces, comme libpgtcl, fournissent des méthodes de plus haut niveau pour gérer les événements de notification ; en fait, avec libpgtcl, le développeur de l'application ne devrait même pas lancer `LISTEN` ou `UNLISTEN` directement. Voir la documentation de l'interface que vous utilisez pour plus de détails.

*NOTIFY* contient une discussion plus détaillée sur l'utilisation de `LISTEN` et `NOTIFY`.

## Paramètres

*nom*
: Nom d'une condition de notification (tout identifiant).

## Exemples

Configure et exécute une séquence listen/notify à partir de psql :

```
LISTEN virtual;
NOTIFY virtual;
```

```
Asynchronous notification "virtual" received from server process with PID 8448.
```

## Compatibilité

Il n'existe pas d'instruction `LISTEN` dans le standard SQL.

## Voir aussi

*NOTIFY*, *UNLISTEN*

# LOAD

## Nom

LOAD −− charge ou décharge une bibliothèque partagée

## Synopsis

```
LOAD 'fichier'
```

## Description

Cette commande charge une bibliothèque partagée dans l'espace d'adressage de PostgreSQL. Si le fichier a été chargé auparavant, il est tout d'abord déchargé. Cette commande est principalement utile pour décharger et recharger une bibliothèque partagée qui a été modifiée depuis que le serveur l'avait chargée la première fois. Pour utiliser une bibliothèque partagée, les fonctions doivent être déclarées en utilisant la commande *CREATE FUNCTION*.

Le nom du fichier est spécifié de la même façon que pour les noms de bibliothèques partagées dans *CREATE FUNCTION* ; en particulier, vous pourriez vous reposer sur un chemin de recherche et un ajout automatique de l'extension de la bibliothèque partagée, suivant les standards système. Voir Section 31.9 pour plus d'informations sur ce thème.

## Compatibilité

`LOAD` est une extension PostgreSQL.

## Voir aussi

*CREATE FUNCTION*

# LOCK

## Nom

LOCK −− verrouille une table

## Synopsis

```
LOCK [ TABLE ] nom [, ...] [ IN mode_verrou MODE ] [ NOWAIT ]

où mode_verrou fait partie
de :

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSI
```

## Description

`LOCK TABLE` obtient un verrou de niveau table, attendant si nécessaire que tous les verrous en conflit soient lâchés. Si `NOWAIT` est spécifié, `LOCK TABLE` n'attend pas l'acquisition du verrou désiré : s'il ne peut pas être obtenu immédiatement, la commande est annulée et une erreur est émise. Une fois obtenu, le verrou est gardé jusqu'à la fin de la transaction en cours. (Il n'y a pas de commande `UNLOCK TABLE` ; les verrous sont toujours abandonnés à la fin de la transaction.)

Lors de l'acquisition automatique de verrous pour les commandes qui référencent des tables, PostgreSQL utilise toujours le mode de verrou le moins restrictif possible. `LOCK TABLE` est fourni pour les cas où vous pourriez avoir besoin de verrous plus restrictifs. Par exemple, supposez qu'une application exécute une transaction au niveau d'isolation de lecture validé (Read Committed) pour s'assurer que les données de la table restent stables pendant la durée de la transaction. Pour réaliser ceci, vous pouvez obtenir un mode de verrou `SHARE` sur la table avant d'envoyer la requête. Ceci empêche toute modification concurrente des données et assure que les lectures de la table voient une vue stable des données validées parce que le mode de verrou `SHARE` est en conflit avec le verrou `ROW EXCLUSIVE` acquis par les modificateurs et votre instruction `LOCK TABLE` *nom* `IN SHARE MODE` attend jusqu'à ce que tous déteneurs concurrents de verrous en mode `ROW EXCLUSIVE` valident ou annulent. Du coup, une fois le verrou obtenu, il ne reste aucune écriture en attente ; de plus, aucune ne peut commencer tant que vous n'avez pas supprimé le verrou.

Pour obtenir un effet similaire lors de l'exécution d'une transaction au niveau d'isolation sérialisable, vous devez exécuter l'instruction `LOCK TABLE` avant d'exécuter toute instruction `SELECT` ou toute instruction de modification de données. La vue des données par une transaction sérialisable des données est gelée à la première instruction `SELECT` ou à la première instruction de modification des données. Un `LOCK TABLE` plus tard empêche toujours les écritures concurrentes — mais il n'assure pas que ce que la transaction lit correspond aux dernières données validées.

Si une transaction de cette sorte va modifier les données de la table, alors elle doit utiliser le mode de verrou `SHARE ROW EXCLUSIVE` au lieu du mode `SHARE`. Ceci nous assure que seule une transaction de ce type est en exécution à la fois. Sans cela, un verrou mortel est possible : deux transactions pourraient acquérir à la fois le mode `SHARE` et être ensuite incapable d'acquérir aussi le mode `ROW EXCLUSIVE` pour réellement effectuer leur mises à jour. (Notez que les propres verrous d'une transaction ne sont jamais en conflit, donc

une transaction peut acquérir le mode ROW EXCLUSIVE lorsqu'il tient le mode SHARE — mais pas si quelqu'un d'autre détient le mode SHARE.) Pour éviter les verrous bloquants, assurez−vous que toutes les transactions acquièrent des verrous sur les mêmes objets dans le même ordre, et si des modes multiples de verrous sont impliqués pour un seul objet, alors les transactions doivent toujours acquérir en premier le mode le plus restrictif.

Plus d'informations sur les modes de verrou et les stratégies de verrouillage sont disponibles dans Section 12.3.

# Paramètres

*nom*

Le nom d'une table existante à verrouiller (pouvant être qualifié du nom du schéma).

La commande LOCK a, b; est équivalente à LOCK a; LOCK b;. Les tables sont verrouillées une par une dans l'ordre spécifié dans la commande LOCK TABLE.

*modeverrou*

Le mode verrou spécifie avec quels verrous ce verrou entre en conflit. Les modes de verrous sont décrits dans Section 12.3.

Si aucun mode de verrou n'est spécifié, alors ACCESS EXCLUSIVE, le mode le plus restrictif, est utilisé.

NOWAIT

Spécifie que LOCK TABLE n'attend pas que les verrous conflictuels soient annulés : si le verrou spécifié ne peut être acquis immédiatement, sans attendre, la transaction est annulée.

# Notes

LOCK TABLE ... IN ACCESS SHARE MODE requiert les droits SELECT sur la table cible. Tous les autres formats de LOCK requièrent les droits UPDATE et/ou DELETE.

LOCK TABLE est utile seulement dans un bloc de transaction (paire BEGIN/COMMIT), car le verrou est supprimé aussitôt que la transaction se termine. Une commande LOCK apparaissant à l'extérieur de tout bloc de transaction forme une transaction contenue dans elle−même, donc le verrou est supprimé dès qu'il est obtenu.

LOCK TABLE s'occupe seulement des verrous au niveau table et du coup, les noms de mode impliquant ROW sont tous mal nommés. Ces noms de modes doivent généralement être compris comme indiquant l'intention de l'utilisateur d'acquérir des verrous de niveau ligne à l'intérieur de la table verrouillée. De plus, le mode ROW EXCLUSIVE est un verrou de table partageable. Gardez en tête que tous les modes de verrou ont des sémantiques identiques en ce qui concerne LOCK TABLE, différant seulement dans les règles de conflit entre les modes. Pour des informations sur la façon d'acquérir un réel verrou au niveau ligne, voir Section 12.3.2 et *Clause FOR UPDATE* dans la documentation de référence de SELECT.

# Exemples

Obtenir un verrou SHARE sur une table avec clé primaire avant de réaliser des insertions dans une table disposant de la clé étrangère :

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE nom = 'Star Wars: Episode I - The Phantom Menace';
-- Effectuer un ROLLBACK si aucun enregistrement n'est retourné
INSERT INTO commentaires_films VALUES
    (_id_, 'SUPER ! Je l''attendais depuis si longtemps !');
COMMIT WORK;
```

Prendre un verrou SHARE ROW EXCLUSIVE sur une table avec clé primaire lors du début des opérations de suppression :

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM commentaires_films WHERE id IN
    (SELECT id FROM films WHERE score < 5);
DELETE FROM films WHERE score < 5;
COMMIT WORK;
```

# Compatibilité

LOCK TABLE n'existe pas dans le standard SQL, qui utilise à la place SET TRANSACTION pour spécifier des niveaux de concurrence sur les transactions. PostgreSQL a aussi cela ; voir *SET TRANSACTION* pour les détails.

Sauf pour les modes de verrous ACCESS SHARE, ACCESS EXCLUSIVE et SHARE UPDATE EXCLUSIVE, les modes de verrou PostgreSQL et la syntaxe LOCK TABLE sont compatibles avec ceux présents dans Oracle.

# MOVE

## Nom

MOVE −− positionne un curseur

## Synopsis

```
MOVE [ direction { FROM | IN } ] nom du curseur
```

## Description

`MOVE` repositionne le curseur sans ramener aucune donnée. `MOVE` fonctionne exactement comme la commande `FETCH` sauf que `MOVE` positionne seulement le curseur et ne retourne aucune ligne.

Référez−vous à ***FETCH*** pour les détails sur la syntaxe et l'utilisation.

## Sortie

Lors d'un achèvement réussi, une commande `MOVE` retourne une balise de commande de la forme

```
MOVE compteur
```

Le *`compteur`* est le nombre de lignes qu'une commande `FETCH` avec les mêmes paramètres aurait renvoyée (zéro étant une valeur possible).

## Exemples

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- Saute les 5 premières lignes :
MOVE FORWARD 5 IN liahona;
MOVE 5

-- Récupère la 6ème ligne à partir du curseur liahona :
FETCH 1 FROM liahona;
 code  | titre  | did | date_prod  | genre  |  longueur
-------+--------+-----+------------+--------+-----------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- Ferme le curseur liahona et termine la transaction :
CLOSE liahona;
COMMIT WORK;
```

## Compatibilité

Il n'existe pas d'instruction `MOVE` dans le standard SQL.

## Voir aussi

*CLOSE*, *DECLARE*, *FETCH*

# NOTIFY

## Nom

NOTIFY −− génère une notification

## Synopsis

```
NOTIFY nom
```

## Description

La commande `NOTIFY` envoie un événement de notification à chaque application cliente qui a exécuté précédemment la commande `LISTEN` *`nom`* pour le nom de notification spécifié dans la base de données courante.

`NOTIFY` fournit une simple forme de signal ou un mécanisme de communication interprocessus pour une collection de processus accédant à la même base de données PostgreSQL. Des mécanismes de haut niveau peuvent être construit en utilisant les tables dans la base de données pour passer des données supplémentaires (en plus d'un simple nom de notification) du notifieur aux écouteurs.

L'information passée au client pour un événement de notification inclut le nom de notification et le PID du processus serveur de la session le notifiant. C'est au concepteur de la base de données de définir les noms de notification qui sont utilisés dans une base de données donnée et ce que chacun signifie.

Habituellement, le nom de notification est le même que le nom d'une table dans la base de données. L'événement notify signifie essentiellement << J'ai modifié cette table, jetez−y un œil pour vérifier ce qu'il y a de nouveau >>. Mais cette signification n'est pas imposée par les commandes `NOTIFY` et `LISTEN`. Par exemple, un concepteur de bases de données pourrait utiliser plusieurs noms de notification différentes pour signaler différentes sortes de modifications vers une seule table.

Lorsque `NOTIFY` est utilisé pour signaler l'occurrence des modifications pour une table particulière, une technique de programmation utile est de placer le `NOTIFY` dans une règle qui est déclenchée par les mises à jour de table. De cette façon, la notification arrive automatiquement quand la table est modifiée et le programmeur d'application ne peut oublier accidentellement de le faire.

`NOTIFY` interagit fortement avec les transactions SQL. Tout d'abord, si un `NOTIFY` est exécuté à l'intérieur d'une transaction, les événements notify ne sont pas délivrés jusqu'à ce que et à condition que la transaction soit validée. Ceci est approprié car, si la transaction est annulée, toutes les commandes qu'elle contenait sont annulées, y compris `NOTIFY`. Mais, cela peut être déconcertant si on s'attend à ce que les événements de notification soient délivrés immédiatement. Ensuite, si une session en écoute reçoit un signal de notification alors qu'il est à l'intérieur d'une transaction, l'événement de notification n'est pas délivré au client connecté tant que la transaction n'est pas terminée (soit validée soit annulée). Encore une fois, le raisonnement est que si une notification a été délivrée à l'intérieur d'une transaction qui a été finalement annulée, on voudrait que la notification soit annulée — mais le serveur ne peut pas << récupérer >> une notification une fois qu'elle a été envoyée au client. Donc, les événements de notification sont seulement délivrés entre les

transactions. Ce qui en découle est que les applications utilisant NOTIFY pour des signaux en temps réel doivent essayer d'avoir des transactions courtes.

NOTIFY se comporte comme les signaux Unix sur un seul point : si le même nom de notification est signalé plusieurs fois en des successions rapides, il est possible que les récepteurs ne reçoivent qu'un seul événement de notification pour plusieurs exécutions de NOTIFY. Donc, c'est une mauvaise idée de dépendre du nombre de notifications reçues. À la place, utilisez NOTIFY pour réveiller vos applications qui ont besoin de faire attention à quelque chose et utilisez un objet de bases de données (tel qu'une séquence) pour garder trace de ce qui s'est passé ou du nombre de fois où cela s'est passé.

Il est courant pour un client qui exécute NOTIFY d'écouter le même nom de notification lui−même. Dans ce cas, il récupère un événement de notification, comme toutes les autres sessions en écoute. Suivant la logique de l'application, ceci pourrait résulter en un travail inutile, par exemple lire une table de la base de données pour trouver les mêmes mises à jour que cette session a écrit. Il est possible d'éviter un travail supplémentaire verifiant si le PID du processus serveur de la session notifiante (fourni dans le message d'événement de la notification) est le même que le PID sur process serveur de sa propre session (disponible à partir de libpq). Quand ils sont identiques, l'événement de notification est le retour de son propre travail et peut être ignoré. (En dépit de ce qui est dit dans le précédent paragraphe, c'est une technique sûre. PostgreSQL distingue les notifications propres des notifications arrivant des autres sessions, de façon à ne pas oublier une notification externe en ignorant vos propres notifications.)

# Paramètres

*nom*
: Nom de la notification à signaler (un identifiant).

# Exemples

Configure et exécute une séquence listen/notify à partir de psql :

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

# Compatibilité

Il n'y a pas d'instruction NOTIFY dans le standard SQL.

# Voir aussi

*LISTEN*, *UNLISTEN*

# PREPARE

## Nom

PREPARE −− prépare une instruction pour exécution

## Synopsis

```
PREPARE nom_plan [ (type_donnees [, ...] ) ] AS instruction
```

## Description

`PREPARE` crée une instruction préparée. Une instruction préparée est un objet côté serveur qui peut être utilisé pour optimiser les performances. Quand l'instruction `PREPARE` est exécutée, l'instruction spécifiée est analysée, réécrite et planifiée. Quand une commande `EXECUTE` est lancée par la suite, l'instruction préparée a seulement besoin d'être exécutée. Du coup, les étapes d'analyse, de réécriture et de planification sont réalisées une seule fois, à la place de chaque fois que l'instruction est exécutée.

Les instructions préparées peuvent prendre des paramètres : les valeurs sont substituées dans l'instruction lorsqu'elle est exécutée. Pour inclure les paramètres dans une instruction préparée, fournissez une liste des types de données dans l'instruction `PREPARE`, et, dans l'instruction à préparer elle−même, référez−vous aux paramètres par position en utilisant `$1`, `$2`, etc. Lors de l'exécution de l'instruction, spécifiez les valeurs réelles pour ces paramètres dans l'instruction `EXECUTE`. Référez−vous à ***EXECUTE*** pour plus d'informations à ce sujet.

Les instructions préparées sont seulement stockées pour la durée de la session en cours. Lorsque la session se termine, l'instruction préparée est oubliée et, du coup, elle doit être recréée avant d'être utilisée de nouveau. Ceci signifie aussi qu'une seule instruction préparée ne peut pas être utilisée par plusieurs clients de bases de données simultanément ; néanmoins, chaque client peut créer sa propre instruction préparée à utiliser. L'instruction préparée peut être supprimée manuellement en utilisant la commande `DEALLOCATE`.

Les instructions préparées sont principalement intéressantes quand une seule session est utilisée pour exécuter un grand nombre d'instructions similaires. La différence de performances est particulièrement significative si les instructions sont complexes à planifier ou à réécrire, par exemple, si la requête implique une jointure de plusieurs tables ou requiert l'application de différentes règles. Si l'instruction est relativement simple à planifier ou à réécrire mais assez coûteuse à exécuter, l'avantage de performance des instructions préparées est moins net.

## Paramètres

*nom_plan*
: Un nom quelconque donné à cette instruction préparée particulière. Il doit être unique dans une session et est utilisé par la suite pour exécuter ou désallouer cette instruction préparée.

*type_donnees*

Le type de données d'un paramètre de l'instruction préparée. Pour référencer les paramètres dans l'instruction préparée elle−même, utilisez `$1`, `$2`, etc.

*`instruction`*

Toute instruction `SELECT`, `INSERT`, `UPDATE` ou `DELETE`.

# Notes

Dans certaines situations, le plan de requête produit par une instruction préparée est inférieur au plan qui aurait été produit si l'instruction avait été soumise et exécutée normalement. C'est parce que, quand l'instruction est planifiée et que le planificateur tente de déterminer le plan de requête optimal, les valeurs réelles de tous les paramètres spécifiés dans l'instruction ne sont pas disponibles. PostgreSQL récupère les statistiques de la distribution des données dans la table et peut utiliser les valeurs constantes dans une instruction pour deviner le résultat probable de l'exécution de l'instruction. Comme cette donnée n'est pas disponible lors de la planification d'instructions préparées avec paramètres, le plan choisi pourrait ne pas être optimal. Pour examiner le plan de requête que PostgreSQL a choisi pour une instruction préparée, utilisez *EXPLAIN*.

Pour plus d'informations sur la planification de la requête et les statistiques récupérées par PostgreSQL dans ce but, voir la documentation de *ANALYZE*.

# Exemples

Crée une requête préparée pour une instruction `INSERT`, puis l'exécute :

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Crée une requête préparée pour une instruction `SELECT`, puis l'exécute :

```
PREPARE usrrptplan (int, date) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

# Compatibilité

Le standard SQL inclut une instruction `PREPARE` mais il est seulement utilisé en SQL embarqué. Cette version de l'instruction `PREPARE` utilise aussi une syntaxe quelque peu différente.

# Voir aussi

*DEALLOCATE*, *EXECUTE*

# REINDEX

## Nom

REINDEX −− reconstruit les index

## Synopsis

```
REINDEX { DATABASE | TABLE | INDEX } nom [ FORCE ]
```

## Description

`REINDEX` reconstruit un index basé sur les données stockées dans la table, remplaçant l'ancienne copie de l'index. Il y a deux raisons principales pour utiliser `REINDEX` :

- Un index a été corrompu et ne contient plus de données valides. Bien qu'en théorie, ceci ne devrait jamais arriver, en pratique, les index peuvent se corrompre à cause de bogues dans le logiciel ou d'échecs matériels. `REINDEX` fournit une méthode de récupération.
- L'index en question contient beaucoup de pages d'index mortes qui ne sont pas réclamées. Ceci peut arriver avec des index B−tree dans PostgreSQL sous certains modèles d'accès. `REINDEX` fournit un moyen de réduire la consommation d'espace de l'index en écrivant une nouvelle version de l'index sans les pages mortes. Voir Section 21.2 pour plus d'informations.

## Paramètres

`DATABASE`
: Recrée tous les index système d'une base de données spécifiée. Les index sur les tables utilisateur ne sont pas traités. De plus, les index sur les catalogues système partagés ne sont pas pris en compte sauf dans le mode autonome (voir ci−dessous).

`TABLE`
: Recrée tous les index d'une table spécifiée. Si la table a une seconde table << TOAST >>, elle est aussi réindexée.

`INDEX`
: Recrée un index spécifié.

*nom*
: Le nom de la base de données, table ou index spécifique à réindexer. Les noms de table et d'index peuvent être qualifiés du nom du schéma. Actuellement, `REINDEX DATABASE` ne peut réindexer que la base de données en cours, donc ce paramètre doit correspondre au nom de la base de données en cours.

`FORCE`
: Ceci est une option obsolète ; elle est ignorée si elle est spécifiée.

# Notes

Si vous suspectez une corruption d'un index sur une table utilisateur, vous pouvez simplement reconstruire cet index, ou tous les index de la table, en utilisant `REINDEX INDEX` ou `REINDEX TABLE`.

Les choses sont plus difficiles si vous avez besoin de récupérer de la corruption d'un index sur une table système. Dans ce cas, il est important pour le système de ne pas avoir utilisé lui−même un des index suspects. (En fait, dans ce type de scénario, vous pourriez trouver que les processus serveur s'arrêtent brutalement immédiatement au lancement, à cause du besoin des index corrompus.) Pour récupérer proprement, le serveur doit être lancé avec l'option `−P`, qui l'empêche d'utiliser les index pour les recherches dans les catalogues système.

Une façon de faire ceci est d'arrêter le postmaster et de lancer le serveur PostgreSQL en mode autonome avec l'option `−P` placée sur la ligne de commande. Ensuite, `REINDEX DATABASE`, `REINDEX TABLE` ou `REINDEX INDEX` peuvent être lancés suivant ce que vous souhaitez reconstruire. En cas de doute, utilisez `REINDEX DATABASE` pour sélectionner la reconstruction de tous les index système dans la base de données. Enfin, quittez la session autonome du serveur et relancez le serveur habituel. Voir la page de référence de postgres pour plus d'informations sur l'interaction avec l'interface du serveur autonome.

Une session standard du serveur peut aussi être lancée avec `−P` dans les options de la ligne de commande. La méthode pour ce faire varie entre les clients mais dans tous les clients basés sur libpq, il est possible de configurer la variable d'environnement `PGOPTIONS` à `−P` avant de lancer le client. Notez que, bien que cette méthode ne verrouille pas les autres clients, il est conseillé d'empêcher les autres utilisateurs de se connecter à la base de données endommagée jusqu'à la fin des réparations.

Si une corruption est suspectée dans les index d'un des catalogues système partagés (`pg_database`, `pg_group`, `pg_shadow` ou `pg_tablespace`), alors un serveur autonome doit être utilisé pour le réparer. `REINDEX` ne traite pas les catalogues partagés dans le mode multiutilisateur.

Pour tous les index sauf les catalogues système partagés, `REINDEX` est protégé contre les arrêts brutaux et utilise les transactions. `REINDEX` n'est pas protégé pour les index partagés, ce qui explique pourquoi ce cas est désactivé pendant les opérations normales. Si un échec survient lors de la réindexation d'un de ces catalogues dans le mode autonome, il n'est pas possible de relancer le serveur en mode normal jusqu'à ce que le problème soit rectifié. (Le symptome typique d'un index partagé reconstruit partiellement est une erreur << index n'est pas un btree >>.)

`REINDEX` est similaire à une suppression et à une nouvelle création de l'index dans le fait que le contenu de l'index est complètement recréé. Néanmoins, les considérations de verrouillage sont assez différentes. `REINDEX` verrouille les écritures mais pas les lectures de la table mère de l'index. Il prend aussi un verrou exclusif sur l'index en cours de traitement, ce qui bloque les lectures qui tentent d'utiliser l'index. Au contraire, `DROP INDEX` crée temporairement un verrou exclusif sur la table parent, bloquant ainsi écritures et lectures. Le `CREATE INDEX` qui suit verrouille les écritures mais pas les lectures ; comme l'index n'existe pas, aucune lecture ne peut être tentée, signifiant qu'il n'y a aucun blocage et que les lectures sont probablement forcées de réaliser des parcours séquentiels complets. Un autre point important est que l'approche suppression/création invalide tous plans de requête en cache utilisant cet index, problème que `REINDEX` ne connaît pas.

Avant PostgreSQL 7.4, `REINDEX TABLE` ne traitait pas automatiquement les tables TOAST et, du coup, elles devaient être réindexées par des commandes séparées. C'est toujours possible mais redondant.

# Exemples

Recrée les index sur la table `ma_table` :

```
REINDEX TABLE ma_table;
```

Reconstruit un index simple :

```
REINDEX INDEX my_index;
```

Reconstruit tous les index système d'une base de données particulière sans croire qu'elles sont déjà valides :

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

# Compatibilité

Il n'existe pas de commande `REINDEX` dans le standard SQL.

# RELEASE SAVEPOINT

## Nom

RELEASE SAVEPOINT −− détruit un point de sauvegarde défini précédemment

## Synopsis

```
RELEASE [ SAVEPOINT ] nom_pointsauvegarde
```

## Description

`RELEASE SAVEPOINT` détruit un point de sauvegarde précédemment défini dans la transaction courante.

Détruire un point de sauvegarde le rend indisponible comme point de retour mais il n'y a pas d'autre comportement visible par l'utilisateur. Il ne défait pas les commandes exécutées après l'établissement du point de sauvegarde. (Pour cela, voir *ROLLBACK TO SAVEPOINT*.) Détruire un point de sauvegarde quand il n'est plus nécessaire peut permettre au système de récupérer certaines ressources avant la fin de la transaction.

`RELEASE SAVEPOINT` détruit aussi tous les points de sauvegarde qui ont été créés après l'établissement du point de sauvegarde indiqué.

## Paramètres

*`nom_pointsauvegarde`*
: Le nom du point de sauvegarde à détruire.

## Notes

Spécifier un nom de point de sauvegarde qui n'a pas été défini est une erreur.

Il n'est pas possible de libérer un point de sauvegarde lorsque la transaction est dans un état d'annulation.

Si plusieurs points de transaction ont le même nom, seul celui qui a été défini le plus récemment est libéré.

## Exemples

Pour établir puis détruire un point de sauvegarde :

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (4);
```

```
    RELEASE SAVEPOINT mon_pointsauvegarde;
COMMIT;
```

La transaction ci−dessus insère à la fois 3 et 4.

# Compatibilité

Cette commande est conforme au standard SQL:2003. Le standard spécifie que le mot clé `SAVEPOINT` est obligatoire mais PostgreSQL autorise son omission.

# Voir aussi

*BEGIN*, *COMMIT*, *ROLLBACK*, *ROLLBACK TO SAVEPOINT*, *SAVEPOINT*

# RESET

## Nom

RESET −− remet un paramètre d'exécution à sa valeur par défaut

## Synopsis

```
RESET nom
RESET ALL
```

## Description

`RESET` restaure les paramètres d'exécution à leur valeur par défaut. `RESET` est une alternative à

```
SET paramètre TO DEFAULT
```

Référez−vous à *SET* pour plus de détails.

La valeur par défaut est définie comme la valeur qu'aurait la variable si aucune commande `SET` n'avait été lancée sur elle pour la session en cours. La source réelle de cette valeur peut être les valeurs par défaut compilées, le fichier de configuration, les options de la ligne de commande ou les paramétrages spécifiques à la base de données ou à l'utilisateur. Voir Section 16.4 pour les détails.

Voir la page de référence de `SET` pour les détails sur le comportement des transactions avec `RESET`.

## Paramètres

*nom*
: Le nom du paramètre d'exécution. Voir *SET* pour une liste.

`ALL`
: Réinitialise tous les paramètres configurables à l'exécution à leur valeur par défaut.

## Exemples

Pour initialiser `geqo` à sa valeur par défaut :

```
RESET geqo;
```

## Compatibilité

`RESET` est une extension de PostgreSQL.

# REVOKE

## Nom

REVOKE −− supprime les droits d'accès

## Synopsis

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] nom_table [, ...]
    FROM { nom_utilisateur | GROUP
    nom_groupe | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE nom_base [, ...]
    FROM { nom_utilisateur | GROUP
    nom_groupe | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION nom_fonction
([type, ...]) [, ...]
    FROM { nom_utilisateur | GROUP
    nom_groupe | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE nom_langage [, ...]
    FROM { nom_utilisateur | GROUP
    nom_groupe | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA nom_schéma [, ...]
    FROM { nom_utilisateur | GROUP
    nom_groupe | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE nom_espacelogique [, ...]
    FROM { nom_utilisateur | GROUP
nom_groupe | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

# Description

La commande REVOKE retire des droits précédemment attribués à un ou plusieurs utilisateurs ou groupes d'utilisateurs. Le mot clé PUBLIC fait référence au groupe implicitement défini de tous les utilisateurs.

Voir la description de la commande *GRANT* pour connaître la signification des types de droits.

Notez qu'un utilisateur possède la somme des droits qui lui ont été donnés directement, des droits qui ont été donnés à un groupe dont il est membre et des droits donnés à PUBLIC. Du coup, par exemple, retirer les droits de SELECT à PUBLIC ne veut pas nécessairement dire que plus personne n'a le droit de faire de SELECT sur l'objet : ceux qui en avaient obtenu le droit directement ou via un groupe l'ont toujours.

Si GRANT OPTION FOR est précisé, seul l'option de transmission de droit (grant option) est supprimée, pas le droit lui même. Sinon, le droit et l'option de transmission de droits sont révoqués.

Si un utilisateur détient un privilège avec le droit de le transmettre, et qu'il l'a transmis à d'autres utilisateurs, alors les droits de ceux−ci sont appelés des droits dépendants. Si les droits ou le droit de transmettre du premier utilisateur sont supprimés, et que des droits dépendants existent, alors ces droits dépendants sont aussi supprimés si l'option CASCADE est utilisée, sinon la suppression de droits est refusée. Cette révocation récursive n'affecte que les droits qui avaient été attribués à travers une chaîne d'utilisateurs traçable jusqu'à l'utilisateur qui subit la commande REVOKE. Du coup, les utilisateurs affectés peuvent finalement garder le droit s'il avait aussi été attribué via d'autres utilisateurs.

# Notes

Utilisez la commande \z de psql pour afficher les droits donnés sur des objets existants. Voir *GRANT* pour des informations sur le format.

Un utilisateur ne peut révoquer que les droits qu'il a donné directement. Si, par exemple, un utilisateur A a donné un droit et la possibilité de le transmettre à un utilisateur B, et que B à son tour l'a donné à C, alors A ne peut pas retirer directement le droit de C. À la place, il peut supprimer le droit de transmettre à B et utiliser l'option CASCADE pour que le droit soit automatiquement supprimé à C. Autre exemple, si A et B ont donné le même droit à C, A peut révoquer son propre don de droit mais pas celui de B, donc C dispose toujours de ce droit.

Lorsqu'un utilisateur, non propriétaire de l'objet, essaie de révoquer (REVOKE) des droits sur l'objet, la commande échoue si l'utilisateur n'a aucun droit sur l'objet. Tant que certains droits sont disponibles, la commande s'exécute mais ne sont supprimés que les droits dont l'utilisateur a l'option de transmission. La forme REVOKE ALL PRIVILEGES affiche un message d'avertissement si les options de transmissions pour un des droits nommés spécifiquement dans la commande ne sont pas possédés. (En principe, ces instructions s'appliquent aussi au propriétaire de l'objet mais comme le propriétaire est toujours traité comme celui détenant toutes les options de transmission, ces cas n'arrivent jamais.)

Si un superutilisateur choisit d'exécuter une commande GRANT ou REVOKE, la commande est exécutée comme si elle était lancée par le propriétaire de l'objet affecté. Comme tous les droits proviennent du propriétaire d'un objet (directement ou via une chaîne de transmissions de droits), un superutilisateur peut supprimer tous les droits sur un objet mais cela peut nécessiter l'utilisation de CASCADE comme expliqué précédemment.

# Exemples

Enlève au groupe public le droit d'insérer des lignes dans la table `films` :

```
REVOKE INSERT ON films FROM PUBLIC;
```

Supprime tous les droits de l'utilisateur `manuel` sur la vue `genres` :

```
REVOKE ALL PRIVILEGES ON genres FROM manuel;
```

Notez que ceci signifie en fait << révoque tous les droits que j'ai donné >>.

# Compatibilité

La note de compatibilité de la commande *GRANT* s'applique par analogie à `REVOKE`. Le résumé de la syntaxe est :

```
REVOKE [ GRANT OPTION FOR ] privileges
    ON objet [ ( colonne [, ...] ) ]
    FROM { PUBLIC | nom_utilisateur
[, ...] }
    { RESTRICT | CASCADE }
```

Le standard nécessite qu'une des options `RESTRICT` ou `CASCADE` soit indiquée, mais PostgreSQL utilise `RESTRICT` par défaut.

# Voir aussi

*GRANT*

# ROLLBACK

## Nom

ROLLBACK −− annule la transaction en cours

## Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

## Description

`ROLLBACK` annule la transaction en cours et fait que les modifications effectuées par la transaction sont annulées.

## Paramètres

`WORK`
`TRANSACTION`
: Mots clés optionnels. Ils sont sans effet.

## Notes

Utilisez *COMMIT* pour terminer une transaction avec succès.

Lancer `ROLLBACK` à l'extérieur d'une transaction ne pose pas de problème mais affiche un message d'avertissement.

## Exemples

Pour annuler toutes les modifications :

```
ROLLBACK;
```

## Compatibilité

Le standard SQL spécifie seulement les deux formes `ROLLBACK` et `ROLLBACK WORK`. Sinon, cette commande est totalement compatible.

# Voir aussi

*BEGIN*, *COMMIT*, *ROLLBACK TO SAVEPOINT*

# ROLLBACK TO SAVEPOINT

## Nom

ROLLBACK TO SAVEPOINT −− annule les instructions depuis un point de sauvegarde

## Synopsis

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] nom_pointsauvegarde
```

## Description

Annule toutes les commandes qui ont été exécutées après l'établissement du point de sauvegarde. Le point de sauvegarde reste valide et peut être annulé de nouveau plus tard si nécessaire.

`ROLLBACK TO SAVEPOINT` détruit implicitement tous les points de sauvegarde qui ont été établis après le point de sauvegarde indiqué.

## Paramètres

*`nom_pointsauvegarde`*
: Le point de sauvegarde où retourner.

## Notes

Utilisez *RELEASE SAVEPOINT* pour détruire un point de sauvegarde sans annuler les effets de commandes exécutées après son établissement.

Spécifier un nom de point de sauvegarde inexistant est une erreur.

Les curseurs ont un comportement quelque peu non transactionnel en ce qui concerne les points de sauvegarde. Tout curseur ouvert à l'intérieur d'un point de sauvegarde est fermé lorsque le point de sauvegarde est annulé. Si un curseur ouvert précédemment est affecté à une commande `FETCH` à l'intérieur d'un point de sauvegarde annulé un peu plus tard, la position du curseur reste à la position que `FETCH` lui a donné (c'est−à−dire que `FETCH` n'est pas annulé). Fermer un curseur n'est pas non plus défait par une annulation. Un curseur dont l'exécution cause l'annulation d'une transaction est placé dans un état non exécutable si bien que, bien que la transaction puisse être restaurée avec `ROLLBACK TO SAVEPOINT`, le curseur ne peut plus être utilisé.

# Exemples

Pour annuler les effets des commandes exécutées après l'établissement de `mon_pointsauvegarde` :

```
ROLLBACK TO SAVEPOINT mon_pointsauvegarde;
```

La position d'un curseur n'est pas affectée par l'annulation des points de sauvegarde :

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

# Compatibilité

Le standard SQL:2003 spécifie que le mot clé `SAVEPOINT` est obligatoire mais PostgreSQL et Oracle autorisent son omission. SQL:2003 autorise seulement `WORK`, pas `TRANSACTION`, comme mot après `ROLLBACK`. De plus, SQL:2003 dispose d'une clause optionnelle `AND [ NO ] CHAIN` qui n'est actuellement pas supportée par PostgreSQL. Sinon, cette commande est conforme au standard SQL.

# Voir aussi

*BEGIN*, *COMMIT*, *RELEASE SAVEPOINT*, *ROLLBACK*, *SAVEPOINT*

# SAVEPOINT

## Nom

SAVEPOINT −− définit un nouveau point de sauvegarde à l'intérieur de la transaction en cours

## Synopsis

```
SAVEPOINT nom_pointsauvegarde
```

## Description

`SAVEPOINT` établit un nouveau point de sauvegarde à l'intérieur de la transaction en cours.

Un point de sauvegarde est une marque spéciale à l'intérieur d'une transaction qui autorise l'annulation de toutes les commandes exécutées après son établissement, restaurant la transaction dans l'état où elle était au moment de l'établissement du point de sauvegarde.

## Paramètres

*`nom_pointsauvegarde`*
: Le nom du nouveau point de sauvegarde.

## Notes

Utilisez *ROLLBACK TO SAVEPOINT* pour annuler un point de sauvegarde. Utilisez *RELEASE SAVEPOINT* pour détruire un point de sauvegarde, conservant l'effet des commandes exécutées après son établissement.

Les points de sauvegarde peuvent seulement être établis à l'intérieur d'un bloc de transaction. Plusieurs points de sauvegarde peuvent être définis dans une transaction.

## Exemples

Pour établir un point de sauvegarde et annuler plus tard les effets des commandes exécutées après son établissement :

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

La transaction ci−dessus insère les valeurs 1 et 3, mais pas 2.

Pour établir puis détruire un point de sauvegarde :

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT mon_pointsauvegarde;
COMMIT;
```

La transaction ci−dessus insère à la fois les valeurs 3 et 4.

# Compatibilité

SQL requiert la destruction automatique d'un point de sauvegarde quand un autre point de sauvegarde du même nom est créé. Avec PostgreSQL, l'ancien point de sauvegarde est conservé, mais seul le plus récent est utilisé pour une annulation ou une libération. (Libérer le point de sauvegarde le plus récent fait que l'ancien est de nouveau accessible aux commandes `ROLLBACK TO SAVEPOINT` et `RELEASE SAVEPOINT`.) Sinon, `SAVEPOINT` est totalement conforme à SQL.

# Voir aussi

*BEGIN*, *COMMIT*, *RELEASE SAVEPOINT*, *ROLLBACK*, *ROLLBACK TO SAVEPOINT*

# SELECT

## Nom

SELECT −− récupère des lignes d'une table ou d'une vue

## Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS nom_d_affichage ] [, ...]
    [ FROM éléments_from [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC |
DESC | USING opérateur ] [, ...] ]
    [ LIMIT { nombre | ALL } ]
    [ OFFSET début ]
    [ FOR UPDATE [ OF nom_table [,
...] ] ]

où éléments_from fait partie de:

    [ ONLY ] nom_table [ * ] [ [
AS ] alias [ ( alias_colonne [, ...] ) ] ]
    ( select ) [ AS ] alias [ ( alias_colonne [, ...] ) ]
    nom_fonction ( [ argument [, ...] ] ) [ AS ] alias [ ( alias_colonne [, ...] | définition_col
    nom_fonction ( [ argument [, ...] ] ) AS ( définition_colonne [, ...] )
    éléments_from [ NATURAL ]
type_jointure éléments_from [ ON condition_jointure | USING ( colonne_jointure [, ...] ) ]
```

## Description

SELECT récupère des lignes à partir d'une ou plusieurs tables. Le traitement général de SELECT est le suivant :

1. Tous les éléments de la liste FROM sont calculés. (Chaque élément dans la liste FROM est une table réelle ou virtuelle.) Si plus d'un élément est spécifié dans la liste FROM, ils sont joints ensemble. (Voir *Clause FROM* ci−dessous.)
2. Si la clause WHERE est spécifiée, toutes les lignes qui ne satisfont pas les conditions sont éliminées de l'affichage. (Voir *Clause WHERE* ci−dessous.)
3. Si la clause GROUP BY est spécifiée, l'affichage est divisé en groupes de lignes qui correspondent à une ou plusieurs valeurs. Si la clause HAVING est présente, elle élimine les groupes qui ne satisfont pas la condition donnée. (Voir *Clause GROUP BY* et *Clause HAVING* ci−dessous.)
4. Les lignes en sortie sont traitées en utilisant les expressions de sortie de SELECT pour chaque ligne sélectionnée. (Voir *Liste SELECT* ci−dessous.)
5. En utilisant les opérateurs UNION, INTERSECT et EXCEPT, l'affichage de plus d'une instruction SELECT peut être combiné pour former un ensemble de résultats. L'opérateur UNION renvoie toutes les lignes qui sont dans un ou plusieurs ensembles de résultats. L'opérateur INTERSECT renvoie

toutes les lignes qui sont dans les deux ensembles de résultats. L'opérateur EXCEPT renvoie les lignes qui sont présentes dans le premier ensemble de résultats mais pas dans le deuxième. Dans les trois cas, les lignes dupliquées sont éliminées sauf si ALL est spécifié. (Voir *Clause UNION*, *Clause INTERSECT* et *Clause EXCEPT* ci−dessous.)

6. Si la clause ORDER BY est spécifiée, les lignes renvoyées sont triées dans l'ordre spécifié. Si ORDER BY n'est pas indiqué, les lignes sont renvoyées dans l'ordre que le système trouve le plus rapide à fournir. (Voir *Clause ORDER BY* ci−dessous.)
7. DISTINCT élimine les lignes dupliquées du résultat. DISTINCT ON élimine les lignes qui correspondent à toutes les expressions données. ALL (la valeur par défaut) renvoie toutes les lignes candidates, y compris les lignes dupliquées. (Voir *Clause DISTINCT* ci−dessous.)
8. Si les clauses LIMIT ou OFFSET sont spécifiées, l'instruction SELECT ne renvoie qu'un sous−ensemble de lignes de résultats. (Voir *Clause LIMIT* ci−dessous.)
9. La clause FOR UPDATE fait que l'instruction verrouille les lignes sélectionnées contre les mises à jour concurrentes. (Voir *Clause FOR UPDATE* ci−dessous.)

Vous devez avoir le droit SELECT sur une table pour lire ses valeurs. L'utilisation de FOR UPDATE requiert de plus le droit UPDATE.

# Paramètres

## Clause FROM

La clause FROM spécifie une ou plusieurs tables source pour SELECT. Si plusieurs sources sont spécifiées, le résultat est un produit cartésien (jointure croisée) de toutes les sources. Mais habituellement, des conditions de qualification sont ajoutées pour restreindre les lignes renvoyées à un petit sous−ensemble de produit cartésien.

La clause FROM peut contenir les éléments suivants :

*nom_table*
: Le nom (pouvant être qualifié du nom du schéma) d'une table existante ou d'une vue. Si ONLY est spécifié, seule cette table est parcourue. Si ONLY n'est pas spécifié, la table et toutes ses descendantes (si elles existent) sont parcourues. * peut être ajouté au nom de la table pour indiquer que les tables descendantes doivent être parcourues mais, dans la version actuelle, c'est le comportement par défaut. (Dans les versions précédant la 7.1, ONLY était le comportement par défaut.) Le comportement par défaut peut être modifié en changeant l'option de configuration sql_inheritance.

*alias*
: Un nom de substitution pour l'élément FROM contenant l'alias. Un alias est utilisé par brièveté ou pour éliminer toute ambiguïté pour les jointures où la même table est parcourue plusieurs fois. Quand un alias est fourni, il cache complètement le nom réel de la table ou fonction ; par exemple, avec FROM foo AS f, le reste du SELECT doit faire référence à cet élément de FROM par f et non pas par foo. Si un alias est donné, une liste d'alias de colonnes peut aussi être saisi pour fournir des noms de substitution pour une ou plusieurs colonnes de la table.

*select*
: Un sous−SELECT peut apparaître dans la clause FROM. Ceci agit comme si sa sortie était transformée en une table temporaire pour la durée de cette seule commande SELECT. Notez que le sous−SELECT doit être entouré par des parenthèses et qu'un alias *doit* être fourni pour le sous−SELECT.

*nom_fonction*

Des appels de fonctions peuvent apparaître dans la clause FROM. (Ceci est particulièrement utile pour les fonctions renvoyant des ensembles de résultats mais toute fonction peut être utilisée.) Ceci agit comme si la sortie était transformée en une table temporaire pour la durée de cette seule commande SELECT. Un alias peut aussi être utilisé. Si un alias est donné, une liste d'alias de colonnes peut être ajoutée pour fournir des noms de substitution pour un ou plusieurs attributs du type composé de retour de la fonction. Si la fonction a été définie comme renvoyant le type de données record, alors un alias ou un mot clé AS doit être présent, suivi par une liste de définitions de colonnes de la forme ( *nom_colonne type_données* [, ... ] ). La liste de définitions de colonnes doit correspondre au nombre réel et aux types réels des colonnes renvoyées par la fonction.

*type_jointure*

Fait partie de

- [ INNER ] JOIN
- LEFT [ OUTER ] JOIN
- RIGHT [ OUTER ] JOIN
- FULL [ OUTER ] JOIN
- CROSS JOIN

Pour les types de jointures INNER et OUTER, une condition de jointure doit être spécifiée, parmi NATURAL, ON *condition_jointure* ou USING (*colonne_jointure* [, ...]). Voir ci–dessous pour la signification. Pour CROSS JOIN, aucune de ces clauses ne doit apparaître.

Une clause JOIN combine deux éléments FROM. Utilisez les parenthèses si nécessaire pour déterminer l'ordre d'imbrication. En l'absence de parenthèses, les JOIN sont imbriqués de gauche à droite. Dans tous les cas, JOIN est plus prioritaire que les virgules séparant les éléments FROM.

CROSS JOIN et INNER JOIN produisent un simple produit cartésien, le même résultat que vous obtenez à partir de la liste de deux éléments au niveau haut du FROM, mais restreint par la condition de jointure (si elle existe). CROSS JOIN est équivalent à INNER JOIN ON (TRUE), c'est–à–dire qu'aucune ligne n'est supprimée par qualification. Ces types de jointure sont juste une aide à la notation car ils ne font rien de plus qu'un simple FROM et WHERE.

LEFT OUTER JOIN renvoie toutes les lignes dans le produit cartésien qualifié (c'est–à–dire toutes les lignes combinées qui réussissent la condition de jointure), plus une copie de chaque ligne dans le côté gauche de la table pour laquelle il n'y a pas de côté droit qui a réussi la condition de jointure. Cette ligne côté gauche est étendue à la largeur complète de la table jointe par insertion de valeurs NULL pour les colonnes côté droit. Notez que seule la condition de la clause JOIN est utilisée pour décider des lignes qui correspondent. Les conditions externes sont appliquées après coup.

Au contraire, RIGHT OUTER JOIN renvoie toutes les lignes jointes plus une ligne pour chaque ligne côté droit sans correspondance (complétée par des NULL pour le côté gauche). Ceci est une simple aide à la notation car vous pourriez la convertir en LEFT en inversant les entrées gauche et droite.

FULL OUTER JOIN renvoie toutes les lignes jointes, plus chaque ligne gauche sans correspondance (étendue par des NULL à droite), plus chaque ligne droite sans correspondance (étendue par des NULL à gauche).

ON *condition_jointure*

*condition_jointure* est une expression qui renvoie une valeur de type boolean (similaire à une clause WHERE) qui spécifie quelles lignes d'une jointure doivent correspondre.

USING (*colonne_jointure* [, ...])

Une clause de la forme `USING ( a, b, ... )` est un raccourci pour `ON left_table.a = right_table.a AND left_table.b = right_table.b ...`. De plus, `USING` implique que seule une des paires de colonnes équivalentes est inclue dans la sortie de la jointure, pas les deux.

`NATURAL`

`NATURAL` est un raccourci pour une liste `USING` qui mentionne toutes les colonnes dans les deux tables qui ont le même nom.

## Clause `WHERE`

La clause `WHERE` optionnelle a la forme générale

```
WHERE condition
```

où *`condition`* est une expression qui s'évalue en un résultat de type `boolean`. Toute ligne ne satisfaisant pas cette condition est éliminée de la sortie. Une ligne satisfait la condition si elle renvoie vrai quand les valeurs réelles de la ligne sont substituées à toute référence de variable.

## Clause `GROUP BY`

La clause `GROUP BY` optionnelle a la forme générale

```
GROUP BY expression [, ...]
```

`GROUP BY` condense en une seule ligne toutes les lignes sélectionnées qui partagent les mêmes valeurs pour les expressions groupées. *`expression`* peut être un nom de colonne en entrée, ou le nom ou le nombre ordinal d'une colonne en sortie (élément de la liste `SELECT`), ou une expression arbitraire formée par des valeurs de colonnes en entrée. Dans le cas d'une ambiguïté, un nom `GROUP BY` est interprété comme un nom de colonne en entrée plutôt qu'un nom de colonne en sortie.

Les fonctions d'agrégat, si elles sont utilisées, sont calculées pour toutes les lignes composant chaque groupe, produisant une valeur séparée pour chaque groupe (alors que sans `GROUP BY`, un agrégat produit une valeur unique calculée avec toutes les lignes sélectionnées). Quand `GROUP BY` est présent, il n'est pas valide que les expressions de liste `SELECT` fassent référence aux colonnes non groupées sauf à l'intérieur de fonctions d'agrégat car il y aurait plus d'une valeur possible à renvoyer pour une colonne non groupée.

## Clause `HAVING`

La clause `HAVING` optionnelle a la forme générale

```
HAVING condition
```

où *`condition`* est identique à celle spécifiée pour la clause `WHERE`.

`HAVING` élimine les lignes groupées qui ne satisfont pas à la condition. `HAVING` est différent de `WHERE` : `WHERE` filtre les lignes individuelles avant l'application de `GROUP BY` alors que `HAVING` filtre les lignes groupées créées par `GROUP BY`. Chaque colonne référencée dans *`condition`* doit référencer sans ambiguïté une colonne groupée, sauf si la référence apparaît dans une fonction d'agrégat.

## Liste SELECT

La liste SELECT (entre les mots clés SELECT et FROM) spécifie les expressions qui forment les lignes en sortie de l'instruction SELECT. Les expressions peuvent faire (et en général font) référence aux colonnes traitées dans la clause FROM. En utilisant la clause AS *nom_sortie*, un autre nom peut être indiqué pour une colonne en sortie. Ce nom est principalement utilisé pour nommer la colonne à l'affichage. Il peut aussi être utilisé pour référencer la valeur de la colonne dans les clauses ORDER BY et GROUP BY, mais pas dans les clauses WHERE ou HAVING ; là, vous devez écrire les expressions à la place.

Au lieu d'une expression, on peut utiliser * dans la liste de sortie comme raccourci pour toutes les colonnes des lignes sélectionnées. De plus, vous pouvez écrire *nom_table*.* comme raccourci pour toutes les colonnes provenant de cette table.

## Clause UNION

La clause UNION a la forme générale :

```
instruction_select UNION [ ALL ]
instruction_select
```

*instruction_select* est toute instruction SELECT sans clause ORDER BY, LIMIT ou FOR UPDATE. (ORDER BY et LIMIT peuvent être attachés à une sous−expression si elle est englobée dans des parenthèses. Sans parenthèses, ces clauses s'appliquent au résultat de l'UNION, et non pas à son expression côté droit.)

L'opérateur UNION calcule l'union ensembliste des lignes renvoyées par les instructions SELECT impliquées. Une ligne est dans l'union de deux ensembles de résultats si elle apparaît dans au moins un des ensembles de résultats. Les deux instructions SELECT qui représentent les opérandes directs de l'UNION doivent produire le même nombre de colonnes et les colonnes correspondantes doivent être d'un type de données compatible.

Le résultat de UNION ne doit pas contenir de lignes dupliquées sauf si l'option ALL est spécifiée. ALL empêche l'élimination des lignes dupliquées. (Du coup, UNION ALL est significativement plus rapide qu'UNION ; utilisez ALL quand vous le pouvez.)

Plusieurs opérateurs UNION dans la même instruction SELECT sont évalués de gauche à droite sauf si c'est indiqué autrement par des parenthèses.

Actuellement, FOR UPDATE ne peut pas être spécifié pour un résultat UNION ou pour toute entrée d'un UNION.

## Clause INTERSECT

La clause INTERSECT a cette forme générale :

```
instruction_select INTERSECT [ ALL ] instruction_select
```

*instruction_select* est toute instruction SELECT sans clause ORDER BY, LIMIT ou FOR UPDATE.

L'opérateur INTERSECT calcule l'intersection des lignes renvoyées par les instructions SELECT impliquées.

Une ligne est dans l'intersection des deux ensembles de résultats si elle apparaît dans chacun des deux ensembles.

Le résultat d'INTERSECT ne contient pas de lignes dupliquées sauf si l'option ALL est spécifiée. Avec ALL, une ligne qui a *m* lignes dupliquées dans la table gauche et *n* lignes dupliquées dans la table droite apparaît min(*m*,*n*) fois dans l'ensemble de résultats.

Plusieurs INTERSECT dans la même instruction SELECT sont évalués de gauche à droite sauf si des parenthèses dictent le contraire. INTERSECT est plus prioritaire que UNION. C'est–à–dire que A UNION B INTERSECT C est lu comme A UNION (B INTERSECT C).

Actuellement, FOR UPDATE ne peut pas être spécifié pour un résultat d'INTERSECT ou pour une entrée d'INTERSECT.

## Clause EXCEPT

La clause EXCEPT a cette forme générale :

```
instruction_select EXCEPT [ ALL ]
instruction_select
```

*instruction_select* est toute instruction SELECT sans clause ORDER BY, LIMIT ou FOR UPDATE.

L'opérateur EXCEPT calcule l'ensemble de lignes qui sont dans le résultat de l'instruction SELECT de gauche mais pas dans le résultat de celle de droite.

Le résultat de EXCEPT ne contient aucune ligne dupliquée sauf si l'option ALL est spécifiée. Avec ALL, une ligne qui a *m* dupliquées dans la table gauche et *n* dupliquées dans la table droite apparaît max(*m*–*n*,0) fois dans l'ensemble de résultat.

Plusieurs opérateurs EXCEPT dans la même instruction SELECT sont évalués de gauche à droite sauf si des parenthèses dictent le contraire. EXCEPT a la même priorité qu'UNION.

Actuellement, FOR UPDATE ne peut pas être spécifié dans un résultat EXCEPT ou pour une entrée d'un EXCEPT.

## Clause ORDER BY

La clause optionnelle ORDER BY a cette forme générale :

```
ORDER BY expression [ ASC | DESC |
USING opérateur ] [, ...]
```

*expression* peut être le nom ou le numéro ordinal d'une colonne en sortie (élément de la liste SELECT) ou il peut être une expression quelconque formée à partir des valeurs des colonnes en entrée.

La clause ORDER BY fait que les lignes de résultat sont triées suivant les expressions données. Si deux lignes sont identiques suivant l'expression la plus à gauche, elles sont comparées avec l'expression suivante et ainsi de suite. Si elles sont identiques pour toutes les expressions de tri, elles sont renvoyées dans un ordre

dépendant de l'implémentation.

Le numéro ordinal fait référence à la position ordinale (de gauche à droite) de la colonne de résultat. Cette fonctionnalité rend possible de définir un ordre sur la base d'une colonne qui n'a pas un nom unique. Ceci n'est jamais absolument nécessaire parce qu'il est toujours possible d'affecter un nom à une colonne résultat en utilisant la clause AS.

Il est aussi possible d'utiliser des expressions quelconques dans la clause ORDER BY, incluant des colonnes qui n'apparaissent pas dans la liste de résultat du SELECT. Du coup, l'instruction suivante est valide :

```
SELECT nom FROM distributeurs ORDER BY code;
```

Une limitation de cette fonctionnalité est que la clause ORDER BY s'appliquant au résultat d'une clause UNION, INTERSECT ou EXCEPT peut seulement spécifier un nom ou numéro de colonne en sortie, pas une expression.

Si une expression ORDER BY est un simple nom qui correspond à la fois à un nom de colonne résultat et à un nom de colonne en entrée, ORDER BY l'interprète comme le nom de la colonne résultat. Ceci est le contraire du choix que GROUP BY fait dans la même situation. Cette incohérence est nécessaire pour être compatible avec le standard SQL.

Optionnellement, vous pourriez ajouter le mot clé ASC (ascendant) ou DESC (descendant) après toute expression de la clause ORDER BY. Sans indication, ASC est la valeur supposée par défaut. Autrement, un nom d'opérateur d'ordre spécifique peut être fourni dans la clause USING. ASC est habituellement équivalent à USING < et DESC est habituellement équivalent à USING >. (Mais le créateur d'un type de données défini par l'utilisateur peut définir à sa guise le tri par défaut et il pourrait correspondre à des opérateurs de nom différent.)

La valeur NULL est triée plus haut que toute autre valeur. En d'autres termes, avec un ordre de tri ascendant, les valeurs NULL sont triées à la fin et avec un ordre de tri descendant, elles sont triées au début.

Les données de chaînes de caractères sont triées suivant l'ordre spécifique à la locale, ordre établi au moment de la création du groupe de bases de données.

## Clause DISTINCT

Si DISTINCT est spécifié, toutes les lignes dupliquées sont supprimées de l'ensemble de résultat (une ligne est conservée pour chaque groupe de lignes dupliquées). ALL spécifie le contraire : toutes les lignes sont conservées ; ceci est la valeur par défaut.

DISTINCT ON ( *expression* [, ...] ) conserve seulement la première ligne de chaque ensemble de lignes où les expressions sont évaluées comme identiques. Les expressions DISTINCT ON sont interprétées en utilisant les mêmes règles que pour ORDER BY (voir ci–dessus). Notez que la << première ligne >> de chaque ensemble n'est pas prévisible sauf si ORDER BY est utilisé pour s'assurer que la ligne désirée apparaît en premier. Par exemple,

```
SELECT DISTINCT ON (emplacement) emplacement, heure, rapport
FROM rapports_meteo
ORDER BY emplacement, heure DESC;
```

récupère le rapport météo le plus récent pour chaque emplacement. Mais si nous n'avions pas utilisé ORDER BY pour forcer l'ordre descendant des valeurs heure pour chaque emplacement, nous aurions obtenu un rapport à partir d'une heure non prévisible pour chaque emplacement.

Le(s) expression(s) DISTINCT ON doi(ven)t correspondre à l'ordre des expression ORDER BY. La clause ORDER BY contient normalement des expressions supplémentaires qui déterminent la précédence désirée des lignes à l'intérieur de chaque groupe DISTINCT ON.

## Clause LIMIT

La clause LIMIT est constituée de deux sous−clauses indépendantes :

```
LIMIT { nombre | ALL }
OFFSET début
```

*nombre* spécifie le nombre maximum de lignes à renvoyer alors que *début* spécifie le nombre de lignes à passer avant de commencer à renvoyer des lignes. Quand les deux sont spécifiés, *début* lignes sont passées avant de commencer à compter les *nombre* lignes à renvoyer.

Lors de l'utilisation de LIMIT, utiliser la clause ORDER BY est une bonne idée pour contraindre les lignes de résultat en un ordre unique. Sinon, vous obtenez un sous−ensemble non prévisible de lignes de la requête — vous pouvez demander les lignes 10 à 20 mais de la 10 à la 20 dans quel ordre ? Vous ne savez pas l'ordre tant que vous ne le spécifiez pas avec ORDER BY.

Le planificateur de requêtes prend LIMIT en compte lors de la génération d'un plan de requêtes, donc vous avez beaucoup de chances d'obtenir des plans différents (récupération d'ordre de lignes différents) suivant ce que vous utilisez pour LIMIT et OFFSET. Du coup, l'utilisation de valeurs différentes pour LIMIT/OFFSET pour sélectionner des sous−ensembles d'un résultat de requête *donne éventuellement des résultats incohérents* sauf si vous forcez un ordre de résultat prévisible avec ORDER BY. Ceci n'est pas un bogue ; c'est une conséquence inhérente au fait que SQL ne promet pas de délivrer les résultats d'une requête dans un ordre particulier sauf si ORDER BY est utilisé pour forcer cet ordre.

## Clause FOR UPDATE

La clause FOR UPDATE a cette forme :

```
FOR UPDATE [ OF nom_table [, ...] ]
```

FOR UPDATE fait que les lignes récupérées par l'instruction SELECT sont verrouillées pour modification. Ceci les empêche d'être modifiées ou supprimées par les autres transactions jusqu'à la fin de la transaction en cours. C'est−à−dire que les autres transactions tentant des UPDATE, DELETE ou SELECT FOR UPDATE de ces lignes sont bloquées jusqu'à ce que la transaction en cours se termine. De plus, si un UPDATE, DELETE ou SELECT FOR UPDATE à partir d'une autre transaction a déjà verrouillé une ligne ou un ensemble de lignes, SELECT FOR UPDATE attend la fin de l'autre transaction puis verrouille et renvoie la ligne modifiée (ou aucune ligne si elle a été supprimée). Pour plus d'informations, voir Chapitre 12.

Si des tables spécifiques sont nommées dans FOR UPDATE, alors seules les lignes provenant de ces tables sont verrouillées  toute autre table utilisée dans le SELECT est simplement lue comme d'habitude.

`FOR UPDATE` ne peut pas être utilisé dans les contextes où les lignes renvoyées ne peuvent pas être clairement identifiées avec des lignes d'une table individuelle ; par exemple, elle ne peut pas être utilisée avec des agrégats.

Il est possible pour une commande `SELECT` utilisant les deux clauses `LIMIT` et `FOR UPDATE` de renvoyer moins de lignes que celles spécifiées par `LIMIT`. Ceci est dû au fait que `LIMIT` sélectionne un nombre de lignes mais pourrait ensuite bloquer lors de la demande d'un verrou `FOR UPDATE`. Une fois que le `SELECT` est débloqué, la qualification de la requête pourrait ne pas être rencontrée et la ligne ne serait pas renvoyée par le `SELECT`.

`FOR UPDATE` peut apparaître avant `LIMIT` pour la compatibilité avec les versions de PostgreSQL antérieures à la 7.3. Il s'exécute néanmoins après `LIMIT`, et c'est donc l'emplacement recommandée pour l'écrire.

# Exemples

Pour joindre la table `films` avec la table `distributeurs` :

```
SELECT f.titre, f.did, d.nom, f.date_prod, f.genre
    FROM distributeurs d, films f
    WHERE f.did = d.did

       titre       | did |     nom      | date_prod  |   genre
-------------------+-----+--------------+------------+------------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drame
 The African Queen | 101 | British Lion | 1951-08-11 | Romantique
 ...
```

Pour additionner la colonne `longueur` de tous les films, grouper les résultats par `genre` :

```
SELECT genre, sum(longueur) AS total FROM films GROUP BY genre;

   genre    | total
------------+-------
 Action     | 07:34
 Comédie    | 02:58
 Drame      | 14:28
 Musical    | 06:42
 Romantique | 04:38
```

Pour additionner la colonne `longueur` de tous les films, grouper les résultats par `genre` et afficher les groupes dont les totaux font moins de cinq heures :

```
SELECT genre, sum(longueur) AS total
    FROM films
    GROUP BY genre
    HAVING sum(longueur) < interval '5 hours';

   genre    | total
------------+-------
 Comedie    | 02:58
 Romantique | 04:38
```

Les deux exemples suivants sont des façons identiques de trier les résultats individuels suivant le contenu de la deuxième colonne (`nom`) :

```
SELECT * FROM distributeurs ORDER BY nom;
SELECT * FROM distributeurs ORDER BY 2;

 did |       nom
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

Le prochain exemple montre comment obtenir l'union des tables `distributeurs` et `acteurs`, restreignant les résultats à ceux commençant avec la lettre W dans chaque table. Seules les lignes distinctes sont voulues, du coup le mot clé `ALL` est omis.

```
distributeurs:               acteurs:
 did |     nom               id |     nom
-----+--------------        ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                          ...

SELECT distributeurs.nom
    FROM distributeurs
    WHERE distributeurs.nom LIKE 'W%'
UNION
SELECT actors.nom
    FROM acteurs
    WHERE acteurs.nom LIKE 'W%';

      nom
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

Cet exemple montre comment utiliser une fonction dans la clause `FROM`, à la fois avec et sans une liste de définition de colonnes :

```
CREATE FUNCTION distributeurs(int) RETURNS SETOF distributeurs AS $$
    SELECT * FROM distributeurs WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributeurs(111);
 did |    name
```

```
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributeurs_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributeurs WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributeurs_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

# Compatibilité

Bien sûr, l'instruction `SELECT` est compatible avec le standard SQL. Mais il y a des extensions et quelques fonctionnalités manquantes.

## Clauses `FROM` omises

PostgreSQL vous permet d'omettre la clause `FROM`. Cela permet par exemple de calculer le résultat d'expressions simples :

```
SELECT 2+2;

 ?column?
----------
        4
```

D'autre bases de données SQL ne le permettent pas, sauf en introduisant une table d'une seule ligne à partir de laquelle la commande `SELECT` s'exécute.

Une utilisation moins évidente est de raccourcir un `SELECT` normal à partir des tables :

```
SELECT distributeurs.* WHERE distributeurs.name = 'Westward';

 did |   name
-----+----------
 108 | Westward
```

Ceci fonctionne parce qu'un élément `FROM` implicite est ajouté pour chaque table référencée dans d'autres parties de l'instruction `SELECT` mais non mentionnée dans `FROM`.

Bien qu'il s'agisse d'un raccourci agréable, sa mauvaise utilisation est facile. Par exemple, la commande

```
SELECT distributeurs.* FROM distributeurs d;
```

est probablement une erreur ; il est probable que l'utilisateur souhaitait

```
SELECT d.* FROM distributeurs d;
```

plutôt que la jointure sans contrainte

```
SELECT distributeurs.* FROM distributeurs d, distributeurs distributeurs;
```

qu'il obtient réellement. Pour aider à la détection de ce type d'erreur, PostgreSQL avertit de l'utilisation de la fonctionnalité du `FROM` implicite dans une instruction `SELECT` qui contient aussi une clause `FROM` explicite. De plus, il est possible de désactiver la fonctionnalité du `FROM` implicite en initialisant le paramètre add_missing_from à false.

## Mot clé `AS`

Dans le SQL standard, le mot clé optionnel `AS` est seulement du bruit et peut être omis sans affecter la signification. L'analyseur PostgreSQL requiert ce mot clé lors du renommage des colonnes en sortie parce que les fonctionnalités d'extension du type créent des ambiguïtés en son absence. `AS` est néanmoins optionnel pour les éléments `FROM`.

## Espace logique disponible pour `GROUP BY` et `ORDER BY`

Dans le standard SQL–92, une clause `ORDER BY` peut seulement utiliser des noms ou des numéros de colonnes alors qu'une clause `GROUP BY` peut seulement utiliser des expressions basées sur les noms de colonne en entrée. PostgreSQL étend chacune de ces clauses pour permettre aussi un autre choix (mais il utilise l'interprétation du standard s'il y a ambiguïté). PostgreSQL autorise aussi les deux clauses à spécifier des expressions quelconques. Notez que les noms apparaissant dans une expression sont toujours pris en tant que noms des colonnes en entrée, et non pas en tant que noms des colonnes du résultat.

SQL:1999 utilise une définition légèrement différente qui n'est pas totalement compatible avec SQL–92. Néanmoins, dans la plupart des cas, PostgreSQL interpréte une expression `ORDER BY` ou `GROUP BY` de la même façon que ce que fait SQL:1999.

## Clauses non standard

Les clauses `DISTINCT ON`, `LIMIT` et `OFFSET` ne sont pas définies dans le standard SQL.

# SELECT INTO

## Nom

SELECT INTO −− définit une nouvelle table à partir des résultats d'une requête

## Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS nom_en_sortie ] [, ...]
    INTO [ TEMPORARY | TEMP ] [ TABLE ] nouvelle_table
    [ FROM élément_from [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC |
DESC | USING opérateur ] [, ...] ]
    [ LIMIT { nombre | ALL } ]
    [ OFFSET début ]
    [ FOR UPDATE [ OF nomtable
[, ...] ] ]
```

## Description

`SELECT INTO` crée une nouvelle table en la remplissant avec des données récupérées par une requête. Les données ne sont pas renvoyées au client comme le fait habituellement l'instruction `SELECT`. Les nouvelles colonnes de la table ont les noms et les types de données associés avec les colonnes en sortie du `SELECT`.

## Paramètres

`TEMPORARY` ou `TEMP`
: Si spécifié, la table est créée comme une table temporaire. Référez−vous à ***CREATE TABLE*** pour plus de détails.

*`new_table`*
: Le nom de la table à créer (pouvant être qualifié par le nom du schéma).

Tous les autres paramètres sont décrits en détail dans ***SELECT***.

## Notes

***CREATE TABLE AS*** est fonctionnellement équivalent à `SELECT INTO`. `CREATE TABLE AS` est la syntaxe recommandée car cette forme de `SELECT INTO` n'est pas disponible dans ECPG ou PL/pgSQL. En effet, ils interprètent la clause `INTO` différemment. De plus, `CREATE TABLE AS` offre un ensemble de fonctionnalités plus important que celui de `SELECT INTO`.

Avant PostgreSQL 8.0, la table créée par `SELECT INTO` incluait toujours des OID. À partir de PostgreSQL 8.0, l'ajout des OID dans la table créée par `SELECT INTO` est contrôlé par la variable de configuration default_with_oids. Cette variable est à true par défaut mais cette valeur pourrait changer dans une prochaine version de PostgreSQL.

# Exemples

Crée une nouvelle table `films_recent` ne contenant que les entrées récentes de la table `films`:

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

# Compatibilité

Le standard SQL utilise `SELECT INTO` pour représenter la sélection de valeurs dans des variables scalaires d'un programme hôte plutôt que la création d'une nouvelle table. Ceci est en fait l'utilisation trouvée dans ECPG (voir Chapitre 29) et dans PL/pgSQL (voir Chapitre 35). L'usage de PostgreSQL de `SELECT INTO` pour représenter une création de table est historique. Il est préférable d'utiliser `CREATE TABLE AS` dans un nouveau programme.

# Voir aussi

*CREATE TABLE AS*

# SET

## Nom

SET −− change un paramètre d'exécution

## Synopsis

```
SET [ SESSION | LOCAL ] nom { TO | = } { valeur | 'valeur' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { fuseau-horaire | LOCAL | DEFAULT }
```

## Description

La commande SET change des paramètres d'exécution. Un grand nombre de paramètres d'exécution listés dans Section 16.4 peut être modifié à la volée avec la commande SET. (Mais certains nécessitent d'être un superutilisateur pour les changer, et d'autres ne peuvent être changés après le démarrage du serveur ou de la session.) SET ne modifie que les paramètres utilisés par la session courante.

Si SET ou SET SESSION sont utilisés dans une transaction qui est ensuite abandonnée, les effets de la commande SET disparaissent lorsque la transaction est annulée. (Ce comportement a changé par rapport aux versions antérieures à la 7.3, pour lesquelles les effets de la commande SET n'étaient pas annulés dans ce cas.) Une fois que la transaction englobant la commande est validée, les effets de la commande persistent jusqu'à la fin de la session, à moins qu'ils ne soient annulés par une autre commande SET.

Les effets de SET LOCAL ne durent que jusqu'à la fin de la transaction en cours, qu'elle soit validée ou non. Dans le cas particulier d'une commande SET suivie par SET LOCAL dans une même transaction, la valeur de SET LOCAL est utilisée jusqu'à la fin de la transaction, et celle de SET prend effet ensuite (si la transaction est validée).

## Paramètres

SESSION
: Indique que la commande prend effet pour la session courante. C'est la valeur par défaut si ni SESSION ni LOCAL n'apparaissent.

LOCAL
: Indique que la commande n'est effective que pour la transaction courante. Après COMMIT ou ROLLBACK, la valeur utilisée pour la session redevient effective. Notez qu'une commande SET LOCAL semble sans effet si elle est exécutée en dehors d'un bloc BEGIN car la transaction prend immédiatement fin.

*nom*
: Nom d'un paramètre ajustable pendant l'exécution. La liste des paramètres disponibles est documentée dans Section 16.4 et ci−dessous.

*valeur*
: Nouvelle valeur du paramètre. Les valeurs peuvent être indiquées sous forme de constantes de chaîne, d'identifiants, de nombres ou de listes de ceux−ci, séparés par des virgules. DEFAULT peut être utilisé

pour remettre le paramètre à sa valeur par défaut.

En plus des paramètres de configuration documentés dans Section 16.4, il y en a quelques autres qui ne peuvent être utilisés qu'avec la commande `SET` ou qui ont une syntaxe spéciale.

`NAMES`
: `SET NAMES` *`valeur`* est un équivalent de `SET client_encoding TO` *`valeur`*.

`SEED`
: Donne une valeur initiale au générateur de nombres aléatoires (la fonction `random`). Les valeurs autorisées sont des nombres à virgule flottante entre 0 et 1, qui sont ensuite multipliés par $2^{31}-1$.

  Le générateur de nombres aléatoires peut aussi être initialisé en appelant la fonction `setseed` :

```
SELECT setseed(valeur);
```

`TIME ZONE`
: `SET TIME ZONE` *`valeur`* est équivalent à `SET timezone TO` *`valeur`*. La syntaxe `SET TIME ZONE` permet d'utiliser une syntaxe spéciale pour indiquer le fuseau horaire. Voici des exemples de valeurs valides :

  `'PST8PDT'`
  : Le fuseau horaire de Berkeley, Californie.

  `'Europe/Rome'`
  : Le fuseau horaire de l'Italie.

  `-7`
  : Le fuseau horaire situé 7 heures à l'ouest de l'UTC (équivalent à PDT). Les valeurs positives sont à l'est de l'UTC.

  `INTERVAL '-08:00' HOUR TO MINUTE`
  : Le fuseau horaire situé 8 heures à l'ouest de l'UTC (équivalent à PST).

  `LOCAL`
  `DEFAULT`
  : Utilise le fuseau horaire local (celui du système d'exploitation du serveur).

  Voir Section 8.5 pour de plus amples informations sur les fuseaux horaires. De même, Annexe B a une liste des noms reconnus pour les fuseaux horaires.

# Notes

La fonction `set_config` propose des fonctionnalités équivalentes. Voir Section 9.20.

# Exemples

Mettre à jour le chemin de recherche :

```
SET search_path TO my_schema, public;
```

Utiliser le style de date traditionnel POSTGRES avec comme convention de saisie << les jours avant les mois >> :

```
SET datestyle TO postgres, dmy;
```

Utiliser le fuseau horaire de Berkeley, Californie :

```
SET TIME ZONE 'PST8PDT';
```

Utiliser le fuseau horaire de l'Italie :

```
SET TIME ZONE 'Europe/Rome';
```

# Compatibilité

`SET TIME ZONE` étend la syntaxe définie dans le standard SQL. Le standard ne permet que des fuseaux horaires numériques alors que PostgreSQL est plus souple dans les syntaxes acceptées. Toutes les autres fonctionnalités de `SET` sont des extensions de PostgreSQL.

# Voir aussi

*RESET*, *SHOW*

# SET CONSTRAINTS

## Nom

SET CONSTRAINTS −− initialise le mode de vérification de contrainte de la transaction en cours

## Synopsis

```
SET CONSTRAINTS { ALL | nom [, ...] } { DEFERRED | IMMEDIATE }
```

## Description

`SET CONSTRAINTS` initialise le comportement de la vérification des contraintes dans la transaction en cours. Les contraintes `IMMEDIATE` sont vérifiées à la fin de chaque instruction. Les contraintes `DEFERRED` ne sont vérifiées qu'à la validation de la transaction. Chaque contrainte a son propre mode `IMMEDIATE` ou `DEFERRED`.

À la création, une contrainte se voit donnée une des trois caractéristiques : `DEFERRABLE INITIALLY DEFERRED`, `DEFERRABLE INITIALLY IMMEDIATE` ou `NOT DEFERRABLE`. La troisième classe est toujours `IMMEDIATE` et n'est pas affectée par la commande `SET CONSTRAINTS`. Les deux premières classes commencent chaque transaction dans le mode indiqué mais leur comportement peut changer à l'intérieur d'une transaction par `SET CONSTRAINTS`.

`SET CONSTRAINTS` avec une liste de noms de contraintes modifie le mode de ces contraintes (qui doivent toutes être différables). S'il existe plusieurs contraintes correspondant à un nom donné, elles sont toutes affectées. `SET CONSTRAINTS ALL` modifie le mode de toutes les contraintes différables.

Lorsque `SET CONSTRAINTS` modifie le mode d'une contrainte de `DEFERRED` à `IMMEDIATE`, le nouveau mode prend effet rétroactivement : toute modification de données qui aurait été vérifiée à la fin de la transaction est en fait vérifiée lors de l'exécution de la commande `SET CONSTRAINTS`. Si une contrainte est violée, la commande `SET CONSTRAINTS` échoue (et ne change pas le mode de contrainte). Du coup, `SET CONSTRAINTS` peut être utilisée pour forcer la vérification de contraintes à un point spécifique d'une transaction.

Actuellement, seules les contraintes de clés étrangères sont affectées par ce paramétrage. Les contraintes de vérification et les contraintes uniques sont toujours immédiates.

## Notes

Cette commande altère seulement le comportement des contraintes à l'intérieur de la transaction en cours. Du coup, si vous exécutez cette commande en dehors d'un bloc de transaction (pair `BEGIN`/`COMMIT`), elle ne semble pas avoir d'effet.

# Compatibilité

Cette commande est compatible avec le comportement défini par le standard SQL à part le fait que dans PostgreSQL, elle ne s'applique qu'aux contraintes de clés étrangères.

Le standard SQL indique que les noms de contraintes apparaissant dans `SET CONSTRAINTS` peut être qualifiés avec le nom du schéma. Ceci n'est pas encore supporté par PostgreSQL : les noms ne doivent pas être qualifiés et toutes les contraintes correspondant à la commande sont affectées quelque soit le schéma où elles se trouvent.

# SET SESSION AUTHORIZATION

## Nom

SET SESSION AUTHORIZATION −− Initialise l'identifiant de l'utilisateur de la session et l'identifiant de l'utilisateur courant de la session courante

## Synopsis

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION utilisateur
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

## Description

Cette commande initialise l'identifiant de l'utilisateur de la session et l'identifiant de l'utilisateur courant pour le contexte courant de la session SQL comme étant *utilisateur*. Le nom de l'utilisateur pourrait être soit un identifiant soit une chaîne littérale. En utilisant cette commande, il est possible, par exemple, de devenir temporairement un utilisateur non privilégié et de redevenir plus tard un superutilisateur.

L'identifiant de l'utilisateur de la session est initialement configuré pour être le nom utilisateur fourni par le client (que s'est éventuellement authentifié). L'identifiant de l'utilisateur courant est normalement identique à l'identifiant de l'utilisateur de la session mais cela pourrait changer temporairement dans le contexte des fonctions << setuid >> et des mécanismes similaires. L'identifiant de l'utilisateur courant est important pour la vérification des droits.

L'identifiant de l'utilisateur de la session pourrait être ne peut être modifié que si l'utilisateur initial de la session (l'*utilisateur authentifié*) a le droit de superutilisateur. Sinon, la commande est acceptée seulement si le nom de l'utilisateur authentifié est spécifié.

Les modificateurs `SESSION` et `LOCAL` agissent de la même façon que la commande standard ***SET***.

Les formes `DEFAULT` et `RESET` réinitialisent les identifiant de l'utilisateur de la session et de l'utilisateur courant à ceux de l'utilisateur originellement authentifié. Ces formes peuvent être exécutées par tout utilisateur.

## Exemples

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;
```

```
 session_user | current_user
--------------+--------------
 paul         | paul
```

## Compatibilité

Le standard SQL autorise l'apparition de quelques autres expressions à la place du littéral *utilisateur* qui ne sont pas importantes en pratique. PostgreSQL autorise la syntaxe de l'identifiant (`"utilisateur"`) alors que SQL ne le permet pas. SQL n'autorise pas cette commande lors d'une transaction ; PostgreSQL ne fait pas cette restriction parce qu'il n'y a aucune raison de le faire. Les droits nécessaires pour exécuter cette commande sont laissés à la définition de l'implémentation par le standard.

# SET TRANSACTION

## Nom

SET TRANSACTION −− initialise les caractéristiques de la transaction actuelle

## Synopsis

```
SET TRANSACTION mode_transaction [, ...]
SET SESSION CHARACTERISTICS AS TRANSACTION mode_transaction [, ...]

où mode_transaction fait partie
de :

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
```

## Description

La commande `SET TRANSACTION` initialise les caractéristiques de la transaction actuelle. Il n'a aucun effet sur les transactions suivantes. `SET SESSION CHARACTERISTICS` initialise les caractéristiques de transaction par défaut pour les prochaines transactions d'une session. Ces valeurs peuvent être surchargées par `SET TRANSACTION` pour une transaction individuelle.

Les caractéristiques disponibles de la transaction sont le niveau d'isolation de la transaction et le mode d'accès de la transaction (lecture/écriture ou lecture seule).

Le niveau d'isolation d'une transaction détermine les données que la transaction peut voir quand d'autres transactions fonctionnent en même temps :

`READ COMMITTED`
: Une instruction peut seulement voir les lignes validées avant qu'elle ne commence. C'est la valeur par défaut.

`SERIALIZABLE`
: Toutes les instructions de la transaction en cours peuvent seulement voir les lignes validées avant que la première requête ou la première instruction de modification de données ait été exécutée dans cette transaction.

Le standard SQL définit deux niveaux supplémentaires, `READ UNCOMMITTED` et `REPEATABLE READ`. Dans PostgreSQL, `READ UNCOMMITTED` est traité comme `READ COMMITTED` alors que `REPEATABLE READ` est traité comme `SERIALIZABLE`.

Le niveau d'isolation de la transaction ne peut pas être modifié après que la première requête ou la première instruction de modification de données (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH` ou `COPY`) d'une transaction ait été exécutée. Voir Chapitre 12 pour plus d'informations sur l'isolation de la transaction et le contrôle de concurrence.

Le mode d'accès de la transaction détermine si la transaction est en lecture/écriture ou en lecture seule. Lecture/écriture est la valeur par défaut. Quand une transaction est en lecture seule, les commandes SQL suivantes sont interdites : `INSERT`, `UPDATE`, `DELETE` et `COPY TO` si la table qu'elles modifieraient n'est pas une table temporaire ; toutes les commandes `CREATE`, `ALTER` et `DROP` ; `COMMENT`, `GRANT`, `REVOKE`, `TRUNCATE` ; `EXPLAIN ANALYZE` et `EXECUTE` si la commande qu'ils exécuteraient est parmi celles listées. Ceci est une notion de haut niveau de lecture seule qui n'empêche pas toutes les écritures sur disque.

# Notes

Si `SET TRANSACTION` est exécuté sans un `START TRANSACTION` ou un `BEGIN` le précédant, il est sans effet car la transaction se termine immédiatement.

Il est possible de compenser `SET TRANSACTION` en spécifiant le *`mode_transaction`* désiré dans `BEGIN` ou `START TRANSACTION`.

Les modes de transaction par défaut d'une session peuvent aussi être configurés en initialisant les paramètres de configuration default_transaction_isolation et default_transaction_read_only. (En fait, `SET SESSION CHARACTERISTICS` est un équivalent verbeux de la configuration de ces variables avec `SET`.) Ceci signifie que les valeurs par défaut peuvent être initialisées dans le fichier de configuration, via `ALTER DATABASE`, etc. Consultez Section 16.4 pour plus d'informations.

# Compatibilité

Les deux commandes sont définies dans le standard SQL. `SERIALIZABLE` est le niveau d'isolation par défaut de la transaction dans le standard. Dans PostgreSQL, la valeur par défaut est habituellement `READ COMMITTED`, mais vous pouvez la modifier comme mentionné ci−dessus. À cause d'un manque de verrou du prédicat, le niveau `SERIALIZABLE` n'est pas vraiment sérialisable. Voir Chapitre 12 pour plus de détails.

Dans le standard SQL, il existe une autre caractéristique de transaction pouvant être configurée avec ces commandes : la taille de l'aire de diagnostique. Ce concept est seulement à utiliser en SQL embarqué et, du coup, n'est pas implémenté dans le serveur PostgreSQL.

Le standard SQL requiert des virgules entre chaque *`mode_transaction`* mais, pour des raisons historiques, PostgreSQL autorise l'omission des virgules.

# SHOW

## Nom

SHOW −− affiche la valeur d'un paramètre d'exécution

## Synopsis

```
SHOW nom
SHOW ALL
```

## Description

`SHOW` affiche la configuration courante des paramètres d'exécution. Ces variables peuvent être initialisées en utilisant l'instruction `SET`, en éditant le fichier de configuration `postgresql.conf`, via la variable d'environnement `PGOPTIONS` (lors de l'utilisation de libpq ou en utilisant une application basée sur libpq), ou via des options en ligne de commande en lançant `postmaster`. Voir Section 16.4 pour plus de détails.

## Paramètres

*nom*

Le nom d'un paramètre d'exécution. Les paramètres disponibles sont documentés dans Section 16.4 et sur la page de référence *SET*. De plus, il existe quelques paramètres qui peuvent être affichés mais ne sont pas initialisables :

`SERVER_VERSION`
Affiche le numéro de version du serveur.

`SERVER_ENCODING`
Affiche le codage de l'ensemble de caractères du côté serveur. À présent, ce paramètre peut être affiché mais non initialisé parce que le codage est déterminé au moment de la création de la base de données.

`LC_COLLATE`
Affiche le paramètre locale de la base de données pour le tri du texte. À présent, ce paramètre est affichable mais ne peut pas être initialisé parce que la configuration est déterminée lors de l'`initdb`.

`LC_CTYPE`
Affiche le paramètre locale de la base de données pour la classification des caractères. À présent, ce paramètre peut être affiché mais non initialisé parce que la configuration est déterminée lors de l'`initdb`.

`IS_SUPERUSER`
Vrai si l'identifiant d'autorisation de la session actuelle a des droits de superutilisateur.

`ALL`

Affiche les valeurs de tous les paramètres de configuration.

# Notes

La fonction `current_setting` produit une sortie équivalente. Voir Section 9.20.

# Exemples

Affiche la configuration actuelle du paramètre `DateStyle` :

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

Affiche la configuration actuelle du paramètre `geqo` :

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

Affiche tous les paramètres :

```
SHOW ALL;
              name               |                        setting
---------------------------------+-----------------------------------------------
 add_missing_from                | on
 archive_command                 | unset
 australian_timezones            | off
    .
    .
    .
 work_mem                        | 1024
 zero_damaged_pages              | off
(140 rows)
```

# Compatibilité

La commande `SHOW` est une extension PostgreSQL.

# Voir aussi

*SET*, *RESET*

# START TRANSACTION

## Nom

START TRANSACTION −− débute un bloc de transaction

## Synopsis

```
START TRANSACTION [ mode_transaction [, ...] ]

où mode_transaction fait partie
de :

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
```

## Description

Cette commande lance un nouveau bloc de transaction. Si le niveau d'isolation ou un mode lecture/écriture est spécifié, la nouvelle transaction a ces caractéristiques, comme si *SET TRANSACTION* avait été exécuté. Elle est identique à la commande *BEGIN*.

## Paramètres

Référez−vous à *SET TRANSACTION* pour la signification des paramètres de cette instruction.

## Compatibilité

Dans le standard, il n'est pas nécessaire de lancer `START TRANSACTION` pour commencer un bloc de transaction : toute commande SQL commence implicitement un bloc. Le comportement de PostgreSQL peut être vu comme lançant implicitement un `COMMIT` après chaque commande non précédée de `START TRANSACTION` (ou `BEGIN`). C'est pourquoi ce comportement est souvent appelé << autocommit >>. D'autres systèmes de bases de données relationnelles pourraient peuvent offrir une fonctionnalité de validation automatique optionnelle.

Le standard SQL requiert des virgules entre les *modes_transaction* successifs mais, pour des raisons historiques, PostgreSQL autorise l'omission des virgules.

Voir aussi la section de compatibilité de *SET TRANSACTION*.

# Voir aussi

*BEGIN*, *COMMIT*, *ROLLBACK*, *SAVEPOINT*, *SET TRANSACTION*

# TRUNCATE

## Nom

TRUNCATE −− vide une table

## Synopsis

```
TRUNCATE [ TABLE ] nom
```

## Description

La commande `TRUNCATE` supprime rapidement toutes les lignes d'une table. Elle a le même effet qu'un `DELETE` non qualifié mais, comme elle ne parcourt par la table, elle est plus rapide. Cette commande est principalement utile pour les grandes tables.

## Paramètres

*name*
: Le nom de la table à tronquer (pouvant être qualifié avec le schéma).

## Notes

Seul le propriétaire d'une table peut la tronquer (`TRUNCATE`).

`TRUNCATE` ne peut pas être utilisé s'il existe des références de clés étrangères de la table vers d'autres tables. Dans de tels cas, la vérification nécessiterait des parcours complets de tables, ce qui n'est pas le but de la commande `TRUNCATE`.

`TRUNCATE` ne déclenche pas les déclencheurs `ON DELETE` définis par l'utilisateur qui pourraient exister sur la table.

## Exemples

Tronquer la table `grossetable` :

```
TRUNCATE TABLE grossetable;
```

# Compatibilité

Il n'existe pas de commande `TRUNCATE` dans le standard SQL.

# UNLISTEN

## Nom

UNLISTEN −− arrête l'écoute d'une notification

## Synopsis

```
UNLISTEN { nom | * }
```

## Description

`UNLISTEN` est utilisé pour supprimer un abonnement existant aux événements `NOTIFY`. `UNLISTEN` annule tout abonnement pour la session PostgreSQL en cours sur la notification *`nom`*. Le caractère joker `*` annule tous les abonnements de la session en cours.

*NOTIFY* contient une discussion plus complète de l'utilisation de `LISTEN` et de `NOTIFY`.

## Paramètres

*`nom`*
: Nom d'une notification (un identificateur quelconque).

`*`
: Tous les abonnements de cette session sont annulés.

## Notes

Vous pouvez vous désabonner de quelque chose auquel vous n'êtes pas abonné ; vous n'obtenez aucun message d'avertissement ou d'erreur.

À la fin de chaque session, `UNLISTEN *` est exécuté automatiquement.

## Exemples

Pour s'abonner :

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Une fois que `UNLISTEN` a été exécuté, les commandes `NOTIFY` suivantes sont ignorées :

```
UNLISTEN virtual;
```

```
NOTIFY virtual;
-- aucun événement NOTIFY n'est reçu
```

## Compatibilité

Il n'y a pas de commande `UNLISTEN` dans le standard SQL.

## Voir aussi

*LISTEN*, *NOTIFY*

# UPDATE

## Nom

UPDATE −− met à jour les lignes d'une table

## Synopsis

```
UPDATE [ ONLY ] table SET colonne = { expression | DEFAULT } [, ...]
    [ FROM liste_from ]
    [ WHERE condition ]
```

## Description

`UPDATE` modifie les valeurs des colonnes spécifiées pour toutes les lignes qui satisfont la condition. Seules les colonnes à modifier doivent être mentionnées dans la clause `SET` ; Les autres colonnes conservent leur précédente valeur.

Par défaut, `UPDATE` met à jour les lignes de la table spécifiée et toutes ses sous−tables. Si vous souhaitez ne mettre à jour que la table spécifique mentionnée, vous pouvez utiliser la clause `ONLY`.

Il existe deux façons de modifier une table en utilisant les informations contenues dans d'autres tables de la base de données : en utilisant des sous−requêtes ou en spécifiant des tables supplémentaires dans la clause `FROM`. La technique la plus appropriée dépend des circonstances spécifiques.

Vous devez avoir le droit `UPDATE` sur la table pour la mettre à jour, ainsi que le droit `SELECT` sur toutes les tables dont les valeurs sont lues dans *`expression`* ou *`condition`*.

## Paramètres

*`table`*
: Le nom de la table à mettre à jour (pouvant être qualifié du nom du schéma).

*`colonne`*
: Le nom d'une colonne dans *`table`*. Le nom de la colonne peut être qualifié avec un nom de sous−champ ou un indice de tableau si nécessaire.

*`expression`*
: Une expression à affecter à la colonne. L'expression peut utiliser les anciennes valeurs de cette colonne ou d'autres colonnes de la table.

`DEFAULT`
: Initialise sa colonne à la valeur par défaut (qui vaut NULL si aucune expression spécifique par défaut ne lui a été affectée).

*`liste_from`*
: Une liste d'expressions de tables, permettant aux colonnes des autres tables d'apparaître dans la condition `WHERE` et les expressions de mise à jour. Ceci est similaire à la liste de tables pouvant être spécifiée dans ***Clause FROM*** d'une instruction `SELECT`. Notez que la table cible ne doit pas

apparaître dans *liste_from*, sauf si vous comptez faire une auto−jointure (auquel cas elle doit apparaître avec un alias dans *liste_from*).

*condition*

Une expression qui renvoie une valeur de type boolean. Seules les lignes pour lesquelles cette expression renvoie true sont mises à jour.

# Sorties

En cas de succès, une commande UPDATE renvoie un message de la forme

```
UPDATE total
```

Le *total* est le nombre de lignes mises à jour. Si *total* vaut 0, aucune ligne ne correspond à *condition* (ceci n'est pas considéré comme une erreur).

# Notes

Quand une clause FROM est présente, la table cible est jointe aux tables mentionnées dans *liste_from*, et chaque ligne en sortie de la jointure représente une opération de mise à jour pour la table cible. Lors de l'utilisation de FROM, vous devriez vous assurer que la jointure produit au moins une ligne en sortie pour chaque ligne à modifier. En d'autres termes, une ligne cible ne doit pas être jointe plus d'une fois à une ligne d'une autre table. Si c'est le cas, alors seulement une des lignes de jointures est utilisée pour mettre à jour la ligne cible mais celle qui est utilisée n'est pas prévisible.

À cause de ce manque de déterminisme, il est plus sûr de ne référencer d'autres tables qu'à l'intérieur de sous−requêtes, même si c'est plus difficile à lire et plus lent que l'utilisation d'une jointure.

# Exemples

Modifie le mot Drame en Dramatique dans la colonne genre de la table films :

```
UPDATE films SET genre = 'Dramatique' WHERE genre = 'Drame';
```

Ajuste les entrées de température et réinitialise la précipitation à sa valeur par défaut dans une ligne de la table temps :

```
UPDATE temps SET temp_basse = temp_basse+1, temp_haute = temp_basse+15, prcp =
DEFAULT
  WHERE ville = 'San Francisco' AND date = '2003-07-03';
```

Incrémente le total des ventes de la personne qui gère le compte d'Acme Corporation, en utilisant la syntaxe de la clause du FROM :

```
UPDATE employes SET total_ventes = total_ventes + 1 FROM comptes
  WHERE compte.nom = 'Acme Corporation'
  AND employes.id = compte.vendeur;
```

Réalise la même opération en utilisant une sous−requête dans la clause WHERE :

```
UPDATE employes SET total_ventes = total_ventes + 1 WHERE id =
  (SELECT vendeur FROM comptes WHERE nom = 'Acme Corporation');
```

Tente d'insérer un nouvel élément dans le stock avec la quantité. Si l'élément existe déjà, à la place, met à jour le total du stock de l'élément existant. Pour faire cela sans échec dans la transaction entière, utilise les points de sauvegarde.

```
BEGIN;
-- autres opérations
SAVEPOINT sp1;
INSERT INTO vins VALUES('Chateau Lafite 2003', '24');
-- Suppose que l'instruction ci-dessus échoue à cause d'une violation de clé
-- unique, donc nous lançons maintenant ces commandes:
ROLLBACK TO sp1;
UPDATE vins SET stock = stock + 24 WHERE nomvin = 'Chateau Lafite 2003';
-- continue avec les autres opérations, et enfin
COMMIT;
```

# Compatibilité

Cette commande est conforme au standard SQL, à l'exception de la clause `FROM` qui est une extension PostgreSQL.

D'autres systèmes de bases de données offrent une option `FROM` dans laquelle la table cible est supposée être de nouveau indiquée dans le `FROM`. Ce n'est pas ainsi que PostgreSQL interprète `FROM`. Faites attention pour le portage d'applications utilisant cette extension.

# VACUUM

## Nom

VACUUM −− récupère l'espace inutilisé et, optionnellement, analyse une base

## Synopsis

```
VACUUM [ FULL | FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL | FREEZE ] [ VERBOSE ] ANALYZE [ table [ (colonne [, ...] ) ] ]
```

## Description

VACUUM récupère l'espace de stockage occupé par des lignes supprimées. Lors des opérations normales de PostgreSQL, les lignes supprimées ou rendues obsolètes par une mise à jour ne sont pas physiquement supprimées de leur table. Elles restent présentes jusqu'à ce qu'un VACUUM soit lancé. C'est pourquoi, il est nécessaire de faire un VACUUM régulièrement, spécialement sur les tables fréquemment mises à jour.

Sans paramètre, VACUUM traite toutes les tables de la base de données courante. Avec un paramètre, VACUUM ne traite que cette table.

VACUUM ANALYZE fait un VACUUM, puis un ANALYZE sur chaque table sélectionnée. C'est une combinaison pratique pour les scripts de maintenance de routine. Voir *ANALYZE* pour avoir plus de détails sur ce qu'il traite.

Le VACUUM standard (sans FULL) récupère simplement l'espace et le rend disponible pour une réutilisation. Cette forme de la commande peut opérer en parallèle avec les opérations normales de lecture et d'écriture de la table, car elle n'utilise pas de verrou exclusif. VACUUM FULL fait un traitement plus complet et, en particulier, déplace des lignes dans d'autres blocs pour compacter la table au maximum sur le disque. Cette forme est beaucoup plus lente et pose un verrou exclusif sur la table pour faire son traitement.

FREEZE est une option particulière qui déclare les lignes comme << gelées >> aussi vite que possible, au lieu d'attendre qu'elles soient assez âgées. Si c'est fait à un moment où il n'y a aucune autre transaction ouverte sur la base, alors il est garanti que toutes les lignes de cette base sont << gelées >> et ne seront pas sujettes à des problèmes de dépassement d'ID de transaction, même si la base est laissée très longtemps sans vacuum. FREEZE n'est pas recommandé en usage normal. Il est seulement destiné à faire partie de la préparation de modèles de bases de données définis par l'utilisateur ou pour d'autres bases de données qui sont exclusivement en lecture seule et ne subiront pas de VACUUM de maintenance régulière. Voir Chapitre 21 pour plus de détail.

## Paramètres

FULL
: Choisit un vacuum << full >>, qui récupère plus d'espace, mais est beaucoup plus long et prend un verrou exclusif sur la table.

FREEZE

Choisit un << gel >> agressif des lignes.

`VERBOSE`
: Affiche un rapport détaillé de l'activité de vacuum sur chaque table.

`ANALYZE`
: Met à jour les statistiques utilisées par l'optimiseur pour déterminer la méthode la plus efficace pour exécuter une requête.

*`table`*
: Le nom (optionnellement qualifié par le nom d'un schéma) d'une table à traiter par vacuum. Par défaut, toutes les tables de la base de données courante sont traitées.

*`colonne`*
: Le nom d'une colonne spécifique à analyser. Par défaut, toutes les colonnes.

## Sorties

Lorsque `VERBOSE` est précisé, `VACUUM` indique sa progression par des messages indiquant la table en cours de traitement. Différentes statistiques sur les tables sont aussi affichées.

## Notes

Nous recommandons que les bases de données actives de production soient traitées par vacuum fréquemment (au moins toutes les nuits), pour supprimer les lignes expirées. Après avoir ajouté ou supprimé un grand nombre de lignes, il peut être utile de faire un `VACUUM ANALYZE` sur la table affectée. Cela met les catalogues système à jour de tous les changements récents et permet à l'optimiseur de requêtes de PostgreSQL de faire de meilleurs choix lors de l'optimisation des requêtes.

L'option `FULL` n'est pas recommandée en usage normal, mais elle peut être utile dans certains cas. Par exemple, si vous avez supprimé l'essentiel des lignes d'une table et si vous voulez que la table diminue physiquement sur le disque pour n'occuper que l'espace réellement nécessaire. Généralement, `VACUUM FULL` réduit plus la table qu'un simple `VACUUM`.

## Exemples

Ce qui suit est un exemple de lancement de `VACUUM` sur une table de la base de données regression.

```
regression=# VACUUM VERBOSE ANALYZE onek;
INFO:  vacuuming "public.onek"
INFO:  index "onek_unique1" now contains 1000 tuples in 14 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.18 sec.
INFO:  index "onek_unique2" now contains 1000 tuples in 16 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.07u sec elapsed 0.23 sec.
INFO:  index "onek_hundred" now contains 1000 tuples in 13 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.17 sec.
INFO:  index "onek_stringu1" now contains 1000 tuples in 48 pages
```

```
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.09u sec elapsed 0.59 sec.
INFO:  "onek": removed 3000 tuples in 108 pages
DETAIL:  CPU 0.01s/0.06u sec elapsed 0.07 sec.
INFO:  "onek": found 3000 removable, 1000 nonremovable tuples in 143 pages
DETAIL:  0 dead tuples cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.07s/0.39u sec elapsed 1.56 sec.
INFO:  analyzing "public.onek"
INFO:  "onek": 36 pages, 1000 rows sampled, 1000 estimated total rows
VACUUM
```

# Compatibilité

Il n'y a pas de commande `VACUUM` dans le standard SQL.

# Voir aussi

*vacuumdb*

# II. Applications clientes de PostgreSQL

Cette partie contient les informations de référence pour les applications clientes et les outils de PostgreSQL. Ces commandes ne sont pas toutes destinées à l'ensemble des utilisateurs. Certaines nécessitent des privilèges spécifiques. La caractéristique commune à toutes ces applications est qu'elles peuvent fonctionner sur n'importe quelle machine, en toute indépendance vis−à−vis du serveur sur lequel se trouve le serveur de base de données.

**Table des matières**

# clusterdb

## Nom

clusterdb −− groupe les bases de données de PostgreSQL

## Synopsis

`clusterdb` [*`options_connexion`*...] [−−table | −t *`table`* ] [*`nom_db`*]
`clusterdb` [*`options_connexion`*...] [−−all | −a]

## Description

clusterdb est un outil de regroupement des tables dans une base de données PostgreSQL. Il trouve des tables qui ont été précédemment groupées et les regroupe encore une fois sur le même index qui a été utilisé précédemment. Les tables qui n'ont jamais été groupées n'ont pas été affectées.

clusterdb est un emballage autour de la commande SQL ***CLUSTER***. Il n'y a pas de différence réelle entre les bases de données groupées via cet outil et via d'autres méthodes pour accéder au serveur.

## Options

clusterdb accepte les arguments suivants en ligne de commande :

`-a`
`--all`
: Groupe toutes les bases de données.

`[-d]` *`nom_db`*
`[--dbname]` *`nom_db`*
: Spécifie le nom de la base de données à grouper. S'il n'est pas spécifié et `-a` (ou `--all`) n'est pas utilisé, le nom de la base de données est lu à partir de la variable d'environnement `PGDATABASE`. Si elle n'est pas configurée, le nom de l'utilisateur spécifié pour la connexion est utilisé.

`-e`
`--echo`
: Affiche les commandes que clusterdb génère et envoie au serveur.

`-q`
`--quiet`
: N'affiche aucune réponse.

`-t` *`table`*
`--table` *`table`*
: Groupe uniquement *`table`*.

clusterdb accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *hôte*
`--host` *hôte*
: Spécifie le nom de l'hôte de la machine sur lequel le serveur fonctionne. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *port*
`--port` *port*
: Spécifie le port TCP ou l'extension du fichier du socket local de domaine Unix sur lequel le serveur attend les connexions.

`-U` *nomutilisateur*
`--username` *nomutilisateur*
: Nom de l'utilisateur qui se connecte

`-W`
`--password`
: Force la demande du mot de passe.

# Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`
: Paramètres de connexion par défaut

# Diagnostiques

En cas de difficulté, voir *CLUSTER* et psql pour des discussions sur les problèmes et messages d'erreur potentiels. Le serveur de la base de données doit être en cours d'exécution sur l'hôte cible. De plus, toutes les configurations de connexion par défaut et variables d'environnement utilisées par la bibliothèque libpq seront appliquées.

# Exemples

Pour grouper la base de données `test` :

```
$ clusterdb test
```

Pour grouper une seule table `foo` dans une base de données nommée `xyzzy` :

```
$ clusterdb --table foo xyzzy
```

# Voir aussi

*CLUSTER*

# createdb

## Nom

createdb −− crée une nouvelle base de données PostgreSQL

## Synopsis

`createdb` [*option*...] [*nombase*] [*description*]

## Description

createdb crée une nouvelle base de données PostgreSQL.

Normalement, l'utilisateur de la base de données qui exécute cette commande devient le propriétaire de la nouvelle base de données. Néanmoins, un propriétaire différent peut être spécifié via l'option `-O` si l'utilisateur exécutant a les droits appropriés.

createdb est un emballage autour de la commande SQL *CREATE DATABASE*. Il n'y a pas de différence effective entre la création de bases de données via cet outil et d'autres méthodes pour accéder au serveur.

## Options

createdb accepte les arguments suivant en ligne de commande :

*nombase*
: Spécifie le nom de la base de données à créer. Le nom doit être unique parmi toutes les bases de données PostgreSQL de ce groupe. La valeur par défaut est de créer une base de données avec le même nom que l'utilisateur système actuel.

*description*
: Spécifie un commentaire à associer avec la base de données nouvellement créée.

`-D` *espace_logique*
`--location` *espace_logique*
: Spécifie l'espace logique par défaut de la base de données.

`-e`
`--echo`
: Affiche les commandes que createdb génère et envoie au serveur.

`-E` *locale*
`--encoding` *locale*
: Spécifie le codage des caractères à utiliser dans cette base de données. Les ensembles de caractères supportés par le serveur PostgreSQL sont décrits dans Section 20.2.1.

`-O` *propriétaire*
`--owner` *propriétaire*
: Spécifie le propriétaire de la base de données.

`-q`
`--quiet`
: N'affiche pas de réponse.

`-T` *modèle*
`--template` *modèle*
: Spécifie la base de données modèle à partir de laquelle construire cette base de données.

Les options `-D`, `-E`, `-O` et `-T` correspondent aux options de la commande SQL sous-jacente *CREATE DATABASE* ; voir ici pour plus d'informations sur elles.

createdb accepte aussi les arguments suivant en ligne de commande pour les paramètres de connexion :

`-h` *hôte*
`--host` *hôte*
: Spécifie le nom de l'hôte sur lequel le serveur est en cours d'exécution. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *port*
`--port` *port*
: Spécifie le port TCP ou l'extension du fichier socket de domaine Unix local sur lequel le serveur attend les connexions.

`-U` *nomutilisateur*
`--username` *nomutilisateur*
: Nom de l'utilisateur à connecter

`-W`
`--password`
: Force la demande du mot de passe.

# Environnement

`PGDATABASE`
: Si configuré, le nom de la base de données à créer sauf si surchargé sur la ligne de commande.

`PGHOST`
`PGPORT`
`PGUSER`
: Paramètres de connexion par défaut. `PGUSER` détermine aussi le nom de la base de données à créer s'il n'est pas spécifié sur la ligne de commande ou par `PGDATABASE`.

# Diagnostiques

En cas de difficulté, voir *CREATE DATABASE* et psql pour des discussions sur des problèmes potentiels et des messages d'erreurs. Le serveur de la base de données doit être en cours d'exécution sur l'hôte cible. De plus, tout paramètre de connexion et variable d'environnement par défaut utilisé par la bibliothèque d'interface libpq sera appliqué.

# Exemples

Pour créer la base de données `demo` en utilisant le serveur de bases de données par défaut :

```
$ createdb demo
CREATE DATABASE
```

La réponse est la même que celle reçue si vous aviez lancé la commande SQL `CREATE DATABASE`.

Pour créer la base de données `demo` en utilisant le serveur sur l'hôte `eden`, port 5000, en utilisant le codage `LATIN1` avec un regard à la commande sous–jacente :

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

# Voir aussi

dropdb, *CREATE DATABASE*

# createlang

## Nom

createlang −− définit un langage de procédure pour PostgreSQL

## Synopsis

`createlang` [*options_connexion*...] *nom_langage* [*nom_db*]
`createlang` [*options_connexion*...] −−list | −l *nom_db*

## Description

createlang est un outil pour ajouter un nouveau langage de programmation à une nouvelle base de données PostgreSQL. createlang peut gérer tous les langages fournis dans la distribution PostgreSQL par défaut, mais aucun langage fournis pas d'autres parties.

Bien que les langages de programmation du moteur peuvent être ajoutés directement en utilisant différentes commandes SQL, il est recommandé d'utiliser createlang parce qu'il réalise un certain nombre de vérification et qu'il est plus facile à utiliser. Voir ***CREATE LANGUAGE*** pour des informations supplémentaires.

## Options

createlang accepte les arguments suivants en ligne de commande :

*nom_langage*
: Spécifie le nom du langage de programmation de procédures à être défini.

`[-d]` *nom_db*
`[--dbname]` *nom_db*
: Spécifie pour quelle base de données a été ajouté le langage. La valeur par défaut est d'utiliser la base de données avec le même nom que l'utilisateur système.

`-e`
`--echo`
: Affiche les commandes SQL exécutées.

`-l`
`--list`
: Affiche une liste de langages déjà installés dans la base de données cible.

`-L` *repertoire*
: Spécifie le répertoire dans lequel l'interpréteur du langage est trouvé. Le répertoire est trouvé normalement automatiquement ; cette option est principalement pour des buts de débogage.

createlang accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *hôte*
`--host` *hôte*

Spécifie le nom de l'hôte de la machine sur lequel le serveur fonctionne. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *port*
`--port` *port*
: Spécifie le port TCP ou l'extension du fichier du socket local de domaine Unix sur lequel le serveur attend les connexions.

`-U` *nomutilisateur*
`--username` *nomutilisateur*
: Nom de l'utilisateur qui se connecte

`-W`
`--password`
: Force la demande du mot de passe.

## Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`
: Paramètres de connexion par défaut

## Diagnostiques

La plupart des messages d'erreur s'expliquent d'eux−mêmes. Dans le cas contraire, lancez createlang avec l'option `--echo` et jetez un œil à la commande SQL respective pour les détails.

## Notes

Utilisez droplang pour supprimer un langage.

## Exemples

Pour installer le langage `pltcl` dans la base de données `template1` :

```
$ createlang pltcl template1
```

## Voir aussi

droplang, *CREATE LANGUAGE*

# createuser

## Nom

createuser −− définit un nouveau compte utilisateur PostgreSQL

## Synopsis

`createuser` [*option*...] [*nom_utilisateur*]

## Description

createuser crée un nouvel utilisateur PostgreSQL. Seuls les superutilisateurs (utilisateurs disposant de `usesuper` dans la table `pg_shadow`) peut créer de nouveaux utilisateurs PostgreSQL, donc createuser doit être appelé par quelqu'un qui peut se connecter en tant que superutilisateur PostgreSQL.

Être un superutilisateur implique aussi la capacité de passer certaines vérifications de droits d'accès dans la base de données, donc cette fonction ne devrait pas être donnée à la légère.

createuser est un emballage autour de la commande SQL *CREATE USER*. Il n'y a pas de différence réelle entre la création d'utilisateurs via cet outil ou via d'autres méthodes pour accéder au serveur.

## Options

createuser accepte les arguments suivant en ligne de commande

*nomutilisateur*
: Spécifie le nom de l'utilisateur PostgreSQL à créer. Ce nom doit être unique parmi tous les utilisateurs de cette installation de PostgreSQL.

`-a`
`--adduser`
: Le nouvel utilisateur est autorisé à créer de nouveaux utilisateurs. (Note : en fait, ceci transforme le nouvel utilisateur en *superutilisateur*. Cette option est mal nommée.)

`-A`
`--no-adduser`
: Le nouvel utilisateur n'a pas le droit de créer d'autres utilisateurs (c'est−à−dire que le nouvel utilisateur est un utilisateur normal, pas un superutilisateur). Ceci est la valeur par défaut.

`-d`
`--createdb`
: Le nouvel utilisateur est autorisé à créer des bases de données.

`-D`
`--no-createdb`
: Le nouvel utilisateur n'a pas le droit de créer des bases de données. Ceci est la valeur par défaut.

`-e`
`--echo`

Affiche les commandes que createuser génère et envoie au serveur.

`-E`
`--encrypted`
: Crypte le mot de passe de l'utilisateur stocké dans la base de données. Si non spécifié, le comportement par défaut des mots de passe est utilisé.

`-i` *`nombre`*
`--sysid` *`nombre`*
: Vous permet de prendre un identifiant utilisateur autre que celui par défaut pour le nouvel utilisateur. Ce n'est pas nécessaire mais certaines personnes apprécient.

`-N`
`--unencrypted`
: Ne crypte pas le mot de passe stocké dans la base de donnée. Si non spécifié, le comportement par défaut des mots de passe est utilisé.

`-P`
`--pwprompt`
: Si indiqué, createuser affichera une invite pour la saisie du mot de passe du nouvel utilisateur. Ceci n'est pas nécessaire si vous pense ne pas utiliser l'authentification par mot de passe.

`-q`
`--quiet`
: N'affiche aucune réponse.

Il vous sera demandé un nom et toute autre information manquante si elles ne sont pas spécifiées sur la ligne de commande.

createuser accepte aussi les arguments suivant en ligne de commande pour les paramètres de connexion :

`-h` *`hôte`*
`--host` *`hôte`*
: Spécifie le nom de l'hôte de la machine sur lequel le serveur est en cours d'exécution. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *`port`*
`--port` *`port`*
: Spécifie le port TCP ou l'extension du fichier socket de domaine Unix sur lequel le serveur attend des connexions.

`-U` *`nomutilisateur`*
`--username` *`nomutilisateur`*
: Nom utilisateur pour se connecter (pas celui à créer).

`-W`
`--password`
: Force la demande de mot de passe (pour se connecter au serveur, pas le mot de passe du nouvel utilisateur).

# Environnement

`PGHOST`
`PGPORT`
`PGUSER`
: Paramètres de connexion par défaut

# Diagnostiques

En cas de problèmes, voir *CREATE USER* et psql pour des discussions sur les problèmes potentiels et les messages d'erreur. Le serveur de la base de données doit être en cours d'exécution sur l'hôte cible. De plus, tout paramétrage de connexion par défaut et toute variable d'environnement utilisé par le client de la bibliothèque libpq s'appliquera.

# Exemples

Pour créer un utilisateur `joe` sur le serveur de bases de données par défaut :

```
$ createuser joe
Shall the new user be allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

Pour créer le même utilisateur `joe` en utilisant le serveur `eden`, port 5000, en évitant les demandes et en jetant un œil à la commande sous–jacente :

```
$ createuser -h eden -p 5000 -D -A -e joe
CREATE USER joe NOCREATEDB NOCREATEUSER;
CREATE USER
```

Pour créer le superutilisateur `joe` et lui affecter immédiatement un mot de passe :

```
$ createuser -P -d -a -e joe
Enter password for new user: xyzzy
Enter it again: xyzzy
CREATE USER joe PASSWORD 'xyzzy' CREATEDB CREATEUSER;
CREATE USER
```

Dans l'exemple ci–dessus, le mot de passe n'est pas affiché lors de sa saisie mais nous le montrons pour plus de clareté. Néanmoins, le mot de passe *apparaîtra* dans la commande affichée — donc, il ne faut pas utiliser l'option `-e` lors de l'affectation d'un mot de passe si quelqu'un peut voir votre écran.

# Voir aussi

dropuser, *CREATE USER*

# dropdb

## Nom

dropdb −− supprime une base de données PostgreSQL

## Synopsis

`dropdb` [*option*...] *nomdb*

## Description

dropdb détruit une base de données PostgreSQL existante. L'utilisateur qui exécute cette commande doit être le superutilisateur des bases de données ou le propriétaire de la base de données.

dropdb est un enrobage de la commande SQL *DROP DATABASE*. Il n'y a aucune différence réelle entre supprimer des bases de données avec cet outil ou via d'autres méthodes d'accès au serveur.

## Options

dropdb accepte les arguments suivants en ligne de commande :

*dbname*
: Indique le nom de la base de données à supprimer.

`-e`
`--echo`
: Affiche les commandes que dropdb génère et envoie au serveur.

`-i`
`--interactive`
: Demande une confirmation avant de détruire quoi que ce soit.

`-q`
`--quiet`
: N'affiche aucune réponse.

dropdb accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *hôte*
`--host` *hôte*
: Spécifie le nom d'hôte de la machine sur laquelle le serveur fonctionne. Si la valeur commence avec un slash, elle est utilisée comme répertoire de la socket de domaine Unix.

`-p` *port*
`--port` *port*
: Spécifie le port TCP ou l'extension du fichier de la socket locale de domaine Unix sur laquelle le serveur attend les connexions.

`-U` *nomutilisateur*
`--username` *nomutilisateur*
: Nom de l'utilisateur qui se connecte

`-W`
`--password`
: Force la demande du mot de passe.

## Environnement

`PGHOST`
`PGPORT`
`PGUSER`
: Paramètres de connexion par défaut

## Diagnostiques

En cas de difficulté, voir *DROP DATABASE* et psql pour des discussions sur les problèmes et messages d'erreurs potentiels. Le serveur de base de données doit être en cours d'exécution sur l'hôte cible. De plus, toutes les configurations de connexion par défaut et toutes les variables d'environnement utilisées par la bibliothèque libpq sont utilisées.

## Exemples

Pour détruire la base de données `demo` sur le serveur de bases de données par défaut :

```
$ dropdb demo
DROP DATABASE
```

Pour détruire la base de données `demo` en utilisant le serveur situé sur l'hôte `eden`, port 5000, avec vérification et quelques informations sur les commandes exécutées :

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
DROP DATABASE
```

## Voir aussi

createdb, *DROP DATABASE*

# droplang

## Nom

droplang −− supprime un langage de procédure de PostgreSQL

## Synopsis

`droplang` [*option_connexion*...] *nom_langage* [*nom_db*]
`droplang` [*option_connexion*...] −−list | −l *nom_db*

## Description

droplang est un outil pour supprimer un langage de procédure existant d'une base de données PostgreSQL. droplang peut supprimer tout langage de procédure, même ceux non fournis dans la distribution de PostgreSQL.

Bien que les langages de programmation du moteur puissent être supprimés directement en utilisant quelques commandes SQL, il est recommandé d'utiliser droplang car il réalise quelques vérifications et est plus simple à utiliser. Voir *DROP LANGUAGE* pour plus d'informations.

## Options

droplang accepte les arguments en ligne de commande :

*langname*
: Spécifie le nom du langage de programmation à supprimer.

`[-d]` *nom_db*
`[--dbname]` *nom_db*
: Spécifie à partir de quelle base de données le langage doit être supprimé. Par défaut, la commande utilise la base de données du nom de l'utilisateur système courant.

`-e`
`--echo`
: Affiche les commandes SQL exécutées.

`-l`
`--list`
: Affiche une liste des langages installés dans la base de données cible.

droplang accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *hôte*
`--host` *hôte*
: Spécifie le nom d'hôte de la machine sur lequel le serveur fonctionne. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *port*
`--port` *port*
: Spécifie le port TCP ou l'extension du fichier de la socket de domaine Unix sur lequel le serveur attend les connexions.

`-U` *nomutilisateur*
`--username` *nomutilisateur*
: Nom de l'utilisateur qui se connecte

`-W`
`--password`
: Force la demande du mot de passe.

## Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`
: Paramètres de connexion par défaut

## Diagnostiques

La plupart des messages d'erreurs s'expliquent d'eux−mêmes. Dans le cas contraire, lancez droplang avec l'option `--echo` et regardez sous la commande SQL correspondante pour avoir plus de détails.

## Notes

Utilisez createlang pour ajouter un langage.

## Exemples

Pour supprimer le langage `pltcl` :

```
$ droplang pltcl nomdb
```

## Voir aussi

createlang, *DROP LANGUAGE*

# dropuser

## Nom

dropuser −− supprime un compte utilisateur PostgreSQL

## Synopsis

`dropuser` [*option*...] [*nomutilisateur*]

## Description

dropuser supprime un utilisateur PostgreSQL existant *et* les bases de données qu'il possède. Seuls les superutilisateurs (utilisateurs ayant `usesuper` initialisé dans la table `pg_shadow`) peut détruire les utilisateurs PostgreSQL.

dropuser est un emballage autour de la commande SQL ***DROP USER***. Il n'y a pas de différence réelle entre la suppression des utilisateurs via cet outil et via d'autres méthodes d'accès du serveur.

## Options

dropuser accepte les arguments suivants en ligne de commande :

*username*
: Indique le nom de l'utilisateur PostgreSQL à supprimer. Un nom vous est demandé si vous n'en donnez aucun sur la ligne de commande.

`-e`
`--echo`
: Affiche les commandes que dropuser génère et envoie au serveur.

`-i`
`--interactive`
: Demande une confirmation avant de réellement supprimer l'utilisateur.

`-q`
`--quiet`
: N'affiche aucune réponse.

dropuser accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *hôte*
`--host` *hôte*
: Indique le nom d'hôte de la machine sur lequel le serveur fonctionne. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *port*
`--port` *port*

Indique le port TCP ou l'extension du fichier du socket local de domaine Unix sur lequel le serveur attend les connexions.

`-U` *`nomutilisateur`*
`--username` *`nomutilisateur`*
: Nom de l'utilisateur qui se connecte

`-W`
`--password`
: Force la demande du mot de passe.

# Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`
: Paramètres de connexion par défaut

# Diagnostiques

En cas de difficulté, voir *DROP USER* et psql pour des discussions sur les problèmes et messages d'erreur potentiels. Le serveur de la base de données doit être en cours d'exécution sur l'hôte cible. De plus, toutes les configurations de connexion par défaut et variables d'environnement utilisées par la bibliothèque libpq sont appliquées.

# Exemples

Pour supprimer l'utilisateur `joe` de la base de données par défaut :

```
$ dropuser joe
DROP USER
```

Pour supprimer l'utilisateur `joe` en utilisant le serveur sur l'hôte `eden`, port 5000, avec vérification et des informations supplémentaires sur la commande sous–jacente :

```
$ dropuser -p 5000 -h eden -i -e joe
User "joe" and any owned databases will be permanently deleted.
Are you sure? (y/n) y
DROP USER "joe"
DROP USER
```

# Voir aussi

createuser, *DROP USER*

# ecpg

## Nom

ecpg −− préprocesseur SQL embarqué pour le C

## Synopsis

`ecpg` [*option*...] *fichier*...

## Description

`ecpg` est le préprocesseur SQL embarqué pour les programmes C. Il convertit des programmes C avec des instructions SQL embarquées en du code C normal en remplaçant les appels au SQL avec les appels spéciaux de fonctions. Les fichiers en sortie peuvent être traités avec tout ensemble d'outils de compilation C.

`ecpg` convertit chaque fichier donné en entrée sur la ligne de commande en un fichier C correspondant. Les fichiers en entrées ont de préférence l'extension `.pgc`, auquel cas l'extension est remplacée par `.c` pour déterminer le nom de fichier en sortie. Si l'extension du fichier en entrée n'est pas `.pgc`, alors le nom de fichier en sortie est calculé en ajoutant `.c` au nom complet du fichier. Le nom de fichier en sortie peut aussi être surchargé en utilisant l'option `-o`.

Cette page de référence ne décrit pas le langage SQL embarqué. Voir Chapitre 29 pour plus d'informations sur ce thème.

## Options

`ecpg` accepte les arguments suivants en ligne de commande :

`-c`
: Génère automatiquement certain code C à partir de code SQL. Actuellement, ceci fonctionne pour `EXEC SQL TYPE`.

`-C` *mode*
: Initialise un mode de compatibilité. *mode* peut être soit `INFORMIX` soit `INFORMIX_SE`.

`-D` *symbol*
: Définit un symbole pour le préprocesseur C.

`-i`
: Analyse aussi les fichiers d'en−tête du système.

`-I` *répertoire*
: Spécifie un chemin d'inclusion supplémentaire, utilisé pour trouver les fichiers inclus via `EXEC SQL INCLUDE`. Par défaut, il s'agit de `.` (répertoire actuel), `/usr/local/include`, du répertoire d'en−tête de PostgreSQL qui a été défini au moment de la compilation (par défaut : `/usr/local/pgsql/include`) puis de `/usr/include` dans cet ordre.

`-o` *filename*
: Indique que `ecpg` doit écrire toute sa sortie dans le *fichier* spécifié.

`-r` *`option`*
: Sélectionne un comportement en exécution. En même temps, *`option`* peut seulement valoir `no_indicator`.

`-t`
: Active la validation automatique (autocommit) des transactions. Dans ce mode, chaque commande SQL est validée automatiquement sauf si elle est à l'intérieur d'un bloc de transaction explicite. Dans le mode par défaut, les commandes sont validées seulement lorsque `EXEC SQL COMMIT` est exécutée.

`-v`
: Affiche des informations supplémentaires dont la version et le chemin des en–têtes.

`--help`
: Affiche un bref résumé de l'utilisation de la commande puis quitte

`--version`
: Affiche la version puis quitte.

# Notes

Lors de la compilation de fichiers de code C prétraités, le compilateur a besoin d'être capable de trouver les fichiers d'en–tête ECPG dans le répertoire des en–têtes de PostgreSQL. Du coup, il fait généralement utiliser l'option `-I` lors de l'appel du compilateur (c'est–à–dire `-I/usr/local/pgsql/include`).

Les programmes C utilisant du SQL embarqué doivent être liés avec la bibliothèque `libecpg`, par exemple en utilisant les options de l'éditeur de liens `-L/usr/local/pgsql/lib -lecpg`.

La valeur réelle des répertoires correspondant à votre installation peut être trouvée en utilisant pg_config.

# Exemples

si vous avez un fichier source en SQL C embarqué nommé `prog1.pgc`, vous pouvez créer un programme exécutable en utilisant la séquence de commandes suivante :

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

# pg_config

## Nom

pg_config −− récupère des informations sur la version installée de PostgreSQL

## Synopsis

`pg_config` {−−bindir | −−includedir | −−includedir−server | −−libdir | −−pkglibdir | −−pgxs | −−configure | −−version}...

## Description

L'outil pg_config affiche les paramètres de configuration de la version installée de PostgreSQL. Il a pour but, par exemple, d'être utilisé par des paquets logiciels qui souhaitent s'interfacer avec PostgreSQL pour faciliter la recherche des fichiers d'en−têtes requis et des bibliothèques.

## Options

Pour utiliser pg_config, fournissez une ou plusieurs des options suivantes :

`--bindir`
: Affiche l'emplacement des exécutables utilisateur. Par exemple, utilisez ceci pour trouver le programme `psql`. C'est aussi normalement l'emplacement du programme `pg_config`.

`--includedir`
: Affiche l'emplacement des fichiers d'en−têtes C des interfaces clientes.

`--includedir-server`
: Affiche l'emplacement des fichiers d'en−têtes C pour la programmation du serveur.

`--libdir`
: Affiche l'emplacement des bibliothèques.

`--pkglibdir`
: Affiche l'emplacement des modules chargeables dynamiquement ou celui où le serveur irait chercher. (D'autres fichiers de données dépendant de l'architecture peuvent aussi être installés dans ce répertoire.)

`--pgxs`
: Affiche l'emplacement des fichiers makefile d'extension.

`--configure`
: Affiche les options données au script `configure` lorsque PostgreSQL a été configuré pour sa construction. Ceci peut être utilisé pour reproduire une configuration identique ou pour trouver avec quelles options un paquet binaire a été construit. (Notez néanmoins que les paquets binaires contiennent souvent des correctifs personnalisés par le vendeur.)

`--version`
: Affiche la version de PostgreSQL et quitte.

Si plus d'une option est donnée (sauf pour `--version`), l'information est affichée dans cet ordre, un élément par ligne.

# Notes

L'option `--includedir-server` est apparue pour PostgreSQL 7.2. Dans les versions précédentes, les fichiers d'en−têtes du serveur étaient installés dans le même emplacement que les en−têtes client, qui pouvaient être récupérés avec l'option `--includedir`. Pour que votre paquet gère les deux cas, essayez la nouvelle option en premier, testez le code de sortie pour savoir si la commande a réussi.

Dans les versions précédant PostgreSQL 7.1, avant que `pg_config` ne soit disponible, aucune méthode de récupération de ces informations de configuration n'existait.

# Historique

L'outil `pg_config` est apparu avec la version 7.1 de PostgreSQL.

# pg_dump

## Nom

pg_dump −− sauvegarde une base de données PostgreSQL dans un script ou un autre fichier d'archive

## Synopsis

`pg_dump` [*`option`*...] [*`nom_base`*]

## Description

pg_dump est un utilitaire pour sauvegarder une base de données PostgreSQL. Il réalise des sauvegardes consistantes même si la base de données est en cours d'utilisation concurrente. pg_dump ne bloque pas l'accès des autres utilisateurs (ni en lecture ni en écriture).

Les sauvegardes peuvent être sous forme de script ou de fichier d'archive. Les sauvegardes script sont au format texte et contiennent les commandes SQL requises pour reconstruire la base de données dans l'état où elle était au moment de la sauvegarde. Pour restaurer à partir de ces scripts, utilisez psql. Ils peuvent être utilisés pour reconstruire la base de données y compris sur d'autres machines et d'autres architectures, et même avec quelques modifications, sur d'autres produits de bases de données SQL.

Les autres formats de fichiers d'archive doivent être utilisés avec pg_restore pour reconstruire la base de données. Ils permettent aussi à pg_restore de sélectionner ce qui doit être restauré ou même de réordonner les éléments avant qu'ils ne soient restaurés. Les formats d'archive permettent aussi de sauvegarder et restaurer les << objets larges >>, ce qui n'est pas possible dans une sauvegarde scriptée. Les fichiers d'archive sont aussi conçus pour être portable suivant les architectures.

Lorsqu'il est utilisé avec un des formats de fichier d'archive et combiné avec pg_restore, pg_dump fournit un mécanisme flexible d'archivage et de transfert. pg_dump peut être utilisé pour sauvegarder une base de données entière, puis pg_restore peut être utilisé pour examiner l'archive et/ou sélectionner quelles parties de la base de données seront restaurées. Le format de fichier en sortie le plus flexible est le format << personnalisé >> (`-Fc`). Il permet la sélection et le réordonnancement de tous les éléments archivés et est compressé par défaut. Le format tar (`-Ft`) n'est pas compressé et il n'est pas possible de réordonner les données au chargement mais est assez flexible ; de plus, il est manipulable avec d'autres outils Unix standard comme `tar`.

Lorsque pg_dump est en cours d'exécution, il il faudrait examiner la sortie à la recherche de tout message d'avertissement (affiché sur la sortie standard des erreurs), spécialement à la lumière des limitations indiquées ci−dessous.

## Options

Les options suivantes en ligne de commande contrôlent le contenu et le format de sortie.

*nom_base*
: Spécifie le nom de la base de données à sauvegarder. Si elle n'est pas spécifiée, la variable d'environnement PGDATABASE est utilisée. Dans le cas contraire, le nom de l'utilisateur spécifié pour la connexion est utilisé.

-a
--data-only
: Sauvegarde seulement les données, pas le schéma (définition des données).

  Cette option est seulement intéressante pour le format texte. Pour les formats d'archive, vous devez spécifier l'option a l'appel de pg_restore.

-b
--blobs
: Inclut les objets larges dans la sauvegarde. Un format non textuel doit être choisi.

-c
--clean
: Ajoute les commandes pour nettoyer (supprimer) les objets de la base avant (les commandes pour) les créer.

  Cette option est seulement intéressante pour le format texte. Pour les formats d'archive, vous devez spécifier l'option à l'appel de pg_restore.

-C
--create
: Commence la sortie avec une commande de création de la base de données elle–même et de reconnexion à la nouvelle base de données. (Avec un script de cette forme, peu importe la base de données à laquelle vous vous connectez avant de lancer le script.)

  Cette option est seulement intéressante pour le format texte. Pour les formats d'archive, vous devez spécifier l'option à l'appel de pg_restore.

-d
--inserts
: Sauvegarde les données avec des commandes INSERT (plutôt qu'une commande COPY). Ceci ralentit la restauration ; c'est principalement utile pour créer des sauvegardes qui seront chargées dans des bases de données autres que PostgreSQL. Notez que la restauration pourrait échouer si vous avez réordonné l'ordre des colonnes. L'option -D est plus sûre, bien qu'encore plus lente.

-D
--column-inserts
--attribute-inserts
: Sauvegarde les données avec des commandes INSERT et des noms de colonnes explicites (INSERT INTO *table* (*colonne*, ...) VALUES ...). Ceci ralentit de beaucoup la restauration ; c'est principalement utile pour créer des sauvegardes qui seront chargées dans des bases de données autres que PostgreSQL..

-f *file*
--file=*file*
: Envoie la sortie dans le fichier spécifié. Si ceci est omis, la sortie standard est utilisée.

-F *format*
--format=*format*
: Sélectionne le format de la sortie. *format* correspond à un des éléments suivants :

  p
  : Crée un fichier de scripts SQL en texte simple (par défaut).

  t

Crée une archive `tar` utilisable par pg_restore. Utiliser ce format d'archives permet le réordonnancement et/ou l'exclusion d'objets de la base lors de la restauration de la base de données. Il est aussi possible de sélectionner les données rechargées au moment de la restauration.

`c`

Crée une archive personnalisée convenable pour pg_restore. C'est le format le plus flexible dans le fait qu'il permet le réordonnancement du chargement des données ainsi que la définition des objets. Ce format est aussi compressé par défaut.

`-i`
`--ignore-version`

Ignore les différences de version entre pg_dump et le serveur de bases de données.

pg_dump peut gérer des bases de données à partir des versions précédentes de PostgreSQL mais les très anciennes versions ne sont plus supportées (actuellement celles précédant la 7.0). Utilisez cette option si vous avez besoin de ne pas vérifier la version (et si pg_dump échoue à ce moment, ne prétendez pas que vous n'avez pas été prévenu).

`-n` *`schéma`*
`--schema=`*`schema`*

Sauvegarde seulement le contenu de *`schema`*. Si cette option n'est pas spécifiée, tous les schémas non système de la base de données cible sont sauvegardés.

**Note :** Dans ce mode, pg_dump ne tente pas de sauvegarder tous les objets de la base de données qui ne font pas partie du schéma sélectionné. Du coup, il n'y a pas de garantie que la sauvegarde d'un seul schéma puisse être restaurée avec succès dans une base de données propre.

`-o`
`--oids`

Sauvegarde les identifiants d'objets (OID) comme faisant parti des données de chaque table. Utilisez cette option si votre application référence les colonnes OID (par exemple dans une contrainte de clé étrangère). Sinon, cette option ne devrait pas être utilisée.

`-O`
`--no-owner`

N'affiche pas les commandes d'initialisation du propriétaire des objets pour correspondre à la base de données originale. Par défaut, pg_dump lance des instructions `ALTER OWNER` ou `SET SESSION AUTHORIZATION` pour initialiser le propriétaire des objets de la base de données. Ces instructions échouent lorsque le script n'est pas lancé par un superutilisateur (ou par l'utilisateur qui possède tous les objets de ce script). Pour créer un script qui peut restaurer tous les utilisateurs et remplace le propriétaire de tous les objets, spécifiez `-O`.

Cette option est seulement utile pour le format texte. Pour les formats d'archive, vous devez spécifier l'option à l'appel de `pg_restore`.

`-R`
`--no-reconnect`

Cette option est obsolète mais est toujours acceptée pour des raisons de compatibilité ascendante.

`-s`
`--schema-only`

Sauvegarde uniquement la définition des objets (le schéma), pas les données.

`-S` *`nomutilisateur`*
`--superuser=`*`nomutilisateur`*

Spécifie le nom du superutilisateur à utiliser lors de la désactivation des déclencheurs. Ceci est seulement utile si `--disable-triggers` est utilisé. (Habituellement, il est mieux ne pas utiliser

ceci et de lancer le script en tant que ce superutilisateur.)

`-t table`
`--table=table`
: Sauvegarde uniquement les données de *table*. Il est possible d'avoir plusieurs tables avec le même nom dans différents schémas ; si c'est le cas, toutes les tables correspondantes sont sauvegardées. Spécifiez à la fois `--schema` et `--table` pour sélectionner seulement une table.

> **Note :** Dans ce mode, pg_dump ne fait aucune tentative de sauvegarde des autres objets de la base de données dont la table sélectionnée pourrait dépendre. Du coup, il n'existe aucune garantie que cette sauvegarde d'une seule table pourra être restaurée toute seule dans une base de données propre.

`-v`
`--verbose`
: Spécifie le mode verbeux. Ceci fait que pg_dump affiche des commentaires détaillés sur les objets et les heures de début et de fin pour le fichier de sauvegarde, ainsi que des messages de progression sur la sortie standard des erreurs.

`-x`
`--no-privileges`
`--no-acl`
: Empêche la sauvegarde des droits d'accès (commandes grant/revoke).

`-X disable-dollar-quoting`
`--disable-dollar-quoting`
: Cette option désactive l'utilisation des guillemets dollar pour les corps des fonctions et les force à être entre guillemets en utilisant la syntaxe standard des chaînes dans SQL.

`-X disable-triggers`
`--disable-triggers`
: Cette option est seulement utile pour créer une sauvegarde des données seules. Elle demande à pg_dump d'inclure des commandes pour désactiver temporairement les déclencheurs sur les tables cibles pendant que les données sont en cours de chargement. Utilisez ceci si vous avez des vérifications d'intégrité ou d'autres déclencheurs sur les tables, déclencheurs que vous ne souhaitez pas exécuter pendant le chargement des données.

  Actuellement, les commandes émises pour `--disable-triggers` doivent être lancées par le superutilisateur. Donc, vous devez aussi spécifier le nom du superutilisateur avec `-S` ou, de préférence, faire attention à lancer le script résultant en tant que superutilisateur.

  Cette option est seulement utile pour le format texte. Pour les formats d'archive, vous devez spécifier l'option à l'appel de `pg_restore`.

`-X use-set-session-authorization`
`--use-set-session-authorization`
: Affiche les commandes SET SESSION AUTHORIZATION du standard SQL au lieu des commandes OWNER TO. Ceci rend la sauvegarde plus compatible avec le standard mais, suivant l'historique des objets dans la sauvegarde, pourrait ne pas restaurer correctement.

`-Z 0..9`
`--compress=0..9`
: Spécifie le niveau de compression à utiliser dans les formats d'archive qui supportent la compression. (Actuellement, seul le format d'archive personnalisée supporte la compression.)

pg_dump accepte aussi les arguments suivants comme paramètres de connexion :

`-h` *hôte*
`--host` *hôte*
: Indique le nom d'hôte de la machine sur laquelle le serveur de bases de données est exécuté. Si la valeur commence par une barre oblique (/), elle est utilisée comme répertoire pour le socket de domaine Unix.

`-p` *port*
`--port` *port*
: Indique le port TCP ou le fichier local de socket de domaine Unix sur lequel le serveur écoute pour la connexion.

`-U` *nomutilisateur*
`--username` *nomutilisateur*
: Nom d'utilisateur à utiliser pour se connecter.

`-W`
`--password`
: Force la demande d'un mot de passe.

## Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`
: Paramètres de connexion par défaut.

## Diagnostiques

pg_dump exécute en interne des instructions `SELECT`. Si vous avez des problèmes en lançant pg_dump, assurez−vous d'être capable de sélectionner des informations de la base de données en utilisant, par exemple, psql.

## Notes

Si votre groupe de bases de données a des ajouts supplémentaires dans la base de données `template1`, faites attention à restaurer la sortie de pg_dump dans une base de données réellement vide ; sinon vous obtiendrez certainement des erreurs dûes à la définition dupliquée des objets ajoutés. Pour faire qu'une base de données soit vide et sans ajout local, copiez à partir de `template0`, et non pas à partir de `template1`, par exemple :

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

pg_dump a quelques limitations :

- Lors de la sauvegarde d'une table seule ou en texte standard, pg_dump ne gère pas les objets larges. Ceux−ci doivent être sauvegardés avec la base de données complète en utilisant un des autres formats d'archive.

- Lorsqu'une sauvegarde des données seules est choisie et que l'option `--disable-triggers` est utilisée, pg_dump émet des commandes pour désactiver les déclencheurs sur les tables utilisateur avant d'insérer les données et des commandes pour les réactiver après que les données sont insérées. Si la restauration est stoppée en plein milieu, les catalogues système pourraient être laissés dans le mauvais état.

Les membres des archives tar sont limités à une taille inférieure à 8 Go. (Ceci est une limitation inhérente au format des fichiers tar.) Du coup, ce format ne peut pas être utilisé si la représentation textuelle d'une table dépasse cette taille. La taille totale d'une archive tar et de tout autre format de sortie n'est pas limitée, sauf peut–être par le système d'exploitation.

Le fichier de sauvegarde produit par pg_dump ne contient pas les statistiques utilisées par l'optimiseur pour prendre les décisions de planification des requêtes. Du coup, il est conseillé de lancer `ANALYZE` après avoir restauré une sauvegarde pour s'assurer de bonnes performances.

# Exemples

Pour sauvegarder une base de données :

```
$ pg_dump mabase > base.out
```

Pour recharger cette base de données :

```
$ psql -d base -f base.out
```

Pour sauvegarder une base de données nommée `mabase`, contenant des objets larges dans un fichier `tar` :

```
$ pg_dump -Ft -b mabase > base.tar
```

Pour recharger cette base de données (avec des objets larges) dans une base de données existante appelée `nouvellebase` :

```
$ pg_restore -d nouvellebase base.tar
```

# Historique

L'outil pg_dump est d'abord apparu dans Postgres95 version 0.02. Les formats de sortie non texte ont été introduits dans la version 7.1 de PostgreSQL.

# Voir aussi

pg_dumpall, pg_restore, psql

# pg_dumpall

## Nom

pg_dumpall −− extrait un groupe de bases de données PostgreSQL dans un fichier script

## Synopsis

`pg_dumpall` [*option*...]

## Description

pg_dumpall est un outil pour extraire (<< sauvegarder >>) toutes les bases de données PostgreSQL d'un groupe dans un fichier script. Celui−ci contient les commandes SQL pouvant être utilisées pour restaurer les bases de données avec psql. Il fait ceci en appelant pg_dump pour chaque base de données du groupe. pg_dumpall sauvegarde aussi les objets globaux, communs à toutes les bases de données. (pg_dump ne sauvegarde pas ces objets.) Cela inclut aussi les informations concernant les utilisateurs et groupes de la base de données ainsi que les droits d'accès s'appliquant aux bases de données.

Du coup, pg_dumpall est une solution intégrée pour sauvegarder vos bases de données. Mais, notez une limitation : il ne peut pas sauvegarder les << objets larges >>, car pg_dump ne peut pas sauvegarder de tels objets dans des fichiers texte. Si vous disposez de bases de données contenant des objets larges, elles devront être sauvegardées en utilisant un des modes de sorties non textuelles de pg_dump.

Comme pg_dumpall lit les tables de toutes les bases de données, il est en pratique nécessaire d'avoir les droits de superutilisateur des bases de données pour avoir une sauvegarde complète. De plus, il faut des droits de superutilisateur à l'exécution du script produit, pour avoir le droit de créer des utilisateurs et des groupes et de créer des bases de données.

Le script SQL écrit sur la sortie standard. Les opérateurs shell doivent être utilisés pour la rediriger dans un fichier.

pg_dumpall a besoin de se connecter plusieurs fois au serveur PostgreSQL (une fois par base de données). Si vous utilisez l'authentification par mot de passe, il pourrait avoir besoin de demander un mot de passe à chaque fois. Il est intéressant d'avoir un fichier `~/.pgpass` dans de tels cas. Voir Section 27.12 pour plus d'informations.

## Options

Les options suivantes en ligne de commande contrôlent le contenu et le format de la sortie.

`-a`
`--data-only`
: Sauvegarde seulement les données, pas le schéma (définition des données).

`-c`
`--clean`
: Inclut les commandes SQL pour nettoyer (supprimer) les bases de données avant de les recréer.

`-d`
`--inserts`
: Sauvegarde les données en tant que commandes `INSERT` (plutôt que des `COPY`). Ceci ralentit la restauration ; c'est principalement utile pour créer des sauvegardes qui peuvent être chargées dans des bases de données autres que PostgreSQL. Notez que la restauration pourrait aussi échouer si vous avez modifié l'ordre des colonnes. L'option `-D` est plus sûre, bien que plus lente.

`-D`
`--column-inserts`
`--attribute-inserts`
: Sauvegarde les données en tant que commandes `INSERT` avec les noms de colonnes explicites (`INSERT INTO` *`table`* `(`*`colonne`*`, ...) VALUES ...`). Ceci ralentit la restauration ; c'est principalement utile pour créer des sauvegardes qui peuvent être chargées dans des bases de données autres que PostgreSQL.

`-g`
`--globals-only`
: Sauvegarde seulement les objets globaux (utilisateurs et groupes), pas les bases de données.

`-i`
`--ignore-version`
: Ignore la différence de version entre pg_dumpall et le serveur de bases de données.

  pg_dumpall peut sauvegarder des bases de données de versions précédentes de PostgreSQL, mais les très anciennes versions ne sont plus supportées (avant la 7.0). Utilisez cette option si vous avez besoin de surcharger la vérification de la version (et si pg_dumpall échoue, ne dites pas que vous n'avez pas été prévenu).

`-o`
`--oids`
: Sauvegarde les identifiants des objets (OID) comme faisant partie des données de chaque table. Utilisez cette option si votre application référence les colonnes OID (par exemple, dans une contrainte de clé étrangère). Sinon, cette option ne doit pas être utilisée.

`-O`
`--no-owner`
: Ne produit pas les commandes pour mettre les propriétaires des objets à ceux de la base de données originale. Par défaut, pg_dumpall lance les instructions `ALTER OWNER` ou `SET SESSION AUTHORIZATION` pour configurer le propriétaire des éléments créés. Ces instructions échouent lorsque le script est lancé par un utilisateur ne disposant pas des droits de superutilisateur (ou ne possédant pas les droits du propriétaire de tous les objets compris dans ce script). Pour que ce script puisse être restauré par n'importe quel utilisateur mais donne la propriété des objets à l'utilisateur l'exécutant, spécifiez `-O`.

`-s`
`--schema-only`
: Sauvegarde seulement les définitions des objets (schéma), pas les données.

`-S` *`username`*
`--superuser=`*`username`*
: Spécifie le nom du superutilisateur à utiliser lors de la désactivation des déclencheurs. Ceci est seulement pris en compte si `--disable-triggers` est utilisé. (Habituellement, il est mieux de ne pas utiliser cette option et de lancer à la place le script résultant en tant que superutilisateur.)

`-v`
`--verbose`

Spécifie le mode verbeux. Ceci fait que pg_dumpall affiche les heures de démarrage/arrêt dans le fichier de sauvegarde et les messages de progression sur la sortie standard. Il active aussi la sortie verbeuse dans pg_dump.

`-x`
`--no-privileges`
`--no-acl`
Empêche la sauvegarde des droits d'accès (commandes grant/revoke).

`-X disable-dollar-quoting`
`--disable-dollar-quoting`
Cette option désactive l'utilisation du dollar comme guillemet pour les corps des fonctions et le force à être entre guillemets en utilisant la syntaxe standard du SQL.

`-X disable-triggers`
`--disable-triggers`
Cette option est utile uniquement lors de la création d'une sauvegarde des données seules. Elle indique à pg_dumpall d'inclure les commandes pour désactiver temporairement les déclencheurs sur les tables cibles alors que les données sont rechargées. Utilisez ceci si vous avez des vérifications d'intégrité référencielle ou d'autres déclencheurs sur les tables que vous ne voulez pas appeler lors du rechargement des données.

Actuellement, les commandes émises pour `--disable-triggers` doivent être lancées en tant que superutilisateur. Donc, vous devez aussi spécifier un nom de superutilisateur avec `-S` ou, mieux, lancer le script résultant en tant que superutilisateur.

`-X use-set-session-authorization`
`--use-set-session-authorization`
Affiche les commandes SET SESSION AUTHORIZATION du standard SQL à la place des commandes OWNER TO. Ceci rend la sauvegarde plus compatible avec les standards mais, suivant l'historique des objets dans la sauvegarde, pourrait ne pas être restauré proprement.

Les options suivantes de la ligne de commande contrôlent les paramètres de connexion à la base de données.

`-h` *`hôte`*
Spécifie le nom d'hôte de la machine sur laquelle le serveur de bases de données est en cours d'exécution. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix. La valeur par défaut est prise à partir de la variable d'environnement `PGHOST`, si elle est initialisée, sinon une connexion socket de domaine Unix est tentée.

`-p` *`port`*
Spécifie le port TCP ou l'extension du fichier socket de domaine Unix local sur lequel le serveur est en écoute des connexions. La valeur par défaut est la variable d'environnement `PGPORT`, si elle est initialisée, ou la valeur compilée.

`-U` *`nomutilisateur`*
Se connecter en tant que cet utilisateur.

`-W`
Force une demande de mot de passe. Ceci devrait arriver automatiquement si le serveur réclame une authentification par mot de passe.

# Environnement

`PGHOST`
`PGPORT`

`PGUSER`
: Paramètres de connexion par défaut

# Notes

Comme pg_dumpall appelle pg_dump en interne, certains messages de diagnostique se réfèrent en fait à pg_dump.

Une fois la restauration effectuée, il est conseillé de lancer `ANALYZE` sur chaque base de données, de façon à ce que l'optimiseur dispose de statistiques utiles. Vous pouvez aussi lancer `vacuumdb -a -z` pour analyser toutes les bases de données.

# Exemples

Pour sauvegarder toutes les bases de données :

```
$ pg_dumpall > db.out
```

Pour recharger cette base de données, utilisez par exemple :

```
$ psql -f db.out template1
```

(La base de données à laquelle vous vous connectez n'a pas d'importance ici car le fichier de script créé par pg_dumpall contient les commandes appropriées pour créer et se connecter aux bases de données sauvegardées.)

# Voir aussi

pg_dump. Vous trouverez ici les détails sur les messages d'erreurs possibles.

# pg_restore

## Nom

pg_restore −− restaure une base de données PostgreSQL à partir d'un fichier d'archive créé par pg_dump

## Synopsis

`pg_restore` [*option*...] [*nom_fichier*]

## Description

pg_restore est un outil pour restaurer une base de données PostgreSQL à partir d'une archive créée par pg_dump dans un des formats non textuel. Il lance les commandes nécessaires pour reconstruire la base de données dans l'état où elle était au moment de sa sauvegarde. Les fichiers d'archive permettent aussi à pg_restore d'être sélectif sur ce qui est restauré ou même de réordonner les éléments à restaurer. Les fichiers d'archive sont conçus pour être portables entre les architectures.

pg_restore peut opérer dans deux modes : si un nom de base de données est spécifié, l'archive est restaurée directement dans la base de données. (Les gros objets peuvent seulement être restaurés en utilisant une connexion directe à la base de données.) Sinon, un script contenant les commandes SQL nécessaires pour reconstruire la base de données est créé (et écrit dans un fichier ou sur la sortie standard), similaire à ceux créés par le format en texte plein de pg_dump. Quelques−unes des options contrôlant la sortie du script sont du coup analogues aux options de pg_dump.

De toute évidence, pg_restore ne peut pas restaurer l'information qui ne se trouve pas dans le fichier d'archive. Par exemple, si l'archive a été réalisée en utilisant l'option donnant les << données sauvegardées par des commandes `INSERT` >>, pg_restore ne sera pas capable de charger les données en utilisant des instructions `COPY`.

## Options

pg_restore accepte les arguments suivants en ligne de commande.

*nom_fichier*
: Spécifie l'emplacement du fichier d'archive à restaurer. S'il n'est pas spécifié, l'entrée standard est utilisée.

`-a`
`--data-only`
: Restaure seulement les données, pas les schémas (définitions des données).

`-c`
`--clean`
: Nettoie (supprime) les objets de la base de données avant de les créer.

`-C`
`--create`

Crée la base de données avant de la restaurer. (Quand cette option est utilisée, la base de données nommée avec `-d` est utilisée seulement pour lancer la commande initiale `CREATE DATABASE`. Toutes les données sont restaurées dans le nom de la base de données qui apparaît dans l'archive.)

`-d` *`nom_base`*
`--dbname=`*`nom_base`*

Se connecte à la base de données *`nom_base`* et restaure directement dans la base de données.

`-e`
`--exit-on-error`

Quitte si une erreur est rencontrée lors de l'envoi des commandes SQL à la base de données. La valeur par défaut est de continuer et d'afficher le nombre d'erreurs à la fin de la restauration.

`-f` *`nom_fichier`*
`--file=`*`filename`*

Spécifie le fichier en sortie pour le script généré ou pour la liste lorsqu'elle est utilisée avec `-l`. Par défaut, il s'agit de la sortie standard.

`-F` *`format`*
`--format=`*`format`*

Spécifie le format de l'archive. Il n'est pas nécessaire de le spécifier car pg_restore détermine le format automatiquement. Si spécifié, il peut être un des suivants :

`t`

L'archive est une archive `tar`. Utiliser ce format d'archive permet le réordonnancement et/ou l'exclusion d'éléments de schéma au moment de la restauration de la base de données. Il est aussi possible de limiter les données à recharger au moment de la restauration.

`c`

L'archive est dans le format personnalisé de pg_dump. Ceci est le format le plus flexible dans le fait qu'il permet le réordonnancement du chargement des données ainsi que des éléments de schéma. Ce format est aussi compressé par défaut.

`-i`
`--ignore-version`

Ignore la vérification de version de la base de données.

`-I` *`index`*
`--index=`*`index`*

Restaure uniquement la définition des index nommés.

`-l`
`--list`

Liste le contenu de l'archive. La sortie de cette opération peut être utilisée avec l'option `-L` pour restreindre et réordonner les éléments à restaurer.

`-L` *`fichier_liste`*
`--use-list=`*`fichier_liste`*

Restaure seulement les éléments dans *`fichier_liste`* et dans leur ordre d'apparition dans le fichier. Les lignes peuvent être déplacées et peuvent aussi être commentées en plaçant un `;` au début de la ligne. (Voir les exemples ci-dessous.)

`-O`
`--no-owner`

Ne pas donner les commandes initialisant les propriétaires des objets pour correspondre à la base de données originale. Par défaut, pg_restore lance des instructions `ALTER OWNER` ou `SET SESSION AUTHORIZATION` pour configurer le propriétaire des éléments du schéma créé. Ces instructions échouent sauf si la connexion initiale à la base de données est réalisée par un superutilisateur (ou le même utilisateur que le propriétaire des objets du script). Avec `-O`, tout nom d'utilisateur peut être utilisé pour la connexion initiale et cet utilisateur est le propriétaire des objets créés.

`-P` *`nom_fonction(argtype [, ...])`*
`--function=`*`nom_fonction(argtype [, ...])`*
: Restaure seulement la fonction nommée. Faites attention à épeler le nom de la fonction et les arguments exactement comme ils apparaissent dans la table des matières du fichier de sauvegarde.

`-r`
`--no-reconnect`
: Cette option est obsolète mais est toujours acceptée pour des raisons de compatibilité ascendante.

`-s`
`--schema-only`
: Restaure uniquement le schéma (définition des données), et non pas les données elles−même. Les valeurs de séquence sont réinitialisées.

`-S` *`nom_utilisateur`*
`--superuser=`*`nom_utilisateur`*
: Spécifie le nom d'utilisateur du superutilisateur à utiliser pour désactiver les déclencheurs. Ceci est seulement nécessaire si `--disable-triggers` est utilisé.

`-t` *`table`*
`--table=`*`table`*
: Restaure uniquement la définition et/ou les données de la table nommée.

`-T` *`trigger`*
`--trigger=`*`trigger`*
: Restaure uniquement le déclencheur nommé.

`-v`
`--verbose`
: Spécifie le mode verbeux.

`-x`
`--no-privileges`
`--no-acl`
: Empêche la restauration des droits d'accès (commandes grant/revoke).

`-X use-set-session-authorization`
`--use-set-session-authorization`
: Affiche les commandes SET SESSION AUTHORIZATION du standard SQL à la place des commandes OWNER TO. Ceci rend la sauvegarde plus compatible avec les standards mais, suivant l'historique des objets dans la sauvegarde, pourrait restaurer correctement.

`-X disable-triggers`
`--disable-triggers`
: Cette option n'est pertinente que lors d'une restauration des données seules. Elle demande à pg_restore d'exécuter des commandes pour désactiver temporairement les déclencheurs sur les tables cibles pendant que les données sont rechargées. Utilisez ceci si vous avez des vérifications d'intégrité référentielle sur les tables que vous ne voulez pas appeler lors du rechargement des données.

  Actuellement, les commandes émises pour `--disable-triggers` doivent être exécutées par un superutilisateur. Donc, vous devriez aussi spécifier un nom de superutilisateur avec `-S` ou, de préférence, lancer pg_restore en tant que superutilisateur PostgreSQL.

pg_restore accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *`hôte`*
`--host=`*`hôte`*
: Spécifie le nom d'hôte de la machine sur lequel le serveur est en cours d'exécution. Si la valeur commence par un slash, elle est utilisée comme répertoire du socket de domaine Unix. La valeur par défaut est prise dans la variable d'environnement `PGHOST`, si elle est initialisée, sinon une connexion

socket de domaine Unix est tentée.

`-p port`
`--port=port`
: Spécifie le port TCP ou l'extension du fichier socket de domaine Unix sur lequel le serveur écoute les connexions. Par défaut, l'outil utilise la variable d'environnement `PGPORT`, si elle est configurée, sinon il utilise la valeur indiquée à la compilation.

`-U nom_utilisateur`
: Se connecte en tant que cet utilisateur

`-W`
: Force une demande de mot de passe. Ceci devrait arriver automatiquement si le serveur requiert une authentification par mot de passe.

# Environnement

`PGHOST`
`PGPORT`
`PGUSER`
: Paramètres de connexion par défaut

# Diagnostiques

Quand une connexion directe à la base de données est spécifiée avec l'option `-d`, pg_restore exécute en interne des instructions SQL. Si vous avez des problèmes en exécutant pg_restore, assurez−vous d'être capable de sélectionner des informations à partir de la base de données en utilisant, par exemple à partir de psql.

# Notes

Si votre installation dispose d'ajouts locaux à la base de données `template1`, faites attention à charger la sortie de pg_restore dans une base de données réellement vide ; sinon, vous avez des risques d'obtenir des erreurs dûes aux définitions dupliquées des objets ajoutés. Pour créer une base de données vide sans ajout local, copiez à partir de `template0`, et non pas de `template1`, par exemple :

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Les limitations de pg_restore sont détaillées ci−dessous.

- Lors de la restauration des données dans une table pré−existante et que l'option `--disable-triggers` est utilisée, pg_restore émet des commandes pour désactiver les déclencheurs sur les tables utilisateur avant d'insérer les données, puis émet les commandes pour les réactiver après l'insertion des données. Si la restauration est stoppée en plein milieu, les catalogues système pourraient être abandonnés dans le mauvais état.
- pg_restore ne restaure pas les objets larges pour une seule table. Si une archive contient des objets larges, alors ils sont tous restaurés.

Voir aussi la documentation de pg_dump pour les détails sur les limitations de pg_dump.

Une fois la restauration terminée, il est conseillé de lancer `ANALYZE` sur chaque table restaurée de façon à ce que l'optimiseur dispose de statistiques utiles.

# Exemples

Pour sauvegarder une base de données appelée `ma_base`, contenant des objets larges dans un fichier `tar` :

```
$ pg_dump -Ft -b ma_base > base.tar
```

Pour recharger cette base de données (avec les objets larges) dans une base de données existante appelée `nouvelle_base` :

```
$ pg_restore -d nouvelle_base base.tar
```

Pour réordonner les éléments de la base de données, il est tout d'abord nécessaire de sauvegarder la table des matières de l'archive :

```
$ pg_restore -l archive.fichier > archive.liste
```

Le fichier de liste consiste en un en–tête et d'une ligne par élément, par exemple

```
;
; Archive created at Fri Jul 28 22:28:36 2000
;     dbname: birds
;     TOC Entries: 74
;     Compression: 0
;     Dump Version: 1.4-0
;     Format: CUSTOM
;
;
; Selected TOC Entries:
;
2; 145344 TABLE species postgres
3; 145344 ACL species
4; 145359 TABLE nt_header postgres
5; 145359 ACL nt_header
6; 145402 TABLE species_records postgres
7; 145402 ACL species_records
8; 145416 TABLE ss_old postgres
9; 145416 ACL ss_old
10; 145433 TABLE map_resolutions postgres
11; 145433 ACL map_resolutions
12; 145443 TABLE hs_old postgres
13; 145443 ACL hs_old
```

Les points virgules commencent un commentaire et les numéros au début des lignes se réfèrent à l'ID d'archive interne affectée à chaque élément.

Les lignes dans le fichier peuvent être commentées, supprimées et réordonnées. Par exemple,

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

peut être utilisé en entrée de pg_restore et ne restaure que les éléments 10 et 6 dans cet ordre :

```
$ pg_restore -L archive.liste archive.fichier
```

# Historique

L'outil pg_restore est apparu dans PostgreSQL version 7.1.

# Voir aussi

pg_dump, pg_dumpall, psql

# psql

## Nom

psql −− terminal interactif PostgreSQL

## Synopsis

`psql` [*option*...] [*nombase* [*nomutilisateur*]]

## Description

psql est une interface en mode texte pour PostgreSQL. Il vous permet de saisir des requêtes de façon interactive, de les exécuter sur PostgreSQL et de voir les résultats de ces requêtes. Alternativement, les entrées peuvent êtres lues à partir d'un fichier. De plus, il fournit un certain nombre de méta−commandes et plusieurs fonctionnalités style shell pour faciliter l'écriture des scripts et automatiser un nombre varié de tâches.

## Options

`-a`
`--echo-all`
: Affiche toutes les lignes en entrée sur la sortie standard lorsqu'elles sont lues. Ceci est plus utile dans le traitement de scripts que dans le mode interactif. C'est équivalent à initialiser la variable `ECHO` à `all`.

`-A`
`--no-align`
: Bascule dans le mode d'affichage non aligné. (Le mode d'affichage par défaut est aligné.)

`-c` *commande*
`--command` *commande*
: Indique que psql doit exécuter une chaîne de commande, *commande*, puis s'arrêter. Cette option est utile dans les scripts shell.

  *commande* doit être soit une chaîne de commandes complètement analysable par le serveur (c'est−à−dire qui ne contient aucune des fonctionnalités spécifiques de psql), soit une seule commande antislash. Du coup, vous ne pouvez pas mixer les commandes SQL et les méta−commandes psql. Pour réussir ceci, vous pouvez envoyer la chaîne dans un tube vers psql comme ceci : `echo "\x \\ select * from foo;" | psql`.

  Si la chaîne de commandes contient plusieurs commandes SQL, elles sont traitées dans une seule transaction sauf si des commandes BEGIN/COMMIT explicites sont inclues dans la chaîne pour la diviser en plusieurs transactions. Ceci est différent du comportement adopté quand la même chaîne est envoyée dans l'entrée standard de psql.

`-d` *nombase*
`--dbname` *nombase*

Indique le nom de la base de données où se connecter. Ceci est équivalent à spécifier *nombase* comme premier argument de la ligne de commande qui n'est pas une option.

`-e`
`--echo-queries`
Copie toutes les commandes qui sont envoyées au serveur sur la sortie standard. Ceci est équivalent à initialiser la variable `ECHO` à `queries`.

`-E`
`--echo-hidden`
Affiche les requêtes réelles générées par `\d` et autres commandes antislash. Vous pouvez utiliser ceci pour étudier les opérations internes de psql. Ceci est équivalent à initialiser la variable `ECHO_HIDDEN` dans psql.

`-f` *nomfichier*
`--file` *nomfichier*
Utilise le fichier *nomfichier* comme source des commandes au lieu de lire les commandes de façon interactive. Une fois que le fichier est entièrement traité, psql se termine. Ceci est globalement équivalent à la commande interne `\i`.

Si *nomfichier* est un - (tiret), alors l'entrée standard est lue.

Utiliser cette option est légèrement différent d'écrire `psql < nomfichier`. En général, les deux feront ce que vous souhaitez mais utiliser `-f` active certaines fonctionnalités intéressantes comme les messages d'erreur avec les numéros de ligne. Il y a aussi une légère chance qu'utiliser cette option réduira la surcharge du lancement. D'un autre côté, la variante utilisant la redirection de l'entrée du shell doit (en théorie) pour ramener exactement le même affichage que celui que vous auriez eu en saisissant tout manuellement.

`-F` *séparateur*
`--field-separator` *séparateur*
Utilisez *séparateur* comme champ séparateur. Ceci est équivalent à `\pset fieldsep` ou `\f`.

`-h` *nomhôte*
`--host` *nomhôte*
Indique le nom d'hôte de la machine sur lequel le serveur est en cours d'exécution. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix.

`-H`
`--html`
Active l'affichage en tableau HTML. Ceci est équivalent à `\pset format html` ou à la commande `\H`.

`-l`
`--list`
Liste toutes les bases de données disponibles puis quitte. Les autres option non relatives à la connexion sont ignorées. Ceci est similaire à la commande interne `\list`.

`-o` *nomfichier*
`--output` *nomfichier*
Dirige tous les affichages de requêtes dans le fichier *nomfichier*. Ceci est équivalent à la commande `\o`.

`-p` *port*
`--port` *port*
Indique le port TCP ou l'extension du fichier socket de domaine local Unix sur lequel le serveur attend les connexions. Par défaut, il s'agit de la valeur de la variable d'environnement `PGPORT` ou, si elle n'est pas initialisée, le port spécifié au moment de la compilation, habituellement 5432.

`-P` *affectation*
`--pset` *affectation*

Vous permet de spécifier les options d'affichage dans le style de `\pset` sur la ligne de commande. Notez que, ici, vous devez séparer nom et valeur avec un signe égal au lieu d'un espace. Du coup, pour initialiser le format d'affichage en LaTeX, vous devez écrire `-P format=latex`.

`-q`
`--quiet`

Indique que psql doit travailler silencieusement. Par défaut, il affiche des messages de bienvenue et des informations diverses. Si cette option est utilisée, rien de ceci n'est affiché. C'est utile avec l'option `-c`. À l'intérieur de psql, vous pouvez aussi initialiser la variable `QUIET` pour arriver au même effet.

`-R` *`séparateur`*
`--record-separator` *`séparateur`*

Utilisez *`séparateur`* comme séparateur d'enregistrement. Ceci est équivalent à la commande `\pset recordsep`.

`-s`
`--single-step`

S'exécute en mode étape par étape. Ceci signifie qu'une intervention de l'utilisateur est nécessaire avant l'envoi de chaque commande au serveur, avec une option pour annuler l'exécution. Utilisez cette option pour déboguer des scripts.

`-S`
`--single-line`

S'exécute en mode simple ligne où un retour à la ligne termine une commande SQL, de la même façon qu'un point–virgule.

> **Note :** Ce mode est fourni pour ceux qui insistent pour l'avoir, mais vous n'êtes pas nécessairement encouragé à l'utiliser. En particulier, si vous mixez SQL et méta–commandes sur une ligne, l'ordre d'exécution n'est pas toujours clair pour l'utilisateur non expérimenté.

`-t`
`--tuples-only`

Désactive l'affichage des noms de colonnes et le pied de page contenant le nombre de résultats, etc. Ceci est équivalent à la méta–commande `\t`.

`-T` *`options_table`*
`--table-attr` *`options_table`*

Permet d'indiquer les options à placer à l'intérieur d'une balise `table` en HTML. Voir `\pset` pour plus de détails.

`-u`

Force psql à demander le nom et le mot de passe de l'utilisateur avant de se connecter à la base de données.

Cette option est obsolète, car elle est conceptuellement mauvaise. (Demander un nom d'utilisateur autre que celui par défaut et demander un mot de passe parce que le serveur l'exige sont deux choses tout à fait différentes.) Il est conseillé d'utiliser les options `-U` et `-W` à la place.

`-U` *`nomutilisateur`*
`--username` *`nomutilisateur`*

Se connecte à la base de données en tant que l'utilisateur *`nomutilisateur`* au lieu de celui par défaut. (Vous devez aussi avoir le droit de le faire, bien sûr.)

`-v` *`affectation`*
`--set` *`affectation`*
`--variable` *`affectation`*

Réalise une affectation de variable comme la commande interne `\set`. Notez que vous devez séparer nom et valeur par un signe égal sur la ligne de commande. Pour désinitialiser une variable, enlevez le

signe d'égalité. Pour simplement initialiser une variable sans valeur, utilisez le signe égal sans passer de valeur. Ces affectations sont réalisées lors de la toute première étape du lancement, du coup les variables réservées à des buts internes peuvent être écrasées plus tard.

`-V`
`--version`
Affiche la version de psql et quitte.

`-W`
`--password`
Force psql à demander un mot de passe avant de se connecter à une base de données.

psql devrait automatiquement demander un mot de passe chaque fois que le serveur réclame une authentification par mot de passe. Néanmoins, la détection de demande de mot de passe n'est actuellement pas totalement fiable, d'où cette option pour forcer une demande. Si aucune demande de mot de passe n'est effectuée et que le serveur requiert une authentification par mot de passe, la tentative de connexion échoue.

Cette option reste configurée pour la session complète, même si vous modifiez la connexion de la base de données avec la méta–commande `\connect`.

`-x`
`--expanded`
Active le mode de formatage de table étendu. Ceci est équivalent à la commande `\x`.

`-X,`
`--no-psqlrc`
Ne lit pas le fichier de démarrage (ni le fichier système `psqlrc` ni celui de l'utilisateur `~/.psqlrc`).

`-?`
`--help`
Affiche de l'aide sur les arguments en ligne de commande de psql et quitte.

# Code de sortie

psql renvoie 0 au shell s'il se termine normalement, 1 s'il y a eu une erreur fatale de son fait (pas assez de mémoire, fichier introuvable), 2 si la connexion au serveur s'est interrompue ou a été annulée, 3 si une erreur est survenue dans un script et si la variable `ON_ERROR_STOP` a été initialisée.

# Usage

## Se connecter à une base de données

psql est une application client PostgreSQL standard. Pour se connecter à une base de données, vous devez connaître le nom de votre base de données cible, le nom de l'hôte et le numéro de port du serveur ainsi que le nom de l'utilisateur que vous souhaitez connecter. psql peut connaître ces paramètres à partir d'options en ligne de commande, respectivement `-d`, `-h`, `-p` et `-U`. Si un argument autre qu'une option est rencontré, il est interprété comme le nom de la base de données (ou le nom de l'utilisateur si le nom de la base de données est déjà donné). Toutes les options ne sont pas requises, des valeurs par défaut sont applicables. Si vous omettez le nom de l'hôte, psql se connecte via un socket de domaine Unix à un serveur sur l'hôte local ou via TCP/IP

sur `localhost` pour les machines qui n'ont pas sockets de domaine Unix. Le numéro de port par défaut est déterminé au moment de la compilation. Comme le serveur de bases de données utilise la même valeur par défaut, vous n'aurez pas besoin de spécifier le port dans la plupart des cas. Le nom de l'utilisateur par défaut est votre nom d'utilisateur Unix, de même pour le nom de la base de données par défaut. Notez que vous ne pouvez pas simplement vous connecter à n'importe quelle base de données avec n'importe quel nom d'utilisateur. Votre administrateur de bases de données doit vous avoir informé de vos droits d'accès.

Quand les valeurs par défaut ne sont pas correctes, vous pouvez vous simplifier la vie en configurant les variables d'environnement `PGDATABASE`, `PGHOST`, `PGPORT` et/ou `PGUSER` avec les valeurs appropriées. (Pour les variables d'environnement supplémentaires, voir Section 27.11.) Il est aussi intéressant d'avoir un fichier `~/.pgpass` pour éviter d'avoir régulièrement à saisir des mots de passe. Voir Section 27.12 pour plus d'informations.

Si la connexion ne peut pas se faire, quelle qu'en soit la raison (c'est–à–dire droits non suffisants, serveur arrêté sur l'hôte cible, etc.), psql renvoie une erreur et s'arrête.

## Saisir des commandes SQL

Dans le cas normal, psql fournit une invite avec le nom de la base de données sur laquelle psql est connecté suivi par la chaîne =>. Par exemple,

```
$ psql testdb
Welcome to psql 8.0.5, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=>
```

À l'invite l'utilisateur peut saisir des commandes SQL. Ordinairement, les lignes en entrée sont envoyées vers le serveur quand un point–virgule de fin de commande est saisi. Une fin de ligne ne termine pas une commande. Du coup, les commandes peuvent être saisies sur plusieurs lignes pour plus de clarté. Si la commande est envoyée et exécutée sans erreur, les résultats de la commande sont affichés sur l'écran.

À chaque fois qu'une commande est exécutée, psql vérifie aussi les événements de notification générés par *LISTEN* et *NOTIFY*.

## Meta–commandes

Tout ce que vous saisissez dans psql qui commence par un antislash non échappé est une méta–commande psql qui est traitée par psql lui–même. Ces commandes aident à rendre psql plus utile pour l'administration ou pour l'écriture de scripts. Les méta–commandes sont plus souvent appelées les commandes slash ou antislash.

Le format d'une commande psql est l'antislash suivi immédiatement d'un verbe de commande et de ses arguments. Les arguments sont séparés du verbe de la commande et les uns des autres par un nombre illimité d'espaces blancs.

Pour inclure des espaces blancs dans un argument, vous devez l'envelopper dans des guillemets simples. Pour inclure un guillemet simple dans un argument, précédez–le d'un antislash. Tout ce qui est contenu entre des guillemets simples est de plus sujet à des substitutions style C pour `\n` (nouvelle ligne), `\t` (tabulation), `\`*`chiffres`*, `\0`*`chiffres`* et `\0x`*`chiffres`* (le caractère dont le code est donné en décimal, octal ou hexadécimal).

Si un argument sans guillemets commence avec un caractère **:**, il est pris pour une variable psql et la valeur de la variable est utilisée à la place de l'argument.

Les arguments placés entre guillemets arrières (`) sont pris comme une ligne de commande passée au shell. L'affichage de la commande (sans l'éventuel saut de ligne à la fin) est pris comme valeur de l'argument. Cela s'applique aussi aux séquences d'échappement ci–dessus.

Quelques commandes prennent un identifiant SQL (comme un nom de table) en argument. Ces arguments suivent les règles de la syntaxe SQL : les lettres sans guillemets sont forcées en minuscule alors que les guillemets doubles (") protègent les lettres de la conversion de casse et autorisent l'incorporation d'espaces blancs dans l'identifiant. À l'intérieur des guillemets doubles, les guillemets doubles en paire se réduisent à un seul guillemet double dans le nom résultant. Par exemple, `FOO"BAR"BAZ` est interprété comme `fooBARbaz` et `"Un nom ""bizarre"` devient `Un nom "bizarre`.

L'analyse des arguments se termine quand d'autres antislash non entre guillemets surviennent. Ceci est pris pour le début d'une nouvelle méta–commande. La séquence spéciale `\\` (deux antislashes) marque la fin des arguments et continue l'analyse des commandes SQL, si elles existent. De cette façon, les commandes SQL et psql peuvent être mixées librement sur une ligne. Mais dans tous les cas, les arguments d'une meta–commande ne peuvent pas continuer après la fin de la ligne.

Les meta–commandes suivantes sont définies :

`\a`
: Si le format actuel d'affichage d'une table est non aligné, il est basculé à aligné. S'il n'est pas non aligné, il devient non aligné. Cette commande est conservée pour des raisons de compatibilité. Voir `\pset` pour une solution plus générale.

`\cd [ `*`répertoire`*` ]`
: Modifie le répertoire courant par *`répertoire`*. Sans argument, le répertoire personnel de l'utilisateur devient le répertoire courant.

> **Astuce :** Pour afficher votre répertoire courant, utilisez `\!pwd`.

`\C [ `*`titre`*` ]`
: Initialise ou supprime le titre des tables affichées en résultat d'une requête. Cette commande est équivalente à `\pset title `*`titre`*. (Le nom de cette commande provient de << caption >>, car elle avait précédemment pour seul but d'initialiser l'en–tête dans une table HTML.)

`\connect` (or `\c`) [ *`nomdb`* [ *`nomutilisateur`* ] ]
: Établit une connexion à une nouvelle base de données et/ou sous un nom d'utilisateur. La connexion précédente est fermée. Si *`nomdb`* est –, le nom de la base de données actuelle est supposé.

  Si *`nomutilisateur`* est omis, le nom de l'utilisateur courant est supposé.

  Comme règle spéciale, `\connect` sans autre argument se connecte à la base de données par défaut en tant que l' utilisateur par défaut (comme ce que vous auriez obtenu en lançant psql sans arguments).

Si la tentative de connexion échoue (mauvais nom utilisateur, accès interdit, etc.), la connexion précédente est conservée si et seulement si psql est en mode interactif. Lorsqu'il exécute un script non interactif, le traitement s'arrête immédiatement avec une erreur. Cette distinction a été choisie pour empêcher les problèmes de typographie et comme un mécanisme de sécurité pour que les scripts n'agissent pas sur la mauvaise base de données.

`\copy` *`table`* `[ (` *`liste_colonne`* `) ] { from | to } {` *`nomfichier`* `| stdin | stdout | pstdin | pstdout } [ with ] [ oids ] [ delimiter [ as ]` '*`caractère`*' `] [ null [ as ] '`*`chaîne`*`' ] [ csv [ quote [ as ] '`*`caractère`*`' ] [ escape [ as ] '`*`caractère`*`' ] [ force quote` *`liste_colonne`* `] [ force not null` *`liste_colonne`* `] ]`

Réalise une opération de copy côté client. C'est une opération qui exécute une commande SQL, ***COPY***, mais au lieu que le serveur lise ou écrive le fichier spécifié, psql lit ou écrit le fichier en faisant le routage des données entre le serveur et le système de fichiers local. Ceci signifie que l'accès et les droits du fichier sont ceux de l'utilisateur local, pas celui du serveur, et qu'aucun droit de superutilisateur n'est requis.

La syntaxe de la commande est similaire à celle de la commande ***COPY*** SQL. Notez que, à cause de cela, des règles spéciales d'analyse s'appliquent à la commande `\copy`. En particulier, les règles de substitution de variable et d'échappement antislash ne s'appliquent pas.

`\copy` *`table`* `from` *`stdin | stdout`* lit/écrit basé sur l'entrée standard de la commande ou sa sortie standard respectivement. Toutes les lignes sont lues à partir de la même source qui a lancé la commande, en continuant jusqu'à ce que `\.` soit lu ou que le flux parvienne à EOF. La sortie est envoyée au même endroit que la sortie de la commande. Pour lire/écrire à partir de l'entrée et de la sortie standard de psql, utilisez `pstdin` ou `pstdout`. Cette option est utile pour peupler des tables en ligne à l'intérieur d'un fichier script SQL.

> **Astuce :** Cette opération n'est pas aussi efficace que la commande `COPY` en SQL parce que toutes les données doivent passer au travers de la connexion client/serveur. Pour les grosses masses de données, la commande SQL est préférable.

`\copyright`

Affiche le copyright et les termes de distribution de PostgreSQL.

`\d [` *`modèle`* `]`
`\d+ [` *`modèle`* `]`

Pour chaque relation (table, vue, index ou séquence) correspondant au *`modèle`*, affiche toutes les colonnes, leur types, le tablespace (s'il ne s'agit pas du tablespace par défaut) et tout attribut spécial tel que `NOT NULL` ou les valeurs par défaut. Les index, contraintes, règles et déclencheurs associés sont aussi affichés, ainsi que la définition de la vue si la relation est une vue. (Ce qui << Correspond au modèle >> est défini ci−dessous.)

Le forme de la commande `\d+` est identique, sauf que des informations plus complètes sont affichées : tout commentaire associé avec les colonnes de la table est affiché, ainsi que la présence d'OID dans la table.

> **Note :** Si `\d` est utilisé sans argument *`modèle`*, c'est équivalent en plus commode à `\dtvs` qui affiche une liste de toutes les tables, vues et séquence.

`\da [` *`modèle`* `]`

Liste toutes les fonctions d'agrégat disponibles, avec les types de données sur lesquels elles opèrent. Si *`modèle`* est spécifié, seuls les agrégats dont les noms commencent par le modèle sont affichés.

`\db [` *`modèle`* `]`
`\db+ [` *`modèle`* `]`

Liste tous les tablespaces disponibles. Si *modèle* est spécifié, seuls les tablespaces dont le nom correspond au modèle sont affichés. Si + est ajouté au nom de la commande, chaque objet est listé avec les droits associés.

`\dc [ modèle ]`

Liste toutes les conversions disponibles entre les encodages de jeux de caractères. Si *modèle* est spécifié, seules les conversions dont le nom correspond au modèle sont listées.

`\dC`

Liste toutes les conversions de types disponibles.

`\dd [ modèle ]`

Affiche les descriptions des objets correspondant au *modèle* ou de tous les objets si aucun argument n'est donné. Mais dans tous les cas, seuls les objets qui ont une description sont listés. (Le terme << objet >> couvre les agrégats, les fonctions, les opérateurs, les types, les relations (tables, vues, index, séquences, objets larges), les règles et les déclencheurs.) Par exemple, :

```
=> \dd version
                      Object descriptions
   Schema   |  Name   |  Object  |        Description
------------+---------+----------+---------------------------
 pg_catalog | version | function | PostgreSQL version string
(1 row)
```

Les descriptions des objets peuvent être ajoutées avec la commande SQL *COMMENT*.

`\dD [ modèle ]`

Liste tous les domaines disponibles. Si *modèle* est spécifié, seuls les domaines correspondant sont affichés.

`\df [ modèle ]`

`\df+ [ modèle ]`

Liste toutes les fonctions disponibles avec leurs arguments et les types en retour. Si *modèle* est spécifié, seules les fonctions dont le nom correspond au modèle sont affichées. Si la forme `\df+` est utilisée, des informations supplémentaires sur chaque fonction, dont le langage et la description, sont proposées.

> **Note :** Pour rechercher des fonctions prenant un argument ou des valeurs de retour d'un type spécifique, utilisez les capacités de recherche du paginateur pour parcourir `\df` output.
>
> Pour réduire les redondances, `\df` n'affiche pas les fonctions d'entrées/sorties des types de données. Ceci est implémenté en ignorant les fonctions qui acceptent ou renvoient `cstring`.

`\dg [ modèle ]`

Liste tous les groupes de bases de données. Si *modèle* est spécifié, seuls les groupes dont le nom correspond au modèle sont listés.

`\distvS [ modèle ]`

Ceci n'est pas le nom réel de la commande : les lettres `i`, `s`, `t`, `v`, `S` correspondent respectivement à index, séquence, table, vue et table système. Vous pouvez spécifier une ou toutes ces lettres, dans n'importe quel ordre, pour obtenir une liste de tous les objets correspondants. La lettre S restreint la liste aux objets système ; sans `S`, seuls les objets non système sont affichés. Si + est ajouté au nom de la commande, chaque objet est listé avec sa description associée, si celle–ci est disponible.

Si *modèle* est spécifié, seuls les objets dont les noms correspondent au modèle sont listés.

`\dl`

Ceci est un alias pour `\lo_list`, qui affiche une liste des objets larges.

`\dn [ modèle ]`
`\dn+ [ modèle ]`
: Liste tous les schémas disponibles (espace logiques). Si *modèle* (une expression régulière) est spécifiée, seuls les schémas dont le nom correspond au modèle sont listés. Tout schéma temporaire non local est supprimé. Si + est ajoutée au nom de la commande, chaque objet est listé avec ses droits et description associés.

`\do [ modèle ]`
: Liste tous les opérateurs disponibles avec leur opérande et type en retour. Si *modèle* est spécifié, seuls les opérateurs dont le nom correspond au modèle sont listés.

`\dp [ modèle ]`
: Produit une liste de toutes les tables, vues et séquences disponibles avec leur droits d'accès associés. Si *modèle* est spécifié, seules les tables, vues et séquences dont le nom correspond au modèle sont listées.

  Les commandes GRANT et REVOKE sont utilisées pour configurer les droits d'accès. Voir GRANT pour plus d'informations.

`\dT [ modèle ]`
`\dT+ [ modèle ]`
: Liste tous les types de données ou seulement ceux dont le nom correspond à *modèle*. La commande `\dT+` affiche des informations supplémentaires.

`\du [ modèle ]`
: Liste tous les utilisateurs de la base de données ou seulement ceux dont le nom correspond au *modèle*.

`\edit` (ou `\e`) `[ nomfichier ]`
: Si *nomfichier* est spécifié, le fichier est édité ; en quittant l'éditeur, son contenu est recopié dans le tampon de requête. Si aucun argument n'est fourni, le tampon de requête actuel est copié dans un fichier temporaire qui est ensuite édité de la même façon.

  Le nouveau tampon de requête est ensuite ré−analysé suivant les règles habituelles de psql, où le tampon complet est traité comme une seule ligne. (Du coup, vous ne pouvez pas faire de scripts de cette façon. Utilisez `\i` pour cela.) Ceci signifie aussi que si la requête se termine avec (ou plutôt contient) un point−virgule, elle est immédiatement exécutée. Dans les autres cas, elle attend simplement dans le tampon de requête.

  > **Astuce :** psql recherche les variables d'environnement `PSQL_EDITOR`, `EDITOR` et `VISUAL` (dans cet ordre) pour connaître l'éditeur à utiliser. Si aucun n'est initialisé, `vi` est utilisé sur les systèmes Unix, `notepad.exe` sur les systèmes Windows.

`\echo texte [ ... ]`
: Affiche les arguments sur la sortie standard séparés par un espace et suivi par une nouvelle ligne. Ceci peut être utile pour intégrer des informations sur la sortie des scripts. Par exemple :

  ```
  => \echo `date`
  Tue Oct 26 21:40:57 CEST 1999
  ```

  Si le premier argument est `−n` sans guillemets, alors la fin de ligne n'est pas écrite.

  > **Astuce :** Si vous utilisez la commande `\o` pour rediriger la sortie de la requête, vous pourriez souhaiter utiliser `\qecho` au lieu de cette commande.

`\encoding [ codage ]`
: Initialise le codage de l'ensemble de caractères du client. Sans argument, cette commande affiche le codage actuel.

`\f [ chaîne ]`
: Initialise le champ séparateur pour la sortie de requête non alignée. La valeur par défaut est la barre verticale (`|`). Voir aussi `\pset` comme moyen générique de configuration des options d'affichage.

`\g [ { nomfichier | |commande } ]`
: Envoie le tampon de requête en entrée vers le serveur et stocke en option la sortie de la requête dans *nomfichier* ou envoie dans un tube la sortie vers un autre shell Unix exécutant *commande*. Un simple `\g` est virtuellement équivalent à un point−virgule. Un `\g` avec argument est une alternative en << un coup >> à la commande `\o`.

`\help` (ou `\h`) `[ commande ]`
: Donne la syntaxe sur la commande SQL spécifiée. Si *commande* n'est pas spécifiée, alors psql liste toutes les commandes pour lesquelles une aide en ligne est disponible. Si *commande* est un astérisque (`*`), alors l'aide en ligne de toutes les commandes SQL est affichée.

> **Note :** Pour simplifier la saisie, les commandes qui consistent en plusieurs mots n'ont pas besoin d'être entre guillemets. Du coup, il est correct de saisir **`\help alter table`**.

`\H`
: Active le format d'affichage HTML des requêtes. Si le format HTML est déjà activé, il est basculé au format d'affichage défaut (texte aligné). Cette commande est pour la compatibilité mais voir `\pset` pour configurer les autres options d'affichage.

`\i nomfichier`
: Lit l'entrée à partir du fichier *nomfichier* et l'exécute comme si elle avait été saisie sur le clavier.

> **Note :** Si vous voulez voir les lignes sur l'écran au moment de leur lecture, vous devez initialiser la variable `ECHO` à `all`.

`\l` (ou `\list`)
`\l+` (ou `\list+`)
: Liste les noms, propriétaires et codages des ensembles de caractères de toutes les bases de données du serveur. Si + est ajouté au nom de la commande, les descriptions des bases de données sont aussi affichées.

`\lo_export loid nomfichier`
: Lit l'objet large d'OID *loid* à partir de la base de données et l'écrit dans *nomfichier*. Notez que ceci est subtilement différent de la fonction serveur `lo_export`, qui agit avec les droits de l'utilisateur avec lequel est exécuté le serveur de base de données et sur le système de fichiers du serveur.

> **Astuce :** Utilisez `\lo_list` pour trouver l'OID de l'objet large.

`\lo_import nomfichier [ commentaire ]`
: Stocke le fichier dans un objet large PostgreSQL. En option, il associe le commentaire donné avec l'objet. Exemple :

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'une
photo de moi'
lo_import 152801
```

La réponse indique que l'objet large a reçu l'ID 152801, dont vous devez vous rappeler si vous souhaitez accéder de nouveau à l'objet. Pour cette raison, il est recommandé de toujours associer un commentaire lisible par un humain avec chaque objet. Ils sont ensuite visibles avec la commande `\lo_list`.

Notez que cette commande est subtilement différente de la fonction serveur `lo_import` car elle agit en tant qu'utilisateur local sur le système de fichier local plutôt qu'en tant qu'utilisateur du serveur et de son système de fichiers.

`\lo_list`

Affiche une liste de tous les objets larges PostgreSQL actuellement stockés dans la base de données, avec tous les commentaires fournis par eux.

`\lo_unlink` *`loid`*

Supprime l'objet large d'OID *`loid`* de la base de données.

**Astuce :** Utilisez `\lo_list` pour trouver l'OID d'un objet large.

`\o` [ {*`nomfichier`*|`|`*`commande`*} ]

Sauvegarde les résultats des requête suivantes dans le fichier *`nomfichier`* ou envoie via un tube les résultats à venir dans un shell Unix séparé pour exécuter *`command`*. Si aucun argument n'est spécifié, l'affichage de la requête est redirigé vers la sortie standard.

Les << résultats de requête >> incluent toutes les tables, réponses de commande et messages d'avertissement obtenus du serveur de bases de données, ainsi que la sortie de différentes commandes antislash qui envoient des requêtes à la base de données (comme `\d`), mais sans messages d'erreur.

**Astuce :** Pour intégrer du texte entre les résultats de requête, utilisez `\qecho`.

`\p`

Affiche le tampon de requête actuel sur la sortie standard.

`\pset` *`paramètre`* `[` *`valeur`* `]`

Cette commande initialise les options affectant l'affichage des tables résultat de la requête. *`paramètre`* décrit l'option à initialiser. La sémantique de *`valeur`* en dépend.

Les options ajustables d'affichage sont :

`format`

Initialise le format d'affichage parmi `unaligned`, `aligned`, `html` ou `latex`. Les abréviations uniques sont autorisées. (ce qui signifie qu'une lettre est suffisante.)

<< Unaligned >> écrit toutes les colonnes d'une ligne sur une seule ligne séparées par le séparateur de champ actif. Ceci a pour but de créer un affichage lisible par d'autres programmes (séparé par des tabulations, séparé par des virgules). Le mode << Aligned >> est l'affichage texte standard, lisible par un humain, proprement formaté. C'est aussi la valeur par défaut. Les modes << HTML >> et << LaTeX >> produisent des tables destinées à être inclues dans des documents utilisant le langage de marques respectif. Ce ne sont pas des documents complets ! (Ce n'est pas dramatique en HTML mais en LaTeX vous devez avoir une structure de document complet.)

`border`

Le second argument doit être un nombre. En général, plus grand est ce nombre, plus les tables ont de bordure et de ligne mais ceci dépend du format. Dans le mode HTML, ceci sera traduit directement avec l'attribut `border=...`. Avec les autres, seules les valeurs 0 (sans bordure), 1 (lignes internes de division) et 2 (forme de table) ont un sens.

`expanded` (ou `x`)

Bascule entre le format standard et étendu. Lorsque le format étendu est activé, tout l'affichage a deux colonnes avec le nom de colonne sur la gauche et la donnée sur la droite. Ce mode est utile si les données ne tiennent pas sur l'écran dans le mode << horizontal >>.

Le mode étendu est supporté par les quatre formats d'affichage.

`NULL`
: Le second argument est une chaîne qui est affichée quand une colonne est NULL. La valeur par défaut est de ne rien afficher, ce qui peut être facilement pris pour, disons, une chaîne vide. Du coup, vous pouvez choisir d'écrire `\pset NULL '(NULL)'`.

`fieldsep`
: Indique le séparateur de champ à utiliser dans le mode d'affichage non aligné. De cette façon, vous pouvez créer, par exemple, une sortie séparée par des tabulations ou des virgules, que d'autres programmes pourraient préférer. Pour configurer une tabulation comme champ séparateur, saisissez `\pset fieldsep '\t'`. Le séparateur de champ par défaut est `'|'` (une barre verticale).

`footer`
: Bascule l'affichage du bas de page par défaut `(x lignes)`.

`recordsep`
: Indique le séparateur d'enregistrement (ligne) à utiliser dans le mode d'affichage non aligné. La valeur par défaut est un caractère de retour chariot.

`tuples_only` (ou `t`)
: Bascule entre les lignes seules et l'affichage complet. Ce dernier peut afficher des informations supplémentaires telles que les en−têtes de colonnes, les titres et différents bas de page. Dans le mode lignes seules, seules les données réelles de la table sont affichées.

`title [ texte ]`
: Initialise le titre de la table pour toutes les tables affichées ensuite. Ceci peut être utilisé pour ajouter des balises de description à l'affichage. Si aucun argument n'est donné, le titre n'est pas initialisé.

`tableattr` (ou `T`) `[ texte ]`
: Vous permet de spécifier tout attribut à placer à l'intérieur de la balise `table` en HTML. Ceci pourrait être par exemple `cellpadding` ou `bgcolor`. Notez que vous ne voulez probablement pas spécifier `border` car c'est pris en compte par `\pset border`.

`pager`
: Contrôle l'utilisation d'un paginateur pour les requêtes et les affichages de l'aide de psql. Si la variable d'environnement `PAGER` est configurée, la sortie est envoyée via un tube dans le programme spécifié. Sinon, une valeur par défaut dépendant de la plateforme (comme `more`) est utilisée.

  Lorsque le paginateur est désactivé, il n'est pas utilisé. Quand le paginateur est activé, il est utilisé seulement si nécessaire, c'est−à−dire si l'affichage se fait sur un terminal et qu'il ne tient pas sur l'écran. (psql ne fait pas un boulot parfait pour savoir quand utiliser le paginateur.) `\pset pager` active et désactive le paginateur. Ce dernier peut aussi être configuré à `always`, ce qui fait qu'il est utilisé en permanence.

Des exemples d'utilisation de ces différents formats sont disponibles dans la section ***Exemples***.

**Astuce :** Il existe plusieurs raccourcis de commandes pour `\pset`. Voir `\a`, `\C`, `\H`, `\t`, `\T` et `\x`.

**Note :** C'est une erreur d'appeler `\pset` sans argument. Dans le futur, cet appel pourrait afficher le statut actuel de toutes les options d'affichage.

`\q`
: Quitte le programme psql.

`\qecho texte [ ... ]`
: Cette commande est identique à `\echo` sauf que les affichages sont écrits dans le canal d'affichage des requêtes, configuré par `\o`.

`\r`

Réinitialise (efface) le tampon de requêtes.

`\s [ nomfichier ]`

Affiche ou sauvegarde l'historique des lignes de commandes dans *nomfichier*. Si *nomfichier* est omis, l'historique est écrit sur la sortie standard. Cette option est seulement disponible si psql est configuré pour utiliser la bibliothèque GNU Readline.

> **Note :** Dans la version actuelle, il n'est plus nécessaire de sauvegarder l'historique des commandes car c'est fait automatiquement à la fin du programme. L'historique est aussi chargé automatiquement chaque fois que psql est lancé.

`\set [ nom [ valeur [ ... ]]]`

Initialise la variable interne *nom* à *valeur* ou, si plus d'une valeur est donnée, à la concaténation de toutes les valeurs. Si aucun second argument n'est donné, la variable est simplement initialisée sans valeur. Pour désinitialiser une variable, utilisez la commande `\unset`.

Les noms de variables valides peuvent contenir des caractères, chiffres et tirets bas. Voir la section *Variables* ci−dessous pour les détails. Les noms des variables sont sensibles à la casse.

Bien que vous puissiez configurer toute variable comme vous le souhaitez, psql traite certaines variables de façon spéciale. Elles sont documentées dans la section sur les variables.

> **Note :** Cette commande est totalement séparée de la commande SQL *SET*.

`\t`

Bascule l'affichage des en−têtes de nom de colonne en sortie et celle du bas de page indiquant le nombre de lignes. Cette commande est équivalente à `\pset tuples_only` et est fournie pour en faciliter l'accès.

`\T options_table`

Vous permet de spécifier les attributs à placer à l'intérieur de la balise `table` dans le mode d'affichage en tableau HTML. Cette commande est équivalente à `\pset tableattr options_table`.

`\timing`

Affiche le temps pris par chaque instruction SQL, en millisecondes, ou arrête cet affichage.

`\w {nomfichier | |commande}`

Place le tampon de requête en cours dans le fichier *nomfichier* ou l'envoie via un tube à la commande Unix *commande*.

`\x`

Bascule le mode étendu de formatage en table. C'est équivalent à `\pset expanded`.

`\z [ modèle ]`

Produit une liste de toutes les tables, vues et séquences disponibles avec leur droit d'accès associé. Si un *modèle* est spécifié, seules les tables, vues et séquences dont le nom correspond au modèle sont listées.

Les commandes GRANT et REVOKE sont utilisées pour configurer les droits d'accès . Voir GRANT pour plus d'informations.

Ceci est un alias pour `\dp` (<< affichage des droits >>).

`\! [ commande ]`

Lance un shell Unix séparé ou exécute la commande Unix *commande*. Les arguments ne sont pas interprétés, le shell les voit tel quel.

`\?`

Affiche l'aide sur les commandes antislash.

Les différentes commandes \d acceptent un paramètre *modèle* pour spécifier le(s) nom(s) d'objet à afficher. * signifie << toute séquence de caractères >> et ? signifie << tout caractère simple >>. (Cette notation est comparable aux modèles du shell pour les noms de fichier Unix.) Les utilisateurs avancés peuvent aussi utiliser des notations d'expressions régulières telles que les classes de caractères, par exemple [0-9] pour correspondre à << tout chiffre >>. Pour faire que tous ces caractères de correspondance de modèles soient interprétés de façon littérale, englobez-les dans des guillemets doubles.

Un modèle contenant un point (sans guillemet) est interprété comme un modèle de nom de schéma suivi par un modèle de nom d'objet. Par exemple, \dt foo*.bar* affiche toutes les tables dont le nom du schéma commence avec foo et dont le nom de table commence avec bar. Si aucun point n'apparaît, alors le modèle correspond seulement aux objets visibles dans le chemin actuel de recherche de schéma.

Lorsque *modèle* est complètement omis, les commandes \d affichent tous les objets visibles dans le chemin de recherche actuel du schéma. Pour voir tous les objets dans la base de données, utilisez le modèle *.*.

## Fonctionnalités avancées

### Variables

psql fournit des fonctionnalités de substitution de variable similaire aux shells de commandes Unix. Les variables sont simplement des paires nom/valeur où la valeur peut être toute chaîne, quel que soit sa longueur. Pour initialiser des variables, utilisez la méta-commande psql \set :

```
testdb=> \set foo bar
```

initialise la variable foo avec la valeur bar. Pour récupérer le contenu de la variable, précédez le nom avec un caractère deux-points. Vous pouvez l'utiliser comme argument de toute commande slash :

```
testdb=> \echo :foo
bar
```

> **Note :** Les arguments de \set sont sujets aux même règles de substitution que les autres commandes. Du coup, vous pouvez construire des références intéressantes comme \set :foo 'quelquechose' et obtenir des << liens doux >> ou des << variables de variables >> comme, respectivement, Perl ou PHP. Malheureusement (ou heureusement ?), on ne peut rien faire d'utile avec ces constructions. D'un autre côté, \set bar :foo est un moyen parfaitement valide de copier une variable.

Si vous appelez \set sans second argument, la variable est initialisée avec une chaîne vide. Pour désinitialiser (ou supprimer) une variable, utilisez la commande \unset.

Les noms de variables internes de psql peuvent être constitués de lettres, nombres et tirets bas dans n'importe quel ordre et autant de fois que vous le voulez. Un certain nombre de ces variables sont traitées spécialement par psql. Elles indiquent certaines options qui peuvent changer au moment de l'exécution en modifiant la valeur de la variable ou représentent un certain état de l'application. Bien que vous puissiez utiliser ces variables dans n'importe quel but, ce n'est pas recommandé car le comportement du programme pourrait devenir très rapidement vraiment étrange. Par convention, toutes les variables traitées spécialement sont uniquement composées de lettres majuscules (et peut-être aussi de chiffres et de tirets bas). Pour s'assurer d'un compatibilité maximum dans le futur, évitez l'utilisation de tels noms de variables pour vos propres

besoins. Une liste de toutes les variables traitées spécialement suit.

AUTOCOMMIT
: Si actif (on, valeur par défaut), chaque commande SQL est automatiquement validée si elle se termine avec succès. Pour suspendre la validation dans ce mode, vous devez saisir une commande SQL BEGIN ou START TRANSACTION. Lorsqu'elle est désactivée (off) ou non initialisée, les commandes SQL ne sont plus validées tant que vous ne lancez pas explicitement COMMIT ou END. Le mode sans autocommit fonctionne en lançant implicitement un BEGIN, juste avant toute commande qui n'est pas déjà dans un bloc de transaction et qui n'est pas elle−même un BEGIN ou une autre commande de contrôle de transaction, ou une commande qui ne peut pas être exécutée à l'intérieur d'un bloc de transaction (comme VACUUM).

  **Note :** Dans le mode sans autocommit, vous devez annuler explicitement toute transaction échouée en saisissant ABORT ou ROLLBACK. Gardez aussi en tête que si vous sortez d'une session sans validation, votre travail est perdu.

  **Note :** Le mode auto−commit est le comportement traditionnel de PostgreSQL alors que le mode sans autocommit est plus proche des spécifications SQL. Si vous préférez sans autocommit, vous pouvez le configurer dans le fichier psqlrc global du système ou dans votre fichier ~/.psqlrc.

DBNAME
: Le nom de la base de données à laquelle vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être désinitialisé.

ECHO
: Si cette variable est initialisée à all, toutes les lignes saisies au clavier ou provenant d'un script sont écrites sur la sortie standard avant d'être analysées ou exécutées. Pour sélectionner ce comportement au lancement du programme, utilisez l'option −a. Si ECHO vaut queries, psql affiche simplement toutes les requêtes au moment de leur envoi au serveur. L'option pour ceci est −e.

ECHO_HIDDEN
: Quand cette variable est initialisée et qu'une commande antislash est envoyée à la base de données, la requête est d'abord affichée. De cette façon, vous pouvez étudier le fonctionnement interne de PostgreSQL et fournir des fonctionnalités similaires dans vos propres programmes. (Pour sélectionner ce comportement au lancement du programme, utilisez l'option −E.) Si vous configurez la variable avec la valeur noexec, les requêtes sont juste affichées mais ne sont pas réellement envoyées au serveur ni exécutées.

ENCODING
: Le codage en cours de l'ensemble de caractères du client.

HISTCONTROL
: Si cette variable est configurée à ignorespace, les lignes commençant avec un espace n'entrent pas dans la liste de l'historique. Si elle est initialisée avec la valeur ignoredups, les lignes correspondant aux précédentes lignes de l'historique n'entrent pas dans la liste. Une valeur de ignoreboth combine les deux options. Si elle n'est pas initialisée ou si elle est configurée avec une autre valeur que celles−ci, toutes les lignes lues dans le mode interactif sont sauvegardées dans la liste de l'historique.

  **Note :** Cette fonctionnalité a été plagiée sur Bash.

HISTSIZE
: Le nombre de commandes à stocker dans l'historique des commandes. La valeur par défaut est 500.

  **Note :** Cette fonctionnalité a été plagiée sur Bash.

HOST
: L'hôte du serveur de la base de données où vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être désinitialisé.

IGNOREEOF
: Si non initialisé, envoyer un caractère EOF (habituellement **Ctrl+D**) dans une session interactive de psql ferme l'application. Si elle est configurée avec une valeur numérique, ce nombre de caractères EOF est ignoré avant la fin de l'application. Si la variable est configurée mais n'a pas de valeur numérique, la valeur par défaut est de 10.

  **Note :** Cette fonctionnalité a été plagiée sur Bash.

LASTOID
: La valeur du dernier OID affecté, renvoyée à partir d'une commande `INSERT` ou `lo_insert`. La validité de cette variable est seulement garantie jusqu'à l'affichage du résultat de la commande SQL suivante.

ON_ERROR_STOP
: Par défaut, si les scripts non interactifs rencontrent une erreur, comme une commande SQL mal formée ou une méta−commande interne, le traitement continue. Ceci a été le comportement traditionnel de psql mais il est quelque fois indésirable. Si cette variable est configurée, le traitement du script s'arrête immédiatement. Si le script a été appelé à partir d'un autre script, il se termine de la même façon. Si le script le plus externe n'a pas été appelé à partir d'une session interactive de psql mais plutôt en utilisant l'option `−f`, psql renvoie le code erreur 3 pour distinguer ce cas des conditions d'erreurs fatales (code d'erreur 1).

PORT
: Le port du serveur de la base de données sur lequel vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être désinitialisé.

PROMPT1
PROMPT2
PROMPT3
: Ils spécifient à quoi doit ressembler l'invite psql. Voir *Invite* ci−dessous.

QUIET
: Cette variable est équivalente à l'option `−q` en ligne de commande. Elle n'est probablement pas très utile en mode interactif.

SINGLELINE
: Cette variable est équivalente à l'option `−S` en ligne de commande.

SINGLESTEP
: Cette variable est équivalente à l'option `−s` en ligne de commande.

USER
: L'utilisateur de la base de données où vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être désinitialisé.

VERBOSITY
: Cette variable peut être configurée avec les valeurs `default`, `verbose` (bavard) ou `terse` (succinct) pour contrôler la verbosité des rapports d'erreurs.

### Interpolation SQL

Un fonctionnalité utile supplémentaire des variables psql est que vous pouvez les substituer (<< interpoler >>) dans les instructions SQL standards. La syntaxe pour ceci est encore l'ajout d'un caractère deux−points (`:`)

avant le nom de la variable.

```
testdb=> \set foo 'ma_table'
testdb=> SELECT * FROM :foo;
```

envoie alors la requête pour la table `ma_table`. La valeur de la variable est copiée littéralement, donc elle peut même contenir des guillemets non terminés ou des commandes antislash. Vous devez vous assurer que cela a un sens. L'interpolation de variable n'est pas réalisée dans des entités SQL SQL entre guillemets.

Une application populaire de cette fonctionnalité est de renvoyer le dernier OID inséré dans les instructions suivantes pour construire un scénario de clé étrangère. Une autre utilisation possible de ce mécanisme est de copier le contenu d'un fichier dans une colonne d'une table. Tout d'abord, chargez le fichier dans une variable puis procédez ainsi.

```
testdb=> \set contenu '\'' `cat mon_fichier.txt` '\''
testdb=> INSERT INTO ma_table VALUES (:contenu);
```

Un problème possible avec cette approche est que `mon_fichier.txt` pourrait contenir des guillemets simples. Ils ont besoin d'être échappés pour ne pas provoquer d'erreurs de syntaxe quand la deuxième ligne est traitée. Ceci peut se faire avec le programme `sed` :

```
testdb=> \set contenu '\'' `sed -e "s/'/\\\\\\'/g" <
mon_fichier.txt` '\''
```

Observez le nombre correct d'antislashs (6) ! Cela fonctionne de cette façon : après l'analyse de la ligne par psql, il passe `sed -e "s/'/\\\'/g" < mon_fichier.txt` au shell. Le shell fait son travail à l'intérieur des doubles guillemets et exécute `sed` avec les arguments `-e` et `s/'/\\'/g`. Quand `sed` analyse ceci, il remplace les deux antislashs avec un seul et fait ensuite la substitution. Peut–être qu'à un moment vous pensiez qu'il était génial que toutes les commandes Unix utilisent le même caractère d'échappement. Et ceci ignore le fait que vous pourriez avoir à échapper aussi tous les antislashs parce que les constantes de texte SQL sont aussi sujet à certaines interprétations. Dans ce cas, vous feriez sans doute mieux de préparer le fichier de façon externe.

Comme les caractères deux–points peuvent apparaître légalement dans les commandes SQL, la règle suivante s'applique : la séquence de caractère << :nom >> n'est pas changée sauf si << nom >> est le nom d'une variable réellement initialisée. Dans tous les cas, vous pouvez échapper un deux–points avec un antislash pour le protéger d'une substitution. (La syntaxe deux–points pour les variables fait partie du standard SQL pour les langages de requêtes embarqués, tels que ECPG. La syntaxe deux–points pour les morceaux de tableau et les conversions de types sont des extensions PostgreSQL, d'où le conflit.)

### Invite

Les invites psql peuvent être personnalisées suivant vos préférences. Les trois variables `PROMPT1`, `PROMPT2` et `PROMPT3` contiennent des chaînes et des séquences d'échappement spéciales décrivant l'apparence de l'invite. L'invite 1 est l'invite normale qui est lancée quand psql réclame une nouvelle commande. L'invite 2 est lancée lorsqu'une saisie supplémentaire est attendue lors de la saisie de la commande parce que la commande n'a pas été terminée avec un point–virgule ou parce qu'un guillemet n'a pas été fermé. L'invite 3 est lancée lorsque vous exécutez une commande SQL `COPY` et que vous devez saisir les valeurs des lignes sur le terminal.

La valeur de la variable prompt sélectionnée est affichée littéralement sauf si un signe pourcentage (%) est rencontré. Suivant le prochain caractère, certains autres textes sont substitués. Les substitutions définies sont :

%M
: Le nom complet de l'hôte (avec le nom du domaine) du serveur de la base de données ou [local] si la connexion est établie via une socket de domaine Unix ou [local:*/répertoire/nom*], si la socket de domaine Unix n'est pas dans l'emplacement par défaut défini à la compilation.

%m
: Le nom de l'hôte du serveur de la base de données, tronqué au premier point ou [local] si la connexion se fait via une socket de domaine Unix.

%>
: Le numéro de port sur lequel le serveur de la base de données écoute.

%n
: Le nom d'utilisateur de la session. (L'expansion de cette valeur peut changer pendant une session après une commande SET SESSION AUTHORIZATION.)

%/
: Le nom de la base de données courante.

%~
: Comme %/ mais l'affichage est un ~ (tilde) si la base de données est votre base de données par défaut.

%#
: Si l'utilisateur de la session est un superutilisateur, alors un # sinon un >. (L'expansion de cette valeur peut changer durant une session après une commande SET SESSION AUTHORIZATION.)

%R
: Pour l'invite 1, affiche normalement = mais affiche ^ si on est en mode simple ligne et ! si la session est déconnectée de la base de données (ce qui peut arriver si \connect échoue). Pour l'invite 2, la séquence est remplacée par −, *, un simple guillemet, un double ou un signe dollar, suivant si psql attend une saisie supplémentaire parce que la commande n'est pas terminée, parce que vous êtes à l'intérieur d'un commentaire /* ... */, ou parce que vous n'avez pas terminé un guillemet ou une chaîne échappée avec des dollars. Pour l'invite 3, la séquence ne produit rien.

%x
: État de la Transaction : une chaîne vide lorsque vous n'êtes pas dans un bloc de transaction ou * si vous vous y trouvez, ou ! si vous êtes dans une transaction échouée, ou enfin ? lorsque l'état de la transaction est indéterminé (par exemple à cause d'une rupture de la connexion).

%*chiffres*
: Le caractère avec ce code numérique est substitué. Si *chiffres* commence avec 0x, le reste des caractères est interprété en hexadécimal ; sinon si le premier caractère est 0, les chiffres sont interprétés comme de l'octal  sinon, les chiffres sont lus en tant que nombre décimal.

%:*nom*:
: La valeur de la variable *nom* de psql. Voir la section ***Variables*** pour les détails.

%`*commande*`
: la sortie de la *commande*, similaire à la substitution par << guillemets inverse >> classique.

%[ ... %]
: Les invites peuvent contenir des caractères de contrôle du terminal qui, par exemple, modifient la couleur, le fond ou le style du texte de l'invite, ou modifient le titre de la fenêtre du terminal. Pour que les fonctionnalités d'édition de ligne de Readline fonctionnent correctement, les caractères de contrôle non affichables doivent être indiqués comme invisibles en les entourant avec %[ et %]. Des pairs multiples de ceux−ci pourraient survenir à l'intérieur de l'invite. Par exemple,

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%#%] '
```

a pour résultat une invite en gras (1;), jaune sur noir (33;40) sur les terminaux compatibles VT100.

Pour insérer un pourcentage dans votre invite, écrivez `%%`. Les invites par défaut sont `'%/%R%# '` pour les invites 1 et 2 et `'>> '` pour l'invite 3.

> **Note :** Cette fonctionnalité a été plagiée sur tcsh.

### Édition de la ligne de commande

psql supporte la bibliothèque Readline pour une édition et une recherche simplifiée et conviviale de la ligne de commande. L'historique des commandes est automatiquement sauvegardé lorsque psql quitte et est rechargé quand psql est lancé. La complétion par tabulation est aussi supportée bien que la logique de complétion n'ait pas la prétention d'être un analyseur SQL. Si pour quelques raisons que ce soit, vous n'aimez pas la complétion par tabulation, vous pouvez la désactiver en plaçant ceci dans un fichier nommé `.inputrc` de votre répertoire personnel :

```
$if psql
set disable-completion on
$endif
```

(Ceci n'est pas une fonctionnalité psql mais Readline. Lisez sa documentation pour plus de détails.)

## Environnement

`PAGER`
: Si les résultats d'une requête ne tiennent pas sur l'écran, ils sont envoyés via un tube sur cette commande. Les valeurs typiques sont `more` ou `less`. La valeur par défaut dépend de la plateforme. L'utilisation du paginateur peut être désactivée en utilisant la commande `\pset`.

`PGDATABASE`
: Base de données où se connecter par défaut

`PGHOST`
`PGPORT`
`PGUSER`
: Paramètres de connexion par défaut

`PSQL_EDITOR`
`EDITOR`
`VISUAL`
: Éditeur utilisé par la commande `\e`. Les variables sont examinées dans l'ordre donné ; la première initialisée est utilisée.

`SHELL`
: Commande exécutée par la commande `\!`.

`TMPDIR`
: Répertoire pour stocker des fichiers temporaires. La valeur par défaut est `/tmp`.

## Fichiers

- Avant de se lancer, psql tente de lire et exécuter les commandes provenant du fichier global au système `psqlrc` ou du fichier utilisateur `~/.psqlrc`. (Sur Windows, le fichier de démarrage de l'utilisateur est nommé `%APPDATA%\postgresql\psqlrc.conf`.) Voir

*PREFIX*/share/psqlrc.sample pour plus d'informations sur la configuration du fichier global au système. Il pourrait être utilisé pour configurer le client et le serveur à votre goût (en utilisant les commandes \set et SET).

- À la fois le fichier psqlrc global au système et le fichier ~/.psqlrc de l'utilisateur peuvent être créés en étant spécifiques à une version si vous leur ajouter un tiret et le numéro de version de ~/.psqlrc-8.0.5. Un fichier correspondant à une version spécifique est préféré à un fichier sans indication de version.
- L'historique de la ligne de commandes est stocké dans le fichier ~/.psql_history ou %APPDATA%\postgresql\psql_history sur Windows.

# Notes

- Dans une vie précédente, psql permettait au premier argument d'une commande antislash à une seule lettre de commencer directement après la commande, sans espace séparateur. Pour la compatibilité, ceci est toujours partiellement supporté mais nous n'allons pas expliquer les détails ici car son utilisation n'est pas encouragée. Si vous obtenez des messages étranges, gardez ceci en tête. Par exemple

```
testdb=> \foo
Field separator is "oo".
```

  ce qui n'est probablement pas ce que vous souhaitez.
- psql ne travaille correctement qu'avec des serveurs de la même version. Cela ne signifie pas que les autres combinaisons ne fonctionnenet pas du tout, mais des problèmes subtiles et moins subtiles peuvent survenir. Les commandes antislash sont particulièrement susceptibles d'échouer si le serveur est d'une version différente.

# Notes pour les utilisateurs sous Windows

psql est construit comme une << application de type console >>. Comme les fenêtres console de windows utilisent un codage différent du reste du système, vous devez avoir une attention particulière lors de l'utilisation de caractères sur 8 bits à l'intérieur de psql. Si psql détecte une page de code problématique, il vous avertira au lancement. Pour modifier la page de code de la console, deux étapes sont nécessaires :

- Configurez la page code en saisissant **cmd.exe /c chcp 1252**. (1252 est une page code appropriée pour l'Allemagne ; remplacez–la par votre valeur.) Si vous utilisez Cygwin, vous pouvez placer cette commande dans /etc/profile.
- Configurez la police de la console par << Lucida Console >> parce que la police raster ne fonctionne pas avec la page de code ANSI.

# Exemples

Le premier exemple montre comment envoyer une commande sur plusieurs lignes d'entrée. Notez le changement de l'invite :

```
testdb=> CREATE TABLE ma_table (
testdb(>  premier integer not NULL default 0,
```

```
testdb(>  second text)
testdb-> ;
CREATE TABLE
```

Maintenant, regardons la définition de la table :

```
testdb=> \d ma_table
             Table "ma_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 premier   | integer | not null default 0
 second    | text    |
```

Maintenant, changeons l'invite par quelque chose de plus intéressant :

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Supposons que nous avons rempli la table de données et que nous voulons les regarder :

```
peter@localhost testdb=> SELECT * FROM ma_table;
 premier | second
---------+--------
       1 | one
       2 | two
       3 | three
       4 | four
(4 rows)
```

Vous pouvez afficher cette table de façon différente en utilisant la commande `\pset` :

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM ma_table;
+---------+--------+
| premier | second |
+---------+--------+
|       1 | one    |
|       2 | two    |
|       3 | three  |
|       4 | four   |
+---------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM ma_table;
premier second
------- ------
      1 one
      2 two
      3 three
      4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
```

```
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM
ma_table;
one,1
two,2
three,3
four,4
```

Vous pouvez aussi utiliser les commandes courtes :

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM ma_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

# vacuumdb

## Nom

vacuumdb −− récupère l'espace inutilisé et, en option, analyse une base de données PostgreSQL

## Synopsis

`vacuumdb` [*`option-de-connexion`*...] [−−full | −f] [−−verbose | −v] [−−analyze | −z] [−−table | −t *`table`* [( *`colonne`* [,...] )] ] [*`base-de-donnees`*]
`vacuumdb` [*`options-de-connexion`*...] [−−all | −a] [−−full | −f] [−−verbose | −v] [−−analyze | −z]

## Description

vacuumdb est un outil pour nettoyer une base de données PostgreSQL. vacuumdb génère aussi des statistiques internes utilisées par l'optimiseur de requêtes de PostgreSQL.

vacuumdb est une surcouche de la commande ***VACUUM***. Il n'y a pas de différence effective entre vacuumdb et les autres méthodes de vacuum.

## Options

vacuumdb accepte les arguments suivants sur la ligne de commande :

`-a`
`--all`
: Nettoie toutes les bases de données.

`[-d]` *`base-de-donnees`*
`[--dbname]` *`base-de-donnees`*
: Indique le nom de la base de données à nettoyer ou à analyser. S'il n'est pas précisé et si `-a` (ou `--all`) n'est pas utilisé, le nom de la base de données est cherché dans la variable d'environnement `PGDATABASE`. Si cette variable d'environnement n'a pas de valeur, alors le nom d'utilisateur précisé pour la connexion est utilisé.

`-e`
`--echo`
: Affiche les commandes que vacuumdb génère et envoie au serveur.

`-f`
`--full`
: Exécute un nettoyage << complet >>.

`-q`
`--quiet`
: N'affiche pas de réponse.

`-t` *`table`* `[ (`*`colonne`* `[,...]) ]`
`--table` *`table`* `[ (`*`colonne`* `[,...]) ]`

Ne nettoie ou n'analyse que la table *table*. Des noms de colonnes peuvent être précisés en conjonction avec l'option `--analyze`

> **Astuce :** Si vous indiquez des colonnes, il vous faudra probablement mettre des caractères d'échappement avant les parenthèses. (Voir les exemples plus bas.)

`-v`
`--verbose`
Affiche des informations détaillées durant le traitement.

`-z`
`--analyze`
Calcule des statistiques à l'usage de l'optimiseur.

vacuumdb accepte aussi les arguments suivants comme paramètres de connexion :

`-h` *hôte*
`--host` *hôte*
Indique le nom d'hôte de la machine sur laquelle le serveur de bases de données est en cours d'exécution. Si la valeur commence par une barre oblique (/), elle est utilisée comme répertoire pour la socket de domaine Unix.

`-p` *port*
`--port` *port*
Indique le port TCP ou le fichier local de socket de domaine Unix sur lequel le serveur écoute pour la connexion.

`-U` *utilisateur*
`--username` *utilisateur*
Nom d'utilisateur à utiliser pour se connecter.

`-W`
`--password`
Force la demande d'un mot de passe.

# Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`
Paramètres de connexion par défaut.

# Diagnostiques

En cas de difficulté, voir *VACUUM* et psql pour des discussions sur les problèmes potentiels et les messages d'erreur. Le serveur de base de données doit être en fonctionnement sur le serveur cible. De même, les éventuels paramètres de connexion et variables d'environnement utilisés par la bibliothèque cliente de libpq sont applicables.

# Notes

vacuumdb peut avoir besoin de se connecter plusieurs fois au serveur PostgreSQL et demande un mot de passe à chaque fois. Dans ce cas, il est pratique d'avoir un fichier `~/.pgpass`. Voir Section 27.12 pour avoir plus d'informations.

# Exemples

Pour nettoyer la base de données `test` :

```
$ vacuumdb test
```

Pour nettoyer et analyser une base de données nommée `grossebase` :

```
$ vacuumdb --analyze grossebase
```

Pour nettoyer une seule table `foo` dans une base de données nommée `xyzzy` et analyser une seule colonne `bar` de la table :

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

# Voir aussi

*VACUUM*

# III. Applications relatives au serveur PostgreSQL

Cette partie contient des informations de référence sur les applications et les outils relatifs au serveur PostgreSQL. Ces commandes n'ont d'utilité que si elles ne sont lancées que sur la machine sur lequel le serveur fonctionne. D'autres programmes utilitaires sont listés dans la Référence II, *Applications clientes de PostgreSQL*.

**Table des matières**

# initdb

## Nom

initdb −− crée un nouveau groupe de bases de données PostgreSQL

## Synopsis

`initdb` [*option*...] −−pgdata | −D *répertoire*

## Description

`initdb` crée un nouveau groupe de bases de données PostgreSQL. Un groupe de bases de données est un ensemble de bases de données gérées par une seule instance du serveur.

Créer un groupe de bases de données consiste en la création des répertoires dans lesquels sont stockées les données de la base de données, en la génération des tables partagées du catalogue (tables appartenant à tout le groupe plutôt qu'à une base de données particulière) et en la création de la base de données `template1`. Lorsque vous créerez plus tard une nouvelle base de données, tout ce qui se trouve dans la base de données `template1` est copié. Elle contient les tables catalogues remplies avec certains éléments comme les types internes.

Bien qu'`initdb` tentera de créer le répertoire de données spécifié, il pourrait ne pas en avoir le droit si le répertoire parent du répertoire de données désiré appartient à root. Pour initialiser dans une telle configuration, créer un répertoire de données vide en tant que root, puis utilisez `chown` pour modifier son propriétaire avec le compte de l'utilisateur de la base de données. Enfin, utilisez `su` pour devenir l'utilisateur de la base de données et pouvoir ainsi exécuter `initdb`.

`initdb` doit être exécuté en tant que l'utilisateur possédant le processus serveur parce que le serveur a besoin d'avoir accès aux fichiers et répertoires qu'`initdb` crée. Comme le serveur ne peut pas être exécuté en tant que root, vous ne devez pas non plus lancer `initdb` en tant que root. (En fait, il refuse de le faire.)

`initdb` initialise la locale et le codage de l'ensemble de caractères par défaut du groupe de bases de données. L'ordre de tri (`LC_COLLATE`) et les classes d'ensembles de caractères (`LC_CTYPE`, c'est−à−dire upper (majuscule), lower (minuscule), digit (chiffres)) sont fixes pour toutes les bases de données et ne peuvent pas être modifiées. Les ordres de tri autres que `C` ou `POSIX` sont aussi moins performantes. Pour ces raisons, il est important de choisir la bonne locale lors de l'exécution d'`initdb`. Les catégories de locale restantes peuvent être modifiées plus tard lors du lancement du serveur. Toutes les valeurs de locales (`lc_*`) peuvent être affichées via `SHOW ALL`. Vous trouverez plus de détails dans Section 20.1.

Le codage de l'ensemble de caractères peut être configuré séparément pour une base de données lors de sa création. `initdb` détermine le codage pour la base de données `template1`, qui sert de valeur par défaut pour les autres bases de données. Pour modifier le codage par défaut, utilisez l'option `--encoding`. Plus de détails est disponible dans Section 20.2.

# Options

`-A` *méthode_auth*
`--auth=`*méthode_auth*
: Cette option spécifie la méthode d'authentification pour les utilisateurs locaux dans `pg_hba.conf`. Ne pas utiliser `trust` sauf si vous avez confiance dans tous les utilisateurs locaux de votre système. `Trust` est la valeur par défaut pour faciliter l'installation.

`-D` *répertoire*
`--pgdata=`*répertoire*
: Cette option spécifie le répertoire où le groupe de bases de données doit être stocké. Ceci est la seule information requise par `initdb` mais vous pouvez éviter de l'écrire en configurant la variable d'environnement `PGDATA`, proposition plus intéressante car le serveur de la base de données (`postmaster`) peut ensuite trouver le répertoire par cette même variable.

`-E` *codage*
`--encoding=`*codage*
: Sélectionne le codage de la base de données template. Ceci est aussi considéré comme le codage par défaut de toutes les bases de données créées ultérieurement sauf si vous le surchargez. La valeur par défaut est dérivée de la locale ou est `SQL_ASCII` si l'autre possibilité ne fonctionne pas. Les ensembles de caractères supportés par le serveur PostgreSQL sont décrits dans Section 20.2.1.

`--locale=`*locale*
: Configure la locale par défaut pour le groupe de bases de données. Si cette option n'est pas spécifiée, la locale est héritée de l'environnement d'exécution d'`initdb`. Le support des locales est décrit dans Section 20.1.

`--lc-collate=`*locale*
`--lc-ctype=`*locale*
`--lc-messages=`*locale*
`--lc-monetary=`*locale*
`--lc-numeric=`*locale*
`--lc-time=`*locale*
: Comme `--locale`, mais configure uniquement la locale dans la catégorie spécifiée.

`-U` *nomutilisateur*
`--username=`*nomutilisateur*
: Sélectionne le nom de l'utilisateur qui sera le superutilisateur de la base de données. Il a par défaut le nom de l'utilisateur lançant `initdb`. Le nom du superutilisateur n'est pas vraiment important mais vous pourriez souhaiter conserver le nom de postgres même si le nom de l'utilisateur du système est différent.

`-W`
`--pwprompt`
: Force `initdb` à demander un mot de passe pour le superutilisateur de la base de données. Si vous ne pensez pas utiliser d'authentification par mot de passe, ceci n'est pas important. Sinon, vous ne pourrez pas utiliser l'authentification par mot de passe tant que vous n'aurez pas configuré un mot de passe pour le surperutilisateur.

`--pwfile=`*nomfichier*
: Fait en sorte qu'`initdb` lise le mot de passe du superutilisateur à partir d'un fichier. La première ligne du ficher est le mot de passe.

D'autres paramètres, moins utilisés, sont disponibles :

`-d`
`--debug`

Affiche les informations de débogage du serveur de démarrage et quelques autres messages de moindre intérêt pour le public général. Le serveur de démarrage est le programme qu'`initdb` lance pour créer les tables catalogues. Cette option génère une quantité énorme de messages ennuyeux.

`-L` *répertoire*

Spécifie où `initdb` doit trouver les fichiers d'entrée pour initialiser le groupe de données. Ceci n'est normalement pas nécessaire. Vous serez prévenu si vous avez besoin de spécifier leur emplacement de façon explicite.

`-n`
`--noclean`

Par défaut, quand `initdb` détermine qu'une erreur l'empêche de créer complètement le groupe de bases de données, il supprime tous les fichiers qu'il a créés avant de se rendre compte qu'il ne peut pas finir son travail. Cette option désactive le nettoyage et est donc utile pour le débogage.

# Environnement

`PGDATA`

Spécifie le répertoire où le groupe de bases de données doit être stocké ; peut être surchargé en utilisant l'option `-D`.

# Voir aussi

postgres, postmaster

# ipcclean

## Nom

ipcclean −− supprime la mémoire partagée et les sémaphores d'un serveur PostgreSQL qui a dû s'arrêter brutalement

## Synopsis

```
ipcclean
```

## Description

`ipcclean` supprime tous les segments de mémoire partagée et les ensembles de sémaphores possédés par l'utilisateur courant. Elle a pour but de nettoyer le système après qu'un serveur PostgreSQL se soit arrêté brutalement (postmaster). Notez que relancer immédiatement le serveur nettoie aussi la mémoire partagée et les sémaphores, donc cette commande est de peu d'utilité réelle.

Seul l'administrateur de la base de données devrait exécuter ce programme car il cause un comportement bizarre (c'est−à−dire des plantages brutaux) s'il est lancé lors d'une exécution multi−utilisateur. Si cette commande est exécutée alors que le serveur est en cours d'exécution, la mémoire partagée et les sémaphores alloués par ce serveur sont supprimés, ce qui a des conséquences sévères sur ce serveur.

## Notes

Ce script est une astuce assez sale mais, depuis qu'il a été écrit (il y a bien longtemps), personne n'a proposé une solution aussi efficace et portable. Comme, maintenant, le `postmaster` effectue lui−même ce nettoyage, il est improbable que `ipcclean` soit amélioré dans le futur.

Le script fait des suppositions sur le format de sortie de l'outil `ipcs`, qui pourraient ne pas être vrai suivant le système d'exploitation. Du coup, il pourrait ne pas fonctionner sur votre système. Il est conseillé de lire le script avant de l'essayer.

# pg_controldata

## Nom

pg_controldata −− affiche les informations de contrôle d'un groupe de bases de données PostgreSQL

## Synopsis

`pg_controldata` [*répertoire_données*]

## Description

`pg_controldata` affiche les informations initialisées lors d'`initdb`, telles que la version du catalogue et la locale du serveur. Il affiche aussi des informations sur le traitement des WAL et des points de vérification. Cette information porte sur le groupe complet et n'est pas spécifique à une base de données.

Cet outil peut seulement être lancé par l'utilisateur qui a initialisé le groupe parce qu'il nécessite le droit de lire le répertoire des données. Vous pouvez spécifier le répertoire des données sur la ligne de commande ou utiliser la variable d'environnement `PGDATA`.

## Environnement

`PGDATA`
: Emplacement du répertoire de données par défaut

# pg_ctl

## Nom

pg_ctl −− lance, arrête ou relance le serveur PostgreSQL

## Synopsis

`pg_ctl` start [−w] [−s] [−D *répertoire_données*] [−l *nomfichier*] [−o *options*] [−p *chemin*]
`pg_ctl` stop [−W] [−s] [−D *répertoire_données*] [−m s[mart] | f[ast] | i[mmediate] ]
`pg_ctl` restart [−w] [−s] [−D *répertoire_données*] [−m s[mart] | f[ast] | i[mmediate] ] [−o *options*]
`pg_ctl` reload [−s] [−D *répertoire_données*]
`pg_ctl` status [−D *répertoire_données*]
`pg_ctl` kill [*nom_signal*] [*id_processus*]
`pg_ctl` register [−N *nom_service*] [−U *nom_utilisateur*] [−P *mot_de_passe*] [−D *répertoire_données*] [−w] [−o *options*]
`pg_ctl` unregister [−N *nom_service*]

## Description

pg_ctl est un outil de lancement, d'arrêt et de redémarrage du serveur PostgreSQL (postmaster). Il permet aussi d'afficher le statut d'un serveur en cours d'exécution. Bien que le serveur doive être lancé manuellement, pg_ctl encapsule les tâches comme la redirection des traces ou le détachement du terminal et du groupe de processus. Il fournit aussi des options intéressantes pour un arrêt contrôlé.

Dans le mode `start`, un nouveau serveur est lancé. Le serveur est lancé en tâche de fond et l'entrée standard est attachée à `/dev/null`. La sortie standard et la sortie d'erreurs standard sont soit envoyées dans un journal de traces (si l'option `-l` est utilisée) soit redirigées dans la sortie standard de pg_ctl (et non pas la sortie d'erreurs standard). Si aucun journal de traces n'est donné, la sortie standard de pg_ctl devrait être redirigée dans un fichier ou envoyée via un tube à un autre processus, par exemple un programme de rotation de journaux de trace, comme rotatelogs sinon `postmaster` écrit sa sortie dans le terminal de contrôle (à partir de l'arrière−plan) et ne se détache pas du groupe de processus du shell.

Dans le mode `stop`, le serveur en cours d'exécution dans le répertoire spécifié est arrêté. Trois méthodes différentes d'arrêt peuvent être choisies avec l'option `-m` : le mode << Smart >> attend la déconnexion de tous les clients. C'est la valeur par défaut. Le mode << Fast >> n'attend pas la déconnexion des clients. Toutes les transactions actives sont annulées et les clients sont forcés à se déconnecter. Le serveur est ensuite arrêté. Le mode << Immediate >> tue tous les processus serveurs sans leur laisser la possibilité de faire un arrêt propre. Ceci amènera une tentative de récupération au prochain lancement.

Le mode `restart` exécute en fait un arrêt suivi d'un lancement. Ceci permet de modifier les options en ligne de commande de `postmaster`.

Le mode `reload` envoie simplement au processus `postmaster` un signal SIGHUP ce qui le fait relire ses fichiers de configuration (`postgresql.conf`, `pg_hba.conf`, etc.). Ceci permet de modifier les options du fichier de configuration sans nécessiter un redémarrage complet pour que les modifications soient prises en

compte.

Le mode `status` vérifie si un serveur est toujours en cours d'exécution sur le répertoire de données spécifié. Si c'est le cas, le PID et les options en ligne de commande utilisées lors de son lancement sont affichés.

Le mode `kill` vous permet d'envoyer un signal à un processus spécifique. Ceci est particulièrement utile pour Microsoft Windows, qui ne possède pas de commande kill. Utilisez `--help` pour voir une liste des noms de signaux supportés.

Le mode `register` vous permet d'enregistrer un service système sur Microsoft Windows.

Le mode `unregister` vous permet d'annuler un service système sur Microsoft Windows, précédemment enregistré avec la commande `register`.

# Options

`-D` *`répertoire_données`*
: Spécifie l'emplacement des fichiers de la base de données sur le système de fichiers. Si cette option est omise, la variable d'environnement `PGDATA` est utilisée.

`-l` *`nomfichier`*
: Ajoute la sortie des traces du serveur dans *`nomfichier`*. Si le fichier n'existe pas, il est créé. L'umask est configuré à 077, donc l'accès au journal des traces est interdit aux autres utilisateurs par défaut.

`-m` *`mode`*
: Spécifie le mode d'arrêt. *`mode`* fait partie de `smart`, `fast` ou `immediate`, ou encore de la première lettre d'un de ces trois–là.

`-o` *`options`*
: Spécifie les options à passer directement à la commande `postmaster`.

  Les options sont habituellement entourées par des guillemets simples ou doubles pour s'assurer qu'elles sont passées comme un groupe.

`-p` *`chemin`*
: Spécifie l'emplacement de l'exécutable `postmaster`. Par défaut, l'exécutable `postmaster` est pris à partir du même répertoire que `pg_ctl` ou, si cela échoue, à partir du répertoire d'installation codé en dur. Il n'est pas nécessaire d'utiliser cette option sauf si vous faites quelque chose d'inhabituel et obtenez des erreurs comme quoi l'exécutable `postmaster` serait introuvable.

`-s`
: Affiche seulement les erreurs, pas les messages d'information.

`-w`
: Attend la fin du lancement ou de l'arrêt. S'interrompt au bout de 60 secondes. Ceci est la valeur par défaut pour les arrêts. Un arrêt avec succès est indiqué par la suppression du fichier PID. Pour le lancement, un `psql -l` se terminant avec succès indique que tout va bien. `pg_ctl` tente d'utiliser le bon port pour psql. Si la variable d'environnement `PGPORT` existe, elle est utilisée. Sinon, il cherche si un port a été configuré dans le fichier `postgresql.conf`. Si aucun des deux n'est utilisé, il utilise le port par défaut avec lequel PostgreSQL a été compilé (5432 par défaut). S'il a attendu, `pg_ctl` renvoie un code de sortie précis basé sur le succès du démarrage ou de l'arrêt.

`-W`

Ne pas attendre le lancement ou l'arrêt pour se terminer. Ceci est la valeur par défaut pour les lancements et redémarrages.

## Options Windows

`-N` *`nom_service`*
: Nom du service système à enregistrer. Le nom est utilisé à la fois comme nom de service et comme nom affiché.

`-P` *`mot_de_passe`*
: Mot de passe de l'utilisateur pour lancer le service.

`-U` *`nom_utilisateur`*
: Nom de l'utilisateur qui va lancer le service. Pour les utilisateurs de domaine, utilisez le format `DOMAIN\nom_utilisateur`.

## Environnement

`PGDATA`
: Emplacement du répertoire des données par défaut.

`PGPORT`
: Port par défaut pour psql (utilisé par l'option −w).

Pour les autres, voir postmaster.

## Fichiers

`postmaster.pid`
: Ce fichier dans le répertoire des données est utilisé pour aider pg_ctl à déterminer si le serveur est actuellement en cours d'exécution.

`postmaster.opts.default`
: Si ce fichier existe dans le répertoire des données, pg_ctl (dans le mode `start`) passe le contenu du fichier comme options de la commande `postmaster`, sauf en cas de surcharge par l'option `-o`. Le contenu de ce fichier est aussi affiché en mode `status`

`postmaster.opts`
: Si ce fichier existe dans le répertoire des données, pg_ctl (en mode `restart`) passe le contenu du fichier comme options de postmaster, sauf en cas de surcharge par l'option `-o`.

`postgresql.conf`
: Ce fichier, situé dans le répertoire des données, est analysé pour trouver le bon port à utiliser avec psql lorsque `-w` est donné en mode `start`.

## Notes

Attendre le lancement complet n'est pas une opération bien définie et peut échouer si le contrôle d'accès est configuré pour qu'un client local ne puisse pas se connecter sans interaction manuelle (par exemple, une authentification par mot de passe).

# Exemples

## Lancer le serveur

Pour lancer un serveur :

```
$ pg_ctl start
```

Un exemple de lancement de serveur, bloquant tant que le serveur n'est pas complètement lancé :

```
$ pg_ctl -w start
```

Pour un serveur utilisant le port 5433 et s'exécutant sans `fsync`, utilisez :

```
$ pg_ctl -o "-F -p 5433" start
```

## Arrêt du serveur

```
$ pg_ctl stop
```

arrête le serveur. Utiliser l'option `-m` vous permet de contrôler *comment* le serveur s'arrête.

## Relance du serveur

Relancer le serveur est pratiquement équivalent à l'arrêter puis à le relancer de nouveau sauf que `pg_ctl` sauvegarde et réutilise les options en ligne de commande qui ont été passées à l'instance précédente. Pour relancer le serveur de la façon la plus simple, utilisez :

```
$ pg_ctl restart
```

Pour relancer le serveur, en attendant qu'il s'arrête puis qu'il se relance :

```
$ pg_ctl -w restart
```

Pour relancer en utilisant le port 5433 et en désactivant `fsync` après redémarrage :

```
$ pg_ctl -o "-F -p 5433" restart
```

## Affichage de l'état du serveur

Voici un exemple de statut affiché à partir de pg_ctl :

```
$ pg_ctl status
pg_ctl: postmaster is running (pid: 13718)
Command line was:
/usr/local/pgsql/bin/postmaster '-D' '/usr/local/pgsql/data' '-p' '5433' '-B' '128'
```

Ceci est la ligne de commande qui serait appelée en mode relance.

## Voir aussi

postmaster

# pg_resetxlog

## Nom

pg_resetxlog −− réinitialise les WAL et les autres informations de contrôle d'un groupe de bases de données PostgreSQL

## Synopsis

`pg_resetxlog` [ −f ] [ −n ] [ −o *`oid`* ] [ −x *`xid`* ] [ −l *`timelineid`*,*`fileid`*,*`seg`* ] *`repdonnees`*

## Description

`pg_resetxlog` efface les WAL et réinitialise en option quelques autres informations de contrôle (stockées dans le fichier `pg_control`). Cette fonction est quelque fois nécessaire si ces fichiers ont été corrompus. Elle ne doit être utilisée qu'en dernier ressort quand le serveur ne démarre plus à cause d'une telle corruption.

Après avoir lancé cette commande, il doit être possible de lancer ce serveur, mais gardez en tête que la base de données pourrait contenir des données non cohérentes à cause de transactions partiellement validées. Vous devriez immédiatement sauvegarder vos données, lancer `initdb` et recharger. Après rechargement, vérifiez les incohérences et réparez si nécessaire.

Cet outil peut seulement être lancé par l'utilisateur qui a installé le serveur parce qu'il requiert un accès en lecture/écriture dans le répertoire des données. Pour des raisons de sécurité, vous devez spécifier le répertoire des données sur la ligne de commande. `pg_resetxlog` n'utilise pas la variable d'environnement `PGDATA`.

Si `pg_resetxlog` se plaint de ne pas pouvoir déterminer de données valides pour `pg_control`, vous pouvez malgré tout le forcer à continuer en spécifiant l'option `−f` (force). Dans ce cas, des valeurs probables sont substituées aux données manquantes. La plupart des champs correspondent mais une aide manuelle pourrait être nécessaire pour le prochain OID, le prochain TID, l'adresse de début des WAL et les champs locale des bases de données. Les trois premiers peuvent être initialisés en utilisant les options discutées plus bas. Le propre environnement de `pg_resetxlog` est la source de ses suppositions quant aux champs locale ; faites attention que `LANG` et d'autres correspondent à l'environnement avec lequel `initdb` a été exécuté. Si vous n'êtes pas capable de déterminer les bonnes valeurs pour tous ces champs, `−f` peut toujours être utilisé mais la base de données récupérée doit être traitée avec encore plus de suspicion que d'habitude : une sauvegarde immédiate et un rechargement sont impératifs. *Ne pas* exécuter d'opérations de modifications de données dans la base avant de sauvegarder ; ce type d'action risque de faire empirer la corruption.

Les options `−o`, `−x` et `−l` permettent d'initialiser manuellement le prochain OID, le prochain TID et l'adresse de début des WAL. Elles sont seulement nécessaires si `pg_resetxlog` est incapable de déterminer les valeurs appropriées en lisant `pg_control`. Une valeur saine pour la prochaine ID en transaction peut être déterminée en cherchant le nom de fichier le plus large numériquement dans le répertoire `pg_clog` sous le répertoire des données, en ajoutant un et en multipliant par 1048576. Notez que les noms de fichiers sont en hexadécimal. Il est habituellement plus facile de spécifier la valeur de l'option en hexadécimal aussi. Par exemple, si `0011` est l'entrée la plus large dans `pg_clog`, `-x 0x1200000` fonctionnera (cinq zéros à la fin fournissent le bon multiplieur). L'adresse de début des WAL doit être plus large que tout numéro de fichier

déjà existant dans le répertoire `pg_xlog` sous le répertoire des données. Ces noms sont aussi en hexadécimal et ont trois parties. La première est le << timeline ID >> et doit habituellement être conservée identique. Ne choisissez pas de valeur plus importante que 255 (`0xFF`) pour la troisième partie ; à la place, incrémentez la deuxième partie et réinitialisez la troisième partie à 0. Par exemple, si `000000010000000320000004A` est l'entrée la plus importante de `pg_xlog`, `-l 0x1,0x32,0x4B` fonctionnera ; mais si l'entrée la plus importante est `000000010000003A000000FF`, choisissez `-l 0x1,0x3B,0x0` ou plus. Il n'y a pas de façon plus simple pour déterminer un OID suivant qui se trouve après celui de la base de données mais, heureusement, il n'est pas critique d'obtenir le bon OID suivant.

L'option `-n` (sans opération) demande à `pg_resetxlog` d'afficher les valeurs reconstruites à partir de `pg_control` puis quitte sans rien modifier. C'est principalement un outil de débogage mais qui peut être utile pour une vérification avant de permettre à `pg_resetxlog` de traiter réellement la base.

# Notes

Cette commande ne doit pas être utilisée si le serveur est en cours d'exécution. `pg_resetxlog` refuse de se lancer s'il trouve un fichier de verrouillage du serveur dans le répertoire des données ; Si le serveur s'est arrêté brutalement, il peut subsister un tel fichier. Dans ce cas, vous pouvez supprimer le fichier de verrouillage pour permettre à `pg_resetxlog` de se lancer. Mais avant de le faire, soyez doublement certain qu'il n'y a pas de `postmaster` ou d'autre processus serveur en cours.

# postgres

## Nom

postgres −− lance un serveur PostgreSQL en mode mono utilisateur

## Synopsis

`postgres` [−A 0 | 1 ] [−B *nombre_de_tampons*] [−c *nom*=*valeur*] [−d *niveau-débogage*] [−−describe−config] [−D *répertoire_données*] [−e] [−E] [−f s | i | t | n | m | h ] [−F] [−N] [−o *nom_fichier*] [−O] [−P] [−s | −t pa | pl | ex ] [−S *work-mem*] [−W *secondes*] [−−*nom*=*valeur*] *base_de_données*
`postgres` [−A 0 | 1 ] [−B *nombre_de_tampons*] [−c *nom*=*valeur*] [−d *niveau-débogage*] [−D *répertoire_données*] [−e] [−f s | i | t | n | m | h ] [−F] [−o *nom_fichier*] [−O] [−p *base_de_données*] [−P] [−s | −t pa | pl | ex ] [−S *work-mem*] [−v *protocole*] [−W *secondes*] [−−*nom*=*valeur*]

## Description

L'exécutable `postgres` est le processus serveur réel de PostgreSQL, celui qui traite les requêtes. Il n'est pas normalement appelé directement ; à la place, un serveur multi−utilisateurs postmaster est lancé.

La deuxième forme ci−dessus est la façon dont `postgres` est appelé par postmaster (seulement conceptuellement, car `postmaster` et `postgres` sont en fait le même programme) ; il ne devrait pas être appelé directement de cette façon. La première forme appelle le serveur directement dans un mode interactif mono utilisateur. La principale utilisation pour ce mode est le démarrage de l'initialisation par initdb. Quelque fois, il est utilisé pour le débogage ou la récupération après un problème grave.

Lorsqu'il est appelé en mode interactif à partir du shell, l'utilisateur peut entrer les requêtes et les résultats sont affichés à l'écran mais dans une forme qui est plus utile pour les développeurs que pour les utilisateurs. Mais notez que lancer un serveur en simple utilisateur n'est pas vraiment convenable pour le débogage du serveur car il n'y a pas de communications interprocessus réalistes et qu'aucun verrou n'est posé.

Lors de l'utilisation d'un serveur en mono utilisateur, l'utilisateur de la session est initialisé avec l'ID 1. Cet utilisateur n'a pas besoin d'exister, de façon à ce qu'un serveur en mode mono utilisateur puisse être utilisé pour récupérer manuellement certains types de dommages accidentels pour les catalogues systèmes. Les pouvoirs implicites du super−utilisateur sont donnés à l'utilisateur d'ID 1 dans le mode mono−utilisateur.

## Options

Quand `postgres` est lancé par un postmaster, il hérite de toutes les options initialisées par ce dernier. En plus, des options spécifiques à `postgres` peuvent être passées par `postmaster` avec l'option `-o`.

Vous pouvez éviter d'avoir à saisir ces options en initialisant un fichier de configuration. Voir Section 16.4 pour plus de détails. Quelques options (sûres) peuvent aussi être initialisées à partir du client se connectant

d'une façon dépendante de l'application. Par exemple, si la variable d'environnement PGOPTIONS est initialisée, alors les clients basés sur libpq passent cette chaîne au serveur, qui l'interprète comme des options en ligne de commande postgres.

## But général

Les options -A, -B, -c, -d, -D, -F et --*name* ont les mêmes significations que postmaster sauf que -d 0 empêche le niveau de traces du serveur du postmaster d'être propagé à postgres.

-e
: Initialise le style de date par défaut à << European >>, c'est–à–dire que l'ordre des champs d'entrée de date est DMY et que le jour est affiché avant le mois dans certains formats de sortie des dates. Voir Section 8.5 pour plus d'informations.

-o *nom_fichier*
: Envoie toutes les traces du serveur dans *nom_fichier*. Si postgres est exécuté sous postmaster, cette option est ignorée et la sortie standard des erreurs, stderr, héritée de postmaster est utilisée.

-P
: Ignore les index système lors de la lecture des tables systèmes (mais met toujours à jour les index lors de la modification des tables). Ceci est utile lors de la récupération des index systèmes endommagés.

-s
: Affiche l'information de l'heure et les autres statistiques à la fin de chaque commande. C'est utile pour connaître les performances ou pour configurer finement le nombre de tampons.

-S *work–mem*
: Indique la quantité de mémoire à utiliser par les tris internes et les hachages avant le tri des fichiers temporaires du disque. Voir la description du paramètre de configuration work_mem dans Section 16.4.3.1.

## Options pour le mode autonome

*base_de_données*
: Indique le nom de la base de données à accéder. Si ce paramètre est omis, Postgres prend comme valeur par défaut le nom de l'utilisateur.

-E
: Affiche toutes les commandes.

-N
: Désactive l'utilisation des retours chariot comme délimiteur d'une instruction.

## Options semi–internes

Il existe plusieurs autres options qui pourraient être spécifiées, utilisées principalement dans des buts de débogage. Elles sont listées ici pour être utilisées par les développeurs du système PostgreSQL. *Utiliser une de ces options est hautement découragé.* De plus, ces options sont susceptibles de changer ou de disparaitre sans préavis dans une version future.

-f { s | i | m | n | h }

Interdit l'utilisation de méthodes de parcours et de jointure particulières : `s` et `i` désactivent respectivement les parcours séquentiels et d'index alors que `n`, `m` et `h` désactivent respectivement les jointures à boucles imbriquées, les jointures par fusion et par hachage.

> **Note :** Ni les parcours séquentiels ni les jointures de boucles imbriquées ne peuvent être complètement désactivés ; les options `-fs` et `-fn` découragent simplement l'optimiseur d'utiliser ces types de plan s'il y a d'autres alternatives.

`-O`
Permet la modification de la structure des tables système. Ceci est utilisé par `initdb`.

`-p` *`base_de_données`*
Indique que ce processus a été lancé par un `postmaster` et indique la base de données à utiliser, etc.

`-t pa[rser] | pl[anner] | e[xecutor]`
Affiche les statistiques de durée de chaque requête pour chacun des modules système majeurs. Cette option ne peut pas être utilisée avec l'option `-s`.

`-v` *`protocole`*
Spécifie le numéro de version du protocole client/serveur à utiliser pour cette session particulière.

`-W` *`secondes`*
Dès que cette option est rencontrée, le processus dort pendant le temps spécifié ici en secondes. Ceci donne aux développeurs le temps d'attacher un débogueur sur le processus serveur.

`--describe-config`
Cette option affiche les variables internes de configuration du serveur ainsi que les descriptions et les valeurs par défaut au format `COPY` délimité par des tabulations. Elle est conçue principalement pour être utilisée par des outils d'administration.

# Environnement

`PGDATA`
Emplacement du répertoire de données par défaut.

Pour les autres, qui ont peu d'influence lors d'un mode mono utilisateur, voir postmaster.

# Notes

Pour annuler une requête en cours d'exécution, envoyez le signal `SIGINT` au processus `postgres` exécutant cette commande.

Pour indiquer à `postgres` de recharger les fichiers de configuration, envoyez un signal `SIGHUP`. Normalement, il est mieux de lancer `SIGHUP` à `postmaster` ; le `postmaster` lance à son tour `SIGHUP` à chacun de ces enfants. Mais dans certains cas, il peut être souhaitable de ne recharger les fichiers de configuration que pour un seul processus `postgres`.

Le `postmaster` utilise `SIGTERM` pour indiquer qu'un processus `postgres` quitte normalement et `SIGQUIT` pour terminer sans le nettoyage normal. Ces signaux *ne doivent pas* être utilisés par les utilisateurs. Il est aussi déconseillé d'envoyer `SIGKILL` à un processus `postgres` — le `postmaster` interprète ceci comme un arrêt brutal dans `postgres` et force tous les processus `postgres` à quitter, ce qui fait partie de sa procédure standard de récupération de panne.

# Usage

Lancer un serveur en mode autonome avec une commande comme

```
postgres -D /usr/local/pgsql/data other-options my_database
```

permet de fournir le chemin correct du répertoire de la base de données avec `-D`. On peut aussi s'assurer que la variable d'environnement `PGDATA` est configurée. Cette commande indique aussi le nom de la base de données avec laquelle vous souhaitez travailler.

Normalement, le serveur en mode mono utilisateur traite les retours chariot comme le terminateur d'entrée de commande ; il n'y a pas de compréhension des points virgules comme c'est le cas dans psql. Pour continuer une commande sur plusieurs lignes, vous devez tapez un antislash avant chaque nouvelle ligne sauf la dernière.

Mais si vous utilisez l'option `-N` en ligne de commande, alors un retour chariot ne termine pas l'entrée de la commande. Dans ce cas, le serveur lit l'entrée standard jusqu'au marqueur de fin de fichier (EOF), puis il traite l'entrée comme une seule chaîne de commande. Les retours chariots avec antislash ne sont pas traités de façon spéciale dans ce cas.

Pour quitter la session, tapez EOF (habituellement, **Control+D**). Si vous avez utilisé `-N`, deux EOF consécutifs sont nécessaires pour quitter.

Notez que le serveur en mode autonome ne fournit pas de fonctions avancées d'édition de ligne (pas d'historique de commandes, par exemple).

# Voir aussi

initdb, ipcclean, postmaster

# postmaster

## Nom

postmaster −− serveur de bases de données multi−utilisateurs PostgreSQL

## Synopsis

`postmaster` [−A 0 | 1 ] [−B *nombre_de_tampons*] [−c *nom*=*valeur*] [−d *niveau_de_débogage*] [−D *répertoire_données*] [−F] [−h *nom_hôte*] [−i] [−k *répertoire*] [−l] [−N *nombre_max_connexions*] [−o *options_supplémentaires*] [−p *port*] [−S] [−−*nom*=*valeur*] [−n | −s]

## Description

`postmaster` est le serveur de bases de données multi−utilisateurs PostgreSQL. Pour qu'une application cliente accède à une base de données, elle se connecte (via le réseau ou localement) à un `postmaster` en cours d'exécution. Ensuite, le `postmaster` lance un processus serveur séparé (<< postgres >>) pour gérer la connexion. Le `postmaster` gère aussi la communication entre les processus du serveur.

Par défaut, le `postmaster` se lance en avant plan et affiche les messages des traces dans le flux standard des erreurs. En pratique, le `postmaster` devrait être lancé comme processus d'arrière−plan, par exemple au démarrage.

Un `postmaster` gère toujours les données à partir d'exactement un groupe de bases de données. Ce groupe est une collection de bases de données qui sont stockées à un emplacement commun du système de fichiers (l'<< aire des données >>). Plus d'un processus `postmaster` peut se lancer sur un système à la fois, s'ils utilisent des aires de données et des ports différents (voir ci−dessous). Une aire de données est créée avec initdb.

Quand le `postmaster` se lance, il a besoin de connaître l'emplacement de l'aire des données. L'emplacement doit être indiqué avec l'option `−D` ou la variable d'environnement `PGDATA` ; il n'existe pas de valeur par défaut. Typiquement, `−D` ou `PGDATA` pointe directement vers le répertoire des données créé par initdb. D'autres configurations possibles du fichier sont discutées dans Section 16.4.1.

## Options

`postmaster` accepte les arguments suivants en ligne de commande. Pour une discussion détaillée des options, consultez Section 16.4. Vous pouvez aussi sauvegarder une saisie de la plupart des options en initialisant un fichier de configuration.

`-A 0|1`
: Active les vérifications d'assertion en exécution, qui est une aide de débogage pour détecter les erreurs de programmation. Cette option est seulement disponible si elle était activée lors de la compilation de PostgreSQL. Dans ce cas, la valeur par défaut est << on >>.

`-B` *`nombre_tampons`*
: Initialise le nombre de tampons partagés utilisés par les processus serveur. La valeur par défaut de ce paramètre est choisi par initdb ; référez−vous à Section 16.4.3.1 pour plus d'informations.

`-c` *`nom=valeur`*
: Configure un paramètre d'exécution. Les paramètres de configuration supportés par PostgreSQL sont décrits dans Section 16.4. La plupart des autres options en ligne de commande sont en fait des formes courtes de ces affectations de paramètres. `-c` peut apparaître plusieurs fois pour initialiser plusieurs paramètres.

`-d` *`niveau_de_débogage`*
: Initialise le niveau de débogage. Plus cette valeur est haute, plus la sortie de débogage est importante. Les valeurs vont de 1 to 5.

`-D` *`répertoire_de_données`*
: Spécifie l'emplacement dans le système de fichiers du répertoire de données et des fichiers de configuration. Voir Section 16.4.1 pour plus de détails.

`-F`
: Désactive les appels à `fsync`. Cela permet une amélioration des performances mais fait prendre le risque d'une corruption de données en cas d'arrêt brutal du système. Cette option est équivalente à désactiver le paramètre de configuration fsync. Lire la documentation détaillée avant d'utiliser ceci !

  `--fsync=true` a l'effet contraire de cette option.

`-h` *`nom_hôte`*
: Spécifie le nom d'hôte ou l'adresse IP sur lequel le `postmaster` est en attente de connexions TCP/IP provenant des applications clientes. La valeur peut aussi être une liste d'adresses séparées par des virgules ou `*` pour indiquer d'écouter sur toutes les interfaces disponibles. Une valeur vide indique de ne pas écouter les adresses IP, auquel cas seuls les sockets de domaine Unix peuvent être utilisés pour se connecter au `postmaster`. Par défaut, écoute seulement sur localhost. Spécifier cette option est équivalent à configurer le paramètre listen_addresses.

`-i`
: Autorise la connexion TCP/IP (Internet domain) des clients distants. Sans cette option, seules les connexions locales sont acceptées. Cette option correspond à la configuration `tcpip_socket=true` dans `postgresql.conf`.

  Cette option est obsolète car elle ne permet pas d'accéder à toutes les possibilités de listen_addresses. Il est généralement préférable de configurer `listen_addresses` directement.

`-k` *`répertoire`*
: Spécifie le répertoire du socket de domaine Unix sur lequel le `postmaster` écoute les connexions des applications clientes. La valeur par défaut est normalement `/tmp` mais peut être modifiée au moment de la construction.

`-l`
: Active les connexions sécurisées utilisant SSL. PostgreSQL doit avoir été compilé avec le support de SSL pour que cette option soit disponible. Pour plus d'informations sur l'utilisation de SSL, référez−vous à Section 16.8.

`-N` *`nombre_max_connexions`*
: Initialise le nombre maximum de connexions clientes que ce `postmaster` accepte. Par défaut, cette valeur est 32 mais il peut être initialisé aussi haut que ce que votre système supporte. (Notez que `-B` doit être au moins deux fois la valeur de `-N`. Voir Section 16.5 pour une discussion des besoins de ressources système pour un grand nombre de connexions clientes.) Spécifier cette option est équivalent à configurer le paramètre max_connections.

`-o` *`options_supplémentaires`*
: Les options spécifiées en ligne de commande dans *`options_supplémentaires`* sont passées à tous les processus serveur lancés par ce `postmaster`. Voir postgres pour les possibilités. Si la

chaîne d'option contient des espaces, la chaîne entière doit être entre guillemets.

`-p` *`port`*
: Spécifie le port TCP/IP ou l'extension du fichier socket de domaine Unix local sur lequel écoute le `postmaster` pour la connexion des applications clientes. A pour défaut la valeur de la variable d'environnement `PGPORT` ou, si `PGPORT` n'est pas configurée, alors il a pour valeur par défaut la valeur établie lors de la compilation (normalement 5432). Si vous spécifiez un port autre que celui par défaut, alors toutes les applications clientes doivent spécifier le même port en utilisant soit les options en ligne de commande soit `PGPORT`.

`-S`
: Spécifie que le processus `postmaster` doit se lancer en mode silencieux. C'est–à–dire qu'il est dissocié du terminal (de contrôle) de l'utilisateur, lancé dans son propre groupe de processus et redirige sa propre sortie standard et sa propre sortie standard des erreurs vers `/dev/null`.

  Utiliser cette option désactive tous les messages de traces, ce qui n'est probablement pas ce que vous souhaitez car cela rend le débogage très difficile. Voir ci–dessous un meilleur moyen de lancer `postmaster` en tâche de fond.

  `--silent-mode=false` a l'effet opposé de cette option.

`--`*`nom`*`=`*`valeur`*
: Initialise un paramètre nommé en exécution ; une forme courte de `-c`.

Deux options supplémentaires en ligne de commande sont disponibles pour les débogages de problèmes causant l'arrêt anormal d'un processus de serveur. La stratégie ordinaire dans cette situation est de notifier tous les autres processus serveur qu'ils doivent se terminer et réinitialiser ensuite la mémoire partagée et les sémaphores. Ceci est dû au fait qu'un processus serveur errant peut avoir corrompu quelques états partagés avant de se terminer. Ces options sélectionnent des comportements alternatifs de `postmaster` dans cette situation. *Aucune de ces options n'a pour but d'être utilisée ordinairement.*

Ces options spéciales sont :

`-n`
: `postmaster` réinitialise les structures de données partagées. Un développeur système ayant les connaissances nécessaires peut alors utiliser un débogueur pour examiner l'état des mémoires partagées et des sémaphores.

`-s`
: `postmaster` stoppe tous les autres processus serveur en envoyant le signal `SIGSTOP` mais ne les termine pas. Ceci permet à un développeur système de récupérer manuellement les << core dump >> à partir de tous les processus serveur.

# Environnement

`PGCLIENTENCODING`
: Codage des caractères par défaut utilisé par les clients. (Les clients peuvent surcharger ceci individuellement.) Cette valeur peut aussi être dans le fichier de configuration.

`PGDATA`
: Emplacement par défaut du répertoire de données.

`PGDATESTYLE`
: Valeur par défaut du paramètre d'exécution DateStyle. (L'utilisation de cette variable d'environnement est obsolète.)

PGPORT
: Port par défaut (de préférence, initialisé dans le fichier de configuration)

TZ
: Fuseau horaire du serveur

# Diagnostiques

Un message d'échec mentionnant `semget` ou `shmget` indique probablement que vous avez besoin de configurer votre noyau pour fournir les mémoires partagées et sémaphores adéquats. Pour plus de discussion, voir Section 16.5.

> **Astuce :** Vous pouvez peut−être éviter de reconfigurer votre noyau en décrémentant shared_buffers pour réduire la consommation de mémoire partagée de PostgreSQL et/ou en réduisant max_connections pour réduire la consommation de sémaphores.

Un message d'échec suggérant qu'un autre postmaster est en cours d'exécution doit être vérifié avec attention, par exemple en utilisant la commande

```
$ ps ax | grep postmaster
```

ou

```
$ ps -ef | grep postmaster
```

suivant votre système. Si vous êtes certain qu'aucun postmaster en conflit n'est en cours d'exécution, vous pouvez supprimer le fichier verrou mentionné dans le message et tenter de nouveau.

Un message d'échec indiquant l'incapacité à lier un port peut indiquer que ce port est déjà utilisé par un processus autre que le serveur PostgreSQL. Vous pouvez obtenir cette erreur si vous terminez le `postmaster` et si vous le relancez immédiatement en utilisant le même port ; dans ce cas, vous devez simplement attendre quelques secondes jusqu'à ce que le système d'exploitation ferme le port avant de réessayer. Enfin, vous pouvez obtenir cette erreur si vous spécifiez un numéro de port que votre système d'exploitation considère comme réservé. Par exemple, un grand nombre de versions d'Unix considèrent que les numéros de port inférieurs à 1024 sont << de confiance >> et permettent seulement au superutilisateur Unix d'y accéder.

# Notes

Si c'est possible, *ne pas* utiliser `SIGKILL` pour tuer le processus `postmaster`. L'utilisation de `SIGKILL` empêche `postmaster` de libérer les ressources du système (c'est−à−dire les mémoires partagées et les sémaphores) qu'il détient avant de terminer. Ceci peut causer des problèmes de lancement d'un processus nouveau `postmaster`.

Pour terminer normalement `postmaster`, les signaux `SIGTERM`, `SIGINT` ou `SIGQUIT` peuvent être utilisés. Le premier attend l'arrêt de tous les clients avant de quitter, le second déconnecte de force tous les clients et le troisième quitte immédiatement sans arrêt propre ce qui donne lieu à une opération de récupération lors du redémarrage. Le signal `SIGHUP` recharge les fichiers de configuration du serveur.

L'utilitaire pg_ctl peut être utilisé pour démarrer et arrêter le `postmaster` en toute sûreté et confortablement.

Les options `--` ne fonctionnent pas sur FreeBSD ou OpenBSD. Utilisez `-c` à la place. Ceci est un bogue dans les systèmes d'exploitation affectés ; une prochaine version de PostgreSQL fournit un contournement si ceci n'est pas corrigé.

# Exemples

Pour lancer `postmaster` en tâche de fond en utilisant les valeurs par défaut, saisissez :

```
$ nohup postmaster >logfile 2>&1 </dev/null &
```

Pour lancer `postmaster` sur un port spécifique :

```
$ postmaster -p 1234
```

Cette commande lance un `postmaster` communiquant via le port 1234. Pour se connecter à ce `postmaster` en utilisant psql, il faut lancer par exemple

```
$ psql -p 1234
```

ou initialiser la variable d'environnement `PGPORT` :

```
$ export PGPORT=1234
$ psql
```

Les paramètres d'exécution nommés peuvent être configurés avec un des styles suivants :

```
$ postmaster -c work_mem=1234
$ postmaster --work-mem=1234
```

Chaque forme surcharge le paramétrage pouvant exister pour `work_mem` dans `postgresql.conf`. Notez que les tirets bas dans les noms de paramètre peuvent être écrits soit avec un tiret bas soit avec un tiret sur la ligne de commande.

> **Astuce :** Sauf dans le cas d'expérimentations à court terme, il est conseillé d'éditer le paramétrage dans `postgresql.conf` plutôt que d'utiliser les options de ligne de commande pour configurer un paramètre.

# Voir aussi

initdb, pg_ctl

# VII. Internes

Cette partie contient différentes informations qui peuvent être utiles aux développeurs de PostgreSQL.

**Table des matières**

# Chapitre 40. Présentation des mécanismes internes de PostgreSQL

> **Auteur :** Ce chapitre vient originellement de *Enhancement of the ANSI SQL Implementation of PostgreSQL*, la thèse de Stefan Simkovics préparée à l'université de technologie de Vienne sous la direction du Dr. Georg Gottlob et de son assistante Katrin Seyr.

Ce chapitre présente la structure interne du serveur PostgreSQL. Après avoir lu les sections suivantes, vous devriez avoir une idée sur la façon dont une requête est exécutée. Ce chapitre n'a pas pour but de donner une description détaillée des opérations internes de PostgreSQL car un tel document serait énorme. À la place, ce chapitre a pour but d'aider le lecteur à comprendre la suite générale des opérations arrivant au niveau serveur à partir du moment où une requête est reçue jusqu'au moment où les résultats sont renvoyés au client.

## 40.1. Chemin d'une requête

Ici, nous allons donner un rapide aperçu des étapes qu'une requête doit franchir pour obtenir un résultat.

1. Une connexion doit être établie à partir d'une application au serveur PostgreSQL. L'application transmet une requête au serveur et attend de recevoir les résultats renvoyés par le serveur.
2. L'*étape de l'analyse* vérifie la requête transmise par l'application au niveau de la syntaxe et crée un *arbre représentant la requête*.
3. Le *système de réécriture* prend l'arbre représentant la requête et cherche les *règles* (stockées dans les *catalogues système*) à appliquer à l'arbre de la requête. Il exécute les transformations données dans les *corps des règles*.

   Une application du système de réécriture est la réalisation des *vues*. À chaque fois qu'une requête comprenant une vue (c'est−à−dire une *table virtuelle*) est exécutée, le système de réécriture modifie la requête de l'utilisateur par la *définition de la vue*, c'est−à−dire une requête qui accède aux *tables de base*.
4. Le *planificateur/optimiseur* prend l'arbre (réécrit) de la requête et crée un *plan d'exécution* qui sera l'entrée de l'*exécuteur*.

   Il le fait tout d'abord en créant tous les *chemins* possibles menant au même résultat. Par exemple, s'il existe un index sur une relation à parcourir, il existe deux chemins pour le parcours. Une possibilité est un simple parcours séquentiel et l'autre possibilité est d'utiliser l'index. Ensuite, le coût pour l'exécution de chaque chemin est estimé et le chemin le moins coûteux est choisi. Ce dernier est étendu en un plan complet que l'exécuteur peut utiliser.
5. L'exécuteur passe récursivement dans l'*arbre constitué par le plan* et retrouve les lignes suivant la façon représentée par le plan. L'exécuteur fait usage du *système de stockage* lors du parcours des relations, exécute les *tris* et *jointures*, évalue les *qualifications* et finalement renvoie les lignes en question.

Dans les sections suivantes, nous allons parler de chacun des éléments discutés ci−dessus avec plus de détails pour permettre une meilleure compréhension des structures de données et de contrôle internes de PostgreSQL.

## 40.2. Moyens pour établir des connexions

PostgreSQL est implémenté suivant un modèle client/serveur pour chaque << processus par utilisateur >>. Dans ce modèle, il existe un *processus client* connecté à un seul *processus serveur*. Comme nous ne savons pas par avance combien de connexions seront établies, nous devons utiliser un *processus maître* qui lancera un processus serveur à chaque fois qu'une connexion sera demandée. Ce processus maître s'appelle `postmaster` et écoute sur le port TCP/IP spécifié les connexions entrantes. À chaque fois qu'une demande pour une connexion est détectée, le processus `postmaster` lance un nouveau processus fils appelé `postgres`. Les tâches du serveur (processus `postgres`) communiquent entre elles en utilisant des *sémaphores* et de la *mémoire partagée* pour s'assurer de l'intégrité des données lors d'un accès simultané aux données.

Le processus client peut être tout programme comprenant le protocole PostgreSQL décrit dans le Chapitre 42. Beaucoup de clients sont basés sur la bibliothèque C libpq mais plusieurs implémentations indépendantes du protocole existent, telle que le pilote Java JDBC.

Une fois la connexion établie, le processus client peut envoyer une requête au serveur (*backend*). La requête est transmise en texte simple, c'est−à−dire qu'aucune analyse n'a besoin d'être réalisée au niveau de l'*interface* (client). Le serveur analyse la requête, crée un *plan d'exécution*, exécute le plan et renvoie les lignes trouvées au client par la connexion établie.

## 40.3. Étape d'analyse

L'*étape d'analyse* consiste en deux parties :

- L'*analyseur* défini dans `gram.y` et `scan.l` est construit en utilisant les outils Unix yacc et lex.
- Le *processus de transformation* fait des modifications et des ajouts aux structures de données renvoyées par l'analyseur.

### 40.3.1. Analyseur

L'analyseur doit vérifier que la syntaxe de la chaîne de la requête (arrivant comme un texte ASCII) soit valide. Si la syntaxe est correcte, un *arbre d'analyse* est construit et renvoyé, sinon une erreur est retournée. Les analyseur et vérificateur syntaxiques sont implémentés en utilisant les outils Unix bien connus que sont lex et yacc.

L'*analyseur lexical* est défini dans le fichier `scan.l` et est responsable de la reconnaissance des *identificateurs*, des *mots clés SQL*, etc. Pour chaque mot clé ou identificateur trouvé, un *jeton* est généré et renvoyé à l'analyseur.

L'analyseur est défini dans le fichier `gram.y` et consiste en un ensemble de *règles de grammaire* et en des *actions* à exécuter lorsqu'une règle est découverte. Le code des actions (qui est en langage C) est utilisé pour construire l'arbre d'analyse.

Le fichier `scan.l` est transformé en fichier source C `scan.c` en utilisant le programme lex et `gram.y` est transformé en `gram.c` en utilisant yacc. Après avoir réalisé ces transformations, un compilateur C normal peut être utilisé pour créer l'analyseur. Ne faites jamais de changement aux fichiers C générés car ils seront écrasés la prochaine fois que lex ou yacc seront appelés.

> **Note :** Les transformations et compilations mentionnées sont normalement réalisées automatiquement en utilisant les *makefile* distribués avec les sources de PostgreSQL.

Une description détaillée de yacc ou des règles de grammaire données dans `gram.y` serait en dehors du champ de ce document. Il existe de nombreux livres et documentations en relation avec lex et yacc. Vous devez être familier avec yacc avant de commencer à étudier la grammaire donnée dans `gram.y`, sinon vous ne comprendrez rien à ce qui s'y passe.

### 40.3.2. Processus de transformation

L'étape d'analyse crée un arbre d'analyse utilisant seulement les règles fixes de la structure syntaxique de SQL. Il ne fait aucune recherche dans les catalogues système, donc il n'y a aucune possibilité de comprendre la sémantique détaillée des opérations demandées. Après que l'analyseur ait fini, le *processus de transformation* prend l'arbre résultant de l'analyseur en entrée et réalise l'interprétation sémantique nécessaire pour connaître les tables, fonctions et opérateurs référencés par la requête. La structure de données construite pour représenter cette information est appelé l'*arbre de la requête*.

La raison de la séparation de l'analyse brute et de l'analyse sémantique est que les recherches des catalogues système peuvent seulement se faire à l'intérieur d'une transaction, et nous ne voulons pas commencer une transaction immédiatement après avoir reçu une requête. L'analyse brute est suffisante pour identifier les commandes de contrôle des transactions (`BEGIN`, `ROLLBACK`, etc) et elles peuvent être correctement exécutées sans plus d'analyse. Une fois que nous savons que nous gérons une vraie requête (telle que `SELECT` ou `UPDATE`), nous pouvons commencer une transaction si nous n'y sommes pas déjà. C'est seulement maintenant que le processus de transformation peut être appelé.

La plupart du temps, l'arbre d'une requête créé par le processus de transformation a une structure similaire à l'arbre d'analyse brute mais il existe beaucoup de différences dans le détail. Par exemple, un nœud `FuncCall` dans l'arbre d'analyse représente quelque chose qui ressemble syntaxiquement à l'appel d'une fonction. Il peut être transformé soit en nœud `FuncExpr` soit en nœud `Aggref` suivant que le nom référencé est une fonction ordinaire ou une fonction d'agrégat. De même, des informations sur les types de données réels des colonnes et des résultats sont ajoutées à l'arbre de la requête.

## 40.4. Système de règles de PostgreSQL

PostgreSQL supporte un puissant *système de règles* pour la spécification des *vues* et des *mises à jour de vues* ambiguës. À l'origine, le système de règles de PostgreSQL consistait en deux implémentations :

- Le premier a fonctionné en utilisant le *niveau des lignes* et était implémenté profondément dans l'*exécuteur*. Le système de règles était appelé à chaque fois qu'il fallait accéder à une ligne individuelle. Cette implémentation a été supprimée en 1995 quand la dernière version officielle du projet Berkeley Postgres a été transformé en Postgres95.
- La deuxième implémentation du système de règles est une technique appelée la *réécriture de requêtes*. Le *système de réécriture* est un module qui existe entre l'*étape d'analyse* et le *planificateur/optimiseur*. Cette technique est toujours implémentée.

Le système de réécriture de requêtes est vu plus en détails dans le Chapitre 33, donc il n'est pas nécessaire d'en parler ici. Nous indiquerons seulement qu'à la fois l'entrée et la sortie du système sont des arbres de requêtes, c'est−à−dire qu'il n'y a pas de changement dans la représentation ou le niveau du détail sémantique des arbres. La réécriture peut être imaginée comme une forme d'expansion de macro.

# 40.5. Planificateur/Optimiseur

La tâche du *planificateur/optimiseur* est de créer un plan d'exécution optimal. En fait, une requête SQL donnée (et donc, l'arbre d'une requête) peut être exécutée de plusieurs façons, chacune arrivant au même résultat. Si ce calcul est possible, l'optimiseur de la requête examinera chacun des plans d'exécution possibles pour finir par sélectionner le plan d'exécution estimé comme étant le plus rapide.

> **Note :** Dans certaines situations, examiner chaque moyen d'exécuter une requête prendrait beaucoup de temps et de mémoire. En particulier, cela arrive lors de l'exécution de requêtes impliquant un grand nombre de jointures. Pour déterminer un plan de requête raisonnable (et non optimal) dans un temps raisonnable, PostgreSQL utilise un *Optimiseur génétique de requêtes (Genetic Query Optimizer)*.

La procédure de recherche du planificateur fonctionne en fait avec des structures de données appelés *chemins*, qui sont simplement des représentations minimales de plans contenant seulement l'information nécessaire pour que le planificateur puisse prendre ses décisions. Une fois le chemin le moins coûteux déterminé, un *arbre plan* est construit pour être donné à l'exécuteur. Ceci représente le plan d'exécution désiré avec suffisamment de détails pour que l'exécuteur puisse le lancer. Dans le reste de cette section, nous ignorerons la distinction entre chemins et plans.

## 40.5.1. Générer les plans possibles

Le planificateur/optimiseur commence par générer des plans pour parcourir chaque relation (table) invididuelle utilisée dans la requête. Les plans possibles sont déterminés par les index disponibles pour chaque relation. Il est toujours possible de réaliser un parcours séquentiel sur une relation, donc un plan de parcours séquentiel est toujours créé. Supposons qu'un index soit défini sur une relation (par exemple un index B−tree) et qu'une requête contienne le filtre `relation.attribut OPR constante`. Si `relation.attribut` correspond à la clé de l'index B−tree et `OPR` est un des opérateurs listés dans la *classe d'opérateurs* de l'index, un autre plan est créé en utilisant l'index B−tree pour parcourir la relation. S'il existe d'autres index et que les restrictions de la requête font correspondre une clé à un index, d'autres plans seront considérés.

Une fois que tous les plans probables ont été trouvés pour parcourir les relations simples, des plans pour les relations jointes sont créés. Le planificateur/optimiseur préfère considérer les jointures entre deux relations pour lesquelles il existe une clause de jointure correspondante dans la qualification `WHERE` (c'est−à−dire pour lequel une restriction comme `where rel1.attr1=rel2.attr2` existe). Des paires de jointures sans clause de jointure sont considérées seulement quand il n'existe pas d'autre choix, c'est−à−dire lorsqu'une relation particulière n'a pas de clause de jointure disponible vers toute autre relation. Tous les plans possibles sont générés pour chaque paire de jointure considérée par le planificateur/optimiseur. Les trois stratégies possibles de jointure sont :

- *jointure de boucle imbriquée* (NdT : nested loop join) : la relation de droite est parcourue une fois pour chaque ligne trouvée dans la relation de gauche. Cette stratégie est facile à implémenter mais peut être très coûteuse en temps. (Par contre, si la relation de droite peut être parcourue à l'aide d'un index, ceci peut être une bonne stratégie. Il est possible d'utiliser des valeurs provenant de la ligne actuelle de la relation de gauche comme clés pour un parcours indexé à droite.)
- *jointure tri et assemblage* (NdT : merge sort join) : Chaque relation est triée sur les attributs de la jointure avant que la jointure ne commence. Puis, les deux relations sont parcourues en parallèle et les lignes correspondantes sont combinées pour former des lignes jointes. Ce type de jointure est plus

intéressant parce que chaque relation n'est parcourue qu'une seule fois. Le tri requis pourrait être réalisé soit par une étape explicite de tri soit en parcourant la relation dans le bon ordre en utilisant un index sur la clé de la jointure.

- *jointure de découpage* (hash join) : la relation de droite est tout d'abord parcourue et chargée dans une table de découpage en utilisant ses attributs de jointure comme clés de découpage. Ensuite, la relation de gauche est parcourue et les valeurs appropriées de chaque ligne trouvée sont utilisées comme clés de découpage pour localiser les lignes correspondantes dans la table.

Quand la requête implique plus de deux relations, le résultat final doit être construit par un arbre d'étapes de jointure, chacune ayant deux entrées. Le planificateur examine les séquences de jointure possibles pour trouver le moins cher.

L'arbre de plan terminé est composé de parcours séquentiels ou indexés des relations de base, plus les nœuds des jointures en boucle, jointures tri et rassemblement, et jointures de découpage si nécessaire, plus toutes les étapes auxiliaires nécessaires, tels que les nœuds de tri ou les nœuds de calcul des fonctions d'agrégat. La plupart des types de nœud de plan ont la capacité supplémentaire de faire une *sélection* (rejetant les lignes qui ne correspondent pas à une condition booléenne spécifiée) et une *projection* (calcul d'un ensemble de colonnes dérivées basé sur des valeurs de colonnes données, c'est−à−dire l'évaluation d'expressions scalaires si nécessaire). Une des responsabilités du planificateur est d'attacher les conditions de sélection à partir de la clause `WHERE` et du calcul des expressions de calculs requises aux nœuds les plus appropriés de l'arbre de plan.

## 40.6. Exécuteur

L'*exécuteur* prend le plan envoyé par le planificateur/optimiseur et l'exécute récursivement pour extraire l'ensemble requis de lignes. Il s'agit principalement d'un mécanisme de pipeline en demande−envoi. Chaque fois qu'un nœud du plan est appelé, il doit apporter une ligne supplémentaire ou indiquer qu'il a fini d'envoyer des lignes.

Pour donner un exemple concret, supposons que le nœud supérieur soit un nœud `MergeJoin`. Avant de pouvoir faire une fusion, deux lignes doivent être récupérées (une pour chaque sous−plan). L'exécuteur s'appelle donc récursivement pour exécuter les sous−plans (en commençant par le sous−plan attaché à l'`arbre gauche`). Le nouveau nœud supérieur (le nœud supérieur du sous−plan gauche) est, disons, un nœud `Sort` (NdT : Tri) et un appel récursif est une nouvelle fois nécessaire pour obtenir une ligne en entrée. Le nœud fils de `Sort` pourrait être un nœud `SeqScan`, représentant la lecture réelle d'une table. L'exécution de ce nœud fait que l'exécuteur récupère une ligne à partir de la table et la renvoie au nœud appelant. Le nœud `Sort` appellera de façon répétée son fils pour obtenir toutes les lignes à trier. Quand l'entrée est terminée (indiqué par le nœud fils renvoyant un NULL au lieu d'une ligne), le code de `Sort` est enfin capable de renvoyer sa première ligne en sortie, c'est−à−dire le premier suivant l'ordre de tri. Il conserve les lignes restantes en mémoire de façon à les renvoyer dans le bon ordre en réponse à des demandes ultérieures.

Le nœud `MergeJoin` demande de la même façon la première ligne à partir du sous−plan droit. Ensuite, il compare les deux lignes pour savoir si elles peuvent être jointes ; si c'est le cas, il renvoie la ligne de jointure à son appelant. Au prochain appel, ou immédiatement s'il ne peut pas joindre la paire actuelle d'entrées, il avance sur la prochaine ligne d'une des deux tables (suivant le résultat de la comparaison), et vérifie de nouveau la correspondance. Finalement, un sous−plan est terminé et le nœud `MergeJoin` renvoie NULL pour indiquer qu'il n'y a plus de lignes jointes à former.

Les requêtes complexes peuvent nécessiter un grand nombre de niveaux de nœuds pour les plans, mais l'approche générale est la même : chaque nœud est exécuté et renvoie sa prochaine ligne en sortie à chaque fois qu'il est appelé. Chaque nœud est responsable aussi de l'application de toute expression de sélection ou de projection qui lui ont été confiées par le planificateur.

Le mécanisme de l'exécuteur est utilisé pour évaluer les quatre types de requêtes de base en SQL : `SELECT`, `INSERT`, `UPDATE` et `DELETE`. Pour `SELECT`, le code de l'exécuteur de plus haut niveau a seulement besoin d'envoyer chaque ligne retournée par l'arbre plan de la requête vers le client. Pour `INSERT`, chaque ligne renvoyée est insérée dans la table cible spécifiée par le `INSERT`. (Une simple commande `INSERT ... VALUES` crée un arbre plan trivial consistant en un seul nœud, `Result`, calculant une seule ligne de résultat. Mais `INSERT ... SELECT` peut demander la pleine puissance du mécanisme de l'exécuteur.) Pour `UPDATE`, le planificateur s'arrange pour que chaque ligne calculée inclue toutes les valeurs mises à jour des colonnes, plus le *TID* (tuple ID ou l'identifiant de la ligne) de la ligne de la cible originale ; l'exécuteur de plus haut niveau utilise cette information pour créer une nouvelle ligne mise à jour et pour marquer la suppression de l'ancienne ligne. Pour `DELETE`, la seule colonne renvoyée par le plan est le TID, et l'exécuteur de plus haut niveau utilise simplement le TID pour visiter chaque ligne cible et la marquer comme supprimée.

# Chapitre 41. Catalogues système

Les catalogues système sont le lieu où une base de données relationnelle stocke les métadonnées des schémas, comme les informations sur les tables et les colonnes, et des données de suivi internes. Les catalogues système de PostgreSQL sont de simples tables. Vous pouvez les supprimer et les recréer, ajouter des colonnes, insérer et mettre à jour des valeurs, et mettre un joyeux bazar dans votre système. Normalement, on ne devrait pas modifier les catalogues système soi−même, il y a toujours des commandes SQL pour le faire. (Par exemple, `CREATE DATABASE` insère une ligne dans le catalogue `pg_database` — et crée physiquement la base de données sur le disque.) Il y a des exceptions pour certaines opérations particulièrement ésotériques, comme l'ajout de méthodes d'accès aux index.

## 41.1. Aperçu

Tableau 41−1 liste les catalogues système. Une documentation détaillée sur les catalogues systèmes vient plus loin.

La plupart des catalogues système sont recopiés de la base de données modèle lors de la création de la base de données et sont donc spécifiques à chaque base de données. Un petit nombre de catalogues sont physiquement partagés par toutes les bases de données d'une installation de PostgreSQL. Ils sont indiqués dans les descriptions des catalogues.

**Tableau 41−1. Catalogues système**

| Nom du catalogue | Contenu |
|---|---|
| pg_aggregate | fonctions d'agrégat |
| pg_am | méthodes d'accès aux index |
| pg_amop | opérateurs des méthodes d'accès |
| pg_amproc | procédures de support des méthodes d'accès |
| pg_attrdef | valeurs par défaut des colonnes |
| pg_attribute | colonnes des tables (<< attributs >>) |
| pg_cast | conversions de types de données (cast) |
| pg_class | tables, index, séquences, vues (<< relations >>) |
| pg_constraint | contraintes de vérification, contraintes uniques, contraintes de clés primaires, contraintes de clés étrangères |
| pg_conversion | informations de conversions de codage |
| pg_database | bases de données de l'installation PostgreSQL |
| pg_depend | dépendances entre objets de la base de données |
| pg_description | descriptions ou commentaires des objets de base de données |
| pg_group | groupes d'utilisateurs de la base de données |
| pg_index | informations supplémentaires des index |
| pg_inherits | hiérarchie d'héritage de tables |
| pg_language | langages pour écrire des fonctions |
| pg_largeobject | gros objets |
| pg_listener | support de notification asynchrone |

| | |
|---|---|
| pg_namespace | schémas |
| pg_opclass | classes d'opérateurs de méthodes d'accès aux index |
| pg_operator | opérateurs |
| pg_proc | fonctions et procédures |
| pg_rewrite | règles de réécriture de requêtes |
| pg_shadow | utilisateurs de la base de données |
| pg_statistic | statistiques de l'optimiseur de requêtes |
| pg_tablespace | espaces logiques dans ce groupe de bases de données |
| pg_trigger | déclencheurs |
| pg_type | types de données |

## 41.2. pg_aggregate

Le catalogue pg_aggregate stocke les informations sur les fonctions d'agrégat. Une fonction d'agrégat est une fonction qui opère sur un ensemble de données (typiquement une colonne de chaque ligne qui correspond à une condition de requête) et retourne une valeur unique calculée à partir de toutes ces valeurs. Les fonctions d'agrégat classiques sont sum (somme), count (compteur) et max (plus grande valeur). Chaque entrée dans pg_aggregate est une extension d'une entrée dans pg_proc. L'entrée de pg_proc décrit le nom de l'agrégat, les types de données d'entrée et de sortie, et d'autres informations des fonctions ordinaires.

**Tableau 41−2. Les colonnes de pg_aggregate**

| Nom | Type | Références | Description |
|---|---|---|---|
| aggfnoid | regproc | pg_proc.oid | OID pg_proc de la fonction d'agrégat |
| aggtransfn | regproc | pg_proc.oid | fonction de transition |
| aggfinalfn | regproc | pg_proc.oid | fonction finale (zéro s'il n'y en a pas) |
| aggtranstype | oid | pg_type.oid | Type de la donnée interne de transition (état) de la fonction d'agrégat |
| agginitval | text | | Valeur initiale de la fonction de transition. C'est un champ texte qui contient la valeur initiale dans sa représentation externe en chaîne de caractères. Si la valeur est NULL, la valeur d'état de transition est initialement NULL. |

Les nouvelles fonctions d'agrégat sont enregistrées avec la commande CREATE AGGREGATE. Lisez la Section 31.10 pour avoir plus d'informations sur l'écriture des fonctions d'agrégat et sur la signification des fonctions de transition, etc.

## 41.3. pg_am

Le catalogue pg_am enregistre les informations sur les méthodes d'accès aux index. Il y a une ligne pour chaque méthode d'accès supportée par le système.

**Tableau 41−3. Colonnes de pg_am**

| Nom | Type | Références | Description |
|---|---|---|---|
| amname | name | | Nom de la méthode d'accès |
| amowner | int4 | pg_shadow.usesysid | ID utilisateur du propriétaire (actuellement non utilisé) |
| amstrategies | int2 | | Nombre de stratégies d'opérateur pour cette méthode d'accès |
| amsupport | int2 | | Nombre de routines de support pour cette méthode d'accès |
| amorderstrategy | int2 | | Zéro si l'index n'a pas d'ordre de tri, sinon, numéro de de stratégie de l'opérateur de stratégie qui décrit l'ordre de tri |
| amcanunique | bool | | Est−ce que la méthode d'accès supporte les index uniques ? |
| amcanmulticol | bool | | Est−ce que la méthode d'accès supporte les index multicolonnes ? |
| amindexnulls | bool | | Est−ce que la méthode d'accès supporte les entrées d'index NULLs ? |
| amconcurrent | bool | | Est−ce que la méthode d'accès supporte les mises à jour concurrentes ? |
| amgettuple | regproc | pg_proc.oid | Fonction << prochaine ligne valide >> |
| aminsert | regproc | pg_proc.oid | Fonction << insérer cette ligne >> |
| ambeginscan | regproc | pg_proc.oid | Fonction << commencer un nouveau balayage >> |
| amrescan | regproc | pg_proc.oid | Fonction << redémarrer ce balayage >> |
| amendscan | regproc | pg_proc.oid | Fonction << arrêter ce balayage >> |
| ammarkpos | regproc | pg_proc.oid | Fonction << marquer la position actuelle dans le balayage >> |
| amrestrpos | regproc | pg_proc.oid | Fonction << restaurer une position de balayage sauvegardée >> |
| ambuild | regproc | pg_proc.oid | Fonction << construire un nouvel index >> |
| ambulkdelete | regproc | pg_proc.oid | Fonction de destruction en masse |
| amvacuumcleanup | regproc | pg_proc.oid | Fonction de nettoyage post−VACUUM |
| amcostestimate | regproc | pg_proc.oid | Fonction d'estimation du coût d'un balayage d'index |

Une méthode d'accès qui supporte les colonnes multiples (qui a amcanmulticol à vrai) *doit* supporter l'indexation des valeurs NULL dans les colonnes autres que la première, parce que l'optimiseur de requêtes supposera que le parcours d'index peut être utilisé pour les requêtes portant seulement sur la ou les première(s) colonne(s). Par exemple, supposons un index sur (a,b) et une requête contenant WHERE a = 4. Le système supposera que le parcours d'index peut être utilisé pour balayer les lignes pour lesquelles a = 4, ce qui est faux si l'index omet les lignes pour lesquelles b est nul. Il est cependant possible d'omettre les lignes dont la première colonne d'index est NULL. (GiST le fait). amindexnulls ne doit être mis à vrai que si la méthode d'accès indexe toutes les lignes, y compris toute combinaison de valeurs NULL.

## 41.4. pg_amop

Le catalogue pg_amop stocke les informations sur les opérateurs associés aux classes d'opérateurs de méthodes d'accès aux index. Il y a une ligne pour chaque opérateur qui est membre d'une classe d'opérateurs.

**Tableau 41−4. Colonnes de pg_amop**

| Nom | Type | Références | Description |
|---|---|---|---|
| amopclaid | oid | pg_opclass.oid | La classe d'opérateur d'index de cette entrée. |
| amopsubtype | oid | pg_type.oid | Sous−type pour distinguer plusieurs entrées d'une stratégie ; zéro par défaut |
| amopstrategy | int2 | | Numéro de stratégie d'opérateur |
| amopreqcheck | bool | | Une entrée trouvée dans l'index doit être revérifiée. |
| amopopr | oid | pg_operator.oid | OID de l'opérateur |

# 41.5. pg_amproc

Le catalogue pg_amproc enregistre des informations sur les procédures de support associées aux classes d'opérateurs de méthodes d'accès. Il y a une ligne pour chaque procédure de support appartenant à une classe d'opérateur.

**Tableau 41−5. Colonnes de pg_amproc**

| Nom | Type | Références | Description |
|---|---|---|---|
| amopclaid | oid | pg_opclass.oid | La classe d'opérateurs d'index de cette entrée |
| amprocsubtype | oid | pg_type.oid | Sous−type dans le cas d'une routine intertype, zéro sinon |
| amprocnum | int2 | | Numéro de procédure de support |
| amproc | regproc | pg_proc.oid | OID de la procédure |

# 41.6. pg_attrdef

Le catalogue pg_attrdef stocke les valeurs par défaut des colonnes. Les informations principales des colonnes sont stockées dans pg_attribute (voir plus loin). Seules les colonnes pour lesquelles une valeur par défaut est explicitement spécifiée (quand la table est créée ou quand une colonne est ajoutée) ont une entrée dans pg_attrdef.

**Tableau 41−6. Colonnes de pg_attrdef**

| Nom | Type | Références | Description |
|---|---|---|---|
| adrelid | oid | pg_class.oid | La table de cette colonne. |
| adnum | int2 | pg_attribute.attnum | Numéro de la colonne |
| adbin | text | | Représentation interne de la valeur par défaut de la colonne |
| adsrc | text | | Une représentation lisible de la valeur par défaut |

Le champ adsrc est historique et il est mieux de ne pas l'utiliser parce qu'il ne conserve pas trace des modifications qui pourraient affecter la représentation de la valeur par défaut. La compilation inverse du champ adbin (avec pg_get_expr par exemple) est une meilleure façon d'afficher la valeur par défaut.

# 41.7. pg_attribute

Le catalogue pg_attribute stocke les informations sur les colonnes des tables. Il y a exactement une ligne de pg_attribute pour chaque colonne de chaque table de la base de données. (Il y a aussi des attributs pour les index et, en fait, tous les objets qui possèdent des entrées dans pg_class.)

Le terme attribut est équivalent à colonne et est utilisé pour des raisons historiques.

**Tableau 41–7. Colonnes de pg_attribute**

| Nom | Type | Références | Description |
|---|---|---|---|
| attrelid | oid | pg_class.oid | La table de cette colonne |
| attname | name | | Le nom de la colonne |
| atttypid | oid | pg_type.oid | Le type de données de cette colonne |
| attstattarget | int4 | | attstattarget contrôle le niveau de détail des statistiques accumulées pour cette colonne par *ANALYZE*. Une valeur de zéro indique qu'aucune statistique ne doit être collectée. Une valeur négative indique d'utiliser l'objectif de statistiques par défaut. Le sens exacte d'une valeur positive dépend du type de données. Pour les données scalaires, attstattarget est à la fois le nombre visé de << valeurs les plus courantes >> et le nombre visé d'histogrammes à créer. |
| attlen | int2 | | Une copie de pg_type.typlen pour le type de cette colonne. |
| attnum | int2 | | Le numéro de la colonne. Les colonnes ordinaires sont numérotées en commençant par 1. Les colonnes système, comme les oid, ont des numéros négatifs arbitraires. |
| attndims | int4 | | Nombre de dimensions, si la colonne est de type tableau, sinon 0. (Pour l'instant, le nombre de dimensions des tableaux n'est pas contrôlé, donc une valeur autre que zéro indique que << c'est un tableau >>.) |
| attcacheoff | int4 | | Toujours −1 sur disque, mais peut être mis à jour, en mémoire, pour mettre en cache l'emplacement de l'attribut dans la ligne. |
| atttypmod | int4 | | atttypmod des données spécifiques au type de données précisé lors de la création de la table (par exemple, la taille maximale d'une colonne de type varchar). Il est transmis à des fonctions spécifiques au type d'entrée de données et de vérification de taille. La valeur est généralement de −1 pour les types de données qui n'ont pas besoin de atttypmod. |
| attbyval | bool | | Une copie de pg_type.typbyval pour ce type de données. |
| attstorage | char | | Contient normalement une copie de pg_type.typstorage pour ce type de données. Pour les types de données TOASTables, cette valeur peut être modifiée après la création de la colonne pour contrôler la règle de stockage. |
| attalign | char | | |

| | | | |
|---|---|---|---|
| | | | Contient une copie de pg_type.typalign pour le type de cette colonne. |
| attnotnull | bool | | Indique une contrainte de colonne non NULL. Il est possible de changer cette colonne pour activer ou désactiver cette contrainte. |
| atthasdef | bool | | Indique que cette colonne a une valeur par défaut. Dans ce cas, il y aura une entrée correspondante dans le catalogue pg_attrdef pour définir cette valeur. |
| attisdropped | bool | | Indique que cette colonne a été supprimée et n'est plus valide. Une colonne supprimée est toujours présente physiquement dans la table, mais elle est ignorée par l'analyseur de requête et ne peut être accédée en SQL. |
| attislocal | bool | | Cette colonne est définie localement dans la relation. Notez qu'une colonne peut être définie localement et héritée simultanément. |
| attinhcount | int4 | | Nombre d'ancêtres directs de cette colonne. Une colonne qui a un nombre d'ancêtres différent de zéro ne peut être supprimée ni renommée. |

Dans l'entrée pg_attribute d'une colonne supprimée, atttypid est réinitialisée à zéro mais attlen et les autres champs copiés à partir de pg_type sont toujours valides. Cet arrangement est nécessaire pour s'adapter à la situation où le type de données de la colonne supprimée a été ensuite supprimé et qu'il n'existe donc plus de ligne pg_type. attlen et les autres champs peuvent être utilisés pour interpréter le contenu d'une ligne de la table.

## 41.8. pg_cast

Le catalogue pg_cast stocke les chemins de conversion de type de donnée, qu'ils soient par défaut ou définis avec la commande CREATE CAST.

**Tableau 41–8. Colonnes de pg_cast**

| Nom | Type | Références | Description |
|---|---|---|---|
| castsource | oid | pg_type.oid | OID du type de données source |
| casttarget | oid | pg_type.oid | OID du type de données en sortie |
| castfunc | oid | pg_proc.oid | OID de la fonction à utiliser pour faire cette conversion. Vaut Zéro si les types de données sont binairement compatibles (c'est–à–dire si aucune opération n'est nécessaire pour effectuer la transformation). |
| castcontext | char | | Indique dans quel contexte la conversion peut être utilisée. e si seules les conversions explicites sont autorisées (avec CAST ou ::). a si les conversions implicites lors de l'affectation à une colonne sont autorisées, en plus des conversions explicites. i si les conversions implicites dans les expressions sont autorisées en plus des autres cas. |

Les fonctions de conversion listées dans pg_cast doivent toujours prendre le type source de la conversion comme type du premier argument et renvoyer le type de destination de la conversion comme type de retour. Une fonction de conversion peut avoir jusqu'à trois arguments. Le deuxième argument, s'il est présent, doir être du type integer ; il reçoit le modificateur de type associé avec le type de destination ou -1 s'il n'y en a

pas. Le troisième argument, s'il est présent, doit être du type boolean ; il reçoit true si la conversion est une conversion explicite, false sinon.

Il est légitime de créer une entrée pg_cast dans laquelle les types source et cible sont les mêmes, si la fonction associée prend plus d'un argument. De telles entrées représentent les << fonctions de coercition de longueur >> qui forcent les valeurs du type à être légal pour une valeur particulière du modificateur de type. Néanmoins, notez qu'à présent il n'existe aucun support pour associer des modifications de type autres que ceux de base avec des types de données créés par l'utilisateur. Du coup, cette fonctionnalité est seulement utile pour un petit nombre de types intégrés qui ont une syntaxe de modificateur de type construit dans leur grammaire.

Quand une entrée pg_cast a des types différents pour la source et la destination et qu'une fonction prend plus d'un argument, il représente la conversion d'un type vers un autre et la coercition en une seule étape. Quand une telle entrée est absente, la coercition vers une type qui utilise un modificateur de type implique deux étapes, une pour convertir entre les types de données et une seconde pour appliquer le modificateur.

## 41.9. pg_class

Le catalogue pg_class catalogue les tables, et à peu près tout ce qui a des colonnes ou qui ressemble de près ou de loin à une table. Cela inclut les index (mais il faut aussi aller voir dans pg_index), les séquences, les vues, les types composites et certaines sortes de relations spéciales ; voir relkind. Par la suite, lorsque l'on parle de << relation >>, on parle de tous ces types d'objets. Toutes les colonnes n'ont pas un sens pour tous les types de relations.

**Tableau 41–9. Colonnes de pg_class**

| Nom | Type | Références | Description |
|---|---|---|---|
| relname | name | | Nom de la table, vue, index, etc. |
| relnamespace | oid | pg_namespace .oid | OID de l'espace de noms qui contient cette relation. |
| reltype | oid | pg_type .oid | OID du type de données qui correspond au type de ligne de cette table, s'il y en a un. Zéro pour les index qui n'ont pas d'entrée dans pg_type. |
| relowner | int4 | pg_shadow .usesysid | Propriétaire de la relation. |
| relam | oid | pg_am .oid | Si c'est un index, OID de la méthode d'accès utilisée (B−tree, hash, etc.) |
| relfilenode | oid | | Nom du fichier disque de cette relation ; 0 s'il n'y en a pas. |
| reltablespace | oid | pg_tablespace.oid | L'espace logique dans lequel cette relation est stockée. Si zéro, l'espace logique par défaut de cette base de données est utilisée. |

| | | | |
|---|---|---|---|
| | | | (Sans intérêt si la relation n'a pas de fichier sur disque.) |
| relpages | int4 | | Taille du fichier disque, exprimée en pages (de taille BLCKSZ). Ce n'est qu'une estimation utilisée par l'optimiseur. Elle est mise à jour par les commandes VACUUM, ANALYZE et quelques commandes DDL comme CREATE INDEX. |
| reltuples | float4 | | Nombre de lignes de la table. Ce n'est qu'une estimation utilisée par l'optimiseur. Elle est mise à jour par les commandes VACUUM, ANALYZE et quelques commandes DDL comme CREATE INDEX. |
| reltoastrelid | oid | pg_class .oid | OID de la table TOAST associée à cette table. 0 s'il n'y en a pas. La table TOAST stocke les attributs de grande taille << hors de la ligne >> dans une table secondaire. |
| reltoastidxid | oid | pg_class .oid | Pour une table TOAST, OID de son index. 0 si ce n'est pas une table TOAST. |
| relhasindex | bool | | Vrai si cette table a (ou a eu récemment) un ou plusieurs index. Ce champ est mis à jour par CREATE INDEX, mais pas remis à faux immédiatement après DROP INDEX. VACUUM remet relhasindex à faux s'il s'aperçoit que la table n'a pas d'index. |
| relisshared | bool | | Vrai si cette table est partagée par toutes les bases de données du groupe de bases de données. Seuls certains catalogues système (comme pg_database) sont partagés. |
| relkind | char | | r = table ordinaire, i = index, S = séquence, v = vue, c = type composite, s = spécial, t = table TOAST. |
| relnatts | int2 | | Nombre de colonnes utilisateur dans la relation (sans compter les colonnes système). Il doit y avoir le même nombre d'entrées dans |

| | | | |
|---|---|---|---|
| | | | pg_attribute. Voir aussi pg_attribute.attnum. |
| relchecks | int2 | | Nombre de contraintes de vérification sur la table ; voir le catalogue pg_constraint. |
| reltriggers | int2 | | Nombre de déclencheurs sur la table ; voir le catalogue pg_trigger. |
| relukeys | int2 | | Inutilisé. (Ce n'est *pas* le nombre de clés uniques de la table.) |
| relfkeys | int2 | | Inutilisé. (Ce n'est *pas* le nombre de clés étrangères de la table.) |
| relrefs | int2 | | Inutilisé |
| relhasoids | bool | | Vrai si on génère un OID pour chaque ligne de la relation. |
| relhaspkey | bool | | Vrai si la table a (ou a eu) une clé primaire. |
| relhasrules | bool | | Vrai si la table comprend des règles ; voir le catalogue pg_rewrite. |
| relhassubclass | bool | | Vrai si au moins une table hérite ou a hérité de celle−ci. |
| relacl | aclitem[] | | Droits d'accès ; voir *GRANT* et *REVOKE* pour plus de détails. |

## 41.10. pg_constraint

Le catalogue pg_constraint stocke les vérifications, clés primaires, clés uniques et étrangères des tables (Les contraintes de colonnes ne sont pas traitées de manière particulière. Elles sont équivalentes à des contraintes de tables.) Les contraintes NOT NULL sont représentées dans le catalogue pg_attribute.

Les contraintes de vérification de domaine sont aussi stockées ici.

**Tableau 41−10. Colonnes de pg_constraint**

| Nom | Type | Références | Description |
|---|---|---|---|
| conname | name | | Nom de la contrainte (pas nécessairement unique !) |
| connamespace | oid | pg_namespace .oid | OID de l'espace de noms qui contient cette contrainte. |
| contype | char | | c = contrainte de vérification, f = contrainte de clé étrangère, p = contrainte de clé primaire, u = contrainte de clé unique |
| condeferrable | bool | | La contrainte est−elle différable ? |

| | | | |
|---|---|---|---|
| condeferred | bool | | La contrainte est−elle différée par défaut ? |
| conrelid | oid | pg_class.oid | Table à laquelle appartient la contrainte ; 0 si ce n'est pas une contrainte de table. |
| contypid | oid | pg_type .oid | Domaine auquel appartient la contrainte ; 0 si ce n'est pas une contrainte de domaine. |
| confrelid | oid | pg_class .oid | Si c'est une clé étrangère, la table référencée ; sinon 0 |
| confupdtype | char | | Code de l'action de mise à jour de la clé étrangère |
| confdeltype | char | | Code de l'action de suppression de clé étrangère |
| confmatchtype | char | | Type de vérification de clé étrangère |
| conkey | int2[] | pg_attribute .attnum | Si c'est une contrainte de table, liste des colonnes contraintes |
| confkey | int2[] | pg_attribute .attnum | Si c'est une clé étrangère, liste des colonnes référencées |
| conbin | text | | S'il s'agit d'une contrainte de vérification, représentation interne de l'expression |
| consrc | text | | S'il s'agit d'une contrainte de vérification, représentation compréhensible de l'expression |

**Note :** consrc n'est pas mis à jour lors de modification d'objets référencés ; par exemple, il ne pistera pas les renommages de colonnes. Plutôt que se fier à ce champ, il est mieux d'utiliser pg_get_constraintdef() pour extraire la définition d'une contrainte de vérification.

**Note :** pg_class.relchecks doit connaître le même nombre de contraintes de vérification pour chaque relation.

## 41.11. pg_conversion

Le catalogue pg_conversion décrit les procédures disponibles de conversion de codage. Voir la commande *CREATE CONVERSION* pour plus d'information.

**Tableau 41−11. Colonnes de pg_conversion**

| Nom | Type | Références | Description |
|---|---|---|---|
| conname | name | | Nom de la conversion (unique au sein d'un espace de noms) |
| connamespace | oid | pg_namespace .oid | OID de l'espace de nom qui contient cette conversion. |
| conowner | int4 | pg_shadow .usesysid | Propriétaire de la conversion |
| conforencoding | int4 | | ID du codage source |

| | | | |
|---|---|---|---|
| contoencoding | int4 | | ID du codage de destination |
| conproc | regproc | pg_proc .oid | Procédure de conversion |
| condefault | bool | | Vrai s'il s'agit de la conversion par défaut |

## 41.12. pg_database

Le catalogue pg_database stocke les informations sur les bases de données disponibles. Les bases de données sont créées avec la commande CREATE DATABASE. Consultez le Chapitre 18 pour avoir des détails sur la signification de certains paramètres.

Contrairement à la plupart des catalogues système, pg_database est partagé entre toutes les bases de données d'un groupe de bases : il n'y a qu'une seule copie de pg_database par groupe, pas une par base.

**Tableau 41−12. Colonnes de pg_database**

| Nom | Type | Références | Description |
|---|---|---|---|
| datname | name | | Nom de la base de données |
| datdba | int4 | pg_shadow .usesysid | Propriétaire de la base, généralement l'utilisateur qui l'a créée |
| encoding | int4 | | Codage des caractères pour cette base de données. |
| datistemplate | bool | | Si ce champ vaut vrai, alors la base peut être utilisée dans la clause TEMPLATE de la commande CREATE DATABASE pour créer une nouvelle base qui sera un clone de celle−ci. |
| datallowconn | bool | | Si ce champ vaut faux, alors personne ne peut se connecter à cette base de données. Ceci permet d'empêcher toute altération de la base template0. |
| datlastsysoid | oid | | Dernier OID système de la base de données ; utile en particulier pour pg_dump. |
| datvacuumxid | xid | | Toutes les lignes insérées ou supprimées par des ID de transaction inférieurs à celui−ci ont été marquées << validé de manière avérée >> ou << annulé de manière avérée >> dans cette base de données. C'est utilisé pour déterminer quand l'espace des journaux de validation peut être réutilisé. |
| datfrozenxid | xid | | Toutes les lignes insérées ou supprimées par des ID de transaction inférieurs à celui−ci ont été réétiquetés avec un ID de transaction permanent (<< gelé >> dans cette base de données. C'est utile pour vérifier si une base de données doit être rapidement nettoyée avec VACUUM pour éviter les problèmes créés par une remise à zéro du compteur de transaction. |
| dattablespace | oid | pg_tablespace .oid | L'espace logique par défaut de la base de données. À l'intérieur de cette base de données, toutes les tables pour lesquelles pg_class.reltablespace vaut |

| | | | |
|---|---|---|---|
| | | | zéro seront stockées dans cet espace logique ; en particulier, tous les catalogues système non partagés seront ici. |
| datconfig | text[] | | Valeurs par défaut de la session pour les variables modifiables en cours de fonctionnement. |
| datacl | aclitem[] | | Droits d'accès ; voir *GRANT* et *REVOKE* pour des détails. |

# 41.13. pg_depend

Le catalogue pg_depend enregistre les relations de dépendances entre les objets de la base de données. Cette information permet à la commande DROP de trouver quels autres objets doivent être supprimés par la commande DROP CASCADE ou au contraire empêchent la suppression dans le cas de DROP RESTRICT.

**Tableau 41−13. Colonnes de pg_depend**

| Nom | Type | Références | Description |
|---|---|---|---|
| classid | oid | pg_class.oid | OID du catalogue système dans lequel l'objet dépendant se trouve. |
| objid | oid | toute colonne OID | OID de l'objet dépendant |
| objsubid | int4 | | Pour une colonne de table, ce champ indique le numéro de colonne (les champs objid et classid se réfèrent à la table elle−même). Pour tous les autres types d'objets, cette colonne est à zéro. |
| refclassid | oid | pg_class.oid | OID du catalogue système dans lequel l'objet référencé se trouve. |
| refobjid | oid | toute colonne OID | OID de l'objet référencé |
| refobjsubid | int4 | | Pour une colonne de table, ce champ indique le numéro de colonne (les champs refobjid et refclassid se réfèrent à la table elle même). Pour tous les autres types d'objets, cette colonne est à zéro. |
| deptype | char | | Code définissant la sémantique de cette relation de dépendance. Voir le texte. |

Dans tous les cas, une entrée dans pg_depend indique que l'objet référence ne peut pas être supprimé sans supprimer aussi l'objet dépendant. Néanmoins, il y a des nuances, identifiées par deptype :

DEPENDENCY_NORMAL (n)
: Une relation normale entre des objets créés séparément. L'objet dépendant peut être supprimé sans affecter l'objet référencé. L'objet référencé ne peut être supprimé qu'en précisant l'option CASCADE, auquel cas l'objet dépendant est supprimé lui−aussi. Exemple : une colonne de table a une dépendance normale avec ses types de données.

DEPENDENCY_AUTO (a)
: L'objet dépendant peut être supprimé séparément de l'objet de référence, et doit être automatiquement supprimé si l'objet référencé est supprimé, quel que soit le mode RESTRICT ou CASCADE. Exemple : une contrainte nommée sur une table est auto−dépendante sur la table et sera automatiquement supprimée en même temps que la table.

DEPENDENCY_INTERNAL (i)
: L'objet dépendant a été créé comme une partie de l'objet référencé et n'est réellement qu'une partie de son implémentation interne. Un DROP de l'objet dépendant sera interdit (avec un message à l'utilisateur lui proposant de faire un DROP de l'objet référencé à la place). Une suppression de l'objet référencé sera propagé à l'objet dépendant que CASCADE soit précisé ou non. Exemple : un trigger qui est créé pour vérifier une contrainte de clé étrangère, est rendu dépendant de l'entrée de la contrainte dans pg_constraint.

DEPENDENCY_PIN (p)
: Il n'y a pas d'objet dépendant ; ce type d'entrée signale que le système lui même dépend de l'objet référencé, et donc que l'objet ne doit jamais être supprimé. Les entrées de ce type sont créées uniquement par initdb. Les colonnes pour l'objet dépendant contiennent des zéros.

D'autres types de dépendance pourraient apparaître dans le futur.

## 41.14. pg_description

Le catalogue pg_description stocke des descriptions (commentaires) optionnels pour chaque objet de la base de données. Les descriptions sont manipulées avec la commande COMMENT et lues avec les commandes \d de psql. Les descriptions de nombreux objets internes sont prédéfinies dans pg_description.

**Tableau 41−14. Colonnes de pg_description**

| Nom | Type | Références | Description |
|---|---|---|---|
| objoid | oid | toute colonne OID | OID de l'objet décrit. |
| classoid | oid | pg_class.oid | OID du système catalogue dans lequel apparaît l'objet décrit. |
| objsubid | int4 | | Pour un commentaire sur une table, le numéro de colonne. Les champs objoid et classoid font référence à la table elle−même. Pour tous les autres types de données, cette colonne est à zéro. |
| description | text | | Texte quelconque qui sert de description à cet objet. |

## 41.15. pg_group

Le catalogue pg_group définit les groupes et stocke les utilisateurs qui en sont membres. Les groupes sont créés avec la commande CREATE GROUP. Consultez le Chapitre 17 pour des informations sur la gestion des droits des utilisateurs.

Parce que les utilisateurs et les groupes sont définis pour tout le groupe de bases de données, pg_group est partagé par toutes les bases de données du groupe : il n'y a qu'une seule copie de pg_group par groupe de bases de données, et non pas une par base de données.

**Tableau 41−15. Colonnes de pg_group**

| Nom | Type | Références | Description |
|---|---|---|---|
| groname | name | | Nom du groupe |
| grosysid | int4 | | Un numéro quelconque qui identifie ce groupe. |
| grolist | int4[] | pg_shadow .usesysid | Tableau contenant les identifiants des utilisateurs de ce groupe |

## 41.16. pg_index

Le catalogue pg_index contient une partie des informations sur les index. Le reste se trouve essentiellement dans pg_class.

**Tableau 41–16. Colonnes de pg_index**

| Nom | Type | Références | Description |
|---|---|---|---|
| indexrelid | oid | pg_class .oid | OID de l'entrée dans pg_class pour cet index |
| indrelid | oid | pg_class .oid | OID de l'entrée dans pg_class de la table sur laquelle cet index porte. |
| indkey | int2vector | pg_attribute .attnum | Tableau comportant indnatts valeurs (et pas plus de INDEX_MAX_KEYS), qui précise les colonnes indexées. Par exemple, une valeur de 1 3 indique que la première et la troisième colonne de la table forment la clé de l'index. Une valeur de zéro dans le tableau indique que l'attribut d'index correspondant est une expression sur les colonnes de la table, et non pas une simple référence de colonne. |
| indclass | oidvector | pg_opclass .oid | Pour chaque colonne dans la clé d'index, ce champ contient l'OID de l'opérateur de classe à utiliser. Pour plus de détails, voir pg_opclass. |
| indnatts | int2 | | Nombre de colonnes de l'index (duplique pg_class.relnatts) |
| indisunique | bool | | Vrai s'il s'agit d'un index unique. |
| indisprimary | bool | | Vrai si cet index est la clé primaire de la table. (indisunique doit toujours |

| | | | |
|---|---|---|---|
| | | | être vrai quand ce champ l'est aussi.) |
| indisclustered | bool | | Vrai si la table était organisée en fonction de cet index. |
| indexprs | text | | Arbres d'expressions (en représentation nodeToString()) pour les attributs d'index qui ne sont pas de simples références de colonnes. Il s'agit d'une liste avec un élément pour chaque entrée à zéro dans indkey. Nul si tous les attributs d'index sont de simples références. |
| indpred | text | | Arbre d'expression (en représentation nodeToString()) pour les prédicats d'index partiels. Nul s'il ne s'agit pas d'un index partiel. |

## 41.17. pg_inherits

Le catalogue pg_inherits enregistre l'information sur la hiérarchie d'héritage des tables. Il existe une entrée pour chaque table enfant direct dans la base de données. (L'héritage indirect peut être déterminé en suivant les chaînes des entrées.)

**Tableau 41–17. Colonnes de pg_inherits**

| Nom | Type | Références | Description |
|---|---|---|---|
| inhrelid | oid | pg_class.oid | OID de la table fille. |
| inhparent | oid | pg_class.oid | OID de la table mère. |
| inhseqno | int4 | | S'il y a plus d'un parent direct pour une table fille (héritage multiple), ce nombre indique dans quel ordre les colonnes héritées doivent être arrangées. Le compteur commence à 1. |

## 41.18. pg_language

Le catalogue pg_language enregistre les langages avec lesquels vous pouvez écrire des fonctions ou des procédures stockées. Voir à *CREATE LANGUAGE* et dans le Chapitre 34 pour avoir plus d'information sur les gestionnaires de langages.

**Tableau 41–18. Colonnes de pg_language**

| Nom | Type | Références | Description |
|---|---|---|---|
| lanname | name | | Nom du langage |
| lanispl | bool | | Vaut faux pour les langages internes (comme SQL) et vrai pour les langages définis par l'utilisateur. Pour l'instant, pg_dump utilise ce champ pour déterminer quels langages doivent être sauvegardés mais cela sera peut−être un jour remplacé par un mécanisme différent. |
| lanpltrusted | bool | | Indique un langage sécurisé. S'il s'agit d'un langage interne, cette colonne est sans importance. |
| lanplcallfoid | oid | pg_proc.oid | Pour les langages non−internes, ceci référence le gestionnaire de langage, qui est une fonction spéciale en charge de l'exécution de toutes les fonctions écrites dans ce langage. |
| lanvalidator | oid | pg_proc.oid | Ceci référence une fonction de validation de langage, en charge de vérifier la syntaxe et la validité des nouvelles fonctions lorsqu'elles sont créées. Zéro si aucun validateur n'est fourni. |
| lanacl | aclitem[] | | Droits d'accès ;; voir *GRANT* et *REVOKE* pour les détails. |

## 41.19. pg_largeobject

Le catalogue pg_largeobject contient les données qui décrivent les << objets de grande taille >>. Un gros objet est identifié par un OID qui lui est affecté lors de sa création. Chaque objet de grande taille est coupé en segments ou << pages >> suffisamment petites pour être facilement stockées dans des lignes de pg_largeobject. La taille de données par page est définie par LOBLKSIZE, qui vaut actuellement BLCKSZ/4, soit habituellement 2 Ko).

**Tableau 41−19. Colonnes de pg_largeobject**

| Nom | Type | Références | Description |
|---|---|---|---|
| loid | oid | | Identifiant de l'objet de grande taille auquel appartient cette page. |
| pageno | int4 | | Numéro de cette page parmi son objet de grande taille, en partant de zéro. |
| data | bytea | | Données effectivement stockées dans l'objet de grande taille. Il ne fait jamais plus de LOBLKSIZE mais peut faire moins. |

Chaque ligne de pg_largeobject contient les données d'une page de l'objet de grande taille, en commençant à l'octet (pageno * LOBLKSIZE) de l'objet. Ceci permet un stockage économique : des pages peuvent manquer, et d'autres faire moins de LOBLKSIZE octets même si elles ne sont pas les dernières de leur objet. Les parties manquantes sont considérées comme des suites de zéro.

## 41.20. pg_listener

Le catalogue pg_listener supporte les commandes *LISTEN* et *NOTIFY*. Un notifié (<< listener >>) crée une entrée dans pg_listener pour chaque nom de notification qu'il attend. Un notifieur parcourt pg_listener et met à jour chaque entrée pour montrer qu'une notification est arrivée. Le notifieur envoie aussi un signal (en utilisant le numéro de processus PID) enregistré dans la table pour prévenir le notifié.

**Tableau 41–20. Colonnes de `pg_listener`**

| Nom | Type | Références | Description |
|---|---|---|---|
| `relname` | `name` | | Nom de la notification. (Il ne doit pas spécialement correspondre à un nom de relation de la base. Le nom `relname` est historique.) |
| `listenerpid` | `int4` | | PID du processus serveur qui a créé cette entrée. |
| `notification` | `int4` | | Zéro si aucun événement n'est en attente pour ce notifié. Si un événement est en attente, ce champ contient le PID du processus serveur qui a envoyé cette notification. |

# 41.21. `pg_namespace`

Le catalogue `pg_namespace` stocke les espaces de noms. Un espace de noms est la structure sous–jacente des schémas SQL : chaque espace de noms peut avoir un ensemble séparé de relations, types, etc. sans qu'il y ait de conflit de noms.

**Tableau 41–21. Colonnes de `pg_namespace`**

| Nom | Type | Références | Description |
|---|---|---|---|
| `nspname` | `name` | | Nom de l'espace de noms |
| `nspowner` | `int4` | `pg_shadow .usesysid` | Propriétaire de l'espace de noms |
| `nspacl` | `aclitem[]` | | Droits d'accès ; voir *GRANT* et *REVOKE* pour des détails. |

# 41.22. `pg_opclass`

Le catalogue `pg_opclass` définit les classes d'opérateurs de méthodes d'accès aux index. Chaque classe d'opérateur définit la sémantique pour les colonnes d'index d'un type particulier, pour une méthode d'accès particulière. Notez qu'il peut y avoir plusieurs classes d'opérateurs pour une combinaison donnée de type/méthode d'accès, ce qui permet de supporter différents comportements.

Les classes d'opérateurs sont longuement décrites dans la Section 31.14.

**Tableau 41–22. Colonnes de `pg_opclass`**

| Nom | Type | Références | Description |
|---|---|---|---|
| `opcamid` | `oid` | `pg_am .oid` | Méthode d'accès à l'index pour lequel est la classe d'opérateur |
| `opcname` | `name` | | Nom de la classe d'opérateurs |
| `opcnamespace` | `oid` | `pg_namespace .oid` | Espace de noms de la classe d'opérateurs. |
| `opcowner` | `int4` | `pg_shadow` | Propriétaires de la classe d'opérateurs. |

| | | | |
|---|---|---|---|
| | | .usesysid | |
| opcintype | oid | pg_type .oid | Type de données que la classe d'opérateurs indexe. |
| opcdefault | bool | | Vrai si la classe d'opérateurs est la classe par défaut pour opcintype |
| opckeytype | oid | pg_type .oid | Type de données stocké dans l'index ou zéro s'il s'agit du même que opcintype |

La plus grande partie des informations définissant une classe d'opérateurs n'est pas dans les lignes de pg_opclass mais dans les lignes correspondantes de pg_amop et pg_amproc. Ces lignes sont considérées comme faisant partie de la définition de classe d'opérateurs, un peu de la même façon qu'une relation est définie par une ligne unique de pg_class et par les lignes associées de pg_attribute et d'autres tables.

# 41.23. pg_operator

Le catalogue pg_operator stocke les informations sur les opérateurs. Voir la commande *CREATE OPERATOR* et la Section 31.12 pour plus d'informations.

**Tableau 41–23. Colonnes de pg_operator**

| Nom | Type | Références | Description |
|---|---|---|---|
| oprname | name | | Nom de l'opérateur |
| oprnamespace | oid | pg_namespace .oid | OID de l'espace de nom qui contient cet opérateur. |
| oprowner | int4 | pg_shadow .usesysid | Propriétaire de l'opérateur |
| oprkind | char | | b = infix (<< les deux >>), l = prefix (<< gauche >>), r = postfix (<< droit >>) |
| oprcanhash | bool | | Cet opérateur supporte les jointures par découpage. |
| oprleft | oid | pg_type .oid | Type de l'opérande de gauche |
| oprright | oid | pg_type .oid | Type de l'opérande de droite |
| oprresult | oid | pg_type .oid | Type du résultat |
| oprcom | oid | pg_operator .oid | Commutateur de cet opérateur, s'il en existe un. |
| oprnegate | oid | pg_operator .oid | Négateur de cet opérateur, s'il en existe un. |
| oprlsortop | oid | pg_operator .oid | Si cet opérateur supporte les jointures par fusion (merge join), ce champ contient l'opérateur qui permet de trier le type de l'opérateur de gauche (L<L). |
| oprrsortop | oid | pg_operator .oid | Si cet opérateur supporte les jointures par fusion (merge join), ce champ contient l'opérateur qui permet de trier le type de l'opérateur de droite (R<R) |
| oprltcmpop | oid | pg_operator .oid | Si cet opérateur supporte les jointures par fusion (merge join), ce champ contient |

| | | | |
|---|---|---|---|
| | | | l'opérateur qui permet de comparer les types des opérandes de gauche et de droite(L<R). |
| oprgtcmpop | oid | pg_operator .oid | Si cet opérateur supporte les jointures par fusion (merge join), ce champ contient l'opérateur plus grand que qui compare les types d'opérandes de gauche et de droite (L>R). |
| oprcode | regproc | pg_proc .oid | Fonction qui implémente cet opérateur |
| oprrest | regproc | pg_proc .oid | Fonction d'estimation de sélectivité de restriction pour cet opérateur |
| oprjoin | regproc | pg_proc .oid | Fonction d'estimation de sélectivité de jointure pour cet opérateur |

Les colonnes inutilisées contiennent des zéros, par exemple oprleft vaut zéro pour un opérateur préfixe.

# 41.24. pg_proc

Le catalogue pg_proc stocke les informations sur les fonctions (ou procédures). Voir *CREATE FUNCTION* et Section 31.3 pour plus d'informations.

Cette table contient des données pour les fonctions d'agrégat comme pour les fonctions simples. Si proisagg est vrai, il devrait y avoir une ligne correspondante dans pg_aggregate.

**Tableau 41−24. Colonnes de pg_proc**

| Nom | Type | Références | Description |
|---|---|---|---|
| proname | name | | Nom de la fonction |
| pronamespace | oid | pg_namespace .oid | OID de l'espace de noms auquel cette fonction appartient. |
| proowner | int4 | pg_shadow .usesysid | Propriétaire de la fonction. |
| prolang | oid | pg_language .oid | Langage ou interface d'appel pour cette fonction |
| proisagg | bool | | La fonction est une fonction d'agrégat. |
| prosecdef | bool | | Si vrai, la fonction définit la sécurité (c'est−à−dire une fonction << setuid >>). |
| proisstrict | bool | | Si vrai, la fonction retourne NULL si un de ses arguments au moins est NULL. Dans ce cas, la fonction ne sera en fait pas appelée du tout. Les fonctions qui ne sont pas << strictes >> doivent être préparées à traiter des paramètres NULL. |
| proretset | bool | | Si vrai, la fonction retourne un ensemble (c'est−à−dire des valeurs multiples du type défini). |
| provolatile | char | | provolatile indique si le résultat de la fonction dépend uniquement de ses arguments ou s'il est affecté par des facteurs externes. Il vaut i pour les fonctions << immuables >>, qui donnent toujours le même résultat |

| | | | |
|---|---|---|---|
| | | | quand les paramètres entrés sont les mêmes. Il vaut s pour les fonctions << stables >>, dont le résultat (pour les mêmes paramètres d'entrée) ne change pas tout au long d'un balayage (de table). Il vaut v pour les fonctions << volatiles >>, dont le résultat peut varier à tout instant. (Utilisez v aussi pour les fonctions qui ont des effets de bord, afin que les appels à ces fonctions ne soient pas optimisés.) |
| pronargs | int2 | | Nombre d'arguments |
| prorettype | oid | pg_type .oid | Type de données renvoyé |
| proargtypes | oidvector | pg_type .oid | Tableau contenant les types de données des arguments de la fonction. |
| proargnames | text[] | | Un tableau avec les noms des arguments de la fonction. Les arguments sans nom sont initialisés à des chaînes vides dans le tableau. Si aucun des arguments n'a un nom, ce champ pourrait être NULL. |
| prosrc | text | | Ce champ indique au gestionnaire de fonctions la façon d'invoquer la fonction. Il peut s'agir du code source pour un langage interprété, d'un symbole lié, d'un nom de fichier ou de n'importe quoi d'autre, en fonction du langage ou de la convention d'appel. |
| probin | bytea | | Information supplémentaire sur la façon d'invoquer la fonction. Encore une fois, l'interprétation dépend du langage. |
| proacl | aclitem[] | | Droits d'accès ; voir *GRANT* et *REVOKE* pour plus de détails. |

Pour les fonctions compilées, intégrées et chargées dynamiquement, prosrc contient le nom de la fonction en langage C (symbol lié). Pour tous les autres types de langages, prosrc contient le code source de la fonction. probin est inutilisé, sauf pour les fonctions en C chargées dynamiquement, pour lesquelles il donne le nom de fichier de la bibliothèque partagée qui contient la fonction.

## 41.25. pg_rewrite

Le catalogue pg_rewrite stocke les règles de réécriture pour les tables et les vues.

**Tableau 41–25. Colonnes de pg_rewrite**

| Nom | Type | Références | Description |
|---|---|---|---|
| rulename | name | | Nom de la règle |
| ev_class | oid | pg_class .oid | Table sur laquelle porte cette règle. |
| ev_attr | int2 | | Colonne sur laquelle porte cette règle. Actuellement, cette colonne vaut toujours zéro pour indiquer qu'il s'agit de la table entière. |
| ev_type | char | | Type d'évènement déclenchant la règle : 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE |

| | | | |
|---|---|---|---|
| is_instead | bool | | Vrai s'il s'agit d'une règle INSTEAD (à la place de). |
| ev_qual | text | | Arbre d'expression (sous la forme d'une représentation nodeToString()) pour la condition qualifiant la règle. |
| ev_action | text | | Arbre de requête (sous la forme d'une représentation nodeToString()) pour l'action de la règle. |

**Note :** pg_class.relhasrules doit être vrai si une table a au moins une règle dans ce catalogue.

# 41.26. pg_shadow

Le catalogue pg_shadow contient les informations sur les utilisateurs de la base de données. Le nom provient du fait que cette table ne doit pas être lisible par le public, vu qu'elle contient des mots de passe. pg_user est une vue accessible par tous sur pg_shadow qui masque le champ mot de passe.

Chapitre 17 contient des informations détaillées sur les utilisateurs et la gestion des droits d'accès.

Parce que les identités des utilisateurs sont globales au groupe de bases de données (cluster), pg_shadow est partagé par toutes les bases de données d'un groupe de bases de données : il n'y a qu'une seule copie de pg_shadow par groupe de bases de données, et non pas une par base de données.

**Tableau 41–26. Colonnes de pg_shadow**

| Nom | Type | Références | Description |
|---|---|---|---|
| usename | name | | Nom de l'utilisateur |
| usesysid | int4 | | Identifiant de l'utilisateur (numéro arbitraire utilisé pour référencer cet utilisateur). |
| usecreatedb | bool | | L'utilisateur a le droit de créer des bases de données. |
| usesuper | bool | | L'utilisateur est un super–utilisateur |
| usecatupd | bool | | L'utilisateur a le droit de modifier les catalogues système. (Même un super–utilisateur n'a pas le droit de le faire si cette colonne n'est pas vraie). |
| passwd | text | | Mot de passe (pouvant être crypté) |
| valuntil | abstime | | Date et heure d'expiration du mot de passe (utilisé seulement pour l'authentification par mot de passe). |
| useconfig | text[] | | Valeur par défaut de session pour les variables de configuration lors de l'exécution. |

# 41.27. pg_statistic

Le catalogue pg_statistic stocke des données statistiques sur le contenu de la base de données. Les entrées sont créées par ANALYZE, puis utilisées par l'optimiseur de requêtes. Il y a une entrée pour chaque colonne de table qui a été analysée. Notez que les données statistiques sont par définition des approximations, même si elles sont à jour.

pg_statistic stocke aussi les données statistiques sur les valeurs des expressions d'index. Elles sont décrites comme si elles étaient de vraies colonnes ; en particulier, starelid référence l'index. Néanmoins, aucune entrée n'est effectuée pour une colonne d'index ordinaire sans expression car cela serait redondant avec l'entrée pour la colonne sous−jacente de la table.

Comme des statistiques différentes seront appropriées pour des types de données différents, pg_statistic est prévu pour ne faire qu'un minimum de suppositions sur les types de statistiques qu'il stocke. Seules des statistiques extrêmement générales (comme les valeurs NULL) ont des colonnes dédiées. Tout le reste est stocké dans des << emplacements >>, qui sont des groupes de colonnes associées dont le contenu est identifié par un numéro de code dans l'une des colonnes de l'emplacement. Pour plus d'information, voir src/include/catalog/pg_statistic.h.

pg_statistic ne doit pas être lisible par le public car même les données statistiques peuvent être considérées comme sensibles. (Exemple : les valeurs maximales et minimales d'une colonne de salaire peuvent être assez intéressantes). pg_stats est une vue sur pg_statistic accessible à tous, qui n'expose que les informations sur ces tables qui sont accessibles à l'utilisateur courant.

**Tableau 41−27. Colonnes de pg_statistic**

| Nom | Type | Références | Description |
|---|---|---|---|
| starelid | oid | pg_class .oid | Table ou index à laquelle la colonne décrite appartient. |
| staattnum | int2 | pg_attribute .attnum | Numéro de la colonne décrite. |
| stanullfrac | float4 | | Fraction des entrées de la colonne qui ont une valeur NULL. |
| stawidth | int4 | | Taille moyenne stockée des entrées non NULL. |
| stadistinct | float4 | | Nombre de valeurs distinctes non NULL dans la colonne. Une valeurs positive est le nombre réel de valeurs distinctes. Une valeur négative est le négatif d'une fraction du nombre de lignes dans la table (par exemple, une colonne dans laquelle les valeurs apparaissent environ deux fois en moyenne pourrait être représentée par stadistinct = −0.5). Une valeur à zéro indique que le nombre de valeurs distinctes est inconnu. |
| stakind*N* | int2 | | Numéro de code indiquant quel type de statistiques est stocké dans << l'emplacement >> numéro *N* de la ligne de pg_statistic. |
| staop*N* | oid | pg_operator .oid | Opérateur utilisé pour dériver les statistiques stockées dans << l'emplacement >> numéro *N*. Par exemple, un emplacement d'histogramme montrerait l'opérateur <, qui définit l'ordre de tri des données. |
| stanumbers*N* | float4[] | | Statistiques numériques du genre approprié pour << l'emplacement >> numéro *N* ou NULL si le type d'emplacement ne nécessite pas de valeurs numériques. |
| stavalues*N* | anyarray | | Valeurs de données de la colonne du type approprié pour << l'emplacement >> numéro *N* ou NULL si le type d'emplacement ne stocke aucune valeur de données. Chaque |

| | | | |
|---|---|---|---|
| | | | valeur d'élément du tableau est en fait du type de données de la colonne spécifiée, si bien qu'il n'y a aucun moyen de définir ces colonnes autrement qu'avec le type `anyarray` (tableau quelconque). |

## 41.28. pg_tablespace

Le catalogue `pg_tablespace` enregistre les informations sur les espaces logiques disponibles. Les tables peuvent être placées dans des espaces logiques particuliers pour aider à l'administration des espaces de stockage.

Contrairement à la plupart des catalogues système, `pg_tablespace` est partagée parmi toutes les bases de données du groupe : il n'y a donc qu'une copie de `pg_tablespace` par groupe, et non pas une par base.

**Tableau 41–28. Colonnes de `pg_tablespace`**

| Nom | Type | Références | Description |
|---|---|---|---|
| `spcname` | `name` | | Nom de l'espace logique |
| `spcowner` | `int4` | `pg_shadow.usesysid` | Propriétaire de l'espace logique, habituellement l'utilisateur qui l'a créé |
| `spclocation` | `text` | | Emplacement (chemin vers le répertoire) de l'espace logique |
| `spcacl` | `aclitem[]` | | Droits d'accès ; voir *GRANT* et *REVOKE* pour les détails. |

## 41.29. pg_trigger

Le catalogue `pg_trigger` stocke les informations sur les déclencheurs des tables. Voir la commande *CREATE TRIGGER* pour plus d'informations.

**Tableau 41–29. Colonnes de `pg_trigger`**

| Nom | Type | Références | Description |
|---|---|---|---|
| `tgrelid` | `oid` | `pg_class.oid` | Table sur laquelle porte le déclencheur |
| `tgname` | `name` | | Nom du déclencheur (doit être unique parmi les déclencheurs d'une table). |
| `tgfoid` | `oid` | `pg_proc.oid` | Fonction à appeler |
| `tgtype` | `int2` | | Masque de bits identifiant les conditions du déclencheur. |
| `tgenabled` | `bool` | | Vrai si le déclencheur est activé. Ceci n'est pas vérifié de manière fiable partout où cela devrait, si bien que désactiver un déclencheur ne fonctionne pas de manière |

| | | | |
|---|---|---|---|
| | | | fiable. |
| tgisconstraint | bool | | Vrai si le déclencheur implémente une contrainte d'intégrité référentielle. |
| tgconstrname | name | | Nom de la contrainte d'intégrité référentielle. |
| tgconstrrelid | oid | pg_class.oid | Table référencée par une contrainte d'intégrité référentielle. |
| tgdeferrable | bool | | Vrai si le déclencheur est différable. |
| tginitdeferred | bool | | Vrai si initialement le déclencheur est différé. |
| tgnargs | int2 | | Nombre de chaînes d'arguments passées à la fonction de déclencheur. |
| tgattr | int2vector | | Actuellement inutilisé. |
| tgargs | bytea | | Chaînes d'arguments à passer au déclencheur, chacune terminée par un NULL. |

**Note :** pg_class.reltriggers doit être d'accord avec le nombre de déclencheurs trouvées dans cette table.

# 41.30. pg_type

Le catalogue pg_type stocke les informations sur les types de données. Les types de base (types scalaires) sont créés avec la commande *CREATE TYPE* et les domaines avec *CREATE DOMAIN*. Un type composite est créé automatiquement pour chaque table de la base et représenter ainsi la structure des lignes de la table. Il est aussi possible de créer des types composites avec CREATE TYPE AS .

**Tableau 41−30. Colonnes de pg_type**

| Nom | Type | Références | Description |
|---|---|---|---|
| typname | name | | Nom du type |
| typnamespace | oid | pg_namespace.oid | OID de l'espace de noms qui contient ce type. |
| typowner | int4 | pg_shadow.usesysid | Propriétaire du type |
| typlen | int2 | | Pour les types de taille fixe, typlen est le nombre d'octets de la représentation interne du type. Mais pour les types de longueur variable, typlen est négatif. −1 indique un type << varlena >> (qui a un attribut de longueur), −2 indique une chaîne C terminée par le caractère NULL. |
| typbyval | bool | | typbyval détermine si les routines internes passent une valeur de ce type par valeur ou par référence. typbyval doit être faux si typlen ne vaut pas 1, 2 ou 4 (ou 8 sur les machines dont le mot−machine est de 8 octets). Les types de longueur variable sont toujours passés par référence. Notez que typbyval peut être faux même si la longueur permettrait un passage par valeur ; c'est le cas pour le type float4, par exemple. |

| | | | |
|---|---|---|---|
| typtype | char | | typtype vaut b pour un type de base, c pour un type composite (c'est–à–dire le type d'une ligne de table), d pour un domaine ou p pour un pseudo–type. Voir aussi typrelid et typbasetype. |
| typisdefined | bool | | Vrai si le type est défini et faux s'il ne s'agit que d'un remplissage pour un type qui n'est pas encore défini. Lorsque typisdefined est faux, rien à part le nom du type, l'espace de noms et l'OID n'est fiable. |
| typdelim | char | | Caractère qui sépare deux valeurs de ce type lorsque le programme lit les valeurs d'un tableau en entrée. Notez que le délimiteur est associé au type d'élément du tableau, pas au type tableau. |
| typrelid | oid | pg_class .oid | S'il s'agit d'un type composite (voir typtype), alors cette colonne pointe vers la ligne de pg_class qui définit la table correspondante. Pour un type composite sans table, l'entrée dans pg_class ne représente pas vraiment une table, mais elle est néanmoins nécessaire pour trouver les lignes de pg_attribute liées au type. Zéro pour les types autres que composites. |
| typelem | oid | pg_type .oid | Si typelem ne vaut pas 0, alors il identifie une autre ligne de pg_type. Le type courant peut alors être utilisé comme un tableau contenant des valeurs de type typelem. Un << vrai >> type tableau a une longueur variable (typlen = –1), mais certains types de longueur fixe (typlen > 0) ont aussi un typelem non nul, par exemple name et oidvector. Si un type de longueur fixe a un typelem, alors sa représentation interne doit être un certain nombre de valeurs de ce type typelem, sans autre donnée. Les types de données tableau de taille variable ont un en–tête défini par les sous–routines de tableau. |
| typinput | regproc | pg_proc .oid | Fonction de conversion en entrée (format texte). |
| typoutput | regproc | pg_proc .oid | Fonction de conversion en sortie (format texte). |
| typreceive | regproc | pg_proc .oid | Fonction de conversion en entrée (format binaire), ou 0 s'il n'y en a pas |
| typsend | regproc | pg_proc .oid | Fonction de conversion en sortie (format binaire), ou 0 s'il n'y en a pas |
| typanalyze | regproc | pg_proc.oid | Fonction ANALYZE personnalisée ou 0 pour utiliser la fonction standard |
| typalign | char | | typalign est l'alignement requis pour stocker une valeur de ce type. Cela s'applique au stockage sur disque aussi bien qu'à la plupart des représentations de cette valeur dans PostgreSQL. Lorsque des valeurs multiples sont stockées consécutivement, comme dans la représentation d'une ligne complète sur disque, un remplissage est inséré avant la donnée de ce type pour qu'elle commence à l'alignement spécifié. La référence de l'alignement est le début de la première donnée de la |

| | | | |
|---|---|---|---|
| | | | séquence.<br><br>Les valeurs possibles sont :<br><br>• c = alignement char ce qui signifie qu'aucun alignement n'est nécessaire ;<br>• s = alignement short (deux octets sur la plupart des machines) ;<br>• i = alignement int (quatre octets sur la plupart des machines) ;<br>• d = alignement double (huit octets sur la plupart des machines, mais pas sur toutes).<br><br>**Note :** Pour les types utilisés dans les tables systèmes il est indispensable que les tailles et alignements définis dans pg_type soient en accord avec la façon dont le compilateur disposera la colonne dans une structure représentant une ligne de table. |
| typstorage | char | | typstorage indique, pour les types varlena (ceux pour lesquels typlen = −1), si le type est préparé pour le TOASTage et quelle stratégie par défaut doit être utilisée pour les attributs de ce type. Les valeurs possibles sont :<br><br>• p : La valeur doit être stockée normalement ;<br>• e : La valeur peut être stockée dans une relation << secondaire >> (si la relation en a une, voir pg_class.reltoastrelid) ;<br>• m : La valeurs peut être stockée compressée sur place ;<br>• x : La valeurs peut être stockée compressée sur place ou stockée dans une relation << secondaire >>.<br><br>Notez que les colonnes m peuvent aussi être déplacées dans une table de stockage secondaire, mais seulement en dernier recours (les colonnes e et x sont déplacées d'abord). |
| typnotnull | bool | | typnotnull représente une contrainte non NULL pour ce type. Ceci n'est utilisé que pour les domaines. |
| typbasetype | oid | pg_type .oid | S'il s'agit d'un domaine, (voir typtype), alors typbasetype identifie le type sur lequel celui−ci est basé. Zéro s'il ne s'agit pas d'un domaine. |
| typtypmod | int4 | | Les domaines utilisent typtypmod pour enregistrer le typmod à appliquer à leur type de base (−1 si le type de base n'utilise pas de typmod). −1 si ce type n'est pas un domaine. |
| typndims | int4 | | typndims est le nombre de dimensions de tableau pour |

| | | | |
|---|---|---|---|
| | | | un domaine qui est un tableau (c'est–à–dire dont `typbasetype` est un type tableau ; le `typelem` du domaine correspondra au `typelem` du type de base). Zéro pour les types autres que les domaines tableaux. |
| `typdefaultbin` | `text` | | Si `typdefaultbin` n'est pas NULL, ce champ est la représentation `nodeToString()` d'une expression par défaut pour le type. Ceci n'est utilisé que pour les domaines. |
| `typdefault` | `text` | | `typdefault` est NULL si le type n'a pas de valeur par défaut associée. Si `typdefaultbin` est non NULL, `typdefault` doit contenir une version lisible de l'expression par défaut représentée par `typdefaultbin`. Si `typdefaultbin` est NULL et si `typdefault` ne l'est pas, alors `typdefault` est la représentation externe de la valeur par défaut du type, qui peut être passée à la fonction de conversion en entrée du type pour produire une constante. |

# 41.31. Vues système

En plus des catalogues système, PostgreSQL apporte un certain nombre de vues internes. Quelques vues systèmes apportent un moyen simple et agréable d'accéder à des requêtes habituellement utilisées dans les catalogues systèmes. D'autres vues donnent accès à l'état du serveur interne.

Le schéma information (Chapitre 30) fournit un autre ensemble de vues qui rejoignent les fonctionnalités des vues systèmes. Comme le schéma information est en SQL standard alors que les vues décrites ici sont spécifiques à PostgreSQL, il est généralement mieux d'utiliser le schéma information si celui–ci apporte toutes les informations dont vous avez besoin.

Tableau 41–31 indique les vues systèmes décrites ici. Une documentation plus détaillée de chaque vue suit ceci. Il existe en plus des vues supplémentaires permettant d'accéder aux résultats du collecteur de statistiques elles sont décrites dans le Tableau 23–1.

Sauf lorsque c'est noté, toutes les vues décrites ici sont en lecture seule.

**Tableau 41–31. Vues système**

| Nom de la vue | But |
|---|---|
| `pg_indexes` | index |
| `pg_locks` | verrous actuellement contenus |
| `pg_rules` | règles |
| `pg_settings` | configuration |
| `pg_stats` | statistiques du planificateur |
| `pg_tables` | tables |
| `pg_user` | utilisateurs de la base de données |
| `pg_views` | vues |

# 41.32. pg_indexes

La vue pg_indexes fournit un accès à des informations utiles sur chaque index de la base de données.

**Tableau 41−32. Colonnes de pg_indexes**

| Nom | Type | Références | Description |
|---|---|---|---|
| schemaname | name | pg_namespace.nspname | nom du schéma contenant les tables et index |
| tablename | name | pg_class.relname | nom de la table concernant cet index |
| indexname | name | pg_class.relname | nom de l'index |
| tablespace | name | pg_tablespace.spcname | nom de l'espace logique contenant l'index (NULL s'il s'agit de la valeur par défaut de la base de données) |
| indexdef | text | | définition de l'index (une commande de création reconstruite) |

# 41.33. pg_locks

La vue pg_locks fournit un accès aux informations concernant les verrous détenus par les transactions ouvertes à l'intérieur du serveur de bases de données. Voir le Chapitre 12 pour une discussion plus importante sur les verrous.

pg_locks contient une ligne par objet verrouillable actif, mode de verrouillage demandé et transaction indiquée. Donc, le même objet verrouillable pourrait apparaître plusieurs fois si plusieurs transactions ont pris ou attendent des verrous sur celui−ci. Néanmoins, un objet qui n'est pas verrouillé n'apparaîtra pas du tout. Un objet verrouillé est soit une relation (c'est−à−dire une table) soit un identifiant de transaction.

Notez que cette vue inclut seulement des verrous au niveau de la table, pas au niveau des lignes. Si une transaction attend un verrou ligne, il apparaîtra dans la vue en attente d'un identifiant de transaction du déteneur actuel du verrou.

**Tableau 41−33. Colonnes pg_locks**

| Nom | Type | Références | Description |
|---|---|---|---|
| relation | oid | pg_class.oid | OID de la relation verrouillée ou NULL si l'objet verrouillé est un identifiant de transaction |
| database | oid | pg_database.oid | OID de la base de données dans lequel la relation verrouillée existe ou zéro si la relation verrouillée est une table partagée globalement ou NULL si l'objet verrouillable est un identifiant de transaction |
| transaction | xid | | ID d'une transaction ou NULL si l'objet verrouillable est une relation |
| pid | integer | | |

| | | | |
|---|---|---|---|
| | | | Identifiant (PID) du processus serveur détenant ou attendant ce verrou |
| mode | text | | Nom du mode de verrou détenu ou attendu par ce processus (voir la Section 12.3.1) |
| granted | boolean | | vrai si le verrou est détenu, faux s'il est attendu |

granted est vrai sur une ligne représentant un verrou tenu par la session indiquée. Une valeur faux indique que cette session attend l'acquisition du verrou, ce qui implique qu'une autre session a choisi un mode de verrouillage conflictuel sur le même objet partageable. La session en attente dormira jusqu'à la relâche du verrou (ou jusqu'à ce qu'une situation de blocage soit détectée). Une session seule ne peut attendre d'acquérir qu'au plus un verrou à la fois.

Chaque transaction détient un verrou exclusif sur son identifiant de transaction toute la durée de son exécution. Si une transaction trouve nécessaire d'attendre spécifiquement une autre transaction, elle le fait en essayant d'acquérir un verrou partagé sur l'identifiant de l'autre transaction. Ceci sera couronné de succès seulement lorsque l'autre transaction termine et relâche son verrou.

Quand la vue pg_locks est accédée, les structures de données du gestionnaire interne de verrous sont momentanément verrouillées et une copie est faite pour que la vue s'affiche. Ceci nous assure que la vue produise un ensemble consistant de résultats, tout en ne bloquant pas les opérations habituelles du gestionnaire de verrous plus longuement que nécessaire. Néanmoins, il peut y avoir des impacts sur les performances de la base de données si cette vue est lue fréquemment.

pg_locks fournit une vue globale de tous les verrous du cluster de bases de données, et non pas seulement une vue de ceux de la base en cours. Bien que sa colonne relation peut être jointe avec pg_class.oid pour identifier les relations verrouillées, ceci ne fonctionnera correctement qu'avec les relations de la base en cours (celles pour qui la colonne database est soit l'OID de la base en cours soit zéro).

Si vous avez activé le collecteur de statistiques, la colonne pid peut être jointe à la colonne procpid de la vue pg_stat_activity pour obtenir plus d'information sur le propriétaire de la session ou attendant de détenir le verrou.

## 41.34. pg_rules

La vue pg_rules fournit un accès à des informations utiles sur les règles de réécriture des requêtes.

**Tableau 41–34. Colonnes de pg_rules**

| Nom | Type | Références | Description |
|---|---|---|---|
| schemaname | name | pg_namespace.nspname | nom du schéma contenant la table |
| tablename | name | pg_class.relname | nom de la table pour laquelle est créée la règle |
| rulename | name | pg_rewrite.rulename | nom de la règle |
| definition | text | | définition de la règle (une commande de création reconstruite) |

La vue pg_rules exclut les règles ON SELECT rules des vues ; elles sont disponibles dans pg_views.

# 41.35. pg_settings

La vue pg_settings fournit un accès aux paramètres d'exécution du serveur. C'est essentiellement une autre interface aux commandes SHOW et SET. Elle fournit aussi un accès à certaines informations sur chaque paramètre qui ne sont pas directement accessibles avec SHOW, telles que les valeurs minimales et maximales.

**Tableau 41−35. Colonnes de pg_settings**

| Nom | Type | Références | Description |
|---|---|---|---|
| name | text | | nom du paramètres d'exécution |
| setting | text | | valeur actuelle du paramètre |
| category | text | | groupe logique du paramètre |
| short_desc | text | | une description brève du paramètre |
| extra_desc | text | | information supplémentaire, plus détaillée, sur le paramètre |
| context | text | | contexte requis pour initialiser la valeur du paramètre |
| vartype | text | | type du paramètre (bool, integer, real ou string) |
| source | text | | source de la valeur du paramètre actuel |
| min_val | text | | valeur minimum autorisée du paramètre (NULL pour les valeurs non numériques) |
| max_val | text | | valeur maximum autorisée du paramètre (NULL pour les valeurs non numériques) |

La vue pg_settings n'accepte pas d'insertion ou de suppression mais peut être mise à jour. Un UPDATE appliqué à une ligne de pg_settings est équivalent à l'exécution de la commande *SET* sur ce paramètre nommé. Le changement affecte seulement la valeur utilisée par la session actuelle. Si un UPDATE est lancé à l'intérieur d'une transaction qui est annulé plus tard, les effets de la commande UPDATE disparaisse lorsque les effets de la transaction sont annulés. Une fois que la transaction est validée, les effets persistent jusqu'à la fin de la session, à moins qu'un autre UPDATE ou SET ne modifie la valeur.

# 41.36. pg_stats

La vue pg_stats fournit un accès aux informations stockées dans la table système pg_statistic. Cette vue autorise un accès aux seules lignes de pg_statistic correspondant aux tables pour lequel l'utilisateur a un droit de lecture et, donc, aucun problème de sécurité n'empêche l'accès en lecture de cette vue au public.

pg_stats est aussi conçu pour afficher l'information dans un format plus lisible que le catalogue sous−jacent, au prix d'un schéma qui doit être étendu lorsque de nouveaux types sont définis dans pg_statistic.

**Tableau 41−36. Colonnes de pg_stats**

| Nom | Type | Références | Description |
|---|---|---|---|
| schemaname | name | pg_namespace.nspname | nom du schéma contenant la table |

| | | | |
|---|---|---|---|
| tablename | name | pg_class.relname | nom de la table |
| attname | name | pg_attribute.attname | nom de la colonne décrite par cette ligne |
| null_frac | real | | fraction d'entrées de colonnes qui sont NULL |
| avg_width | integer | | largeur moyenne en octets des entrées de la colonne |
| n_distinct | real | | Si plus grand que zéro, le nombre estimé de valeurs distinctes dans la colonne. Si négatif, le nombre de valeurs distinctes divisé par le nombre de lignes, le tout muliplié par −1. (La forme négative est utilisée quand ANALYZE croit que le nombre de valeurs distinctes a tendance à grossir au fur et à mesure que la table grossit ; la forme positive est utilisé lorsque la commande semble avoir un nombre fixe de valeurs possibles.) Par exemple, −1 indique une colonne unique pour laquelle le nombre de valeurs distinctes est identique aux nombres de lignes. |
| most_common_vals | anyarray | | Une liste des valeurs habituelles dans cette colonne. (NULL si aucune valeur ne semble identique aux autres.) |
| most_common_freqs | real[] | | Une liste des fréquences des valeurs les plus habituelles, c'est−à−dire le nombre d'occurrences de chacune divisé par le nombre total de lignes. (NULL lorsque most_common_vals l'est.) |
| histogram_bounds | anyarray | | Une liste des valeurs qui divisent les valeurs de la colonne en groupes de population approximativement identique. Les valeurs dans most_common_vals, s'ils existent, sont omises de ce calcul d'histogramme. (Cette colonne est NULL si le type de données de la colonne ne dispose pas de l'opérateur < ou si la liste most_common_vals tient compte de la population |

| | | | |
|---|---|---|---|
| | | | complète.) |
| correlation | real | | Corrélation statistique entre l'ordre physique des lignes et l'ordre logique des valeurs de la colonne. Ceci va de −1 à +1. Lorsque la valeur est plus près de −1 ou +1, un parcours de l'index sur la colonne sera estimé moins cher que si cette valeur tend vers zéro, à cause de la réduction d'accès aléatoire au disque. (Cette colonne est NULL si le type de données de la colonne n'a pas l'opérateur < operator.) |

Le nombre maximum d'entrées dans `most_common_vals` et `histogram_bounds` est configurable colonne par colonne en utilisant la commande `ALTER TABLE SET STATISTICS` ou globalement avec le paramètre d'exécution default_statistics_target.

## 41.37. pg_tables

La vue `pg_tables` fournit un accès à des informations utiles sur chaque table de la base de données.

**Tableau 41−37. Colonnes de `pg_tables`**

| Nom | Type | Références | Description |
|---|---|---|---|
| schemaname | name | pg_namespace.nspname | nom du schéma contenant la table |
| tablename | name | pg_class.relname | nom de la table |
| tableowner | name | pg_shadow.usename | nom du propriétaire de la table |
| tablespace | name | pg_tablespace.spcname | nom de l'espace logique contenant la table (NULL s'il s'agit de l'espace logique par défaut pour cette base) |
| hasindexes | boolean | pg_class.relhasindex | vrai si la table a (ou a récemment eu) des index |
| hasrules | boolean | pg_class.relhasrules | vrai si la table dispose de règles |
| hastriggers | boolean | pg_class.reltriggers | vrai si la table dispose de déclencheurs (triggers) |

## 41.38. pg_user

La vue `pg_user` fournit un accès aux informations concernant les utilisateurs de la base de données. C'est une simple vue lisible publiquement de pg_shadow mais qui n'affiche pas le champ du mot de passe.

**Tableau 41−38. Colonnes de `pg_user`**

| Nom | Type | Références | Description |
|---|---|---|---|
| usename | name | | Nom de l'utilisateur |
| usesysid | int4 | | Identifiant de l'utilisateur (un nombre arbitraire utilisé en référence à cet utilisateur) |
| usecreatedb | bool | | L'utilisateur peut créer des bases de données |
| usesuper | bool | | L'utilisateur est un super−utilisateur |
| usecatupd | bool | | L'utilisateur pourrait mettre à jour les tables systèmes. (Même un super−utilisateur ne pourrait pas le faire tant que cette colonne n'est pas initialisée à vrai. |
| passwd | text | | Ce n'est pas le mot de passe (toujours ********) |
| valuntil | abstime | | Temps d'expiration du mot de passe (utilisé seulement pour l'authentification des mots de passe) |
| useconfig | text[] | | Variables d'exécution par défaut de la session |

## 41.39. pg_views

La vue pg_views fournit un accès à d'importantes informations de chaque vue de la base de données.

**Tableau 41−39. Colonnes de pg_views**

| Nom | Type | Références | Description |
|---|---|---|---|
| schemaname | name | pg_namespace.nspname | Nom du schéma contenant la vue |
| viewname | name | pg_class.relname | Nom de la vue |
| viewowner | name | pg_shadow.usename | Nom du propriétaire de la vue |
| definition | text | | Définition de la vue (une requête SELECT reconstruite) |

# Chapitre 42. Protocole client/serveur

PostgreSQL utilise un protocole messages pour la communication entre les clients et les serveurs (<< frontend >> et << backend >>). Le protocole est supporté par TCP/IP et par les sockets de domaine Unix. Le numéro de port 5432 a été enregistré par l'IANA comme numéro de port TCP personnalisé pour les serveurs supportant ce protocole mais en pratique tout numéro de port non privilégié peut être utilisé.

Ce document décrit la version 3.0 de ce protocole, telle qu'implantée dans PostgreSQL depuis la version 7.4. Pour obtenir la description des versions précédentes du protocole, il faudra se reporter aux versions antérieures de la documentation de PostgreSQL. Un même serveur peut supporter plusieurs versions du protocole. Lors de l'établissement de la communication le client indique au serveur la version du protocole qu'il souhaite utiliser. Le serveur suivra ce protocole s'il en est capable.

Les fonctionnalités de haut niveau construites sur ce protocole (par exemple, la façon dont libpq passe certaines variables d'environnement à l'établissement de la connexion) ne sont pas couvertes par ce chapitre.

Pour répondre efficacement à de multiple clients, le serveur lance un nouveau serveur (<< backend >>) pour chaque client. Dans l'implémentation actuelle, un nouveau processus fils est créé immédiatement après la détection d'une connexion entrante. Et cela de façon transparente pour le protocole. Pour le protocole, les termes << backend >> et << serveur >> sont interchangeables ; comme << frontend >>, << interface >> et << client >>.

## 42.1. Aperçu

Le protocole utilise des phases distinctes pour le lancement et le fonctionnement habituel. Dans la phase de lancement, le client ouvre une connexion au serveur et s'authentifie. (Ce qui peut impliquer un message simple, ou plusieurs messages, en fonction de la méthode d'authentification utilisée.) En cas de réussite, le serveur envoie une information de statut au client et entre dans le mode normal de fonctionnement. Exception faite du message initial de demande de lancement, cette partie du protocole est conduite par le serveur.

En mode de fonctionnement normal, le client envoie requêtes et commandes au serveur et celui−ci retourne les résultats de requêtes et autres réponses. Il existe quelques cas (comme `NOTIFY`) pour lesquels le serveur enverra des messages non sollicités. Mais dans l'ensemble, cette partie de la session est conduite par les requêtes du client.

En général, c'est le client qui décide de la clôture de la session. Il arrive, cependant, qu'elle soit forcée par le moteur. Dans tous les cas, lors de la fermeture de la connexion par le serveur, toute transaction ouverte (non terminée) sera annulée.

En mode opérationnel normal, les commandes SQL peuvent être exécutées via deux sous−protocoles. Dans le protocole des << requêtes simples >>, le client envoie juste une chaîne, la requête, qui est analysée et exécutée immédiatement par le serveur. Dans le protocole des << requêtes étendues >>, le traitement des requêtes est découpé en de nombreuses étapes : l'analyse, le lien avec les valeurs de paramètres et l'exécution. Ceci offre flexibilité et gains en performances au prix d'une complexité supplémentaire.

Le mode opérationnel normal offre des sous−protocoles supplémentaires pour certaines opérations comme `COPY`.

### 42.1.1. Aperçu des messages

Toute la communication s'effectue au travers d'un flux de messages. Le premier octet d'un message identifie le type de message et les quatre octets suivants spécifient la longueur du reste du message (cette longueur inclut les 4 octets de longueur, mais pas l'octet du type de message). Le reste du contenu du message est déterminé par le type de message. Pour des raisons historiques, le tout premier message envoyé par le client (le message de lancement) n'a pas l'octet initial de type du message.

Pour éviter de perdre la synchronisation avec le flux de messages, le serveur et le client stocke le message complet dans un tampon (en utilisant le nombre d'octets) avant de tenter de traiter son contenu. Cela permet une récupération simple si une erreur est détectée lors du traitement du contenu. Dans les situations extrêmes (telles que de ne pas avoir assez de mémoire pour placer le message dans le tampon), le récepteur peut utiliser le nombre d'octets pour déterminer le nombre d'entrées à ignorer avant de continuer la lecture des messages.

En revanche, serveurs et clients doivent être attentifs à ne pas envoyer de message incomplet. Ceci est habituellement obtenu en plaçant le message complet dans un tampon avant de commencer l'envoi. Si un échec de communications survient pendant l'envoi ou la réception d'un message, la seule réponse plausible est l'abandon de la connexion. Il y a, en effet, peu d'espoir de resynchronisation des messages.

---

### 42.1.2. Aperçu des requêtes étendues

Dans le protocole des requêtes étendues, l'exécution de commandes SQL est scindée en plusieurs étapes. L'état retenu entre les étapes est représenté par deux types d'objets : les *instructions préparées* et les *portails*. Une instruction préparée représente le résultat de l'analyse syntaxique, de l'analyse sémantique et de la planification d'une chaîne de requête textuelle. Une instruction préparée n'est pas nécessairement prête à être exécutée parce qu'il peut lui manquer certaines valeurs de *paramètres*. Un portail représente une instruction prête à être exécutée ou déjà partiellement exécutée, dont toutes les valeurs de paramètres manquant sont données. (Pour les instructions `SELECT`, un portail est équivalent à un curseur ouvert. Il est choisi d'utiliser un terme différent car les curseurs ne gèrent pas les instructions autres que `SELECT`.)

Le cycle d'exécution complet consiste en une étape d'*analyse syntaxique*, qui crée une instruction préparée à partir d'une chaîne de requête textuelle ; une étape de *liaison*, qui crée un portail à partir d'une instruction préparée et des valeurs pour les paramètres nécessaires ; et une étape d'*exécution* qui exécute une requête du portail. Dans le cas d'une requête qui renvoie des lignes (`SELECT`, `SHOW`, etc), il peut être signalé à l'étape d'exécution que seul un certain nombre de lignes doivent être retournées, de sorte que de multiples étapes d'exécution seront nécessaires pour terminer l'opération.

Le serveur peut garder la trace de multiples instructions préparées et portails (qui n'existent qu'à l'intérieur d'une session, et ne sont jamais partagés entre les sessions). Les instructions préparées et les portails sont référencés par les noms qui leur sont affectés à la création. De plus, il existe une instruction préparée et un portail << non nommés >>. Bien qu'ils se comportent comme les objets nommés, les opérations y sont optimisées en vue d'une exécution unique de la requête avant son annulation puis est annulée. En revanche, les opérations sur les objets nommés sont optimisées pour des utilisations multiples.

---

### 42.1.3. Formats et codes de format

Les données d'un type particulier pouvaient être transmises sous différents *formats*. Depuis PostgreSQL 7.4, les seuls formats supportés sont le << texte >> et le << binaire >> mais le protocole prévoit des extensions futures. Le format désiré pour toute valeur est spécifié par un *code de format*. Les clients peuvent spécifier un code de format pour chaque valeur de paramètre transmise et pour chaque colonne du résultat d'une requête.

Le texte a zéro pour code de format zéro, le binaire un. Tous les autres codes de format sont réservés pour des définitions futures.

La représentation au format texte des valeurs est toute chaîne produite et acceptée par les fonctions de conversion en entrée/sortie pour le type de données particulier. Dans la représentation transmise, il n'y a pas de caractère nul de terminaison de chaîne ; le client doit en ajouter un s'il souhaite traiter les valeurs comme des chaînes C (le format texte n'autorise pas les valeurs nulles intégrées).

Les représentations binaires des entiers utilisent l'ordre d'octet réseau (octet le plus significatif en premier). Pour les autres types de données, il faudra consulter la documentation ou le code source pour connaître la représentation binaire. Les représentations binaires des types de données complexes changent parfois entre les versions du serveur ; le format texte reste le choix le plus portable.

## 42.2. Flux de messages

Cette section décrit le flux des messages et la sémantique de chaque type de message (les détails concernant la représentation exacte de chaque message apparaît dans Section 42.4). Il existe différents sous−protocoles en fonction de l'état de la connexion : lancement, requête, appel de fonction, COPY et clôture. Il existe aussi des provisions spéciales pour les opérations asynchrones (incluant les réponses aux notifications et les annulations de commande), qui peuvent arriver à tout moment après la phase de lancement.

### 42.2.1. Lancement

Pour débuter une session, un client ouvre une connexion au serveur et envoie un message de démarrage. Ce message inclut les noms de l'utilisateur et de la base de données à laquelle le client souhaite se connecter ; il identifie aussi la version particulière du protocole à utiliser. (Optionnellement, le message de démarrage peut inclure des précisions supplémentaires pour les paramètres d'exécution.) Le serveur utilise ces informations et le contenu des fichiers de configuration (tels que `pg_hba.conf`) pour déterminer si la connexion est acceptable et quelle éventuelle authentification supplémentaire est requise.

Le serveur envoie ensuite le message de demande d'authentification approprié, auquel le client doit répondre avec le message de réponse d'authentification adapté (tel un mot de passe). En principe, le cycle demande/réponse d'authentification peut requérir plusieurs itérations mais aucune des méthodes d'authentification actuelles n'utilise plus d'un cycle. Avec certaines méthodes, aucune réponse du client n'est nécessaire  aucune demande d'authentification n'est alors effectuée.

Le cycle d'authentification se termine lorque le serveur rejette la tentative de connexion (ErrorResponse) ou l'accepte (AuthenticationOk).

Les messages possibles du serveur dans cette phase sont :

ErrorResponse
: La tentative de connexion a été rejetée. Le serveur ferme immédiatement la connexion.

AuthenticationOk
: L'échange d'authentification s'est terminé avec succès.

AuthenticationKerberosV4
: Le client doit alors prendre part à un dialogue d'authentification Kerberos V4 (spécification Kerberos, non décrite ici) avec le serveur. En cas de succès, le serveur répond par AuthenticationOk, ErrorResponse sinon.

AuthenticationKerberosV5

Le client doit alors prendre part à un dialogue d'authentification Kerberos V5 (spécification Kerberos, non décrite ici) avec le serveur. En cas de succès, le serveur répond AuthenticationOk, ErrorResponse sinon.

AuthenticationCleartextPassword
: Le client doit alors envoyer un PasswordMessage contenant le mot de passe en clair. Si le mot de passe est correct, le serveur répond AuthenticationOk, ErrorResponse sinon.

AuthenticationCryptPassword
: Le client doit alors envoyer un PasswordMessage contenant le mot de passe chiffré à l'aide de crypt(3), en utilisant le composant salt de deux caractères spécifié dans le message AuthenticationCryptPassword. Si le mot de passe est correct, le serveur répond AuthenticationOk, ErrorResponse sinon.

AuthenticationMD5Password
: Le client doit alors envoyer un PasswordMessage contenant le mot de passe chiffré à l'aide de MD5, en utilisant le composant salt de quatre caractères spécifié dans le message AuthenticationMD5Password. Si le mot de passe est correct, le serveur répond AuthenticationOk, ErrorResponse sinon.

AuthenticationSCMCredential
: Cette réponse est possible uniquement pour les connexions locales de domaine Unix sur les plateformes qui supportent les messages de légitimation SCM. Le client doit fournir un message de légitimation SCM, puis envoyer une donnée d'un octet. Le contenu de cet octet importe peu ; il n'est utilisé que pour s'assurer que le serveur attend assez longtemps pour recevoir le message de légitimation. Si la légitimation est acceptable, le serveur répond AuthenticationOk, ErrorResponse sinon.

Si le client ne supporte pas la méhode d'authentification demandée par le serveur, il doit immédiatement fermer la connexion.

Après la réception du message AuthenticationOk, le client attend d'autres messages du serveur. Au cours de cette phase, un processus serveur est lancé et le client est simplement en attente. Il est encore possible que la tentative de lancement échoue (ErrorResponse) mais, dans la plupart des cas, le serveur enverra les messages ParameterStatus, BackendKeyData et enfin ReadyForQuery.

Durant cette phase, le serveur tentera d'appliquer tous les paramètres d'exécution supplémentaires qui ont été fournis par le message de lancement. En cas de succès, ces valeurs deviennent les valeurs par défaut de la session. Une erreur engendre ErrorResponse et déclenche la sortie.

Les messages possibles du serveur dans cette phase sont :

BackendKeyData
: Ce message fournit une clé secrète que le client doit conserver s'il souhaite envoyer des annulations de requêtes par la suite. Le client ne devrait pas répondre à ce message, mais continuer à attendre un message ReadyForQuery.

ParameterStatus
: Ce message informe le client de la configuration actuelle (initiale) des paramètres du serveur, tels `client_encoding` ou `DateStyle`. Le client peut ignorer ce message ou enregistrer la configuration pour ses besoins futurs ; voir Section 42.2.6 pour plus de détails. Le client ne devrait pas répondre à ce message mais continuer à attendre un message ReadyForQuery.

ReadyForQuery
: Le lancement est terminé. Le client peut dès lors envoyer des commandes.

ErrorResponse
: Le lancement a échoué. La connexion est fermée après l'envoi de ce message.

NoticeResponse
: Un message d'avertissement a été envoyé. Le client devrait afficher ce message mais continuer à attendre un ReadyForQuery ou un ErrorResponse.

Le même message ReadyForQuery est envoyé à chaque cycle de commande. En fonction des besoins de codage du client, il est possible de considérer ReadyForQuery comme le début d'un cycle de commande, ou de le considérer comme terminant la phase de lancement et chaque cycle de commande.

## 42.2.2. Requête simple

Un cycle de requête simple est initié par le client qui envoie un message Query au serveur. Le message inclut une commande SQL (ou plusieurs) exprimée comme une chaîne texte. Le serveur envoie, alors, un ou plusieurs messages de réponse dépendant du contenu de la chaîne représentant la requête et enfin un message ReadyForQuery. ReadyForQuery informe le client qu'il peut envoyer une nouvelle commande. Il n'est pas nécessaire que le client attende ReadyForQuery avant de lancer une autre commande mais le client prend alors la responsabilité de ce qui arrive si la commande précédente échoue et que les commandes suivantes, déjà lancées, réussissent.

Les messages de réponse du serveur sont :

CommandComplete
: Commande SQL terminée normalement.

CopyInResponse
: Le serveur est prêt à copier des données du client vers une table  voir Section 42.2.5.

CopyOutResponse
: Le serveur est prêt à copier des données d'une table vers le client ; voir Section 42.2.5.

RowDescription
: Indique que des lignes vont être envoyées en réponse à une requête `SELECT`, `FETCH`... Le contenu de ce message décrit le placement des colonnes dans les lignes. Le contenu est suivi d'un message DataRow pour chaque ligne envoyée au client.

DataRow
: Un des ensembles de lignes retournés par une requête `SELECT`, `FETCH`...

EmptyQueryResponse
: Une chaîne de requête vide a été reconnue.

ErrorResponse
: Une erreur est survenue.

ReadyForQuery
: Le traitement d'une requête est terminé. Un message séparé est envoyé pour l'indiquer parce qu'il se peut que la chaîne de la requête contienne plusieurs commandes SQL. CommandComplete marque la fin du traitement d'une commande SQL, pas de la chaîne complète. ReadyForQuery sera toujours envoyé que le traitement se termine avec succès ou non.

NoticeResponse
: Un message d'avertissement concernant la requête a été envoyé. Les avertissements sont complémentaires des autres réponses, le serveur continuera à traiter la commande.

La réponse à une requête `SELECT` (ou à d'autres requêtes, telles `EXPLAIN` ou `SHOW`, qui retournent des ensembles de données) consiste normalement en un RowDescription, plusieurs messages DataRow (ou aucun) et pour finir un CommandComplete. `COPY` depuis ou vers le client utilise un protocole spécial décrit dans Section 42.2.5. Tous les autres types de requêtes produisent uniquement un message CommandComplete.

Puisqu'une chaîne de caractères peut contenir plusieurs requêtes (séparées par des points virgules), il peut y avoir plusieurs séquences de réponses avant que le serveur ne finisse de traiter la chaîne. ReadyForQuery est envoyé lorsque la chaîne complète a été traitée et que le serveur est prêt à accepter une nouvelle chaîne de requêtes.

Si une chaîne de requêtes complètement vide est reçue (aucun contenu autre que des espaces fines), la réponse sera EmptyQueryResponse suivie de ReadyForQuery.

En cas d'erreur, ErrorResponse est envoyé suivi de ReadyForQuery. Tous les traitements suivants de la chaîne sont annulés par ErrorResponse (quelque soit le nombre de requêtes restant à traiter). Ceci peut survenir au milieu de la séquence de messages engendrés par une requête individuelle.

En mode de requêtage simple, les valeurs récupérées sont toujours au format texte, sauf si la commande est un `FETCH` sur un curseur déclaré avec l'option `BINARY`. Dans ce cas, les valeurs récupérées sont au format binaire. Les codes de format donnés dans le message RowDescription indiquent le format utilisé.

Un client doit être préparé à accepter des messages ErrorResponse et NoticeResponse quand bien même il s'attendrait à un autre type de message. Voir aussi Section 42.2.6 concernant les messages que le client pourrait engendrer du fait d'événements extérieurs.

La bonne pratique consiste à coder les clients dans un style machine−état qui acceptera tout type de message à tout moment plutôt que de parier sur la séquence exacte des messages.

## 42.2.3. Requête étendue

Le protocole de requête étendu divise le protocole de requêtage simple décrit ci−dessus en plusieurs étapes. Les résultats des étapes de préparation peuvent être réutilisés plusieurs fois pour plus d'efficacité. De plus, des fonctionnalités supplémentaires sont disponibles, telles que la possibilité de fournir les valeurs des données comme des paramètres séparés au lieu d'avoir à les insérer directement dans une chaîne de requêtes.

Dans le protocole étendu, le client envoie tout d'abord un message Parse qui contient une chaîne de requête, optionnellement quelques informations sur les types de données aux emplacements des paramètres, et le nom de l'objet de destination d'une instruction préparée (une chaîne vide sélectionne l'instruction préparée sans nom). La réponse est soit ParseComplete soit ErrorResponse. Les types de données des paramètres peuvent être spécifiés par l'OID ; dans le cas contraire, l'analyseur tente d'inférer les types de données de la même façon qu'il le ferait pour les constantes chaînes littérales non typées.

> **Note :** La chaîne contenue dans un message Parse ne peut pas inclure plus d'une instruction SQL, sinon une erreur de syntaxe est rapportée. Cette restriction n'existe pas dans le protocole de requête simple, mais est présente dans le protocole étendu. En effet, permettre aux instructions préparées ou aux portails de contenir de multiples commandes compliquerait inutilement le procotole.

En cas de succès de sa création, une instruction préparée nommée dure jusqu'à la fin de la session courante, sauf si elle est détruite explicitement. Une instruction préparée non nommée ne dure que jusqu'à la prochaine instruction Parse spécifiant l'instruction non nommée comme destination. Un simple message Query détruit également l'instruction non nommée. Les instructions préparées nommées doivent être explicitement closes avant de pouvoir être redéfinies par un message Parse. Ce n'est pas obligatoire pour une instruction non nommée. Il est également possible de créer des instructions préparées nommées, et d'y accéder, en ligne de commandes SQL à l'aide des instructions `PREPARE` et `EXECUTE`.

Dès lors qu'une instruction préparée existe, elle est déclarée exécutable par un message Bind. Le message Bind donne le nom de l'instruction préparée source (une chaîne vide désigne l'instruction préparée non nommée), le nom du portail destination (une chaîne vide désigne le portail non nommé) et les valeurs à utiliser pour tout emplacement de paramètres présent dans l'instruction préparée. L'ensemble des paramètres fournis doit correspondre à ceux nécessaires à l'instruction préparée. Bind spécifie aussi le format à utiliser pour toutes les données renvoyées par la requête ; le format peut être spécifié complètement ou par colonne. La réponse est, soit BindComplete, soit ErrorResponse.

> **Note :** Le choix entre sortie texte et binaire est déterminé par les codes de format donnés dans Bind, quelque soit la commande SQL impliquée. L'attribut `BINARY` dans les déclarations du curseur n'est pas pertinent lors de l'utilisation du protocole de requête étendue.

En cas de succès de sa création, un objet portail nommé dure jusqu'à la fin de la transaction courante sauf s'il est explicitement détruit. Un portail non nommé est détruit à la fin de la transaction ou dès la prochaine instruction Bind spécifiant le portail non nommé comme destination. Un simple message Query détruit également le portail non nommé. Les portails nommés doivent être explicitement fermés avant de pouvoir être redéfinis par un message Bind. Cela n'est pas obligatoire pour le portail non nommé. Il est également possible de créer des portails nommés, et d'y accéder, en ligne de commandes SQL à l'aide des instructions `DECLARE CURSOR` et `FETCH`.

Dès lors qu'un portail existe, il peut être exécuté à l'aide d'un message Execute. Ce message spécifie le nom du portail (une chaîne vide désigne le portail non nommé) et un nombre maximum de lignes de résultat (zéro signifiant la << récupération de toutes les lignes >>). Le nombre de lignes de résultat a seulement un sens pour les portails contenant des commandes qui renvoient des ensembles de lignes ; dans les autres cas, la commande est toujours exécutée jusqu'à la fin et le nombre de lignes est ignoré. Les réponses possibles d'Execute sont les même que celles décrites ci−dessus pour les requêtes lancées via le protocole de requête simple, si ce n'est qu'Execute ne cause pas l'envoi de ReadyForQuery.

Si Execute se termine avant la fin de l'exécution d'un portail (du fait d'un nombre de lignes de résultats différent de zéro), il enverra un message PortalSuspended ; la survenue de ce message indique au client qu'un autre Execute devrait être lancé sur le même portail pour terminer l'opération. Le message CommandComplete indiquant la fin de la commande SQL n'est pas envoyé avant l'exécution complète du portail. Une phase Execute est toujours terminée par la survenue d'un seul de ces messages : CommandComplete, EmptyQueryResponse (si le portail a été créé à partir d'une chaîne de requête vide), ErrorResponse ou PortalSuspended.

À la réalisation complète de chaque série de messages de requêtes étendues, le client doit lancer un message Sync. Ce message sans paramètre oblige le serveur à fermer la transaction courante si elle n'est pas à l'intérieur d'un bloc de transaction `BEGIN`/`COMMIT` (<< fermer >> signifiant valider en l'absence d'erreur ou annuler sinon). Une réponse ReadyForQuery est alors envoyée. Le but de Sync est de fournir un point de resynchronisation pour les récupérations d'erreurs. Quand une erreur est détectée lors du traitement d'un message de requête étendue, le serveur lance ErrorResponse, puis lit et annule les messages jusqu'à ce qu'un Sync soit atteint. Il envoie ensuite ReadyForQuery et retourne au traitement normal des messages. Aucun échappement n'est réalisé si une erreur est détectée *lors* du traitement de Sync −−− l'unicité du ReadyForQuery envoyé pour chaque Sync est ainsi assuré.

> **Note :** Sync n'impose pas la fermeture d'un bloc de transactions ouvert avec `BEGIN`. Cette situation est détectable car le message ReadyForQuery inclut le statut de la transaction.

En plus de ces opérations fondamentales, requises, il y a plusieurs opérations optionnelles qui peuvent être utilisées avec le protocole de requête étendue.

Le message Describe (variante de portail) spécifie le nom d'un portail existant (ou une chaîne vide pour le portail non nommé). La réponse est un message RowDescription décrivant les lignes qui seront renvoyées par l'exécution du portail ; ou un message NoData si le portail ne contient pas de requête renvoyant des lignes ; ou ErrorResponse le portail n'existe pas.

Le message Describe (variante d'instruction) spécifie le nom d'une instruction préparée existante (ou une chaîne vide pour l'instruction préparée non nommée). La réponse est un message ParameterDescription décrivant les paramètres nécessaires à l'instruction, suivi d'un message RowDescription décrivant les lignes qui seront renvoyées lors de l'éventuelle exécution de l'instrution (ou un message NoData si l'instruction ne renvoie pas de lignes). ErrorResponse est retourné si l'instruction préparée n'existe pas. Comme Bind n'a pas encore été exécuté, les formats à utiliser pour les lignes retournées ne sont pas encore connues du serveur ; dans ce cas, les champs du code de format dans le message RowDescription seront composés de zéros.

> **Astuce :** Dans la plupart des scénarios, le client devra exécuter une des variantes de Describe avant de lancer Execute pour s'assurer qu'il sait interpréter les résultats reçus.

Le message Close ferme une instruction préparée ou un portail et libère les ressources. L'exécution de Close sur une instruction ou un portail inexistant ne constitue pas une erreur. La réponse est en général CloseComplete mais peut être ErrorResponse si une difficulté quelconque est rencontrée lors de la libération des ressources. Clore une instruction préparée ferme implicitement tout autre portail ouvert construit à partir de cette instruction.

Le message Flush n'engendre pas de sortie spécifique, mais force le serveur à délivrer toute donnée restante dans les tampons de sortie. Un Flush doit être envoyé après toute commande de requête étendue, à l'exception de Sync, si le client souhaite examiner le résultat de cette commande avant de lancer d'autres commandes. Sans Flush, les messages retournés par le serveur seront combinés en un nombre minimum de paquets pour minimiser la charge réseau.

> **Note :** Le message Query simple est approximativement équivalent aux séries Parse, Bind, Describe sur un portail, Execute, Close, Sync utilisant les objets de l'instruction préparée ou du portail, non nommés et sans paramètres. Une différence est l'acceptation de plusieurs instructions SQL dans la chaîne de requêtes, la séquence bind/describe/execute étant automatiquement réalisée pour chacune, successivement. Il en diffère également en ne retournant pas les messages ParseComplete, BindComplete, CloseComplete ou NoData.

## 42.2.4. Appel de fonction

Le sous−protocole d'appel de fonction (NDT : Function Call dans la version originale) permet au client d'effectuer un appel direct à toute fonction du catalogue système `pg_proc` de la base de données. Le client doit avoir le droit d'exécution de la fonction.

> **Note :** Le sous−protocole d'appel de fonction est une fonctionnalité qu'il vaudrait probablement mieux éviter dans tout nouveau code. Des résultats similaires peuvent être obtenus en initialisant une instruction préparée qui lance `SELECT function($1, ...)`. Le cycle de l'appel de fonction peut alors être remplacé par Bind/Execute.

Un cycle d'appel de fonction est initié par le client envoyant un message FunctionCall au serveur. Le serveur envoie alors un ou plusieurs messages de réponse en fonction des résultats de l'appel de la fonction et finalement un message de réponse ReadyForQuery. ReadyForQuery informe le client qu'il peut envoyer en toute sécurité une nouvelle requête ou un nouvel appel de fonction.

Les messages de réponse possibles du serveur sont :

ErrorResponse
: Une erreur est survenue.

FunctionCallResponse
: L'appel de la fonction est terminé et a retourné le résultat donné dans le message. Le protocole d'appel de fonction ne peut gérer qu'un résultat scalaire simple, pas un type ligne ou un ensemble de résultats.

ReadyForQuery
: Le traitement de l'appel de fonction est terminé. ReadyForQuery sera toujours envoyé, que le traitement se termine avec succès ou avec une erreur.

NoticeResponse
: Un message d'avertissement relatif à l'appel de fonction a été retourné. Les avertissements sont complémentaires des autres réponses, c'est−à−dire que le seveur continuera à traiter la commande.

## 42.2.5. Opérations COPY

La commande `COPY` permet des transferts rapides de données en lot vers ou à partir du serveur. Les opérations Copy−in et Copy−out basculent chacune la connexion dans un sous−protocole distinct qui existe jusqu'à la fin de l'opération.

Le mode Copy−in (transfert de données vers le serveur) est initié quand le serveur exécute une instruction SQL `COPY FROM STDIN`. Le serveur envoie une message CopyInResponse au client. Le client peut alors envoyer zéro (ou plusieurs) message(s) CopyData, formant un flux de données en entrée. (Il n'est pas nécessaire que les limites du message aient un rapport avec les limites de la ligne, mais cela est souvent un choix raisonnable.) Le client peut terminer le mode Copy−in en envoyant un message CopyDone (permettant une fin avec succès) ou un message CopyFail (qui causera l'échec de l'instruction SQL `COPY` avec une erreur). Le serveur retourne alors au mode de traitement de la commande précédant le début de `COPY`, protocole de requête simple ou étendu. Il enverra enfin CommandComplete (en cas de succès) ou ErrorResponse (sinon).

Si le serveur détecte un erreur en mode copy−in (ce qui inclut la réception d'un message CopyFail), il enverra un message ErrorResponse. Si la commande `COPY` a été lancée à l'aide d'un message de requête étendue, le serveur annulera les messages du client jusqu'à ce qu'un message Sync soit reçu. Il enverra alors un message ReadyForQuery et retournera dans le mode de fonctionnement normal. Si la commande `COPY` a été lancée dans un message simple Query, le reste de ce message est annulé et ReadyForQuery est envoyé. Dans tous les cas, les messages CopyData, CopyDone ou CopyFail suivants envoyés par l'interface seront simplement annulés.

Le serveur ignorera les messages Flush et Sync reçus en mode copy−in. La réception de tout autre type de messages hors−copie constitue une erreur qui annulera l'état Copy−in, comme cela est décrit plus haut. L'exception pour Flush et Sync est faite pour les bibliothèques clientes qui envoient systématiquement Flush ou Sync après un message Execute sans vérifier si la commande à exécuter est `COPY FROM STDIN`.

Le mode Copy−out (transfert de données à partir du serveur) est initié lorsque le serveur exécute une instruction SQL `COPY TO STDOUT`. Le moteur envoie un message CopyOutResponse au client suivi de zéro (ou plusieurs) message(s) CopyData (un par ligne), suivi de CopyDone. Le serveur retourne ensuite au mode de traitement de commande dans lequel il se trouvait avant le lancement de `COPY` et envoie CommandComplete. Le client ne peut pas annuler le tranfert (sauf en fermant la connexion ou en lançant une requête d'annulation, Cancel), mais il peut ignorer les messages CopyData et CopyDone non souhaités.

Si le serveur détecte une erreur en mode Copy−out, il enverra un message ErrorResponse et retournera dans le mode de traitement normal. Le client devrait traiter la réception d'un message ErrorResponse (ou en fait tout type de message autre que CopyData ou CopyDone) par la clôture du mode Copy−out.

Les messages CopyInResponse et CopyOutResponse incluent les champs qui informent le client du nombre de colonnes par ligne et des codes de format de chaque colonne. Dans l'implantation actuelle, toutes les colonnes d'une opération `COPY` donnée utilisent le même format, mais la conception du message n'en tient pas compte.

## 42.2.6. Opérations asynchrones

Il existe plusieurs cas pour lesquels le serveur enverra des messages qui ne sont pas spécifiquement demandés par le flux de commande du client. Les clients doivent être préparés à gérer ces messages à tout moment même si aucune requête n'est en cours. Vérifier ces cas avant de commencer à lire la réponse d'une requête est un minimum.

Il est possible que des messages NoticeResponse soient engendrés en dehors de toute activité ; par exemple, si l'administrateur de la base de données commande un arrêt << rapide >> de la base de données, le serveur enverra un NoticeResponse l'indiquant avant de fermer la connexion. Les clients devraient toujours être prêts à accepter et afficher ces messages, même si la connexion est inactive.

Des messages ParameterStatus seront engendrés à chaque fois que la valeur active d'un paramètre est modifiée, et cela pour tout paramètre que le serveur pense utile au client. Cela survient plus généralement en réponse à une commande SQL `SET` exécutée par le client. Ce cas est en fait synchrone −−− mais il est possible aussi que le changement de statut d'un paramètre survienne à la suite d'une modification par l'administrateur des fichiers de configuration ; changements suivis de l'envoi du signal SIGHUP au postmaster. De plus, si une commande SET est annulée, un message ParameterStatus approprié sera engendré pour rapporter la valeur effective.

À ce jour, il existe un certain nombre de paramètres codés en dur pour lesquels des messages ParameterStatus seront engendrés : on trouve `server_version` (un pseudo paramètre non modifiable après le lancement) ; `client_encoding` ; `is_superuser` ; `session_authorization` ; `DateStyle`. Cet ensemble pourrait changer dans le futur, voire devenir configurable. De toute façon, un client peut ignorer un message ParameterStatus pour les paramètres qu'il ne comprend pas ou qui ne le concernent pas.

Si un client lance une commande `LISTEN`, alors le serveur enverra un message NotificationResponse (à ne pas confondre avec NoticeResponse !) à chaque fois qu'une commande `NOTIFY` est exécutée pour la notification de même nom.

> **Note :** Actuellement, NotificationResponse ne peut être envoyé qu'à l'extérieur d'une transaction. Il ne surviendra donc pas au milieu d'une réponse à une commande, mais il peut survenir juste avant ReadyForQuery. Il est toutefois déconseillé de concevoir un client en partant de ce principe. La bonne pratique est d'être capable d'accepter NotificationResponse à tout moment du protocole.

## 42.2.7. Annulation de requêtes en cours

Pendant le traitement d'une requête, le client peut demander l'annulation de la requête. La demande d'annulation n'est pas envoyée directement au serveur par la connexion ouverte pour des raisons d'efficacité de l'implémentation : il n'est pas admissible que le serveur vérifie constamment les messages émanant du client

lors du traitement des requêtes. Les demandes d'annulation sont relativement inhabituelles ; c'est pourquoi elles sont traitées de manière relativement simple afin d'éviter que ce traitement ne pénalise le fonctionnement normal.

Pour effectuer une demande d'annulation, le client ouvre une nouvelle connexion au serveur et envoie un message CancelRequest à la place du message StartupMessage envoyé habituellement à l'ouverture d'une connexion. Le serveur traitera cette requête et fermera la connexion. Pour des raisons de sécurité, aucune réponse directe n'est faite au message de requête d'annulation.

Un message CancelRequest sera ignoré sauf s'il contient la même donnée clé (PID et clé secrète) que celle passée au client lors du démarrage de la connexion. Si la donnée clé correspond, le traitement de la requête en cours est annulé. (Dans l'implantation existante, ceci est obtenu en envoyant un signal spécial au processus serveur qui traite la requête.)

Le signal d'annulation peut ou non être suivi d'effet −−− par exemple, s'il arrive après la fin du traitement de la requête par le serveur, il n'aura alors aucun effet. Si l'annulation est effective, il en résulte la fin précoce de la commande accompagnée d'un message d'erreur.

De tout ceci, il ressort que, pour des raisons de sécurité et d'efficacité, le client n'a aucun moyen de savoir si la demande d'annulation a abouti. Il continuera d'attendre que le serveur réponde à la requête. Effectuer une annulation permet simplement d'augmenter la probabilité de voir la requête en cours finir rapidement et échouer accompagnée d'un message d'erreur plutôt que réussir.

Comme la requête d'annulation est envoyée via une nouvelle connexion au serveur et non pas au travers du lien de communication client/serveur établi, il est possible que la requête d'annulation soit lancée par un processus quelconque, pas forcément celui du client pour lequel la requête doit être annulée. Cela peut présenter quelques avantages de flexibilité dans la construction d'applications multi−processus ; mais également une faille de sécurité puisque des personnes non autorisées pourraient tenter d'annuler des requêtes. La faille de sécurité est comblé par l'exigence d'une clé secrète, engendrée dynamiquement, pour toute requête d'annulation.

## 42.2.8. Fin

Lors de la procédure normale de fin le client envoie un message Terminate et ferme immédiatement la connexion. À la réception de ce message, le serveur ferme la connexion et s'arrête.

Dans de rares cas (tel un arrêt de la base de données par l'administrateur), le serveur peut se déconnecter sans demande du client. Dans de tels cas, le serveur tentera d'envoyer un message d'erreur ou d'avertissement en donnant la raison de la déconnexion avant de fermer la connexion.

D'autres scénarios de fin peuvent être dus à différents cas d'échecs, tels qu'un << core dump >> côté client ou serveur, la perte du lien de communications, la perte de synchronisation des limites du message, etc. Que le client ou le serveur s'aperçoive d'une fermeture de la connexion, le buffer sera vidé et le processus terminé. Le client a la possibilité de lancer un nouveau processus serveur en recontactant le serveur s'il ne souhaite pas se finir. Il peut également envisager de clore la connexion si un type de message non reconnu est reçu ; en effet, ceci est probablement le résultat de la perte de synchronisation des limite de messages.

Que la fin soit normale ou non, toute transaction ouverte est annulée, non pas validée. Si un client se déconnecte alors qu'une requête autre que SELECT est en cours de traitement, le serveur terminera probablement la requête avant de prendre connaissance de la déconnexion. Si la requête est en dehors d'un

bloc de transaction (séquence `BEGIN` ... `COMMIT`), il se peut que les résultats soient validés avant que la connexion ne soit reconnue.

### 42.2.9. Chiffrement SSL de session

Si PostgreSQL a été construit avec le support de SSL, les communications client/serveur peuvent être chiffrées en l'utilisant. Ce chiffrement assure la sécurité de la communication dans les environnements où des agresseurs pourraient capturer le trafic de la session.

Pour initier une connexion chiffrée par SSL, le client envoie initialement un message SSLRequest à la place d'un StartupMessage. Le serveur répond avec un seul octet contenant `S` ou `N` indiquant respectivement s'il souhaite ou non utiliser le SSL. Le client peut alors clore la connexion s'il n'est pas satisfait de la réponse. Pour continuer après un `S`, il faut échanger une poignée de main SSL (handshake) (non décrite ici car faisant partie de la spécification SSL) avec le serveur. En cas de succès, le StartupMessage habituel est envoyé. Dans ce cas, StartupMessage et toutes les données suivantes seront chiffrés avec SSL. Pour continuer après un `N`, il suffit d'envoyer le StartupMessage habituel et de continuer sans chiffrage.

Le client doit être préparé à gérer une réponse ErrorMessage à un SSLRequest émanant du serveur. Ceci ne peut survenir que si le serveur ne dispose pas du support de SSL. Dans ce cas, la connexion doit être fermée, mais le client peut choisir d'ouvrir une nouvelle connexion et procéder sans SSL.

Un SSLRequest initial peut également être utilisé dans une connexion en cours d'ouverture pour envoyer un message CancelRequest.

Alors que le protocole lui−même ne fournit pas au serveur de moyen de forcer le chiffrage SSL, l'administrateur peut configuer le serveur pour rejeter les sessions non chiffrées, ce qui est une autre façon de vérifier l'authentification.

## 42.3. Types de données des message

Cette section décrit les types de données basiques utilisés dans les messages.

Int*n*(*i*)
: Un entier sur *n* bits dans l'odre des octets réseau (octet le plus significatif en premier). Si *i* est spécifié, c'est exactement la valeur qui apparaîtra, sinon la valeur est variable, par exemple Int16, Int32(42).

Int*n*[*k*]
: Un tableau de *k* entiers sur *n* bits, tous dans l'ordre des octets réseau. La longueur *k* du tableau est toujours déterminée par un champ précédent du message, par exemple, Int16[M].

String(*s*)
: Une chaîne terminée par un octet nul (chaîne style C). Il n'y a pas de limitation sur la longueur des chaînes. Si *s* est spécifié, c'est la valeur exacte qui apparaîtra, sinon la valeur est variable. Par exemple, String("utilisateur").

> **Note :** *Il n'y a aucune limite prédéfinie* à la longueur d'une chaîne retournée par le serveur. Une bonne stratégie de codage de client consiste à utiliser un tampon dont la taille peut croître pour que tout ce qui tient en mémoire puisse être accepté. Si cela n'est pas faisable, il faudra lire la chaîne complète et supprimer les caractères qui ne tiennent pas dans le tampon de taille fixe.

Byte*n*(*c*)

Exactement *n* octets. Si la largeur *n* du champ n'est pas une constante, elle peut toujours être déterminée à partir d'un champ précédent du message. Si *c* est spécifié, c'est la valeur exacte. Par exemple, Byte2, Byte1('\n').

## 42.4. Formats de message

Cette section décrit le format détaillé de chaque message. Chaque message est marqué pour indiquer s'il peut être envoyé par un client (F pour *Frontend*), un serveur (B pour *Backend*) ou les deux (F & B). Bien que chaque message commence par son nombre d'octets, le format du message est défini de telle sorte que la fin du message puisse être trouvée sans ce nombre. Cela contribue à la vérification de la validité. Le message CopyData est une exception car il constitue une partie du flux de données ; le contenu d'un message CopyData individuel n'est, en soi, pas interprétable.

AuthenticationOk (B)

Byte1('R')
: Marqueur de demande d'authentification.

Int32(8)
: Taille du message en octets, y compris la taille elle−même.

Int32(0)
: L'authentification a réussi.

AuthenticationKerberosV4 (B)

Byte1('R')
: Marqueur de demande d'authentification.

Int32(8)
: Taille du message en octets, y compris la taille elle−même.

Int32(1)
: Une authentification Kerberos V4 est requise.

AuthenticationKerberosV5 (B)

Byte1('R')
: Marqueur de demande d'authentification.

Int32(8)
: Taille du message en octets, y compris la taille elle−même.

Int32(2)
: Une authentification Kerberos V5 est requise.

AuthenticationCleartextPassword (B)

Byte1('R')
: Marqueur de demande d'authentification.

Int32(8)
: Taille du message en octets, y compris la taille elle−même.

Int32(3)
: Un mot de passe en clair est requis.

AuthenticationCryptPassword (B)

Byte1('R')
: Marqueur de demande d'authentification.

Int32(10)
: Taille du message en octets, y compris la taille elle−même.

Int32(4)
: Un mot de passe chiffré à l'aide de crypt() est requis.

Byte2
: Composant (salt) à utiliser lors du chiffrement du mot de passe.

AuthenticationMD5Password (B)

Byte1('R')
: Marqueur de demande d'authentification.

Int32(12)
: Taille du message en octets, y compris la taille elle−même.

Int32(5)
: Un mot de passe chiffré par MD5 est requis.

Byte4
: Composant (salt) à utiliser lors du chiffrement du mot de passe.

AuthenticationSCMCredential (B)

Byte1('R')
: Marqueur de demande d'authentification.

Int32(8)
: Taille du message en octets, y compris la taille elle−même.

Int32(6)
: Un message d'accréditation SCM est requis.

BackendKeyData (B)

Byte1('K')
: Marqueur de clé d'annulation. Le client doit sauvegarder ces valeurs s'il souhaite initier des messages CancelRequest par la suite.

Int32(12)
: Taille du message en octets, y compris la taille elle−même.

Int32
: ID du processus du serveur concerné.

Int32
: Clé secrète du serveur concerné.

Bind (F)

Byte1('B')

Marqueur de commande Bind.

Int32

Taille du message en octets, y compris la taille elle−même.

String

Nom du portail de destination (une chaîne vide sélectionne le portail non−nommé).

String

Nom de l'instruction source préparée (une chaîne vide sélectionne l'instruction préparée non−nommée).

Int16

Nombre de codes de format de paramètres qui suivent (notés *C* ci−dessous). Peut valoir zéro pour indiquer qu'il n'y a aucun paramètre ou que tous les paramètres utilisent le format par défaut (texte) ; ou un, auquel cas le code de format spécifié est appliqué à tous les paramètres ; il peut aussi être égal au nombre courant de paramètres.

Int16[*C*]

Codes de format des paramètres. Tous doivent valoir zéro (texte) ou un (binaire).

Int16

Nombre de valeurs de paramètres qui suivent (peut valoir zéro). Cela doit correspondre au nombre de paramètres nécessaires à la requête.

Puis, le couple de champs suivant apparaît pour chaque paramètre :

Int32

Taille de la valeur du paramètre, en octets (ce nombre n'inclut pas la longueur elle−même). Peut valoir zéro. Traité comme un cas spécial, −1 indique une valeur de paramètre NULL. Aucun octet de valeur ne suit le cas NULL.

Byte*n*

Valeur du paramètre, dans le format indiqué par le code de format associé. *n* est la longueur ci−dessus.

Après le dernier paramètre, les champs suivants apparaissent :

Int16

Nombre de codes de format des colonnes de résultat qui suivent (noté *R* ci−dessous). Peut valoir zéro pour indiquer qu'il n'y a pas de colonnes de résultat ou que les colonnes de résultat utilisent le format par défaut (texte) ; ou une, auquel cas le code de format spécifié est appliqué à toutes les colonnes de résultat (s'il y en a) ; il peut aussi être égal au nombre de colonnes de résultat de la requête.

Int16[*R*]

Codes de format des colonnes de résultat. Tous doivent valoir zéro (texte) ou un (binaire).

BindComplete (B)

Byte1('2')

Indicateur de Bind complet.

Int32(4)

Taille du message en octets, y compris la taille elle−même.

CancelRequest (F)

Int32(16)

Taille du message en octets, y compris la taille elle−même.

Int32(80877102)

Code d'annulation de la requête. La valeur est choisie pour contenir `1234` dans les 16 bits les plus significatifs et `5678` dans les 16 bits les moins significatifs. (Pour éviter toute confusion, ce code ne doit pas être le même qu'un numéro de version de protocole.)

Int32
ID du processus du serveur cible.

Int32
Clé secrète du serveur cible.

Close (F)

Byte1('C')
Marqueur de commande Close.

Int32
Taille du message en octets, y compris la taille elle−même.

Byte1
'`S`' pour fermer une instruction préparée ; ou '`P`' pour fermer un portail.

String
Nom de l'instruction préparée ou du portail à fermer (une chaîne vide sélectionne l'instruction préparée ou le portail non−nommé(e)).

CloseComplete (B)

Byte1('3')
Indicateur de complétude de Close.

Int32(4)
Taille du message en octets, y compris la taille elle−même.

CommandComplete (B)

Byte1('C')
Marqueur de réponse de complétude de commande.

Int32
Taille du message en octets, y compris la taille elle−même.

String
Balise de la commande. Mot simple identifiant la commande SQL terminée.

Pour une commande `INSERT`, la balise est `INSERT` *`oid lignes`* où *`lignes`* est le nombre de lignes insérées. *`oid`* est l'ID de l'objet de la ligne insérée si *`lignes`* vaut 1 et que la table cible a des OID ; sinon *`oid`* vaut 0.

Pour une commande `DELETE`, la balise est `DELETE` *`lignes`* où *`lignes`* est le nombre de lignes supprimées.

Pour une commande `UPDATE`, la balise est `UPDATE` *`lignes`* où *`lignes`* est le nombre de lignes mises à jour.

Pour une commande `MOVE`, la balise est `MOVE` *`lignes`* où *`lignes`* est le nombre de lignes de déplacement du curseur.

Pour une commande `FETCH`, la balise est `FETCH` *`lignes`* où *`lignes`* est le nombre de lignes récupérées à partir du curseur.

CopyData (F & B)

Byte1('d')
: Marqueur de données de COPY.

Int32
: Taille du message en octets, y compris la taille elle−même.

Byte*n*
: Données formant une partie d'un flux de données `COPY`. Les messages envoyés depuis le serveur correspondront toujours à des lignes uniques de données, mais les messages envoyés par les clients peuvent diviser le flux de données de façon arbitraire.

CopyDone (F & B)

Byte1('c')
: Indicateur de fin de COPY.

Int32(4)
: Taille du message en octets, y compris la taille elle−même.

CopyFail (F)

Byte1('f')
: Indicateur d'échec de COPY.

Int32
: Taille du message en octets, y compris la taille elle−même.

String
: Message d'erreur rapportant la cause d'un échec.

CopyInResponse (B)

Byte1('G')
: Marqueur de réponse de Start Copy In. Le client doit alors envoyer des données de copie (s'il n'est pas à cela, il enverra un message CopyFail).

Int32
: Taille du message en octets, y compris la taille elle−même.

Int8
: 0 indique que le format de copie complet est textuel (lignes séparées par des retours chariot, colonnes séparées par des caractères de séparation, etc). 1 indique que le format de copie complet est binaire (similaire au format DataRow). Voir *COPY* pour plus d'informations.

Int16
: Nombre de colonnes dans les données à copier (noté *N* ci−dessous).

Int16[*N*]
: Codes de format à utiliser pour chaque colonne. Chacun doit valoir zéro (texte) ou un (binaire). Tous doivent valoir zéro si le format de copie complet est de type texte.

CopyOutResponse (B)

Byte1('H')
: Marqueur de réponse Start Copy Out. Ce message sera suivi de données copy−out.

Int32
: Taille du message en octets, y compris la taille elle−même.

Int8
: 0 indique que le format de copie complet est textuel (lignes séparées par des retours chariots, colonnes séparées par des caractères séparateur, etc). 1 indique que le format de copie complet est binaire (similaire au format DataRow). Voir *COPY* pour plus d'informations.

Int16
: Nombre de colonnes de données à copier (noté *N* ci−dessous).

Int16[*N*]
: Codes de format à utiliser pour chaque colonne. Chaque code doit valoir zéro (texte) ou un (binaire). Tous doivent valoir zéro si le format de copie complet est de type texte.

DataRow (B)

Byte1('D')
: Marqueur de ligne de données.

Int32
: Taille du message en octets, y compris la taille elle−même.

Int16
: Nombre de valeurs de colonnes qui suivent (peut valoir zéro).

Apparaît ensuite le couple de champs suivant, pour chaque colonne :

Int32
: Longueur de la valeur de la colonne, en octets (ce nombre n'inclut pas la longueur elle−même). Peut valoir zéro. Traité comme un cas spécial, −1 indique une valeur NULL de colonne. Aucun octet de valeur ne suit le cas NULL.

Byte*n*
: Valeur de la colonne dans le format indiqué par le code de format associé. *n* est la longueur ci−dessus.

Describe (F)

Byte1('D')
: Marqueur de commande Describe.

Int32
: Taille du message en octets, y compris la taille elle−même.

Byte1
: 'S' pour décrire une instruction préparée ; ou 'P' pour décrire un portail.

String
: Nom de l'instruction préparée ou du portail à décrire (une chaîne vide sélectionne l'instruction préparée ou le portail non−nommé(e)).

EmptyQueryResponse (B)

Byte1('I')
: Marqueur de réponse à une chaîne de requête vide. (C'est un substitut de CommandComplete.)

Int32(4)

Taille du message en octets, y compris la taille elle−même.

ErrorResponse (B)

Byte1('E')
Marqueur d'erreur.
Int32
Taille du message en octets, y compris la taille elle−même.
Le corps du message est constitué d'un ou plusieurs champs identifié(s), suivi(s) d'un octet nul comme terminateur. L'ordre des champs n'est pas fixé. Pour chaque champ, on trouve les informations suivantes :

Byte1
Code identifiant le type de champ ; s'il vaut zéro, c'est la fin du message et aucune chaîne ne suit. Les types de champs définis sont listés dans Section 42.5. De nouveaux types de champs pourraient être ajoutés dans le futur, les clients doivent donc ignorer silencieusement les types non reconnus.
String
Valeur du champ.

Execute (F)

Byte1('E')
Marqueur de commande Execute.
Int32
Taille du message en octets, y compris la taille elle−même.
String
Nom du portail à exécuter (une chaîne vide sélectionne le portail non−nommé).
Int32
Nombre maximum de lignes à retourner si le portail contient une requête retournant des lignes (ignoré sinon). Zéro signifie << aucune limite >>.

Flush (F)

Byte1('H')
Marqueur de commande Flush.
Int32(4)
Taille du message en octets, y compris la taille elle−même.

FunctionCall (F)

Byte1('F')
Marqueur d'appel de fonction.
Int32
Taille du message en octets, y compris la taille elle−même.
Int32
Spécifie l'ID de l'objet représentant la fonction à appeler.
Int16

Nombre de codes de format de l'argument qui suivent (noté *C* ci−dessous). Cela peut être zéro pour indiquer qu'il n'y a pas d'arguments ou que tous les arguments utilisent le format par défaut (texte) ; un, auquel cas le code de format est appliqué à tous les arguments ; il peut aussi être égal au nombre réel d'arguments.

Int16[*C*]

Les codes de format d'argument. Chacun doit valoir zéro (texte) ou un (binaire).

Int16

Nombre d'arguments fournis à la fonction.

Apparaît ensuite, pour chaque argument, le couple de champs suivant :

Int32

Longueur de la valeur de l'argument, en octets (ce nombre n'inclut pas la longueur elle−même). Peut valoir zéro. Traité comme un cas spécial, −1 indique une valeur NULL de l'argument. Aucun octet de valeur ne suit le cas NULL.

Byte*n*

Valeur de l'argument dans le format indiqué par le code de format associé. *n* est la longueur ci−dessus.

Après le dernier argument, le champ suivant apparaît :

Int16

Code du format du résultat de la fonction. Doit valoir zéro (texte) ou un (binaire).

FunctionCallResponse (B)

Byte1('V')

Marqueur de résultat d'un appel de fonction.

Int32

Taille du message en octets, y compris la taille elle−même.

Int32

Longueur de la valeur du résultat de la fonction, en octets (ce nombre n'inclut pas la longueur elle−même). Peut valoir zéro. Traité comme un cas spécial, −1 indique un résultat de fonction NULL. Aucun octet de valeur ne suit le cas NULL.

Byte*n*

Valeur du résultat de la fonction, dans le format indiqué par le code de format associé. *n* est la longueur ci−dessus.

NoData (B)

Byte1('n')

Indicateur d'absence de données.

Int32(4)

Taille du message en octets, y compris la taille elle−même.

NoticeResponse (B)

Byte1('N')

Marqueur d'avertissement.

Int32

Taille du message en octets, y compris la taille elle−même.

Le corps du message est constitué d'un ou plusieurs champs identifié(s), suivi(s) d'un octet zéro comme terminateur. L'ordre des champs n'est pas fixé. Pour chaque champ, on trouve les informations suivantes :

Byte1
Code identifiant le type de champ ; s'il vaut zéro, c'est la fin du message et aucune chaîne ne suit. Les types de champs déjà définis sont listés dans Section 42.5. De nouveaux types de champs pourraient être ajoutés dans le futur, les clients doivent donc ignorer silencieusement les champs de type non reconnu.

String
Valeur du champ.

NotificationResponse (B)

Byte1('A')
Marqueur de réponse de notification.

Int32
Taille du message en octets, y compris la taille elle−même.

Int32
ID du processus serveur ayant procédé à la notification.

String
Nom de la condition à l'origine de la notification.

String
Information supplémentaire provenant du processus à l'origine de la notification. (Cette fonctionnalité n'est , à ce jour, pas implantée, le champ est donc toujours constitué d'une chaîne vide.)

ParameterDescription (B)

Byte1('t')
Marqueur de description de paramètre.

Int32
Taille du message en octets, y compris la taille elle−même.

Int16
Nombre de paramètres utilisé par l'instruction (peut valoir zéro).

Pour chaque paramètre, suivent :

Int32
ID de l'objet du type de données du paramètre.

ParameterStatus (B)

Byte1('S')
Marqueur de rapport d'état de paramètre d'exécution.

Int32
Taille du message en octets, y compris la taille elle−même.

String
Nom du paramètre d'exécution dont le rapport est en cours.

String
Valeur actuelle du paramètre.

Parse (F)

Byte1('P')
Marqueur de commande Parse.
Int32
Taille du message en octets, y compris la taille elle−même.
String
Nom de l'instruction préparée de destination (une chaîne vide sélectionne l'instruction préparée non−nommée).
String
Chaîne de requête à analyser.
Int16
Nombre de types de données de paramètre spécifiés (peut valoir zéro). Ce n'est pas une indication du nombre de paramètres pouvant apparaître dans la chaîne de requête, mais simplement le nombre de paramètres pour lesquels le client veut pré−spécifier les types.
Pour chaque paramètre, on trouve ensuite :

Int32
ID de l'objet du type de données du paramètre. la valeur zéro équivaut à ne pas spécifier le type.

ParseComplete (B)

Byte1('1')
Indicateur de fin de Parse.
Int32(4)
Taille du message en octets, y compris la taille elle−même.

PasswordMessage (F)

Byte1('p')
Marqueur de réponse de mot de passe.
Int32
Taille du message en octets, y compris la taille elle−même.
String
Mot de passe (chiffré à la demande).

PortalSuspended (B)

Byte1('s')
Indicateur de suspension du portail. Apparaît seulement si la limite du nombre de lignes d'un message Execute a été atteint.
Int32(4)
Taille du message en octets, y compris la taille elle−même.

Query (F)

Byte1('Q')
: Marqueur de requête simple.

Int32
: Taille du message en octets, y compris la taille elle−même.

String
: La chaîne de requête elle−même.

ReadyForQuery (B)

Byte1('Z')
: Identifie le type du message. ReadyForQuery est envoyé à chaque fois que le serveur est prêt pour un nouveau cycle de requêtes.

Int32(5)
: Taille du message en octets, y compris la taille elle−même.

Byte1
: Indicateur de l'état transactionnel du serveur. Les valeurs possibles sont '`I`' s'il est en pause (en dehors d'un bloc de transaction) ; '`T`' s'il est dans un bloc de transaction ; ou '`E`' s'il est dans un bloc de transaction échouée (les requêtes seront réjetées jusqu'à la fin du bloc).

RowDescription (B)

Byte1('T')
: Marqueur de description de ligne.

Int32
: Taille du message en octets, y compris la taille elle−même.

Int16
: Nombre de champs dans une ligne (peut valoir zéro).

On trouve, ensuite, pour chaque champ :

String
: Nom du champ.

Int32
: Si le champ peut être identifié comme colonne d'une table spécifique, l'ID de l'objet de la table ; sinon zéro.

Int16
: Si le champ peut être identifié comme colonne d'une table spécifique, le numéro d'attribut de la colonne ; sinon zéro.

Int32
: ID de l'objet du type de données du champ.

Int16
: Taille du type de données (voir `pg_type.typlen`). Les valeurs négatives indiquent des types de largeur variable.

Int32
: Modificateur de type (voir `pg_attribute.atttypmod`). La signification du modificateur est spécifique au type.

Int16
: Code de format utilisé pour le champ. Zéro (texte) ou un (binaire), à l'heure actuelle. Dans un RowDescription retourné par la variante de l'instruction de Describe, le code du format n'est pas encore connu et vaudra toujours zéro.

SSLRequest (F)

Int32(8)
Taille du message en octets, y compris la taille elle−même.
Int32(80877103)
Code de requête SSL. La valeur est choisie pour contenir `1234` dans les 16 bits les plus significatifs, et `5679` dans les 16 bits les moins significatifs (pour éviter toute confusion, ce code ne doit pas être le même que celui d'un numéro de version de protocole).

StartupMessage (F)

Int32
Taille du message en octets, y compris la taille elle−même.
Int32(196608)
Numéro de version du protocole. Les 16 bits les plus significatifs représentent le numéro de version majeure (3 pour le protocole décrit ici). Les 16 bits les moins significatifs représentent le numéro de version mineure (0 pour le protocole décrit ici).
Le numéro de version du protocole est suivi par un ou plusieurs couple(s) nom de paramètre et chaîne de valeur. Un octet zéro est requis comme terminateur après le dernier couple nom/valeur. L'ordre des paramètres n'est pas fixé. Le paramètre `user` est requis, les autres sont optionnels. Chaque paramètre est spécifié de la façon suivante :

String
Nom du paramètre. Les noms actuellement reconnus sont :

`user`
Nom de l'utilisateur de base de données sous lequel se connecter. Requis ; il n'y a pas de valeur par défaut.
`database`
Base de données à laquelle se connecter. Par défaut le nom de l'utilisateur.
`options`
Arguments en ligne de commande pour le serveur. (Rendu obsolète par l'utilisation de paramètres individuels d'exécution.)
En plus de ce qui précède, tout paramètre d'exécution pouvant être initialisé au démarrage du serveur peut être listé. Ces paramètres seront appliqués au démarrage du serveur (après analyse des options en ligne de commande, s'il y en a). Leurs valeurs agiront comme valeurs de session par défaut.
String
Valeur du paramètre.

Sync (F)

Byte1('S')
Marqueur de commande Sync.
Int32(4)
Taille du message en octets, y compris la taille elle−même.

Terminate (F)

Byte1('X')
: Marqueur de fin.

Int32(4)
: Taille du message en octets, y compris la taille elle−même.

---

# 42.5. Champs des messages d'erreur et d'avertissement

Cette section décrit les champs qui peuvent apparaître dans les messages ErrorResponse et NoticeResponse. Chaque type de champ a un motif d'identification codé sur un octet. Tout type de champ donné doit apparaître au plus une fois par message.

`S`
: Gravité (Severity) : le contenu du champ peut être `ERROR`, `FATAL` ou `PANIC` dans un message d'erreur, `WARNING`, `NOTICE`, `DEBUG`, `INFO` ou `LOG` dans un message d'avertissement, ou la traduction régionale de l'un d'eux. Toujours présent.

`C`
: Code : code SQLSTATE de l'erreur (voir Annexe A). Non internationalisable. Toujours présent.

`M`
: Message : premier message d'erreur, en clair. Doit être court et précis (typiquement une ligne). Toujours présent.

`D`
: Détail : deuxième message d'erreur, optionnel, apportant des informations supplémentaires sur le problème. Peut être sur plusieurs lignes.

`H`
: Astuce (Hint) : suggestion optionnelle de résolution du problème. Différent de Détail parce qu'il offre un conseil (potentiellement inapproprié) plutôt que des faits réels. Peut être sur plusieurs lignes.

`P`
: Position : valeur du champ, entier décimal ASCII indiquant un curseur sur la position de l'erreur dans la chaîne de requête originale. Le premier caractère a l'index 1. Les positions sont mesurées en caractères, non pas en octets.

`p`
: Position interne : ceci est défini de la même façon que le champ `P` mais c'est utilisé quand la position du curseur se réfère à une commande générée en interne plutôt qu'une soumise par le client. Le champ `q` apparaîtra toujours quand ce champ apparaît.

`q`
: Requête interne : le texte d'une commande générée en interne et qui a échoué. Ceci pourrait être, par exemple, une requête SQL lancée par une fonction PL/pgSQL.

`W`
: Où (Where) : indication du contexte dans lequel l'erreur est survenue. Inclut, actuellement, une trace de la pile des appels des fonctions PL actives. Cette trace comprend une entrée par ligne, la plus récente en premier.

`F`
: Fichier (File) : nom du fichier de code source comportant l'erreur.

`L`
: Ligne (Line) : numéro de ligne dans le fichier de code source comportant l'erreur.

`R`

Routine : nom de la routine dans le code source comportant l'erreur.

Le client est responsable du formatage adapté à ses besoins des informations affichées ; en particulier par l'ajout de retours chariots sur les lignes longues, si cela s'avérait nécessaire. Les caractères de retour chariot apparaissant dans les champs de messages d'erreur devraient être traités comme des changements de paragraphes, non comme des changements de lignes.

## 42.6. Résumé des modifications depuis le protocole 2.0

Cette section fournit une liste rapide des modifications à l'attention des développeurs essayant d'adapter au protocole 3.0 des bibliothèques client existantes.

Le paquet de démarrage initial utilise un format flexible de liste de chaînes au lieu d'un format fixe. Les valeurs de session par défaut des paramètres d'exécution peuvent même être spécifiés directement dans le paquet de démarrage. (En fait, cela était déjà possible en utilisant le champ `options` ; mais étant donné la largeur limitée d'`options` et l'impossibilité de mettre entre guillemets les espaces fines dans les valeurs, ce n'était pas une technique très sûre.)

Tous les messages possèdent désormais une indication de longueur qui suit immédiatement l'octet du type de message (sauf pour les paquets de démarrage qui n'ont pas d'octet de type). PasswordMessage possède à présent un octet de type.

Les messages ErrorResponse et NoticeResponse ('`E`' et '`N`') contiennent maintenant plusieurs champs, à partir desquels le code client peut assembler un message d'erreur fonction du niveau de verbiage désiré. Des champs individuels ne se termineront plus par un retour chariot alors que la chaîne seule envoyée dans l'ancien protocole le faisait systématiquement.

Le message ReadyForQuery ('`Z`') inclut un indicateur d'état de la transaction.

La distinction entre les types de messages BinaryRow et DataRow est supprimée ; le type de message DataRow seul sert à retourner les données dans tous les formats. La disposition de DataRow a changé pour faciliter son analyse. La représentation des valeurs binaires a également été modifiée : elle n'est plus liée directement à la représentation interne du serveur.

Il existe un nouveau sous−protocole pour les << requêtes étendues >> qui ajoute les types de messages client Parse, Bind, Execute, Describe, Close, Flush et Sync et les types de messages serveur ParseComplete, BindComplete, PortalSuspended, ParameterDescription, NoData et CloseComplete. Les clients existants ne sont pas directement concernés par ce sous−protocole, mais son utilisation apportera des améliorations en terme de performances et de fonctionnalités.

Les données de COPY sont désormais encapsulées dans des messages CopyData et CopyDone. Il y a une façon bien définie de réparer les erreurs lors du COPY. La dernière ligne spéciale << `\.` >> n'est plus nécessaire et n'est pas envoyée lors de COPY OUT. (Elle est toujours reconnue comme un indicateur de fin lors du COPY IN mais son utilisation est obsolète. Elle sera éventuellement supprimée.) Le COPY binaire est supporté. Les messages CopyInResponse et CopyOutResponse incluent les champs indiquant le nombre de colonnes et le format de chaque colonne.

La disposition des messages FunctionCall et FunctionCallResponse a changé. FunctionCall supporte à présent le passage aux fonctions d'arguments NULL. Il peut aussi gérer le passage de paramètres et la récupération de résultats aux formats texte et binaire. Il n'y a plus aucune raison de considérer FunctionCall comme une faille

potentielle de sécurité car il n'offre plus d'accès direct aux représentations internes des données du serveur.

Le serveur envoie des messages ParameterStatus ('S') lors de l'initialisation de la connexion pour tous les paramètres qu'il considère intéressants pour la bibliothèque client. En conséquence, un message ParameterStatus est envoyé à chaque fois que la valeur active d'un de ces paramètres change.

Le message RowDescription ('T') contient les nouveaux champs OID de table et de numéro de colonne pour chaque colonne de la ligne décrite. Il affiche aussi le code de format pour chaque colonne.

Le message CursorResponse ('P') n'est plus engendré par le serveur.

Le message NotificationResponse ('A') a un champ de type chaîne supplémentaire, actuellement vide mais qui pourrait à terme transporter des données supplémentaires engendrées par l'émetteur de l'événement NOTIFY.

Le message EmptyQueryResponse ('I') nécessitait un paramètre chaîne vide ; ce n'est plus le cas.

---

# Chapitre 43. Conventions de codage pour PostgreSQL

## 43.1. Formatage

Le formatage du code source utilise un espacement de quatre colonnes pour les tabulations, avec la préservation de celles–ci (c'est–à–dire que les tabulations ne sont pas converties en espaces). Chaque niveau logique d'indentation est une tabulation supplémentaire. Les règles de disposition (positionnement des parenthèses, etc) suivent les conventions BSD.

Bien que les correctifs (patchs) soumis ne sont absolument pas tenus de suivre ces règles de formatage, il est recommandé de le faire. Votre code sera passé dans pgindent, donc il n'y a pas d'intérêts à ce qu'il soit joli grâce à d'autres ensembles de conventions de formatage.

Pour Emacs, ajoutez ce qui suit (ou quelque chose de semblable) à votre fichier d'initialisation `~/.emacs` :

```
;; vérification des fichiers avec un chemin contenant "postgres" ou "pgsql"
(setq auto-mode-alist
  (cons '("\\(postgres\\|pgsql\\).*\\.[ch]\\'" . pgsql-c-mode)
        auto-mode-alist))
(setq auto-mode-alist
  (cons '("\\(postgres\\|pgsql\\).*\\.cc\\'" . pgsql-c-mode)
        auto-mode-alist))

(defun pgsql-c-mode ()
  ;; Configuration du formatage pour le code C de PostgreSQL
  (interactive)
  (c-mode)
  (setq-default tab-width 4)
  (c-set-style "bsd")             ; positionner c-basic-offset à 4, plus d'autres choses
  (c-set-offset 'case-label '+)   ; ajuste l'indentation de la casse pour correspondre aux usages
  (setq indent-tabs-mode t))      ; pour être sûr que nous gardons les tabulations en indentant
```

Pour vi, votre `~/.vimrc` ou le fichier équivalent devrait contenir ce qui suit :

```
set tabstop=4
```

ou d'une manière équivalente, dans vi, essayez

```
:set ts=4
```

Les outils de parcours de texte more et less peuvent être appelés de la manière suivante

```
more -x4
less -x4
```

pour qu'ils affichent correctement les tabulations.

---

# 43.2. Reporter les erreurs dans le serveur

Les messages d'erreurs, d'alertes et de traces générés à l'intérieur du code du serveur doivent être créés en utilisant `ereport` ou son ancien cousin `elog`. L'utilisation de cette fonction est assez complexe pour que cela requiert quelques explications.

Il y a deux éléments requis pour chaque message : un niveau de sévérité (allant de `DEBUG` à `PANIC`) et un message texte primaire. De plus, il y a des éléments optionnels, le plus commun d'entre eux est le code d'identifiant de l'erreur qui suit les conventions SQLSTATE des spécifications SQL. `ereport` en elle−même n'est qu'une fonction shell qui existe principalement pour des convenances syntaxiques faisant ressembler la génération de messages à l'appel d'une fonction dans un code source C. Le seul paramètre directement accepté par `ereport` est le niveau de sévérité. Le message texte primaire et les autres éléments de messages optionnels sont générés en appelant des fonctions auxiliaires, comme `errmsg`, à l'intérieur de l'appel à `ereport`.

Un appel typique à `ereport` peut ressembler à ceci :

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

Ceci met le niveau de sévérité de l'erreur à `ERROR` (une erreur banale). L'appel à `errcode` spécifie le code d'erreur SQLSTATE en utilisant une macro définie dans `src/include/utils/errcodes.h`. L'appel à `errmsg` fournit le message texte primaire. Notez l'ensemble supplémentaire de parenthèses entourant les appels aux fonctions auxiliaires (cela est ennuyeux mais syntaxiquement nécessaire).

Voici un exemple plus complexe :

```
ereport(ERROR,
        (errcode(ERRCODE_AMBIGUOUS_FUNCTION),
         errmsg("function %s is not unique",
                func_signature_string(funcname, nargs,
                                      actual_arg_types)),
         errhint("Unable to choose a best candidate function. "
                 "You may need to add explicit typecasts.")));
```

Ceci illustre l'utilisation des codes de formatage pour intégrer des valeurs d'exécution dans un message texte. Un message << conseil >>, optionnel, est également fourni.

Les routines auxiliaires disponibles pour `ereport` sont :

- `errcode(sqlerrcode)` spécifie le code SQLSTATE de l'identifiant de l'erreur pour la condition. Si cette routine n'est pas appelée, par défaut, l'identifiant de l'erreur sera `ERRCODE_INTERNAL_ERROR` quand le niveau de sévérité de l'erreur est `ERROR` ou plus haut, `ERRCODE_WARNING` quand le niveau d'erreur est `WARNING` et `ERRCODE_SUCCESSFUL_COMPLETION` pour `NOTICE` et inférieur. Bien que ces valeurs par défaut soient souvent commodes, demandez−vous toujours si elles sont appropriées avant d'omettre l'appel à `errcode()`.
- `errmsg(const char *msg, ...)` spécifie le message texte primaire de l'erreur et les possibles valeurs d'exécutions à insérer dedans. Les insertions sont spécifiées par les codes de formatage dans le style `sprintf`. En plus des codes de formatage standards acceptés par `sprintf`,

le code %m peut être utilisé pour insérer le message d'erreur retourné par strerror pour la valeur courante de errno. [7] %m ne nécessite aucune entrée correspondante dans la liste de paramètres pour errmsg. Notez que la chaîne de caractères du message sera passée à travers gettext pour une possible adaptation linguistique avant que les codes de formatage ne soient exécutés.

- errmsg_internal(const char *msg, ...) fait la même chose que errmsg à l'exception que la chaîne de caractères du message ne sera pas inclue dans le dictionnaire de messages d'internationalisation. Cela devrait être utilisé pour les cas qui << ne peuvent pas arriver >> et pour lesquels il n'est probablement pas intéressant de déployer un effort de traduction.
- errdetail(const char *msg, ...) fournit un message << détail >> optionnel ; cela est utilisé quand il y a des informations supplémentaires qu'il semble inadéquat de mettre dans le message primaire. La chaîne de caractères du message est traitée de la même manière que celle de errmsg.
- errhint(const char *msg, ...) fournit un message << conseil >> optionnel ; cela est utilisé pour offrir des suggestions sur la façon de régler un problème, par opposition aux détails effectifs au sujet de ce qui a mal tourné. La chaîne de caractères du message est traitée de la même manière que celle de errmsg.
- errcontext(const char *msg, ...) n'est normalement pas appelée directement depuis un site de message de ereport mais plutôt elle est utilisée dans les fonctions de rappels error_context_stack pour fournir des informations à propos du contexte dans lequel une erreur s'est produite, comme les endroits courants dans la fonction PL. La chaîne de caractères du message est traitée de la même manière que celle de errmsg. À l'inverse des autres fonctions auxiliaires, celle–ci peut être appelée plus d'une fois dans un appel de ereport ; les chaînes successives ainsi fournies sont concaténées et séparées pas des caractères d'interlignes (NL).
- errposition(int cursorpos) spécifie l'endroit textuel d'une erreur dans la chaîne de caractères de la requête. Actuellement, c'est seulement utile pour les erreurs détectées dans les phases d'analyses lexicales et syntaxiques du traitement de la requête.
- errcode_for_file_access() est une fonction commode qui sélectionne l'identifiant d'erreur SQLSTATE approprié pour une défaillance dans l'appel système relatif à l'accès d'un fichier. Elle utilise le errno sauvegardé pour déterminer quel code d'erreur générer. Habituellement cela devrait être utilisé en combinaison avec %m dans le texte du message d'erreur primaire.
- errcode_for_socket_access() est une fonction commode qui sélectionne l'identifiant d'erreur SQLSTATE approprié pour une défaillance dans l'appel système relatif à une socket.

Il y a une plus ancienne fonction nommée elog, qui est toujours largement utilisée. Un appel à elog

```
elog(niveau, "chaine format", ...);
```

est strictement équivalement à

```
ereport(level, (errmsg_internal("chaine format", ...)));
```

Notez que le code d'erreur SQLSTATE est toujours par défaut et que la chaîne de caractères du message n'est pas inclue dans le dictionnaire de messages d'internationalisation. Par conséquent, elog devrait être utilisé seulement pour les erreurs internes et l'enregistrement de trace de débogage de bas niveau. N'importe quel message qui est susceptible d'être intéressant pour les utilisateurs ordinaires devrait passer par ereport. Néanmoins, il y a assez de contrôles d'erreurs internes qui << ne peuvent pas arriver >> dans le système, que elog est toujours largement utilisé ; pour ces messages, cela est préféré à cause de sa simplicité d'écriture.

Des conseils sur l'écriture de bons messages d'erreurs peuvent être trouvés dans la Section 43.3.

# 43.3. Guide de style des messages d'erreurs

Ce guide de style est fournit dans l'espoir de maintenir une cohérence et un style facile à comprendre dans tous les messages générés par PostgreSQL.

## 43.3.1. Ce qui va où

Le message primaire devrait être court, factuel et éviter les références aux détails d'exécution comme le nom de fonction spécifique. << Court >> veut dire << devrait tenir sur une ligne dans des conditions normales >>. Utilisez un message détail si nécessaire pour garder le message primaire court ou si vous sentez le besoin de mentionner les détails de l'implémentation comme un appel système particulier qui échoue. Les messages primaires et détails doivent être factuels. Utilisez un message conseil pour les suggestions à propos de quoi faire pour fixer le problème, spécialement si la suggestion ne pourrait pas toujours être applicable.

Par exemple, au lieu de

```
IpcMemoryCreate: shmget(clé=%d, taille=%u, 0%o) a échoué : %m
(plus un long supplément qui est basiquement un conseil)
```

écrivez

```
Primaire:    Ne peut pas créer un ségment en mémoire partagée : %m
Détail:      L'appel système qui a échoué était shmget(key=%d, size=%u, 0%o).
Astuce:     Le supplément
```

Raisonnement : garder le message primaire court aide à le garder au point et laisse les clients présenter un espace à l'écran sur la supposition qu'une ligne est suffisante pour les messages d'erreurs. Les messages détails et conseils peuvent être relégués à un mode verbeux ou peut−être dans une fenêtre pop−up détaillant l'erreur. De plus, les détails et les conseils devront normalement être supprimés des traces du serveur pour gagner de l'espace. La référence aux détails d'implémentation est à éviter puisque les utilisateurs n'en connaissent de toute façon pas les détails.

## 43.3.2. Formatage

N'émettez pas d'hypothèses spécifiques à propos du formatage dans les messages textes. Attendez−vous à ce que les clients et les traces du serveur enveloppent les lignes pour correspondre à leurs propres besoins. Dans les messages longs, les caractères d'interlignes (\n) peuvent être utilisés pour indiquer les coupures suggérées d'un paragraphe. Ne terminez pas un message avec un caractère d'interlignes. N'utilisez pas des tabulations ou d'autres caractères de formatage. (Dans les affichages des contextes d'erreurs, les caractères d'interlignes sont automatiquement ajoutés pour séparer les niveaux d'un contexte comme dans les appels aux fonctions.)

Raisonnement : les messages ne sont pas nécessairement affichés dans un affichage de type terminal. Dans les interfaces graphiques ou les navigateurs, ces instructions de formatage sont, au mieux, ignorées.

## 43.3.3. Guillemets

Les textes en anglais devraient utiliser des guillemets doubles quand la mise entre guillemets est appropriée. Les textes dans les autres langues devraient uniformément employer un genre de guillemets qui est conforme aux coutumes de publication et à la sortie visuelle des autres programmes.

Raisonnement : le choix des guillemets doubles sur celui des guillemets simples est quelque peu arbitraire mais tend à être l'utilisation préférée. Certains ont suggéré de choisir le type de guillemets en fonction du type d'objets des conventions SQL (notamment, les chaînes de caractères entre guillemets simples, les identifiants entre guillemets doubles). Mais ceci est un point technique à l'intérieur du langage avec lequel beaucoup d'utilisateurs ne sont pas familiers ; les conventions SQL ne prennent pas en compte les autres genres de termes entre guillemets, ne sont pas traduites dans d'autres langues et manquent un peu de sens aussi.

## 43.3.4. Utilisation des guillemets

Utilisez toujours les guillemets pour délimiter les noms de fichiers, les identifiants fournis par les utilisateurs et les autres variables qui peuvent contenir des mots. Ne les utilisez pas pour marquer des variables qui ne contiennent pas de mots (par exemple, les noms d'opérateurs).

Il y a des fonctions au niveau du serveur qui vont, au besoin, mettre entre guillemets leur propre flux de sortie (par exemple, `format_type_be()`). Ne mettez pas de guillemets supplémentaires autour du flux de sortie de ce genre de fonctions.

Raisonnement : les objets peuvent avoir un nom qui crée une ambiguïté une fois incorporé dans un message. Soyez prudent en indiquant où un nom commence et fini. Mais n'encombrez pas les messages avec des guillemets qui ne sont pas nécessaires ou qui sont dupliqués.

## 43.3.5. Grammaire et ponctuation

Les régles sont différentes pour les messages d'erreurs primaires et pour les messages détails/conseils :

Messages d'erreurs primaires : ne mettez pas en majuscule la première lettre. Ne terminez pas un message avec un point. Ne pensez même pas à finir un message avec un point d'exclamation.

Messages détails et conseils : utilisez des phrases complètes et toutes terminées par des points. Mettez en majuscule le premier mot des phrases.

Raisonnement : éviter la ponctuation rend plus facile, pour les applications clientes, l'intégration du message dans des contextes grammaticaux variés. Souvent, les messages primaires ne sont de toute façon pas des phrases complètes. (Et s'ils sont assez longs pour être sur plusieurs phrases, ils devraient être divisés en une partie primaire et une partie détail.) Cependant, les messages détails et conseils sont longs et peuvent avoir besoin d'inclure de nombreuses phrases. Pour la cohérence, ils devraient suivre le style des phrases complètes même s'il y a seulement une phrase.

## 43.3.6. Majuscule contre minuscule

Utilisez les minuscules pour les mots d'un message, inclus la première lettre d'un message d'erreur primaire. Utilisez les majuscules pour les commandes et les mots−clé SQL s'ils apparaissent dans le message.

Raisonnement : il est plus facile de rendre toutes les choses plus cohérentes au regard de cette façon, puisque certains messages sont des phrases complètes et d'autres non.

### 43.3.7. Éviter la voix passive

Utilisez la voix active. Utilisez des phrases complètes quand il y a un sujet (<< A ne peut pas faire B >>). Utilisez le style télégramme, sans sujet, si le sujet est le programme lui−même ; n'utilisez pas << Je >> pour le programme.

Raisonnement : le programme n'est pas humain. Ne prétendez pas autre chose.

---

### 43.3.8. Présent contre passé

Utilisez le passé si une tentative de faire quelque chose échouait, mais pourrait peut−être réussir la prochaine fois (peut−être après avoir corriger certains problèmes). Utilisez le présent si l'échec est sans doute permanent.

Il y a une différence sémantique non triviale entre les phrases de la forme

```
n'a pas pu ouvrir le fichier "%s": %m
```

et

```
ne peut pas ouvrir le dossier "%s"
```

La première forme signifie que la tentative d'ouverture du fichier a échoué. Le message devrait donner une raison comme << disque plein >> ou << le fichier n'existe pas >>. Le passé est approprié parce que la prochaine fois le disque peut ne plus être plein ou le fichier en question peut exister.

La seconde forme indique que la fonctionnalité d'ouvrir le fichier nommé n'existe pas du tout dans le programme ou que c'est conceptuellement impossible. Le présent est approprié car la condition persistera indéfiniment.

Raisonnement : d'accord, l'utilisateur moyen ne sera pas capable de tirer de grandes conclusions simplement à partir du temps du message mais, puisque la langue nous fournit une grammaire, nous devons l'utiliser correctement.

---

### 43.3.9. Type de l'objet

En citant le nom d'un objet, spécifiez quel genre d'objet c'est.

Raisonnement : sinon personne ne saura ce qu'est << foo.bar.baz >>.

---

### 43.3.10. Crochets

Les crochets sont uniquement utilisés (1) dans les synopsis des commandes pour indiquer des arguments optionnels ou (2) pour indiquer l'indice inférieur d'un tableau.

Raisonnement : rien de ce qui ne correspond pas à l'utilisation habituelle, largement connue troublera les gens.

---

## 43.3.11. Assembler les messages d'erreurs

Quand un message inclut du texte qui est généré ailleurs, intégrez–le dans ce style :

```
n'a pas pu ouvrir le fichier %s: %m
```

Raisonnement : il serait difficile d'expliquer tous les codes d'erreurs possibles pour coller ceci dans une unique phrase douce, ainsi une certaine forme de ponctuation est nécessaire. Mettre le texte inclus entre parenthèses a été également suggéré, mais ce n'est pas naturel si le texte inclus est susceptible d'être la partie la plus importante du message, comme c'est souvent le cas.

---

## 43.3.12. Raisons pour les erreurs

Les messages devraient toujours indiquer la raison pour laquelle une erreur s'est produite. Par exemple :

```
MAUVAIS :  n'a pas pu ouvrir le fichier %s
MEILLEUR : n'a pas pu ouvrir le fichier %s (échec E/S)
```

Si aucune raison n'est connue, vous feriez mieux de corriger le code.

---

## 43.3.13. Nom des fonctions

N'incluez pas le nom de la routine de rapport dans le texte de l'erreur. Nous avons d'autres mécanismes pour trouver cela quand c'est nécessaire et, pour la plupart des utilisateurs, ce n'est pas une information utile. Si le texte de l'erreur n'a plus beaucoup de sens sans le nom de la fonction, reformulez–le.

```
MAUVAIS :  pg_atoi: erreur dans "z": ne peut pas analyser "z"
MEILLEUR : syntaxe en entrée invalide pour l'entier : "z"
```

Évitez de mentionner le nom des fonctions appelées, au lieu de cela dites ce que le code essayait de faire :

```
MAUVAIS :  ouvrir() a échoué : %m
MEILLEUR : n'a pas pu ouvrir le fichier %s: %m
```

Si cela semble vraiment nécessaire, mentionnez l'appel système dans le message détail. (Dans certains cas, fournir les valeurs réelles passées à l'appel système pourrait être une information appropriée pour le message détail.)

Raisonnement : les utilisateurs ne savent pas tout ce que ces fonctions font.

---

## 43.3.14. Mots délicats à éviter

**Incapable.** << Incapable >> est presque la voix passive. Une meilleure utilisation est << ne pouvait pas >> ou << ne pourrait pas >> selon les cas.

**Mauvais.** Les messages d'erreurs comme << mauvais résultat >> sont vraiment difficile à interpréter intelligemment. Cela est mieux d'écrire pourquoi le résultat est << mauvais >>, par exemple, << format invalide >>.

**Illégal.** << Illégal >> représente une violation de la loi, le reste est << invalide >>. Meilleur encore, dites

pourquoi cela est invalide.

**Inconnu.** Essayez d'éviter << inconnu >>. Considérez << erreur : réponse inconnue >>. Si vous ne savez pas qu'elle est la réponse, comment savez−vous que cela est incorrect ? << Non reconnu >> est souvent un meilleur choix. En outre, assurez−vous d'inclure la valeur pour laquelle il y a un problème.

```
MAUVAIS :  type de n&oelig;ud inconnu
MEILLEUR : type de n&oelig;ud non reconnu : 42
```

**Trouver contre Exister.** Si le programme emploie un algorithme non trivial pour localiser une ressource (par exemple, une recherche de chemin) et que l'algorithme échoue, il est juste de dire que le programme n'a pas pû << trouver >> la ressource. D'un autre côté, si l'endroit prévu pour la ressource est connu mais que le programme ne peut pas accéder à celle−ci, alors dites que la ressource n'<< existe >> pas. Utilisez << trouvez >> dans ce cas là semble faible et embrouille le problème.

## 43.3.15. Orthographe appropriée

Orthographiez les mots en entier. Par exemple, évitez :

- spec (NdT : spécification)
- stats (NdT : statistiques)
- params (NdT : paramètres)
- auth (NdT : authentification)
- xact (NdT : transaction)

Raisonnement : cela améliore la cohérence.

## 43.3.16. Adaptation linguistique

Gardez à l'esprit que les textes des messages d'erreurs ont besoin d'être traduit en d'autres langues. Suivez les directives dans la Section 44.2.2 pour éviter de rendre la vie difficile aux traducteurs.

# Chapitre 44. Support natif des langues

## 44.1. Pour le traducteur

Les programmes PostgreSQL (serveur et client) peuvent afficher leur message dans la langue préférée de l'utilisateur — si les messages ont été traduits. Créer et maintenir les ensembles de messages traduits nécessite l'aide de personnes parlant leur propre langue et souhaitant contribuer à PostgreSQL. Il n'est nul besoin d'être un développeur pour cela. Cette section explique comment apporter son aide.

### 44.1.1. Prérequis

Les compétences dans sa langue d'un traducteur ne seront pas jugées — cette section concerne uniquement les outils logiciels. Théoriquement, seul un éditeur de texte est nécessaire. Mais ceci n'est vrai que dans le cas très improbable où un traducteur ne souhaiterait pas tester ses traductions des messages. Lors de la configuration des sources, il faudra s'assurer d'utiliser l'option `--enable-nls`. Ceci assurera également la présence de la bibliothèque libintl et du programme `msgfmt` dont tous les utilisateurs finaux ont indéniablement besoin. Pour tester son travail, il sera utile de suivre les parties pertinentes des instructions d'installation.

Pour commencer un nouvel effort de traduction ou pour faire un assemblage de catalogues de messages (décrit ci–après), il faudra installer respectivement les programmes `xgettext` et `msgmerge` dans une implémentation compatible GNU. Il est prévu dans le futur que `xgettext` ne soit plus nécessaire lorsqu'une distribution empaquetée des sources est utilsée. (À partir du CVS, il en sera toujours besoin.) GNU Gettext 0.10.36 ou ultérieure est actuellement recommandé.

Toute implémentation locale de gettext devrait être disponible avec sa propre documentation. Une partie en est certainement dupliquée dans ce qui suit mais des détails complémentaires y sont certainement disponibles.

### 44.1.2. Concepts

Les couples de messages originaux (anglais) et de leurs (possibles) traductions sont conservés dans les *catalogues de messages*, un pour chaque programme (bien que des programmes liés puissent partager un catalogue de messages) et pour chaque langue cible. Il existe deux formats de fichiers pour les catalogues de messages : le premier est le fichier << PO >> (pour "Portable Object" ou Objet Portable), qui est un fichier texte muni d'une syntaxe spéciale et que les traducteurs éditent. Le second est un fichier << MO >> (pour "Machine Object" ou Objet Machine), qui est un fichier binaire engendré à partir du fichier PO respectif et qui est utilisé lorsque le programme internationalisé est exécuté. Les traducteurs ne s'occupent pas des fichiers MO ; en fait, quasiment personne ne s'en occupe.

L'extension du fichier de catalogue de messages est, sans surprise, soit `.po`, soit `.mo`. Le nom de base est soit le nom du programme qu'il accompagne soit la langue utilisée dans le fichier, suivant la situation. Ceci peut s'avérer être une source de confusion. Des exemples sont `psql.po` (fichier PO pour psql) ou `fr.mo` (fichier MO en français).

Le format du fichier PO est illustré ici :

```
# commentaire

msgid "chaîne originale"
```

```
msgstr "chaîne traduite"

msgid "encore une originale"
msgstr "encore une de traduite"
"les chaînes peuvent être sur plusieurs lignes, comme ceci"

...
```

Les chaînes msgid sont extraites des sources du programme. (Elles n'ont pas besoin de l'être mais c'est le moyen le plus commun). Les lignes msgstr sont initialement vides puis complétées avec les chaînes traduites. Les chaînes peuvent contenir des caractères d'échappement de style C et peuvent être sur plusieurs lignes comme le montre l'exemple ci−dessus. (La ligne suivante doit démarrer au début de la ligne.)

Le caractère # introduit un commentaire. Si une espace fine suit immédiatement le caractère #, c'est qu'il s'agit là d'un commentaire maintenu par le traducteur. On trouve aussi des commentaires automatiques qui n'ont pas d'espace fine suivant immédiatement le caractère #. Ils sont maintenus par les différents outils qui opèrent sur les fichiers PO et ont pour but d'aider le traducteur.

```
#. commentaire automatique
#: fichier.c:1023
#, drapeau, drapeau
```

Les commentaires du style #. sont extraits du fichier source où le message est utilisé. Il est possible que le développeur ait ajouté des informations pour le traducteur, telles que l'alignement attendu. Le commentaire #: indique l'emplacement exact où le message est utilisé dans le source. Le traducteur n'a pas besoin de regarder le source du programme, mais il peut le faire s'il subsiste un doute sur l'exactitude d'une traduction. Le commentaire #, contient des drapeaux décrivant le message d'une certaine façon. Il existe actuellement deux drapeaux : `fuzzy` est positionné si le message risque d'être rendu obsolète par des changements dans les sources. Le traducteur peut alors vérifier ceci et supprimer ce drapeau. Notez que les messages << fuzzy >> ne sont pas accessibles à l'utilisateur final. L'autre drapeau est `c−format` indiquant que le message utilise le format de la fonction C `printf`. Ceci signifie que la traduction devrait aussi être de ce format avec le même nombre et le même type de paramètres fictifs. Il existe des outils qui vérifient que le message est une chaîne au format printf et valident le drapeau c−format en consequence.

## 44.1.3. Créer et maintenir des catalogues de messages

OK, alors comment faire pour créer un catalogue de messages << vide >> ? Tout d'abord, se placer dans le répertoire contenant le programme dont on souhaite traduire les messages. S'il existe un fichier `nls.mk`, alors ce programme est préparé pour la traduction.

S'il y a déjà des fichiers `.po`, alors quelqu'un a déjà réalisé des travaux de traduction. Les fichiers sont nommés *`langue`*`.po`, où *`langue`* est le code de langue sur deux caractères (en minuscules) tel que défini par l'ISO 639−1, c'est−à−dire `fr.po` pour le français. S'il existe réellement un besoin pour plus d'une traduction par langue alors les fichiers peuvent être renommés *`langue_region`*`.po` où *`region`* est le code de langue sur deux caractères (en majuscules), tel que défini par l'ISO 3166−1, c'est−à−dire `pt_BR.po` pour le portuguais du Brésil. Si vous trouvez la langue que vous souhaitez, vous pouvez commencer à travailler sur ce fichier.

Pour commencer une nouvelle traduction, il faudra préalablement exécuter la commande

```
gmake init-po
```

Ceci créera un fichier *nomprog*.pot. (.pot pour le distinguer des fichiers PO qui sont << en production >>. Le T signifie << template >> (NdT : modèle en anglais). On copiera ce fichier sous le nom *langue*.po. On peut alors l'éditer. Pour faire savoir qu'une nouvelle langue est disponible, il faut également éditer le fichier nls.mk et y ajouter le code de la langue (ou de la langue et du pays) avec une ligne ressemblant à ceci :

```
AVAIL_LANGUAGES := de fr
```

(D'autres langues peuvent apparaître, bien entendu.)

À mesure que le programme ou la bibliothèque change, des messages peuvent être modifiés ou ajoutés par les développeurs. Dans ce cas, il n'est pas nécessaire de tout recommencer depuis le début. À la place, on lancera la commande

```
gmake update-po
```

qui créera un nouveau catalogue de messages vides (le fichier pot avec lequel la traduction a été initiée) et le fusionnera avec les fichiers PO existants. Si l'algorithme de fusion a une incertitude sur un message particulier, il le marquera << fuzzy >> comme expliqué ci−dessus. Dans la cas où quelque chose se passerait mal, l'ancien fichier PO est sauvegardé avec une extension .po.old.

## 44.1.4. Éditer les fichiers PO

Les fichiers PO sont éditables avec un éditeur de texte standard. Le traducteur doit seulement modifier l'emplacement entre les guillemets après la directive msgstr, peut ajouter des commentaires et modifier le drapeau fuzzy. (NdA : Il existe, ce qui n'est pas surprenant, un mode PO pour Emacs, que je trouve assez utile.)

Les fichiers PO n'ont pas besoin d'être entièrement remplis. Le logiciel retournera automatiquement à la chaîne originale si une traduction n'est pas disponible ou est laissée vide. Soumettre des traductions incomplètes pour les inclure dans l'arborescence des sources n'est pas un problème ; cela permet à d'autres personnes de récupérer le travail commencé pour le continuer. Néanmoins, les traducteurs sont encouragés à donner une haute priorité à la suppression des entrées fuzzy après avoir fait une fusion. Les entrées fuzzy ne seront pas installées ; elles servent seulement de référence à ce qui pourrait être une bonne traduction.

Certaines choses sont à garder à l'esprit lors de l'édition des traductions :

- S'assurer que si la chaîne originale se termine par un retour chariot, la traduction le fasse bien aussi. De même pour les tabulations, etc.
- Si la chaîne originale est une chaîne au format printf, la traduction doit l'être aussi. La traduction doit également avoir les même spécificateurs de format et dans le même ordre. Quelques fois, les règles naturelles de la langue rendent cela impossible ou tout au moins difficile. Dans ce cas, il est possible de modifier les spécificateurs de format de la façon suivante :

  ```
  msgstr "Die Datei %2$s hat %1$u Zeichen."
  ```

  Le premier paramètre fictif sera alors utilisé par le deuxième argument de la liste. Le *chiffre*$ a besoin de suivre immédiatement le %, avant tout autre manipulateur de format. (Cette fonctionnalité existe réellement dans la famille des fonctions printf, mais elle est peu connue, n'ayant que peu d'utilité en dehors de l'internationalisation des messages.)

- Si la chaîne originale contient une erreur linguistique, on pourra la rapporter (ou la corriger soi−même dans le source du programme) et la traduire normalement. La chaîne corrigée peut être fusionnée lorsque les programmes sources auront été mis à jour. Si la chaîne originale contient une erreur factuelle, on la rapportera (ou la corrigera soi−même) mais on ne la traduira pas. À la place, on marquera la chaîne avec un commentaire dans le fichier PO.
- Maintenir le style et le ton de la chaîne originale. En particulier, les messages qui ne sont pas des phrases (`cannot open file %s`, soit `ne peut pas ouvrir le fichier %s`) ne devraient probablement pas commencer avec une lettre capitale (si votre langue distingue la casse des lettres) ou finir avec un point (si votre langue utilise des marques de ponctuation). Lire Section 43.3 peut aider.
- Lorsque la signification d'un message n'est pas connue ou s'il est ambigü, on pourra demander sa signification sur la liste de diffusion des développeurs. Il est possible qu'un anglophone puisse aussi ne pas le comprendre ou le trouver ambigü. Il est alors préférable d'améliorer le message.

# 44.2. Pour le développeur

## 44.2.1. Mécaniques

Cette section explique comment implémenter le support natif d'une langue dans un programme ou dans une bibliothèque qui fait partie de la distribution PostgreSQL. Actuellement, cela s'applique uniquement aux programmes C.

**Ajouter le support NLS à un programme**

1. Le code suivant est inséré dans la séquence initiale du programme :

   ```
   #ifdef ENABLE_NLS
   #include <locale.h>
   #endif

   ...

   #ifdef ENABLE_NLS
   setlocale(LC_ALL, "");
   bindtextdomain("nomprog", LOCALEDIR);
   textdomain("nomprog");
   #endif
   ```

   (*nomprog* peut être choisi tout à fait librement.)
2. Partout où un message candidat à la traduction est trouvé, un appel à `gettext()` doit être inséré. Par exemple :

   ```
   fprintf(stderr, "panic level %d\n", lvl);
   ```

   devra être changé avec

   ```
   fprintf(stderr, gettext("panic level %d\n"), lvl);
   ```

   (`gettext` est défini comme une opération nulle si NLS n'est pas configuré.)

   Cela peut engendrer du fouillis. Un raccourci habituel consiste à utiliser

```
#define _(x) gettext(x)
```

Une autre solution est envisageable si le programme effectue la plupart de ses communications via une fonctio nou un nombre restreint de fonctions, telle `ereport()` pour le moteur. Le fonctionnement interne de cette fonction peut alors être modifiée pour qu'elle appelle `gettext` pour toutes les chaînes en entrée.

3. Un fichier `nls.mk` est ajouté dans le répertoire des sources du programme. Ce fichier sera lu comme un makefile. Les affectations des variables suivantes doivent être réalisées ici :

   `CATALOG_NAME`
   : Le nom du programme tel que fourni lors de l'appel à `textdomain()`.

   `AVAIL_LANGUAGES`
   : Liste des traductions fournies — initialement vide.

   `GETTEXT_FILES`
   : Liste des fichiers contenant les chaînes traduisibles, c'est−à−dire celles marquées avec `gettext` ou avec une solution altenative. Il se peut que cette liste inclut pratiquement tous les fichiers sources du programme. Si cette liste est trop longue, le premier << fichier >> peut être remplacé par un + et le deuxième mot représenter un fichier contenant un nom de fichier par ligne.

   `GETTEXT_TRIGGERS`
   : Les outils qui engendrent des catalogues de messages pour les traducteurs ont besoin de connaître les appels de fonction contenant des chaînes à traduire. Par défaut, seuls les appels à `gettext()` sont reconnus. Si _ ou d'autres identifiants sont utilisés, il est nécessaire de les lister ici. Si la chaîne traduisible n'est pas le premier argument, l'élément a besoin d'être de la forme `func:2` (pour le second argument).

Le système de construction s'occupera automatiquement de construire et installer les catalogues de messages.

## 44.2.2. Guide d'écriture des messages

Voici quelques lignes de conduite pour l'écriture de messages facilement traduisibles.

- Ne pas construire de phrases à l'exécution, telles que

  ```
  printf("Files were %s.\n", flag ? "copied" : "removed");
  ```

  L'ordre des mots d'une phrase peut être différent dans d'autres langues. De plus, même si gettext() est correctement appelé sur chaque fragment, il pourrait être difficile de traduire séparément les fragments. Il est préférable de dupliquer un peu de code de façon à ce que chaque message à traduire forme un tout cohérent. Seuls les nombres, noms de fichiers et autres variables d'exécution devraient être insérés au moment de l'exécution dans le texte d'un message.
- Pour des raisons similaires, ceci ne fonctionnera pas :

  ```
  printf("copied %d file%s", n, n!=1 ? "s" : "");
  ```

  parce que cette forme présume de la façon dont la forme plurielle est obtenue. L'idée de résoudre ce cas de la façon suivante :

  ```
  if (n==1)
      printf("copied 1 file");
  else
      printf("copied %d files", n):
  ```

sera source de déception. Certaines langues ont plus de deux formes avec des règles particulières. Il se pourrait qu'une solution à ce problème soit trouvée dans le futur, mais actuellement le mieux est encore de l'éviter. Il est préférable d'écrire :

```
printf("number of copied files: %d", n);
```

- Lorsque quelque chose doit être communiqué au traducteur, telle que la façon dont un message doit être aligné avec quelque autre sortie, on pourra faire précéder l'occurrence de la chaîne d'un commentaire commençant par `translator`, par exemple

  ```
  /* translator: This message is not what it seems to be. */
  ```

  Ces commentaires sont copiés dans les catalogues de messages de façon à ce que les traducteurs les voient.

---

# Chapitre 45. Écrire un gestionnaire de langage procédural

Tous les appels vers des fonctions écrites dans un langage autre que celui de l'interface << version 1 >> pour les langages compilés (ceci inclut les fonctions dans les langages procéduraux définis par l'utilisateur, les fonctions écrites en SQL et les fonctions utilisant l'interface de langage compilé version 0), passent par une fonction du *gestionnaire d'appels* spécifique au langage. Il est de la responsabilité du gestionnaire d'appels d'exécuter la fonction de manière significative, comme par exemple en interprétant le texte source fourni. Ce chapitre indique comment le gestionnaire d'appels d'un nouveau langage procédural peut être écrit.

Le gestionnaire d'appel d'un langage procédural est une fonction << normale >> qui doit être écrite dans un langage compilé tel que le C, en utilisant l'interface version−1, et enregistré avec PostgreSQL comme ne prenant aucun argument et retournant le type `language_handler`. Ce pseudo−type spécial identifie la fonction en tant que gestionnaire d'appel et l'empêche d'être appelée directement à partir des commandes SQL.

Le gestionnaire d'appels est appelé de la même façon que n'importe quelle autre fonction : il reçoit un pointeur vers une `structure FunctionCallInfoData` contenant des valeurs en argument et une information sur la fonction appelée. Il doit renvoyer un résultat `Datum` (et, si possible, initialiser le champ `isnull` de la structure `FunctionCallInfoData` s'il souhaite renvoyer un résultat SQL NULL). La différence entre un gestionnaire d'appels et une fonction ordinaire est que le champ `flinfo->fn_oid` de la structure `FunctionCallInfoData` contiendra l'OID de la fonction à appeler, et non pas celui du gestionnaire d'appels lui−même. Le gestionnaire d'appels doit utiliser ce champ pour déterminer la fonction à exécuter. De plus, la liste d'arguments passée a été configurée en accord avec la déclaration de la fonction cible, et non pas en fonction du gestionnaire d'appels.

C'est au gestionnaire d'appels de récupérer l'entrée de la fonction à partir de la table système `pg_proc` et d'analyser les types des arguments et de la valeur de retour de la fonction appelée. La clause `AS` de la commande `CREATE FUNCTION` de la fonction sera trouvée dans la colonne `prosrc` de `pg_proc`. Cela peut être le texte du source du langage procédural lui−même (comme pour PL/Tcl) mais, en théorie, cela peut être un chemin vers un fichier ou tout autre chose qui indique en détails au gestionnaire d'appels ce qu'il doit faire.

Souvent, la même fonction est appelée plusieurs fois par instruction SQL. Un gestionnaire d'appels peut éviter des recherches d'informations répétées sur la fonction appelée en utilisant le champ `flinfo->fn_extra`. Il sera initialement à `NULL` mais pourra être configuré par le gestionnaire d'appels pour pointer vers l'information de la fonction appelée. Pour les appels suivants, si `flinfo->fn_extra` est déjà différent de `NULL`, alors il peut être utilisé et l'étape de recherche d'information pourra être évitée. Le gestionnaire d'appels doit s'assurer que `flinfo->fn_extra` pointe sur une zone mémoire qui restera allouée au moins jusqu'à la fin de la requête en cours car il se peut qu'une structure de données `FmgrInfo` soit conservée aussi longtemps. Une façon de le faire est d'allouer les données supplémentaires dans le contexte mémoire spécifié par `flinfo->fn_mcxt` ; de telles données auront normalement la même espérance de vie que `FmgrInfo` lui−même. Mais le gestionnaire pourrait également choisir d'utiliser un contexte mémoire de plus longue durée de façon à mettre en cache sur plusieurs requêtes des informations sur les définitions des fonctions.

Lorsqu'une fonction en langage procédural est appelée via un déclencheur, aucun argument ne lui est passé de façon habituelle mais le champ `context` de `FunctionCallInfoData` pointe sur une structure `TriggerData`, et n'est pas `NULL` comme c'est le cas dans les appels de fonctions standard. Le gestionnaire d'un langage devrait fournir des mécanismes pour que les fonctions de langages procéduraux obtiennent des

informations du déclencheur.

Voici un modèle de gestionnaire de langage procédural écrit en C :

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * Appelé en tant que procédure d'un déclencheur
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        retval = ...
    }
    else
    {
        /*
         * Appelé en tant que fonction
         */

        retval = ...
    }

    return retval;
}
```

Seules quelques milliers de lignes de codes devront être ajoutées à la place des points pour compléter ce modèle.

Après avoir compilé la fonction du gestionnaire dans un module chargeable (voir Section 31.9.6), les commandes suivantes enregistrent le langage procédural défini dans l'exemple :

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'nomfichier'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Les langages de procédures inclus dans la distribution standard sont de bonnes références lorsque vous essayez d'écrire votre propre gestionnaire d'appels. Regardez dans le sous−répertoire `src/pl` des sources.

# Chapitre 46. Optimiseur génétique de requêtes (*Genetic Query Optimizer*)

**Auteur :** Écrit par Martin Utesch (<utesch@aut.tu-freiberg.de>) de l'institut de contrôle automatique de l'université des mines et de la technologie de Freiberg, Allemagne.

## 46.1. Gestion des requêtes comme un problème complexe d'optimisation

Parmi tous les opérateurs relationnels, le plus difficile à exécuter et à optimiser est la jointure (*join*). Le nombre de plans alternatifs pour répondre à une requête croît de façon exponentielle avec le nombre de jointures inclus. Un effort supplémentaire d'optimisation est dû au support d'une variété de *méthodes de jointure* (par exemple les boucles imbriquées, les jointures de découpage, les jointures d'assemblage dans PostgreSQL) pour exécuter des jointures individuelles et une diversité d'*index* (par exemple R−tree, B−tree, découpage dans PostgreSQL) comme chemins d'accès des relations.

L'implémentation de l'optimiseur de PostgreSQL réalise une *recherche pratiquement exhaustive* sur tout l'espace des stratégies alternatives. Cet algorithme, tout d'abord introduit dans la base de données << System R >>, produit un ordre de jointure presque optimal mais peut prendre beaucoup de temps et d'espace mémoire lorsque le nombre de jointures dans une requête devient important. Ceci rend inapproprié l'optimiseur ordinaire de requêtes de PostgreSQL pour les requêtes établissant une jointure entre un grand nombre de tables.

L'institut de contrôle automatique de l'université des mines et de la technologie, basé à Freiberg, Allemagne, a rencontré les problèmes décrits car ces employés voulaient utiliser le DBMS PostgreSQL comme moteur pour leur système de support pour la maintenance d'une grille de courant électrique. Le DBMS avait besoin de gérer des requêtes comprenant des jointures larges pour la machine d'inférence du système de connaissances.

Les difficultés en terme de performance pour l'exploration des plans de requêtes possibles ont créé la demande du développement d'une nouvelle technique d'optimisation.

Dans la suite, nous décrivons l'implémentation d'un *algorithme génétique* pour résoudre le problème des ordres de jointure d'une façon efficace pour les requêtes impliquant un grand nombre de ces jointures.

## 46.2. Algorithmes génétiques

L'algorithme génétique (GA) est une méthode heuristique d'optimisation qui opère via des recherches non déterministes au hasard. L'ensemble des solutions possibles pour le problème d'optimisation est considéré comme une *population* d'*individus*. Le degré d'adaptation d'un individu dans son environnement est spécifié par sa *forme physique*.

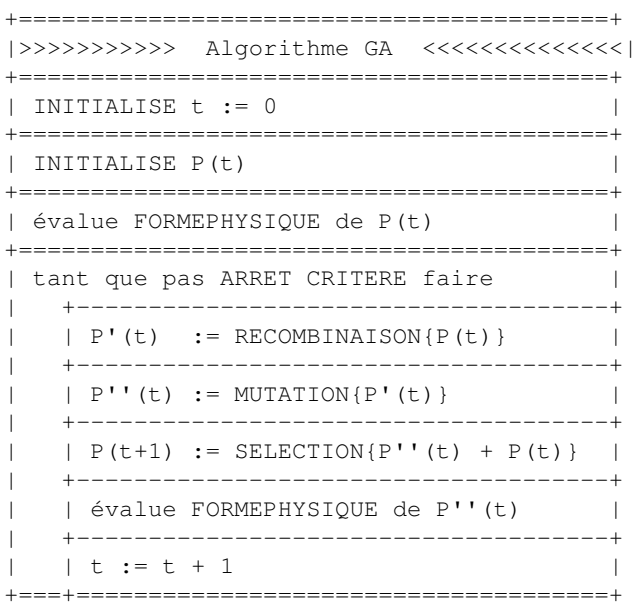
Les coordonnées d'un individu dans l'espace de recherche sont représentées par des *chromosomes*, en fait un ensemble de chaînes de caractères. Un *gène* est une sous−section d'un chromosome qui code la valeur d'un seul paramètre en cours d'optimisation. Les codages typiques pour un gène pourraient être *binary* ou *integer*.

À travers la simulation des opérations évolutives (*recombinaison*, *mutation* et *sélection*), de nouvelles générations de points de recherche sont trouvées affichant une meilleure forme physique que leurs ancêtres.

D'après la FAQ de comp.ai.genetic, il ne peut pas être dit plus fortement qu'un GA n'est pas une recherche effectuée seulement au hasard. Un GA utilise des processus stocastiques mais le résultat n'est pas du tout dû au hasard (mieux que cela).

**Figure 46−1. Diagramme structuré d'un algorithme génétique**

P(t) génération des ancêtres au temps t
P''(t) génération des descendants au temps t




# 46.3. Optimisation génétique des requêtes (GEQO) avec PostgreSQL

Le module GEQO est la solution du problème d'optimisation des requêtes, une solution similaire au problème du voyageur de commerce (TSP). L'approche du module GEQO concernant le problème d'optimisation de requêtes revient au problème bien connu du marchand de commerce (TSP). Les plans de requêtes possibles sont codés comme des chaînes d'entiers. Chaque chaîne représente l'ordre de jointure d'une relation de la requête à une autre. Par exemple, l'arbre de jointure

```
   /\
  /\ 2
 /\ 3
4  1
```

est codé avec la chaîne d'entiers '4−1−3−2', ce qui signifie : première jointure entre les relations '4' et '1', puis '3' et enfin '2', avec 1, 2, 3, 4 les identifiants des relations pour l'optimiseur de PostgreSQL.

Des parties du module GEQO sont adaptées de l'algorithme Genitor de D. Whitley.

Les caractéristiques spécifiques de l'implémentation de GEQO dans PostgreSQL sont :

- Utilisation d'un *état d'équilibre* du GA (remplacement des individus les moins performants d'une population, pas un remplacement d'une génération complète) qui permet une convergence rapide vers les plans de requêtes améliorés. C'est essentiel pour une gestion des requêtes sur un temps raisonnable ;
- Utilisation d'un *croisement de recombinaison de bord* qui convient tout spécialement pour garder bas le nombre de pertes aux bords pour la solution du TSP en utilisant un GA;
- La mutation comme opérateur génétique est obsolète d'une telle façon qu'aucun mécanisme de réparation n'est nécessaire pour générer des tours TSP légaux.

Le module GEQO permet à l'optimiseur de requêtes de PostgreSQL de supporter les requêtes disposant de jointures importantes de manière efficace via une recherche non exhaustive.

### 46.3.1. Tâches pour la future implémentation de GEQO pour PostgreSQL

Un gros travail est toujours nécessaire pour améliorer les paramètres de l'algorithme génétique. Dans le fichier `src/backend/optimizer/geqo/geqo_main.c`, pour les routines `gimme_pool_size` et `gimme_number_generations`, nous devons trouver un compromis dans les paramètres pour satisfaire deux demandes concurrentes :

- Plan de requête optimum
- Temps de calcul

À un niveau plus basique, il n'est pas clair que résoudre l'optimisation d'une requête avec un algorihme GA conçu pour TSP soit appropriée. Dans le cas TSP, le coût associé avec toute sous−chaîne (tour partiel) est indépendant du reste du tour mais ceci n'est certainement pas vrai pour l'optimisation de requêtes. Du coup, la question reste posée quant au fait que la recombinaison soit la procédure de mutation la plus efficace.

## 46.4. Lectures supplémentaires

Les ressources suivantes contiennent des informations supplémentaires sur les algorithmes génétiques :

- The Hitch−Hiker's Guide to Evolutionary Computation (FAQ de comp.ai.genetic)
- Evolutionary Computation and its application to art and design par Craig Reynolds
- *Fundamentals of Database Systems*
- *The design and implementation of the POSTGRES query optimizer*

# Chapitre 47. Fonctions d'estimation du coût des index

> **Auteur :** Écrit par Tom Lane (<tgl@sss.pgh.pa.us>) le 24 janvier 2000.

> **Note :** Ceci pourra faire partie d'un futur chapitre plus étoffé sur la façon d'écrire des nouvelles méthodes d'accès aux index.

Chaque méthode d'accès aux index doit fournir une fonction d'estimation de coût qui est utilisée par l'optimiseur de requêtes. L'OID de procédure de cette fonction est donné dans le champ `amcostestimate` de la ligne correspondant à cette méthode d'accès dans la table `pg_am`.

> **Note :** Avant PostgreSQL 7.0, un mécanisme différent était utilisé pour enregistrer les fonctions d'estimation de coût spécifiques des index.

La fonction amcostestimate reçoit en paramètres une liste de clauses WHERE pour lesquelles il a été déterminé qu'elles peuvent utiliser cet index. Elle doit retourner le coût estimé d'accès à l'index et la sélectivité de la clause WHERE, c'est–à–dire la fraction de la table principale qui sera retournée par le parcours de l'index. Dans les cas simples, pratiquement tout le travail d'estimation de coût peut être réalisé en appelant des routines standard de l'optimiseur ; La fonction amcostestimate ne sert qu'à permettre à la méthode d'accès d'ajouter sa connaissance spécifique du type d'index, au cas où il serait possible d'améliorer l'estimation standard.

Chaque fonction amcostestimate doit avoir la signature suivante :

```
void
amcostestimate (Query *root,
                RelOptInfo *rel,
                IndexOptInfo *index,
                List *indexQuals,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

Les quatre premiers paramètres sont des entrées :

root
: La requête traitée.

rel
: La relation sur laquelle porte l'index.

index
: L'index lui même.

indexQuals
: La liste des clauses qualifiées pour l'index (implicitement reliées par des ET logiques) ; une liste NIL indique qu'aucune clause n'est qualifiée.

Les quatre derniers paramètres sont des sorties passées par référence :

*indexStartupCost

Coût du démarrage du traitement de l'index

*indexTotalCost

Coût total de traitement de l'index

*indexSelectivity

Sélectivité de l'index

*indexCorrelation

Coefficient de corrélation entre l'ordre de lecture de l'index et l'ordre de la table sous−jacente.

Notez que les fonctions d'estimation de coût doivent être écrites en C, pas en SQL ou avec un autre des langages de programmation disponibles, car elles doivent accéder aux structures de données internes de l'optimiseur.

Le coût d'accès aux index doit être calculé avec les unités utilisées dans `src/backend/optimizer/path/costsize.c` : la lecture séquentielle d'un bloc a un coût de 1,0, une lecture non séquentielle a un coût de random_page_cost et le coût de traitement d'une ligne d'index a généralement un coût de cpu_index_tuple_cost (qui est un paramètre réglable par l'utilisateur). De plus, un multiple approprié de cpu_operator_cost doit être ajouté pour chaque opérateur de comparaison appelé lors du traitement de l'index et, spécialement, pour l'évaluation des indexQuals eux−mêmes.

Le coût d'accès doit inclure chaque coût disque et CPU associé au parcours de l'index lui même, mais PAS les coûts associés à la recherche ou au traitement des lignes de la table principale qui sont identifiées par l'index.

Le << coût de départ >> est la part du coût total de parcours qui doit être dépensée avant de pouvoir retrouver la première ligne. Pour la plupart des index, il peut être mis à zéro, mais un index qui a un coût de départ élevé peut vouloir lui donner un coût supérieur à zéro.

La sélectivité de l'index indexSelectivity doit indiquer la fraction de la table principale qui sera retournée par le parcours de l'index. Dans le cas d'un index peu intéressant, la sélectivité sera typiquement supérieure à la fraction des lignes qui passent réellement les conditions données.

indexCorrelation doit donner la corrélation (entre −1,0 et 1,0) entre l'ordre de l'index et l'ordre de la table. Ce paramètre est utilisé pour ajuster le coût de lecture de la table principale.

**Estimation de coût**

Une estimation de coût classique se fait comme suit :

1. Estimez et retournez la fraction de la table principale qui sera visitée, en fonction des conditions qualifiés. En l'absence de connaissance spécifique au type d'index, utilisez la fonction standard de l'optimiseur, `clauselist_selectivity()` :

   ```
   *indexSelectivity = clauselist_selectivity(root, indexQuals,
                                              rel- relid, JOIN_INNER);
   ```

2. Estimez le nombre de lignes d'index qui seront visitées durant le parcours. Pour de nombreux types d'index, cette valeur vaut indexSelectivity fois le nombre de lignes de l'index mais elle peut valoir plus. (Notez que la taille de l'index en pages et en lignes et disponible dans la structure IndexOptInfo.)
3. Estimez le nombre de pages d'index qui seront retournées lors de son parcours. Ce sera peut−être juste indexSelectivity fois la taille de l'index en pages.
4. Calculez le coût d'accès à l'index. Un estimateur générique pourrait procéder comme suit :

   ```
   /*
   ```

```
 * La supposition générale est que les pages d'index vont être lues
 * séquentiellement, si bien qu'elles ont un coût de 1,0 chacune, et
 * non pas de random_page_cost.
 * De plus, on ajoute le coût de l'évaluation des conditions pour chaque
 * ligne d'index. Tous les coûts sont supposés payés incrémentalement
 * lors du parcours de l'index.
 */
cost_qual_eval(&index_qual_cost, indexQuals);
*indexStartupCost = index_qual_cost.startup;
*indexTotalCost = numIndexPages +
    (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

5. Estimez la corrélation d'index. Pour un index simple ordonné sur un seul champ, cela peut se retrouver dans pg_statistic. Si vous ne savez pas, mettez zéro (pas de corrélation).

Des exemples de fonctions d'estimation peuvent être trouvés dans `src/backend/utils/adt/selfuncs.c`.

Par convention, l'entrée dans `pg_proc` pour une fonction `amcostestimate` a huit arguments tous déclarés comme `internal` (car aucun n'a un type connu de SQL), et le type retourné est `void`.

---

# Chapitre 48. Index GiST

## 48.1. Introduction

GiST est un acronyme pour *Generalized Search Tree*, c'est−à−dire arbre de recherche généralisé. C'est une méthode d'accès à une structure de type arbre de manière balancée, qui agit comme un modèle de base dans lequel il est possible d'implémenter des schémas d'indexage arbitraire. B+−trees, R−trees et de nombreux autres schémas d'indexage peuvent être implémentés avec GiST.

Un avantage de GiST est qu'il autorise le développement de types de données personnalisés avec les méthodes d'accès appropriées, par un expert dans le domaine des types de données, plutôt que par un expert des bases de données.

Les quelques informations disponibles ici ont été récupérées du site web du projet d'indexage GiST de l'université de Californie et de la thèse de Marcel Kornacker, Méthodes d'accès pour les systèmes de bases de données de la prochaine génération. L'implémentation GiST de PostgreSQL est principalement maintenu par Teodor Sigaev et Oleg Bartunov. Leur site web, http://www.sai.msu.su/~megera/postgres/gist/, dispose de plus d'informations.

## 48.2. Extensibilité

Traditionnellement, implémenter une nouvelle méthode d'accès à un index signifie un gros travail complexe. Il était nécessaire de comprendre les fonctionnements internes de la base de données, tels que le gestionnaire de verrous ou le WAL. L'interface GiST a un haut niveau d'abstraction, demandant à l'implémenteur de la méthode d'accès de n'implémenter que la sémantique du type de données en cours d'accès. La couche GiST elle−même prend garde aux accès concurrents, aux traces et à la recherche dans la structure en arbre.

L'extensibilité ne devrait pas être confondue avec l'extensibilité des autres arbres de recherche standards en termes de données qu'ils gèrent. Par exemple, PostgreSQL supporte les B+−trees et R−trees extensibles. Ceci signifie que vous pouvez utiliser PostgreSQL pour construire un B+−tree ou un R−tree sur tout type de données que vous souhaitez. Mais, les B+−trees supportent seulement les prédicats sur une séquence (<, =, >) et les R−trees supportent seulement les requêtes sur les séquences n−D (contient, est contenu, équivaut).

Donc, si vous indexez, disons, une collection d'images avec un B+−tree PostgreSQL, vous pouvez seulement lancer des requêtes telles que << est−ce que imagex est égale à imagey >>, << est−ce que imagex est plus petite que imagey >> et << est−ce que imagex est plus grande que imagey >> ? Suivant votre façon de définir le << égal à >>, le << inférieur à >> ou le << supérieur à >> dans ce contexte, cela peut être utile. Néanmoins, en utilisant un index basé sur GiST, vous pouvez créer des moyens de poser des questions spécifiques au domaine, peut−être << trouver toutes les images de chevaux >> ou << trouver toutes les images sur−exposées >>.

Tout ce qui est nécessaire pour obtenir une méthode d'accès GiST fonctionnelle est d'implémenter sept méthodes définies par l'utilisateur, qui définissent le comportement des clés dans l'arbre. Bien sûr, ces méthodes doivent être particulièrement élaborées pour supporter des requêtes avancées mais pour toutes les requêtes standards (B+−trees, R−trees, etc.) elles sont relativement simples. En bref, GiST combine l'extensibilité avec la généralité, la ré−utilisation de code et une interface propre.

## 48.3. Implémentation

Il existe sept méthodes qu'une classe d'opérateur d'index doit fournir pour GiST :

consistent
: Suivant un prédicat `p` sur une page de l'arbre et une requête utilisateur, `q`, cette méthode doit renvoyer false s'il est certain qu'à la fois `p` et `q` ne peuvent pas être vrais pour un élément de données spécifié.

union
: Cette méthode consolide les informations de l'arbre. Suivant une liste d'entrées, cette fonction génère un nouveau prédicat qui est vrai pour toutes les entrées.

compress
: Convertit l'élément de données en un format convenable pour l'emplacement physique dans une page d'index.

decompress
: L'inverse de la fonction `compress`. Convertit la représentation de l'élément donné en un format manipulable par la base de données.

penalty
: Renvoie une valeur indiquant le << coût >> d'une insertion d'une nouvelle entrée dans une branche particulière de l'arbre. Les éléments seront insérés en bas du chemin de la plus petite pénalité (`penalty`) de l'arbre.

picksplit
: Quand une séparation de page est nécessaire, cette fonction décide des entrées qui resteront sur l'ancienne page et de celles qui seront déplacées sur la nouvelle.

same
: Renvoie true si deux entrées sont identiques, false autrement.

## 48.4. Limitations

L'implémentation actuelle de GiST dans PostgreSQL a quelques grosses limitations : l'accès de GiST n'est pas concurrent ; l'interface de GiST ne permet pas le développement de certains types de données, tels que les arbres numériques (voir les papiers d'Aoki) ; et il n'existe pas encore de support pour les WAL de mises à jour dans les index GiST.

Les solutions aux problèmes de concurrence apparaissent dans la thèse de Marcel Kornacker ; néanmoins, ces idées n'ont pas encore été mises en pratiques dans l'implémentation de PostgreSQL.

Le manque de WAL est un simple soucis de programmation mais comme cela n'a pas encore été fait, un arrêt brutal pourrait rendre un index GiST inconsistant, forçant le lancement d'un REINDEX.

## 48.5. Exemples

Pour voir les exemples d'implémentations de méthodes d'indexage utilisant GiST, examinez les modules contrib suivants :

btree_gist
: B−Tree

cube
: Indexage de cube multi−dimensionnel

intarray

RD−Tree pour un tableau à une dimension de valeurs int4

ltree

Indexage des structures de type arbre

rtree_gist

R−Tree

seg

Stockage et accès indexé pour les << nombres flottants >>

tsearch and tsearch2

Indexage de texte complet

---

# Chapitre 49. Stockage physique de la base de données

Ce chapitre fournit un aperçu du format de stockage physique utilisé par les bases de données PostgreSQL.

## 49.1. Emplacement des fichiers de la base de données

Cette section décrit le format de stockage au niveau des fichiers et répertoires.

Toutes les données nécessaires à un groupe de bases de données sont stockées dans le répertoire data du groupe, habituellement référencé en tant que `PGDATA` (d'après le nom de la variable d'environnement qui peut être utilisé pour le définir). Un emplacement courant pour `PGDATA` est `/var/lib/pgsql/data`. Plusieurs groupes, gérés par différents postmasters, peuvent exister sur la même machine.

Le répertoire `PGDATA` contient plusieurs sous−répertoires et fichiers de contrôle, comme indiqué dans le Tableau 49−1. En plus de ces éléments requis, les fichiers de configuration du groupe, `postgresql.conf`, `pg_hba.conf` et `pg_ident.conf` sont traditionnellement stockés dans `PGDATA` (bien qu'il soit possible, à partir de la version 8.0 de PostgreSQL, de les conserver ailleurs).

**Tableau 49−1. Contenu de `PGDATA`**

| Élément | Description |
|---|---|
| `PG_VERSION` | Un fichier contenant le numéro de version majeur de PostgreSQL |
| `base` | Sous−répertoire contenant les sous−répertoires par base de données |
| `global` | Sous−répertoire contenant les tables communes au groupe, telles que `pg_database` |
| `pg_clog` | Sous−répertoire contenant les données d'état de validation des transactions |
| `pg_subtrans` | Sous−répertoire contenant les données d'états des sous−transaction |
| `pg_tblspc` | Sous−répertoire contenant les liens symboliques vers les espaces logiques |
| `pg_xlog` | Sous−répertoire contenant les fichiers WAL (Write Ahead Log) |
| `postmaster.opts` | Un fichier enregistrant les options en ligne de commande avec lesquelles le postmaster a été lancé la dernière fois |
| `postmaster.pid` | Un fichier verrou enregistrant le PID courant du postmaster et l'identifiant du segment de mémoire partagé (absent après l'arrêt de postmaster) |

Pour chaque base de données dans le groupe, il existe un sous−répertoire dans `PGDATA/base`, nommé d'après l'OID de la base de données dans `pg_database`. Ce sous−répertoire est l'emplacement par défaut pour les fichiers de la base de données; en particulier, ses catalogues système sont stockés ici.

Chaque table et index sont stockés dans un fichier séparé, nommé d'après le numéro *filenode* de la table ou de l'index, lequel se trouve dans `pg_class.relfilenode`.

| Attention |
|---|
| Notez que, bien que le filenode de la table correspond souvent à son OID, cela n'est *pas* nécessairement le cas; certaines opérations, comme `TRUNCATE`, `REINDEX`, `CLUSTER` et quelques formes d'`ALTER TABLE`, peuvent modifier le filenode tout en préservant l'OID. Évitez de supposer que filenode et OID sont identiques. |

Quand une table ou un index dépasse 1 Go, il est divisé en *segments* d'un Go. Le nom du fichier du premier segment est identique au filenode; les segments suivants sont nommés filenode.1, filenode.2, etc. Cette disposition évite des problèmes sur les plateformes qui ont des limitations sur les tailles des fichiers. Le contenu des tables et des index est discuté plus en détails dans Section 49.3.

Une table contenant des colonnes avec des entrées potentiellement volumineuses aura une table *TOAST* associée, qui est utilisée pour le stockage de valeurs de champs trop importantes pour conserver des lignes adéquates. `pg_class.reltoastrelid` établit un lien entre une table et sa table TOAST, si elle existe. Voir Section 49.2 pour plus d'informations.

Les espaces logiques (Tablespaces) rendent ce scénario plus compliqués. Chaque espace logique défini par l'utilisateur contient un lien symbolique dans le répertoire `PGDATA/pg_tblspc`, pointant vers le répertoire physique de l'espace logique (comme spécifié dans sa commande `CREATE TABLESPACE`). Le lien symbolique est nommé d'après l'OID de l'espace logique. À l'intérieur du répertoire de l'espace logique, il existe un sous−répertoire pour chacune des bases de données contenant des éléments dans cet espace logique. Ce sous−répertoire est nommé d'après l'OID de la base. Les tables de ce répertoire suivent le schéma de nommage des filenodes. L'espace logique `pg_default` n'est pas accédé via `pg_tblspc` mais correspond à `PGDATA/base`. De façon similaire, l'espace logique `pg_global` n'est pas accédé via `pg_tblspc` mais correspond à `PGDATA/global`.

## 49.2. TOAST

Cette section fournit un aperçu de TOAST (*The Oversized−Attribute Storage Technique*, la technique de stockage des attributs trop grands).

Puisque PostgreSQL utilise une taille de page fixe (habituellement 8 Ko) et n'autorise pas qu'une ligne s'étende sur plusieurs pages, il n'est pas possible de stocker de grandes valeurs directement dans les champs. Avant PostgreSQL 7.1, il existait une limite en dur juste en−dessous d'une page pour la taille totale des données d'une ligne de table. A partir de la version 7.1, cette limite est dépassée en permettant de compresser des valeurs de champ volumineuses et/ou de les diviser en plusieurs lignes physiques. Ceci survient de façon transparente pour l'utilisateur, avec seulement un petit impact sur le code du serveur. Cette technique est connu sous l'acronyme affectueux de TOAST (ou << the best thing since sliced bread >>).

Seuls certains types de données supportent TOAST — il n'est pas nécessaire d'imposer cette surcharge sur les types de données qui ne produisent pas de gros volumes. Pour supporter TOAST, un type de données doit avoir une représentation (*varlena*) à longueur variable, dans laquelle les 32 premiers bits contiennent la longueur totale de la valeur en octets (ceci incluant la longueur elle−même). TOAST n'a aucune contrainte supplémentaire sur la représentation. Toutes les fonctions niveau C qui gèrent un type données supportant TOAST doivent faire attention à gérer les valeurs en entrée TOASTées. (Ceci se fait normalement en appelant `PG_DETOAST_DATUM` avant de faire quoi que ce soit avec une valeur en entrée; mais dans certains cas, des approches plus efficaces sont possibles.)

TOAST empiète les deux bits de poids fort sur le mot contenant la longueur varlena, limitant par conséquent la taille logique de toute valeur d'un type de données TOAST−able à 1 Go ($2^{30} - 1$ octets). Quand les deux bits sont à zéro, cette valeur est une valeur ordinaire, non TOASTée, du type de données. Un des bits initialisé indique que la valeur a été compressée et doit être décompressée avant d'être utilisée. L'autre bit indique que la valeur a été stocké en dehors de la ligne. Dans ce cas, le reste de la valeur est en réalité simplement un pointeur et la véritable donnée se trouve ailleurs. Quand les deux bits sont initialisés, les données sont stockées ailleurs tout en étant compressées. Dans tous les cas, la longueur dans les bits de poids faible du mot

varlena indique la taille réelle de la donnée, pas la taille de la valeur logique qui serait extraite par décompression ou récupération de la donnée en dehors de la ligne.

Si une des colonnes d'une table est TOAST−able, la table disposera d'une table TOAST associé, dont l'OID est stockée dans l'entrée `pg_class.reltoastrelid` de la table. Les valeurs TOASTées hors−ligne sont conservées dans la table TOAST comme décrit avec plus de détails ci−dessous.

La technique de compression utilisée est un simple et rapide membre de la famille des techniques de compression LZ. Voir `src/backend/utils/adt/pg_lzcompress.c` pour les détails.

Les valeurs hors−ligne sont divisées (après compression si nécessaire) en morceaux d'au plus `TOAST_MAX_CHUNK_SIZE` octets (cette valeur est un peu plus petite que `BLCKSZ/4`, soit à peu près 2000 octets par défaut). Chaque morceau est stocké comme une ligne séparée dans la table TOAST de la table propriétaire. Chaque table TOAST contient les colonnes `chunk_id` (un OID identifiant la valeur TOASTée particulière), `chunk_seq` (un numéro de séquence pour le morceau de la valeur) et `chunk_data` (la donnée réelle du morceau). Un index unique sur `chunk_id` et `chunk_seq` offre une récupération rapide des valeurs. Un pointeur datum représentant une valeur TOASTée hors−ligne a par conséquent besoin de stocker l'OID de la table TOAST dans laquelle chercher et l'OID de la valeur spécifique (son `chunk_id`). Par commodité, les pointeurs datums stockent aussi la taille logique du datum (taille de la donnée originale non compressée) et la taille stockée réelle (différente si la compression a été appliquée). A partir du mot d'en−tête varlena, la taille totale d'un pointeur datum TOAST est par conséquent de 20 octets quelque soit la taille réelle de la valeur représentée.

Le code TOAST est déclenché seulement quand une valeur de ligne à stocker dans une table est plus grande que `BLCKSZ/4` octets (habituellement 2 Ko). Le code TOAST compressera et/ou déplacera les valeurs de champ hors la ligne jusqu'à ce que la valeur de la ligne soit plus petite que `BLCKSZ/4` octets ou que plus aucun gain ne puisse être réalisé. Lors d'une opération UPDATE, les valeurs des champs non modifiées sont habituellement préservées telles quelles ; donc un UPDATE sur une ligne avec des valeurs hors ligne n'induit pas de coûts à cause de TOAST si aucune des valeurs hors−ligne n'est modifiée.

Le code TOAST connaît quatre stratégies différentes pour stocker les colonnes TOAST−ables :

- `PLAIN` empêche soit la compression soit le stockage hors−ligne. Ceci est la seule stratégie possible pour les colonnes des types de données non TOAST−ables.
- `EXTENDED` permet à la fois la compression et le stockage hors−ligne. Ceci est la valeur par défaut de la plupart des types de données TOAST−ables. La compression sera tentée en premier, ensuite le stockage hors−ligne si la ligne est toujours trop grande.
- `EXTERNAL` autorise le stockage hors−ligne mais pas la compression. L'utilisation d'`EXTERNAL` rendra plus rapides les opérations sur des sous−chaînes d'importantes colonnes de type `text` et `bytea` (au dépens d'un espace de stockage accrus) car ces opérations sont optimisées pour récupérer seulement les parties requises de la valeur hors−ligne lorsqu'elle n'est pas compressée.
- `MAIN` autorise la compression mais pas le stockage hors−ligne. (En réalité le stockage hors−ligne sera toujours réalisé pour de telles colonnes mais seulement en dernier ressort s'il n'existe aucune autre solution pour diminuer suffisament la taille de la ligne.)

Chaque type de données TOAST−able spécifie une stratégie par défaut pour les colonnes de ce type de donnée, mais la stratégie pour une colonne d'une table donnée peut être modifiée avec `ALTER TABLE SET STORAGE`.

Cette combinaison a de nombreux avantages comparés à une approche plus directe comme autoriser le stockage des valeurs de lignes sur plusieurs pages. En supposant que les requêtes sont habituellement

qualifiées par comparaison avec des valeurs de clé relativement petites, la grosse partie du travail de l'exécuteur sera réalisée en utilisant l'entrée principale de la ligne. Les grandes valeurs des attributs TOASTés seront seulement récupérées (si elles sont sélectionnées) au moment où l'ensemble de résultats est envoyé au client. Ainsi, la table principale est bien plus petite et un plus grand nombre de ses lignes tiennent dans le cache du tampon partagé, ce qui ne serait pas le cas sans aucun stockage hors−ligne. Le tri l'utilise aussi, et les tris seront plus souvent réalisés entièrement en mémoire. Un petit test a montré qu'une table contenant des pages HTML typiques ainsi que leurs URL étaient stockées en à peu près la moitié de la taille des données brutes en incluant la table TOAST et que la table principale contenait moins de 10 % de la totalité des données (les URL et quelques petites pages HTML). Il n'y avait pas de différence à l'exécution en comparaison avec une table non TOASTée, dans laquelle toutes les pages HTLM avaient été coupées à 7 Ko pour tenir.

## 49.3. Emplacement des pages de la base de données

Cette section fournit un aperçu du format des pages utilisées par les tables et index de PostgreSQL.[8] Les séquences et les tables TOAST tables sont formatées comme des tables standards.

Dans l'explication qui suit, un *octet* contient huit bits. De plus, le terme *élément* fait référence à une valeur de données individuelle qui est stockée dans une page. Dans une table, un élément est une ligne ; dans un index, un élément est une entrée d'index.

Chaque table et index est stocké comme un tableau de *pages* d'une taille fixe (habituellement 8 Ko, bien qu'une taille de page différente peut être sélectionnée lors de la compilation du serveur). Dans une table, toutes les pages sont logiquement équivalentes pour qu'un élément (ligne) particulier puisse être stocké dans n'importe quelle page. Dans les index, la première page est généralement réservée comme *métapage* contenant des informations de contrôle, et il peut exister différents types de pages à l'intérieur de l'index, suivant la méthode d'accès à l'index.

Tableau 49−2 affiche le contenu complet d'une page. Il existe cinq parties pour chaque page.

**Tableau 49−2. Disposition générale d'une page**

| Élément | Description |
|---|---|
| PageHeaderData | Longueur de 20 octets. Contient des informations générales sur la page y compris des pointeurs sur les espaces libres. |
| ItemPointerData | Tableau de paires (décalage,longueur) pointant sur les éléments réels. Quatre octets par élément. |
| Free space | L'espace non alloué. Les pointeurs de nouveaux éléments sont alloués à partir du début de cette région, les nouveaux éléments à partir de la fin. |
| Items | Les éléments eux−mêmes. |
| Special space | Données spécifiques des méthodes d'accès aux index. Différentes méthodes stockent différentes données. Vide pour les tables ordinaires. |

Les vingt premiers octets de chaque page consistent en un en−tête de page (PageHeaderData). Son format est détaillé dans Tableau 49−3. Les deux premiers champs traquent l'entrée WAL la plus récente relative à cette page. Ils sont suivis par trois champs d'entiers sur deux octets (`pd_lower`, `pd_upper` et `pd_special`). Ils contiennent des décalages d'octets à partir du début de la page jusqu'au début de l'espace non alloué, jusqu'à la fin de l'espace non alloué, et jusqu'au début de l'espace spécial. Les deux derniers octets de l'en−tête de page, `pd_pagesize_version`, stockent à la fois la taille de la page et un indicateur de versoin. A partir de la version 8.0 de PostgreSQL, le numéro de version est 2 ; PostgreSQL 7.3 et 7.4 ont

utilisé le numéro de version 1 ; les versions précédentes utilisaient le numéro de version 0. (La disposition fondamentale de la page et le format de l'en−tête n'ont pas changé dans ces versions mais la disposition de l'en−tête des lignes de tête a changé.) La taille de la page est seulement présente comme vérification croisée ; il n'existe pas de support pour avoir plus d'une taille de page dans une installation.

**Tableau 49−3. Disposition de PageHeaderData**

| Champ | Type | Longueur | Description |
|---|---|---|---|
| pd_lsn | XLogRecPtr | 8 octets | LSN : octet suivant le dernier octet de l'enregistrement xlog pour la dernière modification de cette page |
| pd_tli | TimeLineID | 4 octets | TLI de la dernière modification |
| pd_lower | LocationIndex | 2 octets | Décalage jusqu'au début de l'espace libre |
| pd_upper | LocationIndex | 2 octets | Décalage jusqu'à la fin de l'espace libre |
| pd_special | LocationIndex | 2 octets | Décalage jusqu'au début de l'espace spécial |
| pd_pagesize_version | uint16 | 2 octets | Taille de la page et disposition de l'information du numéro de version |

Tous les détails se trouvent dans `src/include/storage/bufpage.h`.

Après l'en−tête de la page se trouvent les identificateurs d'élément (`ItemIdData`), chacun nécessitant quatre octets. Un identificateur d'élément contient un décalage d'octet vers le début d'un élément, sa longueur en octets, et quelques bits d'attributs qui affectent son interprétation. Les nouveaux identificateurs d'éléments sont alloués si nécessaire à partir du début de l'espace non alloué. Le nombre d'identificateurs d'éléments présents peut être déterminé en regardant `pd_lower`, qui est augmenté pour allouer un nouvel identificateur. Comme un identificateur d'élément n'est jamais déplacé tant qu'il n'est pas libéré, son index pourrait être utilisé sur une base à long terme pour référencer un élément, même quand l'élément lui−même est déplacé le long de la page pour compresser l'espace libre. En fait, chaque pointeur vers un élément (`ItemPointer`, aussi connu sous le nom de `CTID`), créé par PostgreSQL consiste en un numéro de page et l'index de l'identificateur d'élément.

Les éléments eux−mêmes sont stockés dans l'espace alloué en marche arrière, à partir de la fin de l'espace non alloué. La structure exacte varie suivant le contenu de la table. Les tables et les séquences utilisent toutes les deux une structure nommée `HeapTupleHeaderData`, décrite ci−dessous.

La section finale est la << section spéciale >> qui pourrait contenir tout ce que les méthodes d'accès souhaitent stocker. Par exemple, les index b−tree stockent des liens vers les enfants gauche et droit de la page ainsi que quelques autres données sur la structure de l'index. Les tables ordinaires n'utilisent pas du tout de section spéciale (indiquée en configurant `pd_special` à la taille de la page).

Toutes les lignes de la table sont structurées de la même façon. Il existe un en−tête à taille fixe (occupant 27 octets sur la plupart des machines), suivi par un bitmap NULL optionnel, un champ ID de l'objet optionnel et les données de l'utilisateur. L'en−tête est détaillé dans Tableau 49−4. Les données réelles de l'utilisateur (les colonnes de la ligne) commencent àu décalage indiqué par `t_hoff`, qui doit toujours être un multiple de la distance MAXALIGN pour la plateforme. Le bitmap NULL est seulement présent si le bit *HEAP_HASNULL* est initialisé dans `t_infomask`. S'il est présent, il commence juste après l'en−tête fixe et occupe suffisamment d'octets pour avoir un bit par colonne de données (c'est−à−dire `t_natts` bits ensemble). Dans cette liste de bits, un bit 1 indique une valeur non NULL, un bit 0 une valeur NULL. Quand le bitmap n'est pas présent, toutes les colonnes sont supposées non NULL. L'ID de l'objet est seulement présent si le bit *HEAP_HASOID* est initialisé dans `t_infomask`. S'il est présent, il apparaît juste avant la limite `t_hoff`.

Tout ajout nécessaire pour faire de `t_hoff` un multiple de MAXALIGN apparaîtra entre le bitmap NULL et l'ID de l'objet. (Ceci nous assure en retour que l'ID de l'objet est convenablement aligné.)

**Tableau 49−4. Disposition de HeapTupleHeaderData**

| Champ | Type | Longueur | Description |
|---|---|---|---|
| t_xmin | TransactionId | 4 octets | insère le tampon XID |
| t_cmin | CommandId | 4 octets | insère le tampon CID |
| t_xmax | TransactionId | 4 octets | supprime le tampon XID |
| t_cmax | CommandId | 4 octets | supprime le tampon CID (surcharge avec t_xvac) |
| t_xvac | TransactionId | 4 octets | XID pour l'opération VACUUM déplaçant une version de ligne |
| t_ctid | ItemPointerData | 6 octets | TID en cours pour cette version de ligne ou pour une version plus récente |
| t_natts | int16 | 2 octets | nombre d'attributs |
| t_infomask | uint16 | 2 octets | différents bits d'options (flag bits) |
| t_hoff | uint8 | 1 octet | décalage vers les données utilisateur |

Tous les détails sont disponibles dans `src/include/access/htup.h`.

Interpréter les données réelles peut seulement se faire avec des informations obtenues à partir d'autres tables, principalement `pg_attribute`. Les valeurs clés nécessaires pour identifier les emplacements des champs sont `attlen` et `attalign`. Il n'existe aucun moyen pour obtenir directement un attribut particulier, sauf quand il n'y a que des champs de largeur fixe et aucune colonne NULL. Tout ceci est emballé dans les fonctions *heap_getattr*, *fastgetattr* et *heap_getsysattr*.

Pour lire les données, vous avez besoin d'examinez chaque attribut à son tour. Commencez par vérifier si le champ est NULL en fonction du bitmap NULL. S'il l'est, allez au suivant. Puis, assurez−vous que vous avez le bon alignement. Si le champ est un champ à taille fixe, alors tous les octets sont placés simplement. S'il s'agit d'un champ à taille variable (attlen = −1), alors c'est un peu plus compliqué. Tous les types de données à longueur variable partagent la même structure commune d'en−tête, `varattrib`, qui inclut la longueur totale de la valeur stockée et quelques bits d'option. Suivant les options, les données pourraient être soit dans la table de base soit dans une table TOAST ; elles pourraient aussi être compressées (voir Section 49.2).

# Chapitre 50. Interface du moteur BKI

Les fichiers d'interface du moteur (BKI) sont des scripts écrits dans un langage spécial, qui est compris par le serveur PostgreSQL quand il est lancé dans le mode spécial << bootstrap >>. Ce mode autorise la création des catalogues systèmes alors que les commandes SQL requièrent l'existence des catalogues. Les fichiers BKI peuvent donc être utilisés pour créer le système de base de données en tout premier lieu. (Et ils ne sont probablement pas utiles pour un autre emploi.)

initdb utilise un fichier BKI pour faire une partie de son travail lors de la création d'un nouveau groupe de base de données. Le fichier en entrée utilisé par initdb est créé lors de la construction et de l'installation de PostgreSQL par un programme nommé `genbki.sh` qui lit les quelques fichiers d'en−têtes C spécialement formatés dans le répertoire `src/include/catalog` des sources. Le fichier BKI créé est appelé `postgres.bki` et est normalement installé dans le sous−répertoire `share` du répertoire d'installation.

Des informations plus complètes sont disponibles dans la documentation pour initdb.

## 50.1. Format des fichiers BKI

Cette section décrit comment le serveur PostgreSQL interprète les fichiers BKI. Cette description sera plus simple à comprendre si le fichier `postgres.bki` se trouve à portée de main comme exemple.

L'entrée BKI consiste en une séquence de commandes. Les commandes sont constituées d'un certain nombre d'éléments, suivant la syntaxe de la commande. Les éléments sont habituellement séparés par des espaces blancs mais n'ont pas besoin de l'être s'il n'y a pas d'ambiguïté. Il n'existe pas de séparateur spécial pour les commandes ; le prochain élément qui ne peut syntaxiquement pas appartenir à la commande précédente en lance une autre. (Habituellement, vous devriez mettre une nouvelle commande sur une nouvelle ligne pour plus de clarté.) Les éléments peuvent être certains mots clés, des caractères spéciaux (parenthèses, virgules, etc.), nombres ou chaînes de caractères entre guillemets doubles. Tous sont sensibles à la casse.

Les lignes commençant avec un # sont ignorées.

## 50.2. Commandes BKI

create [bootstrap] [shared_relation] [without_oids] *`nomtable`* (*`nom1`* = *`type1`* [, *`nom2`* = *`type2`*, ...])
: Crée une table nommée *`nomtable`* avec les colonnes données entre parenthèses.

  Les types de colonnes suivants sont supportés directement par `bootstrap.c`: `bool`, `bytea`, `char` (1 byte), `name`, `int2`, `int4`, `regproc`, `regclass`, `regtype`, `text`, `oid`, `tid`, `xid`, `cid`, `int2vector`, `oidvector`, `_int4` (array), `_text` (array), `_aclitem` (array). Bien qu'il soit possible de créer des tables contenant des colonnes d'autres types, ceci ne peut pas être fait tant que `pg_type` n'est pas créé et rempli avec les entrées appropriées.

  Quand `bootstrap` est spécifié, la table sera seulement construite sur disque ; rien n'est saisi dans `pg_class`, `pg_attribute`, etc, pour elle. Du coup, la table ne sera pas accessible par des opérations SQL standards jusqu'à ce que de nouvelles entrées sont réalisées (avec des commandes `insert`). Cette option est utilisée pour créer `pg_class`, etc.

  La table est créée comme partagée si `shared_relation` est spécifié. Elle aura des OID sauf si

`without_oids` est spécifié.

open *`nomtable`*
: Ouvre la table nommée *`nomtable`* pour plus de manipulations.

close [*`nomtable`*]
: Ferme la table ouverte appelée *`nomtable`*. Une erreur survient si *`nomtable`* n'est pas déjà ouverte. Si *`nomtable`* n'est pas indiqué, alors la table actuellement ouverte est fermée.

insert [OID = *`valeur_oid`*] (*`valeur1 valeur2`* ...)
: Insère une nouvelle ligne dans la table ouverte en utilisant *`valeur1`*, *`valeur2`*, etc., comme valeurs de ses colonnes et *`valeur_oid`* pour son OID. Si *`valeur_oid`* vaut zéro (0) ou si la clause est omise, alors le prochain OID disponible est utilisé.

  Les valeurs NULL peuvent être indiquées en utilisant le mot clé spécial `_null_`. Les valeurs contenant des espaces doivent être entre des guillemets doubles.

declare [unique] index *`nomindex`* on *`nomtable`* using *`nomma`* (*`classeop1 nom1`* [, ...])
: Crée un index nommé *`nomindex`* sur la table nommée *`nomtable`* en utilisant la méthode d'accès nommée *`nomma`*. Les champs de l'index sont appelés *`nom1`*, *`nom2`* etc., et les classes d'opérateur à utiliser sont respectivement *`classeop1`*, *`classeop2`* etc. Le fichier index est créé et les entrées du catalogue appropriées sont ajoutées pour lui, mais le contenu de l'index n'est pas initialisé par cette commande.

build indices
: Remplit les index précédemment déclarés.

## 50.3. Exemple

La séquence suivante de commandes créera la table `test_table` avec deux colonnes `cola` et `colb` de type, respectivement, `int4` et `text` et insèrera deux lignes dans la table.

```
create test_table (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

# VIII. Annexes

**Table des matières**

# Annexe A. Codes d'erreurs de PostgreSQL

Tous les messages émis par le serveur PostgreSQL se voient affectés des codes d'erreur sur cinq caractères suivant les conventions du standard SQL pour les codes << SQLSTATE >>. Les applications qui ont besoin de savoir quelle condition d'erreur est survenue devraient normalement tester le code d'erreur plutôt que de chercher le message d'erreur textuel. Les codes d'erreurs sont moins sujets à changement suivant les versions de PostgreSQL et ne sont pas sujets aux changements dûs à la localisation des messages d'erreur. Notez que seulement certains des codes d'erreur produit par PostgreSQL sont définis par le standard SQL ; quelques codes d'erreur supplémentaires pour les conditions non définies par le standard ont été inventés ou empruntés aux autres bases de données.

Suivant le standard, les deux premiers caractères d'un code d'erreur dénote une classe d'erreurs alors que les trois derniers indiquent une condition spécifique à l'intérieur de cette classe. Du coup, une application ne reconnaissant pas le code d'erreur spécifique pourrait toujours être capable d'inférer ce qu'il convient de faire d'après la classe de l'erreur.

Tableau A−1 liste tous les codes d'erreurs définis dans PostgreSQL 8.0.5. (Certains ne sont pas réellement utilisés mais sont définis par le standard SQL.) Les classes d'erreurs sont aussi affichées. Pour chaque classe d'erreur, il y a un code d'erreur << standard >> ayant les trois derniers caractères `000`. Ce code est utilisé seulement pour les conditions d'erreurs qui tombent dans la classe mais n'ont pas de code plus spécifique affecté.

Le nom de la condition PL/pgSQL pour chaque code d'erreur est le même que la phrase affichée dans la table, les tirets bas étant substitués aux espaces. Par exemple, le code `22012`, DIVISION BY ZERO, a comme nom de condition `DIVISION_BY_ZERO`. Les noms de condition peuvent être écrit en majuscule ou en minuscule. (Notez que PL/pgSQL ne reconnaît pas les messages d'avertissement, contrairement aux erreurs ; il s'agit des classes 00, 01 et 02.)

**Tableau A−1. Codes d'erreurs PostgreSQL**

| Code d'erreur | Signification |
|---|---|
| Class 00 | Successful Completion |
| `00000` | SUCCESSFUL COMPLETION |
| Class 01 | Warning |
| `01000` | WARNING |
| `0100C` | DYNAMIC RESULT SETS RETURNED |
| `01008` | IMPLICIT ZERO BIT PADDING |
| `01003` | NULL VALUE ELIMINATED IN SET FUNCTION |
| `01007` | PRIVILEGE NOT GRANTED |
| `01006` | PRIVILEGE NOT REVOKED |
| `01004` | STRING DATA RIGHT TRUNCATION |
| `01P01` | DEPRECATED FEATURE |
| Class 02 | No Data — this is also a warning class per SQL:1999 |
| `02000` | NO DATA |
| `02001` | NO ADDITIONAL DYNAMIC RESULT SETS RETURNED |

| | |
|---|---|
| Class 03 | SQL Statement Not Yet Complete |
| 03000 | SQL STATEMENT NOT YET COMPLETE |
| Class 08 | Connection Exception |
| 08000 | CONNECTION EXCEPTION |
| 08003 | CONNECTION DOES NOT EXIST |
| 08006 | CONNECTION FAILURE |
| 08001 | SQLCLIENT UNABLE TO ESTABLISH SQLCONNECTION |
| 08004 | SQLSERVER REJECTED ESTABLISHMENT OF SQLCONNECTION |
| 08007 | TRANSACTION RESOLUTION UNKNOWN |
| 08P01 | PROTOCOL VIOLATION |
| Class 09 | Triggered Action Exception |
| 09000 | TRIGGERED ACTION EXCEPTION |
| Class 0A | Feature Not Supported |
| 0A000 | FEATURE NOT SUPPORTED |
| Class 0B | Invalid Transaction Initiation |
| 0B000 | INVALID TRANSACTION INITIATION |
| Class 0F | Locator Exception |
| 0F000 | LOCATOR EXCEPTION |
| 0F001 | INVALID LOCATOR SPECIFICATION |
| Class 0L | Invalid Grantor |
| 0L000 | INVALID GRANTOR |
| 0LP01 | INVALID GRANT OPERATION |
| Class 0P | Invalid Role Specification |
| 0P000 | INVALID ROLE SPECIFICATION |
| Class 21 | Cardinality Violation |
| 21000 | CARDINALITY VIOLATION |
| Class 22 | Data Exception |
| 22000 | DATA EXCEPTION |
| 2202E | ARRAY SUBSCRIPT ERROR |
| 22021 | CHARACTER NOT IN REPERTOIRE |
| 22008 | DATETIME FIELD OVERFLOW |
| 22012 | DIVISION BY ZERO |
| 22005 | ERROR IN ASSIGNMENT |
| 2200B | ESCAPE CHARACTER CONFLICT |
| 22022 | INDICATOR OVERFLOW |
| 22015 | INTERVAL FIELD OVERFLOW |
| 2201E | INVALID ARGUMENT FOR LOGARITHM |
| 2201F | INVALID ARGUMENT FOR POWER FUNCTION |
| 2201G | INVALID ARGUMENT FOR WIDTH BUCKET FUNCTION |
| 22018 | INVALID CHARACTER VALUE FOR CAST |
| 22007 | INVALID DATETIME FORMAT |
| 22019 | INVALID ESCAPE CHARACTER |

| | |
|---|---|
| 2200D | INVALID ESCAPE OCTET |
| 22025 | INVALID ESCAPE SEQUENCE |
| 22010 | INVALID INDICATOR PARAMETER VALUE |
| 22020 | INVALID LIMIT VALUE |
| 22023 | INVALID PARAMETER VALUE |
| 2201B | INVALID REGULAR EXPRESSION |
| 22009 | INVALID TIME ZONE DISPLACEMENT VALUE |
| 2200C | INVALID USE OF ESCAPE CHARACTER |
| 2200G | MOST SPECIFIC TYPE MISMATCH |
| 22004 | NULL VALUE NOT ALLOWED |
| 22002 | NULL VALUE NO INDICATOR PARAMETER |
| 22003 | NUMERIC VALUE OUT OF RANGE |
| 22026 | STRING DATA LENGTH MISMATCH |
| 22001 | STRING DATA RIGHT TRUNCATION |
| 22011 | SUBSTRING ERROR |
| 22027 | TRIM ERROR |
| 22024 | UNTERMINATED C STRING |
| 2200F | ZERO LENGTH CHARACTER STRING |
| 22P01 | FLOATING POINT EXCEPTION |
| 22P02 | INVALID TEXT REPRESENTATION |
| 22P03 | INVALID BINARY REPRESENTATION |
| 22P04 | BAD COPY FILE FORMAT |
| 22P05 | UNTRANSLATABLE CHARACTER |
| Class 23 | Integrity Constraint Violation |
| 23000 | INTEGRITY CONSTRAINT VIOLATION |
| 23001 | RESTRICT VIOLATION |
| 23502 | NOT NULL VIOLATION |
| 23503 | FOREIGN KEY VIOLATION |
| 23505 | UNIQUE VIOLATION |
| 23514 | CHECK VIOLATION |
| Class 24 | Invalid Cursor State |
| 24000 | INVALID CURSOR STATE |
| Class 25 | Invalid Transaction State |
| 25000 | INVALID TRANSACTION STATE |
| 25001 | ACTIVE SQL TRANSACTION |
| 25002 | BRANCH TRANSACTION ALREADY ACTIVE |
| 25008 | HELD CURSOR REQUIRES SAME ISOLATION LEVEL |
| 25003 | INAPPROPRIATE ACCESS MODE FOR BRANCH TRANSACTION |
| 25004 | INAPPROPRIATE ISOLATION LEVEL FOR BRANCH TRANSACTION |
| 25005 | NO ACTIVE SQL TRANSACTION FOR BRANCH TRANSACTION |
| 25006 | READ ONLY SQL TRANSACTION |
| 25007 | SCHEMA AND DATA STATEMENT MIXING NOT SUPPORTED |

| | |
|---|---|
| 25P01 | NO ACTIVE SQL TRANSACTION |
| 25P02 | IN FAILED SQL TRANSACTION |
| Class 26 | Invalid SQL Statement Name |
| 26000 | INVALID SQL STATEMENT NAME |
| Class 27 | Triggered Data Change Violation |
| 27000 | TRIGGERED DATA CHANGE VIOLATION |
| Class 28 | Invalid Authorization Specification |
| 28000 | INVALID AUTHORIZATION SPECIFICATION |
| Class 2B | Dependent Privilege Descriptors Still Exist |
| 2B000 | DEPENDENT PRIVILEGE DESCRIPTORS STILL EXIST |
| 2BP01 | DEPENDENT OBJECTS STILL EXIST |
| Class 2D | Invalid Transaction Termination |
| 2D000 | INVALID TRANSACTION TERMINATION |
| Class 2F | SQL Routine Exception |
| 2F000 | SQL ROUTINE EXCEPTION |
| 2F005 | FUNCTION EXECUTED NO RETURN STATEMENT |
| 2F002 | MODIFYING SQL DATA NOT PERMITTED |
| 2F003 | PROHIBITED SQL STATEMENT ATTEMPTED |
| 2F004 | READING SQL DATA NOT PERMITTED |
| Class 34 | Invalid Cursor Name |
| 34000 | INVALID CURSOR NAME |
| Class 38 | External Routine Exception |
| 38000 | EXTERNAL ROUTINE EXCEPTION |
| 38001 | CONTAINING SQL NOT PERMITTED |
| 38002 | MODIFYING SQL DATA NOT PERMITTED |
| 38003 | PROHIBITED SQL STATEMENT ATTEMPTED |
| 38004 | READING SQL DATA NOT PERMITTED |
| Class 39 | External Routine Invocation Exception |
| 39000 | EXTERNAL ROUTINE INVOCATION EXCEPTION |
| 39001 | INVALID SQLSTATE RETURNED |
| 39004 | NULL VALUE NOT ALLOWED |
| 39P01 | TRIGGER PROTOCOL VIOLATED |
| 39P02 | SRF PROTOCOL VIOLATED |
| Class 3B | Savepoint Exception |
| 3B000 | SAVEPOINT EXCEPTION |
| 3B001 | INVALID SAVEPOINT SPECIFICATION |
| Class 3D | Invalid Catalog Name |
| 3D000 | INVALID CATALOG NAME |
| Class 3F | Invalid Schema Name |
| 3F000 | INVALID SCHEMA NAME |
| Class 40 | Transaction Rollback |
| 40000 | TRANSACTION ROLLBACK |

| | |
|---|---|
| 40002 | TRANSACTION INTEGRITY CONSTRAINT VIOLATION |
| 40001 | SERIALIZATION FAILURE |
| 40003 | STATEMENT COMPLETION UNKNOWN |
| 40P01 | DEADLOCK DETECTED |
| Class 42 | Syntax Error or Access Rule Violation |
| 42000 | SYNTAX ERROR OR ACCESS RULE VIOLATION |
| 42601 | SYNTAX ERROR |
| 42501 | INSUFFICIENT PRIVILEGE |
| 42846 | CANNOT COERCE |
| 42803 | GROUPING ERROR |
| 42830 | INVALID FOREIGN KEY |
| 42602 | INVALID NAME |
| 42622 | NAME TOO LONG |
| 42939 | RESERVED NAME |
| 42804 | DATATYPE MISMATCH |
| 42P18 | INDETERMINATE DATATYPE |
| 42809 | WRONG OBJECT TYPE |
| 42703 | UNDEFINED COLUMN |
| 42883 | UNDEFINED FUNCTION |
| 42P01 | UNDEFINED TABLE |
| 42P02 | UNDEFINED PARAMETER |
| 42704 | UNDEFINED OBJECT |
| 42701 | DUPLICATE COLUMN |
| 42P03 | DUPLICATE CURSOR |
| 42P04 | DUPLICATE DATABASE |
| 42723 | DUPLICATE FUNCTION |
| 42P05 | DUPLICATE PREPARED STATEMENT |
| 42P06 | DUPLICATE SCHEMA |
| 42P07 | DUPLICATE TABLE |
| 42712 | DUPLICATE ALIAS |
| 42710 | DUPLICATE OBJECT |
| 42702 | AMBIGUOUS COLUMN |
| 42725 | AMBIGUOUS FUNCTION |
| 42P08 | AMBIGUOUS PARAMETER |
| 42P09 | AMBIGUOUS ALIAS |
| 42P10 | INVALID COLUMN REFERENCE |
| 42611 | INVALID COLUMN DEFINITION |
| 42P11 | INVALID CURSOR DEFINITION |
| 42P12 | INVALID DATABASE DEFINITION |
| 42P13 | INVALID FUNCTION DEFINITION |
| 42P14 | INVALID PREPARED STATEMENT DEFINITION |
| 42P15 | INVALID SCHEMA DEFINITION |

| | |
|---|---|
| 42P16 | INVALID TABLE DEFINITION |
| 42P17 | INVALID OBJECT DEFINITION |
| Class 44 | WITH CHECK OPTION Violation |
| 44000 | WITH CHECK OPTION VIOLATION |
| Class 53 | Insufficient Resources |
| 53000 | INSUFFICIENT RESOURCES |
| 53100 | DISK FULL |
| 53200 | OUT OF MEMORY |
| 53300 | TOO MANY CONNECTIONS |
| Class 54 | Program Limit Exceeded |
| 54000 | PROGRAM LIMIT EXCEEDED |
| 54001 | STATEMENT TOO COMPLEX |
| 54011 | TOO MANY COLUMNS |
| 54023 | TOO MANY ARGUMENTS |
| Class 55 | Object Not In Prerequisite State |
| 55000 | OBJECT NOT IN PREREQUISITE STATE |
| 55006 | OBJECT IN USE |
| 55P02 | CANT CHANGE RUNTIME PARAM |
| 55P03 | LOCK NOT AVAILABLE |
| Class 57 | Operator Intervention |
| 57000 | OPERATOR INTERVENTION |
| 57014 | QUERY CANCELED |
| 57P01 | ADMIN SHUTDOWN |
| 57P02 | CRASH SHUTDOWN |
| 57P03 | CANNOT CONNECT NOW |
| Class 58 | System Error (errors external to PostgreSQL itself) |
| 58030 | IO ERROR |
| 58P01 | UNDEFINED FILE |
| 58P02 | DUPLICATE FILE |
| Class F0 | Configuration File Error |
| F0000 | CONFIG FILE ERROR |
| F0001 | LOCK FILE EXISTS |
| Class P0 | PL/pgSQL Error |
| P0000 | PLPGSQL ERROR |
| P0001 | RAISE EXCEPTION |
| Class XX | Internal Error |
| XX000 | INTERNAL ERROR |
| XX001 | DATA CORRUPTED |
| XX002 | INDEX CORRUPTED |

# Annexe B. Support de Date/Heure

PostgreSQL utilise un analyseur heuristique interne pour le support des dates/heures données en entrée. Les dates et heures sont entrées en tant que chaînes, et sont découpées en champs distincts avec une détermination préliminaire du type d'information contenu dans le champ. Chaque champ est interpreté et soit il lui est attribué une valeur numérique, soit il est ignoré ou rejeté. Le parseur contient des tables de correspondance internes pour tout les champs textuels y compris les mois, les jours de la semaine et les fuseaux horaires.

Cette annexe contient des informations sur le contenu de ces tables de correspondance et décrit les méthodes utilisées par le parseur pour décoder les dates et heures.

## B.1. Interprétation des Entrées Date/Heure

Les entrées de type date/heure sont toutes décodées en utilisant le processus suivant.

1. Diviser la chaîne d'entrée en marqueurs et cataloguer chaque marqueur en tant que une chaîne, heure, fuseau horaire ou nombre.

    a. Si un marqueur numérique contient les deux points (`:`), ceci est une chaîne d'heure. Inclure tout les chiffres et deux points suivants.
    b. Si un marqueur numérique contient un tiret (`-`), une barre oblique (`/`), ou plus d'un point (`.`), ceci est une chaîne de date qui contient peut être un mois sous forme de texte.
    c. Si un marqueur est uniquement numérique alors il est soit un champ simple ou une date concaténée ISO 8601 (i.e., `19990113` pour le 13 janvier 1999) ou heure (i.e., `141516` pour 14:15:16).
    d. Si le marqueur commence avec un plus (`+`) ou un moins (`-`), alors il est soit un fuseau horaire, soit un champ spécial.
2. Si un marqueur est une chaîne texte, le comparer avec les différentes chaînes possibles.

    a. Faire une recherche binaire dans la table de correspondance pour le marqueur en tant que chaîne spéciale (i.e., `today`), jour (i.e., `Thursday`), mois (i.e., `January`), ou mot bruit (i.e., `at`, `on`).

       Fixer les valeurs pour chaque champ ainsi qu'un masque binaire. Par exemple, fixer l'année, le mois, le jour pour `today`, et aussi l'heure, les minutes, les secondes pour `now`.
    b. Si pas trouvé, faire une recherche binaire similaire dans une table de correspondance pour faire correspondre le marqueur avec un fuseau horaire.
    c. Si toujours pas trouvé, lever une erreur.
3. Quand le marqueur est un nombre ou un champ de nombre:

    a. S'il y a huit ou six chiffres, et aucun autre champ date n'a été lu, alors l'interpréter comme une << date concaténée >> (i.e., `19990118` ou `990118`). L'interprétation est `AAAAMMJJ` ou `AAMMJJ`.
    b. Si le marqueur est de trois chiffres et une année a déjà été lue, alors il sera interprété comme un jour de l'année.
    c. Si quatre ou six chiffres et une année ont déjà été lus, alors il sera interprété comme une heure (`HHMM` or `HHMMSS`).
    d. Si trois chiffres ou plus et aucun champ date n'a été trouvé, il sera interprété comme année (ceci force l'ordre aa-mm-jj des champs dates restants).

e. Sinon l'ordre de champ date est supposé suivre le paramêtre `DateStyle` : mm−jj−aa, jj−mm−aa, ou aa−mm−jj. Lever une erreur si un champ jour ou mois est découvert hors des limites.

4. Si BC a été spécifié, rendre NULL l'année et ajouter un pour le stockage interne. (Il n'y a pas d'année zéro dans le calendrier Grégorien, alors numériquement 1 BC devient l'année zéro.)
5. Si BC n'a pas été spécifié et le champ année fait deux chiffres en longueur, alors ajuster l'année a quatre chiffres. Si le champ vaut moins que 70 alors ajouter 2000, sinon ajouter 1900.

**Astuce :** Les années Grégoriennes AD 1−99 peuvent être entrées en utilisant 4 chiffres avec les zéros de devant (i.e., `0099` est AD 99). Les versions précédentes de PostgreSQL acceptaient les années avec trois chiffres et avec un seul chiffre mais depuis la version 7.0, les regles ont été renforcées pour réduire de possibles ambiguïtés.

## B.2. Mots clés Date/Heure

Tableau B−1 montre les marqueurs qui sont reconnus commme les noms des mois.

**Tableau B−1. Noms des mois**

| Mois | Abréviations |
|---|---|
| Janvier | Jan |
| Février | Feb |
| Mars | Mar |
| Avril | Apr |
| Mai | |
| Juin | Jun |
| Juillet | Jul |
| Août | Aug |
| Septembre | Sep, Sept |
| Octobre | Oct |
| Novembre | Nov |
| Décembre | Dec |

Tableau B−2 montre les marqueurs qui sont reconnus comme les noms des jours de la semaine.

**Tableau B−2. Noms des jours de la semaine**

| Jour | Abréviation |
|---|---|
| Dimanche | Sun |
| Lundi | Mon |
| Mardi | Tue, Tues |
| Mercredi | Wed, Weds |
| Jeudi | Thu, Thur, Thurs |
| Vendredi | Fri |

| | |
|---|---|
| Samedi | Sat |

Tableau B−3 montre les marqueurs qui servent à modifier divers contenus.

**Tableau B−3. Modificateurs de Champs Date/Heure**

| Identifiant | Description |
|---|---|
| `ABSTIME` | Ignoré |
| `AM` | Heure avant 12:00 |
| `AT` | Ignoré |
| `JULIAN`, `JD`, `J` | Le champ suivant est un jour Julien |
| `ON` | Ignoré |
| `PM` | Heure à ou après 12:00 |
| `T` | Le champ suivant est de temps |

Le mot clé `ABSTIME` est ignoré pour des raisons historiques: Dans de très vieilles versions de PostgreSQL, des valeurs invalides de type `abstime` étaient émises en tant que `Invalid Abstime`. Par contre, ceci n'est plus le cas et ce mot clé sera probablement abandonné dans une future version.

Tableau B−4 montre les abréviations de fuseau horaire reconnus par PostgreSQL pour les valeurs de type date/time. Notez que ces noms ne sont *pas* nécessairement utilisés pour l'affichage de date/heure — la sortie est pilotée par les abréviations officielles des fuseaux horaires associées avec le paramètre de configuration timezone actuellement sélectionné. (Il est probable que les prochaines versions utiliseront aussi `timezone` pour les entrées.)

Ce tableau est organisé par décalage du fuseau horaire à partir de UTC, plutôt que par ordre alphabétique. Ceci a été choisi pour faciliter la correspondance entre l'usage local et les abréviations reconnues là ou elles seraient différentes.

**Tableau B−4. Abréviations de fuseaux horaires en entrée**

| Fuseau horaire | Décalage à partir d'UTC | Description |
|---|---|---|
| NZDT | +13:00 | New Zealand Daylight−Saving Time |
| IDLE | +12:00 | International Date Line, East |
| NZST | +12:00 | New Zealand Standard Time |
| NZT | +12:00 | New Zealand Time |
| AESST | +11:00 | Australia Eastern Summer Standard Time |
| ACSST | +10:30 | Central Australia Summer Standard Time |
| CADT | +10:30 | Central Australia Daylight−Saving Time |
| SADT | +10:30 | South Australian Daylight−Saving Time |
| AEST | +10:00 | |

| | | |
|---|---|---|
| | | Australia Eastern Standard Time |
| EAST | +10:00 | East Australian Standard Time |
| GST | +10:00 | Guam Standard Time, Russia zone 9 |
| LIGT | +10:00 | Melbourne, Australia |
| SAST | +09:30 | South Australia Standard Time |
| CAST | +09:30 | Central Australia Standard Time |
| AWSST | +09:00 | Australia Western Summer Standard Time |
| JST | +09:00 | Japan Standard Time, Russia zone 8 |
| KST | +09:00 | Korea Standard Time |
| MHT | +09:00 | Kwajalein Time |
| WDT | +09:00 | West Australian Daylight−Saving Time |
| MT | +08:30 | Moluccas Time |
| AWST | +08:00 | Australia Western Standard Time |
| CCT | +08:00 | China Coastal Time |
| WADT | +08:00 | West Australian Daylight−Saving Time |
| WST | +08:00 | West Australian Standard Time |
| JT | +07:30 | Java Time |
| ALMST | +07:00 | Almaty Summer Time |
| WAST | +07:00 | West Australian Standard Time |
| CXT | +07:00 | Christmas (Island) Time |
| MMT | +06:30 | Myanmar Time |
| ALMT | +06:00 | Almaty Time |
| MAWT | +06:00 | Mawson (Antarctica) Time |
| IOT | +05:00 | Indian Chagos Time |
| MVT | +05:00 | Maldives Island Time |
| TFT | +05:00 | Kerguelen Time |
| AFT | +04:30 | Afghanistan Time |
| EAST | +04:00 | Antananarivo Summer Time |
| MUT | +04:00 | Mauritius Island Time |
| RET | +04:00 | Reunion Island Time |
| SCT | +04:00 | Mahe Island Time |
| IRT, IT | +03:30 | Iran Time |
| EAT | +03:00 | Antananarivo, Comoro Time |
| BT | +03:00 | Baghdad Time |
| EETDST | +03:00 | |

| | | |
|---|---|---|
| | | Eastern Europe Daylight−Saving Time |
| HMT | +03:00 | Hellas Mediterranean Time (?) |
| BDST | +02:00 | British Double Summer Time |
| CEST | +02:00 | Central European Summer Time |
| CETDST | +02:00 | Central European Daylight−Saving Time |
| EET | +02:00 | Eastern European Time, Russia zone 1 |
| FWT | +02:00 | French Winter Time |
| IST | +02:00 | Israel Standard Time |
| MEST | +02:00 | Middle European Summer Time |
| METDST | +02:00 | Middle Europe Daylight−Saving Time |
| SST | +02:00 | Swedish Summer Time |
| BST | +01:00 | British Summer Time |
| CET | +01:00 | Central European Time |
| DNT | +01:00 | *Dansk Normal Tid* |
| FST | +01:00 | French Summer Time |
| MET | +01:00 | Middle European Time |
| MEWT | +01:00 | Middle European Winter Time |
| MEZ | +01:00 | *Mitteleuropéische Zeit* |
| NOR | +01:00 | Norway Standard Time |
| SET | +01:00 | Seychelles Time |
| SWT | +01:00 | Swedish Winter Time |
| WETDST | +01:00 | Western European Daylight−Saving Time |
| GMT | 00:00 | Greenwich Mean Time |
| UT | 00:00 | Universal Time |
| UTC | 00:00 | Universal Coordinated Time |
| Z | 00:00 | Same as UTC |
| ZULU | 00:00 | Same as UTC |
| WET | 00:00 | Western European Time |
| WAT | −01:00 | West Africa Time |
| FNST | −01:00 | Fernando de Noronha Summer Time |
| FNT | −02:00 | Fernando de Noronha Time |
| BRST | −02:00 | Brasilia Summer Time |
| NDT | −02:30 | Newfoundland Daylight−Saving Time |
| ADT | −03:00 | Atlantic Daylight−Saving Time |

| | | |
|---|---|---|
| AWT | –03:00 | (unknown) |
| BRT | –03:00 | Brasilia Time |
| NFT | –03:30 | Newfoundland Standard Time |
| NST | –03:30 | Newfoundland Standard Time |
| AST | –04:00 | Atlantic Standard Time (Canada) |
| ACST | –04:00 | Atlantic/Porto Acre Summer Time |
| EDT | –04:00 | Eastern Daylight–Saving Time |
| ACT | –05:00 | Atlantic/Porto Acre Standard Time |
| CDT | –05:00 | Central Daylight–Saving Time |
| EST | –05:00 | Eastern Standard Time |
| CST | –06:00 | Central Standard Time |
| MDT | –06:00 | Mountain Daylight–Saving Time |
| MST | –07:00 | Mountain Standard Time |
| PDT | –07:00 | Pacific Daylight–Saving Time |
| AKDT | –08:00 | Alaska Daylight–Saving Time |
| PST | –08:00 | Pacific Standard Time |
| YDT | –08:00 | Yukon Daylight–Saving Time |
| AKST | –09:00 | Alaska Standard Time |
| HDT | –09:00 | Hawaii/Alaska Daylight–Saving Time |
| YST | –09:00 | Yukon Standard Time |
| MART | –09:30 | Marquesas Time |
| AHST | –10:00 | Alaska/Hawaii Standard Time |
| HST | –10:00 | Hawaii Standard Time |
| CAT | –10:00 | Central Alaska Time |
| NT | –11:00 | Nome Time |
| IDLW | –12:00 | International Date Line, West |

**Australian Time Zones.** Il y a trois conflits de nommage entre les noms de fuseaux horaires Australiens et les noms de fuseaux horaires utilisés Amérique du Sud et du Nord: `ACST`, `CST` et `EST`. Si l'option de démarrage `australian_timezones` est mis à vrai alors `ACST`, `CST`, `EST` et `SAT` sont interprétés comme des noms de fuseaux horaires Australiens, comme montré dans Tableau B–5. Si ceci est faux (ce qui est le cas par défaut) alors `ACST`, `CST`, et `EST` sont pris comme des noms de fuseaux horaires américains et `SAT` est interprété comme un mot peu significatif indiquant Samedi.

**Tableau B–5. Abreviations de fuseaux horaires australiens en entrée**

| Fuseau Horaire | Offset from UTC | Description |
|---|---|---|
| ACST | +09:30 | Temps Standard d'Australie Centrale |
| CST | +10:30 | Australian Central Standard Time |

| | | |
|---|---|---|
| EST | +10:00 | Australian Eastern Standard Time |
| SAT | +09:30 | South Australian Standard Time |

Tableau B−6 affiche les noms de fuseau horaire reconnu par PostgreSQL comme les valeurs valides pour le paramètre timezone parameter. Notez que ces noms sont conceptuellement aussi différents des noms montrés dans Tableau B−4 : la plupart de ces noms implique une règle locale pour l'avancement de l'heure alors que les noms précédents représentent seulement un décalage fixé à partir d'UTC.

Dans beaucoup de cas, il existe plusieurs noms équivalents pour la même zone. Ils sont affichés sur la même ligne. La table est triée principalement par le nom de la ville principale de la zone.

**Tableau B−6. Noms des zones de fuseaux horaires pour la configuration de `timezone`**

| Fuseau horaire |
|---|
| Africa/Abidjan |
| Africa/Accra |
| Africa/Addis_Ababa |
| Africa/Algiers |
| Africa/Asmera |
| Africa/Bamako |
| Africa/Bangui |
| Africa/Banjul |
| Africa/Bissau |
| Africa/Blantyre |
| Africa/Brazzaville |
| Africa/Bujumbura |
| Africa/Cairo Egypt |
| Africa/Casablanca |
| Africa/Ceuta |
| Africa/Conakry |
| Africa/Dakar |
| Africa/Dar_es_Salaam |
| Africa/Djibouti |
| Africa/Douala |
| Africa/El_Aaiun |
| Africa/Freetown |
| Africa/Gaborone |
| Africa/Harare |
| Africa/Johannesburg |
| Africa/Kampala |
| Africa/Khartoum |
| Africa/Kigali |
| Africa/Kinshasa |

| |
|---|
| Africa/Lagos |
| Africa/Libreville |
| Africa/Lome |
| Africa/Luanda |
| Africa/Lubumbashi |
| Africa/Lusaka |
| Africa/Malabo |
| Africa/Maputo |
| Africa/Maseru |
| Africa/Mbabane |
| Africa/Mogadishu |
| Africa/Monrovia |
| Africa/Nairobi |
| Africa/Ndjamena |
| Africa/Niamey |
| Africa/Nouakchott |
| Africa/Ouagadougou |
| Africa/Porto−Novo |
| Africa/Sao_Tome |
| Africa/Timbuktu |
| Africa/Tripoli Libya |
| Africa/Tunis |
| Africa/Windhoek |
| America/Adak America/Atka US/Aleutian |
| America/Anchorage SystemV/YST9YDT US/Alaska |
| America/Anguilla |
| America/Antigua |
| America/Araguaina |
| America/Aruba |
| America/Asuncion |
| America/Bahia |
| America/Barbados |
| America/Belem |
| America/Belize |
| America/Boa_Vista |
| America/Bogota |
| America/Boise |
| America/Buenos_Aires |
| America/Cambridge_Bay |
| America/Campo_Grande |
| America/Cancun |
| America/Caracas |

| |
|---|
| America/Catamarca |
| America/Cayenne |
| America/Cayman |
| America/Chicago CST6CDT SystemV/CST6CDT US/Central |
| America/Chihuahua |
| America/Cordoba America/Rosario |
| America/Costa_Rica |
| America/Cuiaba |
| America/Curacao |
| America/Danmarkshavn |
| America/Dawson |
| America/Dawson_Creek |
| America/Denver MST7MDT SystemV/MST7MDT US/Mountain America/Shiprock Navajo |
| America/Detroit US/Michigan |
| America/Dominica |
| America/Edmonton Canada/Mountain |
| America/Eirunepe |
| America/El_Salvador |
| America/Ensenada America/Tijuana Mexico/BajaNorte |
| America/Fortaleza |
| America/Glace_Bay |
| America/Godthab |
| America/Goose_Bay |
| America/Grand_Turk |
| America/Grenada |
| America/Guadeloupe |
| America/Guatemala |
| America/Guayaquil |
| America/Guyana |
| America/Halifax Canada/Atlantic SystemV/AST4ADT |
| America/Havana Cuba |
| America/Hermosillo |
| America/Indiana/Indianapolis America/Indianapolis America/Fort_Wayne EST SystemV/EST5 US/East−Indiana |
| America/Indiana/Knox America/Knox_IN US/Indiana−Starke |
| America/Indiana/Marengo |
| America/Indiana/Vevay |
| America/Inuvik |
| America/Iqaluit |
| America/Jamaica Jamaica |
| America/Jujuy |
| America/Juneau |

| |
|---|
| America/Kentucky/Louisville America/Louisville |
| America/Kentucky/Monticello |
| America/La_Paz |
| America/Lima |
| America/Los_Angeles PST8PDT SystemV/PST8PDT US/Pacific US/Pacific−New |
| America/Maceio |
| America/Managua |
| America/Manaus Brazil/West |
| America/Martinique |
| America/Mazatlan Mexico/BajaSur |
| America/Mendoza |
| America/Menominee |
| America/Merida |
| America/Mexico_City Mexico/General |
| America/Miquelon |
| America/Monterrey |
| America/Montevideo |
| America/Montreal |
| America/Montserrat |
| America/Nassau |
| America/New_York EST5EDT SystemV/EST5EDT US/Eastern |
| America/Nipigon |
| America/Nome |
| America/Noronha Brazil/DeNoronha |
| America/North_Dakota/Center |
| America/Panama |
| America/Pangnirtung |
| America/Paramaribo |
| America/Phoenix MST SystemV/MST7 US/Arizona |
| America/Port−au−Prince |
| America/Port_of_Spain |
| America/Porto_Acre America/Rio_Branco Brazil/Acre |
| America/Porto_Velho |
| America/Puerto_Rico SystemV/AST4 |
| America/Rainy_River |
| America/Rankin_Inlet |
| America/Recife |
| America/Regina Canada/East−Saskatchewan Canada/Saskatchewan SystemV/CST6 |
| America/Santiago Chile/Continental |
| America/Santo_Domingo |
| America/Sao_Paulo Brazil/East |
| America/Scoresbysund |

| |
|---|
| America/St_Johns Canada/Newfoundland |
| America/St_Kitts |
| America/St_Lucia |
| America/St_Thomas America/Virgin |
| America/St_Vincent |
| America/Swift_Current |
| America/Tegucigalpa |
| America/Thule |
| America/Thunder_Bay |
| America/Toronto Canada/Eastern |
| America/Tortola |
| America/Vancouver Canada/Pacific |
| America/Whitehorse Canada/Yukon |
| America/Winnipeg Canada/Central |
| America/Yakutat |
| America/Yellowknife |
| Antarctica/Casey |
| Antarctica/Davis |
| Antarctica/DumontDUrville |
| Antarctica/Mawson |
| Antarctica/McMurdo Antarctica/South_Pole |
| Antarctica/Palmer |
| Antarctica/Rothera |
| Antarctica/Syowa |
| Antarctica/Vostok |
| Asia/Aden |
| Asia/Almaty |
| Asia/Amman |
| Asia/Anadyr |
| Asia/Aqtau |
| Asia/Aqtobe |
| Asia/Ashgabat Asia/Ashkhabad |
| Asia/Baghdad |
| Asia/Bahrain |
| Asia/Baku |
| Asia/Bangkok |
| Asia/Beirut |
| Asia/Bishkek |
| Asia/Brunei |
| Asia/Calcutta |
| Asia/Choibalsan |
| Asia/Chongqing Asia/Chungking |

| |
|---|
| Asia/Colombo |
| Asia/Dacca Asia/Dhaka |
| Asia/Damascus |
| Asia/Dili |
| Asia/Dubai |
| Asia/Dushanbe |
| Asia/Gaza |
| Asia/Harbin |
| Asia/Hong_Kong Hongkong |
| Asia/Hovd |
| Asia/Irkutsk |
| Asia/Jakarta |
| Asia/Jayapura |
| Asia/Jerusalem Asia/Tel_Aviv Israel |
| Asia/Kabul |
| Asia/Kamchatka |
| Asia/Karachi |
| Asia/Kashgar |
| Asia/Katmandu |
| Asia/Krasnoyarsk |
| Asia/Kuala_Lumpur |
| Asia/Kuching |
| Asia/Kuwait |
| Asia/Macao Asia/Macau |
| Asia/Magadan |
| Asia/Makassar Asia/Ujung_Pandang |
| Asia/Manila |
| Asia/Muscat |
| Asia/Nicosia Europe/Nicosia |
| Asia/Novosibirsk |
| Asia/Omsk |
| Asia/Oral |
| Asia/Phnom_Penh |
| Asia/Pontianak |
| Asia/Pyongyang |
| Asia/Qatar |
| Asia/Qyzylorda |
| Asia/Rangoon |
| Asia/Riyadh |
| Asia/Riyadh87 Mideast/Riyadh87 |
| Asia/Riyadh88 Mideast/Riyadh88 |
| Asia/Riyadh89 Mideast/Riyadh89 |

| |
|---|
| Asia/Saigon |
| Asia/Sakhalin |
| Asia/Samarkand |
| Asia/Seoul ROK |
| Asia/Shanghai PRC |
| Asia/Singapore Singapore |
| Asia/Taipei ROC |
| Asia/Tashkent |
| Asia/Tbilisi |
| Asia/Tehran Iran |
| Asia/Thimbu Asia/Thimphu |
| Asia/Tokyo Japan |
| Asia/Ulaanbaatar Asia/Ulan_Bator |
| Asia/Urumqi |
| Asia/Vientiane |
| Asia/Vladivostok |
| Asia/Yakutsk |
| Asia/Yekaterinburg |
| Asia/Yerevan |
| Atlantic/Azores |
| Atlantic/Bermuda |
| Atlantic/Canary |
| Atlantic/Cape_Verde |
| Atlantic/Faeroe |
| Atlantic/Madeira |
| Atlantic/Reykjavik Iceland |
| Atlantic/South_Georgia |
| Atlantic/St_Helena |
| Atlantic/Stanley |
| Australia/ACT Australia/Canberra Australia/NSW Australia/Sydney |
| Australia/Adelaide Australia/South |
| Australia/Brisbane Australia/Queensland |
| Australia/Broken_Hill Australia/Yancowinna |
| Australia/Darwin Australia/North |
| Australia/Hobart Australia/Tasmania |
| Australia/LHI Australia/Lord_Howe |
| Australia/Lindeman |
| Australia/Melbourne Australia/Victoria |
| Australia/Perth Australia/West |
| CET |
| EET |
| Etc/GMT+1 |

| |
|---|
| Etc/GMT+2 |
| Etc/GMT+3 |
| Etc/GMT+4 |
| Etc/GMT+5 |
| Etc/GMT+6 |
| Etc/GMT+7 |
| Etc/GMT+8 |
| Etc/GMT+9 |
| Etc/GMT+10 |
| Etc/GMT+11 |
| Etc/GMT+12 |
| Etc/GMT−1 |
| Etc/GMT−2 |
| Etc/GMT−3 |
| Etc/GMT−4 |
| Etc/GMT−5 |
| Etc/GMT−6 |
| Etc/GMT−7 |
| Etc/GMT−8 |
| Etc/GMT−9 |
| Etc/GMT−10 |
| Etc/GMT−11 |
| Etc/GMT−12 |
| Etc/GMT−13 |
| Etc/GMT−14 |
| Europe/Amsterdam |
| Europe/Andorra |
| Europe/Athens |
| Europe/Belfast |
| Europe/Belgrade Europe/Ljubljana Europe/Sarajevo Europe/Skopje Europe/Zagreb |
| Europe/Berlin |
| Europe/Brussels |
| Europe/Bucharest |
| Europe/Budapest |
| Europe/Chisinau Europe/Tiraspol |
| Europe/Copenhagen |
| Europe/Dublin Eire |
| Europe/Gibraltar |
| Europe/Helsinki |
| Europe/Istanbul Asia/Istanbul Turkey |
| Europe/Kaliningrad |
| Europe/Kiev |

| |
|---|
| Europe/Lisbon Portugal |
| Europe/London GB GB−Eire |
| Europe/Luxembourg |
| Europe/Madrid |
| Europe/Malta |
| Europe/Minsk |
| Europe/Monaco |
| Europe/Moscow W−SU |
| Europe/Oslo Arctic/Longyearbyen Atlantic/Jan_Mayen |
| Europe/Paris |
| Europe/Prague Europe/Bratislava |
| Europe/Riga |
| Europe/Rome Europe/San_Marino Europe/Vatican |
| Europe/Samara |
| Europe/Simferopol |
| Europe/Sofia |
| Europe/Stockholm |
| Europe/Tallinn |
| Europe/Tirane |
| Europe/Uzhgorod |
| Europe/Vaduz |
| Europe/Vienna |
| Europe/Vilnius |
| Europe/Warsaw Poland |
| Europe/Zaporozhye |
| Europe/Zurich |
| Factory |
| GMT GMT+0 GMT−0 GMT0 Greenwich Etc/GMT Etc/GMT+0 Etc/GMT−0 Etc/GMT0 Etc/Greenwich |
| Indian/Antananarivo |
| Indian/Chagos |
| Indian/Christmas |
| Indian/Cocos |
| Indian/Comoro |
| Indian/Kerguelen |
| Indian/Mahe |
| Indian/Maldives |
| Indian/Mauritius |
| Indian/Mayotte |
| Indian/Reunion |
| MET |
| Pacific/Apia |
| Pacific/Auckland NZ |

| |
|---|
| Pacific/Chatham NZ−CHAT |
| Pacific/Easter Chile/EasterIsland |
| Pacific/Efate |
| Pacific/Enderbury |
| Pacific/Fakaofo |
| Pacific/Fiji |
| Pacific/Funafuti |
| Pacific/Galapagos |
| Pacific/Gambier SystemV/YST9 |
| Pacific/Guadalcanal |
| Pacific/Guam |
| Pacific/Honolulu HST SystemV/HST10 US/Hawaii |
| Pacific/Johnston |
| Pacific/Kiritimati |
| Pacific/Kosrae |
| Pacific/Kwajalein Kwajalein |
| Pacific/Majuro |
| Pacific/Marquesas |
| Pacific/Midway |
| Pacific/Nauru |
| Pacific/Niue |
| Pacific/Norfolk |
| Pacific/Noumea |
| Pacific/Pago_Pago Pacific/Samoa US/Samoa |
| Pacific/Palau |
| Pacific/Pitcairn SystemV/PST8 |
| Pacific/Ponape |
| Pacific/Port_Moresby |
| Pacific/Rarotonga |
| Pacific/Saipan |
| Pacific/Tahiti |
| Pacific/Tarawa |
| Pacific/Tongatapu |
| Pacific/Truk |
| Pacific/Wake |
| Pacific/Wallis |
| Pacific/Yap |
| UCT Etc/UCT |
| UTC Universal Zulu Etc/UTC Etc/Universal Etc/Zulu |
| WET |

En plus des noms affichés dans la table, PostgreSQL acceptera les noms des fuseaux horaires de la forme *STDdécalage* ou *STDdécalageDST*, où *STD* est une abréviation de fuseau, *décalage* est un décalage

numérique en heures à l'ouest d'UTC et *DST* est une abréviation optionnelle de fuseau pour l'avancement de l'heure, en supposant une heure avant le décalage indiqué. Par exemple, si EST5EDT n'était pas déjà un nom reconnu, il serait accepté et équivalent fonctionnellement au temps USA East Coast. Quand un fuseau horaire d'avancement de l'heure est présent, il est supposé être utilisé suivant les règles du fuseau horaire USA, donc cette fonctionnalité est d'une utilité limitée en dehors de l'Amérique du Nord. Vous devez aussi être prévenu que cette provision peut amener silencieusement des entrées acceptées bien que boguées car il n'y a pas de vérification sur la cohérence des abréviations de fuseau. Par exemple, SET TIMEZONE TO FOOBAR0 fonctionnera, laissant réellement le système utiliser une abréviation particulière pour GMT.

# B.3. Histoire d'Unités

Le Julian Date a été inventé par le savant français Joseph Justus Scaliger (1540−1609) et tient probablement son nom du pere de Scaliger, le scholar Italian Julius Caesar Scaliger (1484−1558). Les astronomes ont utilisés la periode Julian pour donner un nombre unique a chaque jour depuis le 1 Janvier 4713 BC. ceci est la date dite Julian (JD). JD 0 designe les 24 heures de midi UTC le 1 Janvier 4713 BC jusqu'à midi UTC le 2 Janvier 4713 BC.

La << Date Julian >> est différente du << calendrier Julian >>. Le calendrier Julian a été introduit par Julius Caesar en 45 BC. Il était utilisé de maniére courante jusqu'en l'an 1582, ou des pays ont commencés à se convertir au calendrier Grégorien. Dans le calendrier Julian, l'année tropical est arrondie à 365 1/4 jours = 365.25 jours. ceci donne une erreur d'à peu près 1 jour tout les 128 ans.

L'erreur de calendrier grandissante poussa le Pape Gregory XIII a réformé le calendrier en accord avec les instructions du Conseil de Trente. Dans le calendrier Grégorien, l'année tropical est arrondie en 365 + 97 / 400 jours = 365.2425 jours. Donc, il faut à peu prés 3300 années pour l'année tropical pour virer d'un jour par rapport au calendrier Grégorien.

L'arrondi 365+97/400 est obtenu en ayant 97 années bissextiles tous les 400 ans, en utilisant les régles suivantes:

Toutes les années divisibles par 4 sont des années bissextiles.

Par contre, toutes les années divisibles par 100 ne sont pas des années bissextiles.

Par contre, toutes les années divisibles par 400 sont, finalement, des années bissextiles.

Donc 1700, 1800, 1900, 2100 et 2200 ne sont pas des années bissextiles. Mais 1600, 2000 et 2400 sont des années bissextiles. Par opposition, dans le plus ancien calendrier Julian, toutes les années divisibles par 4 sont des années bissextiles.

L'annonce du Pape de février 1582 décrétait que 10 jours devaient étre supprimés du mois d'octobre 1582 pour que le 15 octobre suivent immédiatement après le 4 octobre. Cela a été appliqué en Italie, Pologne, Portugal et Espagne. Les autres pays catholiques ont suivi peu de temps mais les pays protestants ont été plus récalcitrants à changer et la Gréce n'a pas changé avant le début du 20ème siècle. La réforme a été appliquée par la Grande Bretagne et ses colonies (y compris ce qui est maintenant les Etats−Unis) en 1752. Donc le 2 septembre 1752 a été suivi du 14 septembre 1752. ceci est la raison pour laquelle la commande cal produit la sortie suivante:

```
$ cal 9 1752
   septembre 1752
di lu ma me je ve sa
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

> **Note :** Le standard SQL stipule que << Au sein d'une définition d'un < datetime literal> , les < valeurs date/heure> sont restreintes par les règles naturelles pour les dates et heures relatives au calendrier Grégorien >>. Les dates entre 1752−09−03 et 1752−09−13, bien qu'éliminées dans plusieurs pays par ordre du Pape, sont conformes aux << règles naturelles >> et sont donc des dates valables.

Différents calendriers ont été developés dans différentes parties du monde, la plupart précédant le système Grégorien. Par exemple, les débuts du calendrier chinois peuvent être évalués au alentour du 14ème siécle avant J−C. La légende veut que l'empereur Huangdi invente le calendrier en 2637 avant J−C. La République de Chine utilise le calendrier Grégorien pour ses besoins civils. Le calendrier chinois est utilisé pour déterminer les festivals.

---

# Annexe C. Mots−clé SQL

La Tableau C−1 liste tous les éléments qui sont des mots−clé dans le standard SQL et dans PostgreSQL 8.0.5. Des informations sous−jacentes peuvent être trouvées dans Section 4.1.1.

SQL distingue les mots−clé *réservés* et *non réservés*. Selon le standard, les mots−clé réservés sont réellement les seuls mots−clé ; ils ne sont jamais autorisés comme identifiants. Les mots−clé non réservés ont seulement un sens spécial dans certains contextes et peuvent être utilisés comme identifiants dans d'autres contextes. La plupart des mots−clé non réservés sont en fait les noms des tables et des fonctions prédéfinies spécifiés par SQL. Le concept de mots−clé non réservés existe seulement pour indiquer que certains sens prédéfinis sont attachés à un mot dans certains contextes.

Dans l'analyseur de PostgreSQL, la vie est un peu plus compliquée. Il y a différentes classes d'éléments allant de ceux que l'on ne peut jamais utiliser comme identifiants à ceux qui n'ont absolument aucun statut spécial dans l'analyseur par rapport à un identifiant ordinaire (c'est généralement le cas pour les fonctions spécifiées par SQL). Même les mots−clé réservés ne sont pas complètement réservés dans PostgreSQL et peuvent être utilisés comme noms des colonnes (par exemple, `SELECT 55 AS CHECK`, même si `CHECK` est un mot−clé).

Dans Tableau C−1, dans la colonne pour PostgreSQL, nous classons comme << non réservé >> les mots−clé qui sont explicitement connus par l'analyseur mais qui sont autorisés dans la plupart des contextes où un identifiant est attendu. Certains mots−clé qui sont non réservés et qui ne peuvent pas être utilisés comme un nom de fonction ou un type de données sont marqués en conséquence. (La plupart des mots représentent des fonctions prédéfinies ou des types de données avec une syntaxe spéciale. La fonction ou le type est toujours disponible mais il ne peut pas être redéfini par un utilisateur.) Les << réservés >> sont des éléments qui sont seulement admis comme noms de colonnes << AS >> (et peut−être dans très peu d'autres contextes). Certains mots−clé réservés sont autorisés comme noms pour les fonctions ; cela est également montré dans le tableau.

En règle générale, si vous avez des erreurs de la part de l'analyseur pour des commandes qui contiennent un des mots−clés listés comme identifiants, vous devriez essayer de mettre entre guillemets l'identifiant pour voir si le problème disparait.

Il est important de comprendre avant d'étudier la Tableau C−1 que le fait qu'un mot−clé ne soit pas réservé dans PostgreSQL ne signifie pas que la fonctionnalité en rapport avec ce mot n'est pas implémentée. Réciproquement, la présence d'un mot−clé n'indique pas l'existance d'une fonctionnalité.

**Tableau C−1. Mots−clé SQL**

| Mot−clé | PostgreSQL | SQL:2003 | SQL:1999 | SQL−92 |
|---|---|---|---|---|
| `A` | | non réservé | | |
| `ABORT` | non réservé | | | |
| `ABS` | | réservé | non réservé | |
| `ABSOLUTE` | non réservé | non réservé | réservé | réservé |
| `ACCESS` | non réservé | | | |

| | | | | |
|---|---|---|---|---|
| ACTION | non réservé | non réservé | réservé | réservé |
| ADA | | non réservé | non réservé | non réservé |
| ADD | non réservé | non réservé | réservé | réservé |
| ADMIN | | non réservé | réservé | |
| AFTER | non réservé | non réservé | réservé | |
| AGGREGATE | non réservé | | réservé | |
| ALIAS | | | réservé | |
| ALL | réservé | réservé | réservé | réservé |
| ALLOCATE | | réservé | réservé | réservé |
| ALSO | non réservé | | | |
| ALTER | non réservé | réservé | réservé | réservé |
| ALWAYS | | non réservé | | |
| ANALYSE | réservé | | | |
| ANALYZE | réservé | | | |
| AND | réservé | réservé | réservé | réservé |
| ANY | réservé | réservé | réservé | réservé |
| ARE | | réservé | réservé | réservé |
| ARRAY | réservé | réservé | réservé | |
| AS | réservé | réservé | réservé | réservé |
| ASC | réservé | non réservé | réservé | réservé |
| ASENSITIVE | | réservé | non réservé | |
| ASSERTION | non réservé | non réservé | réservé | réservé |
| ASSIGNMENT | non réservé | non réservé | non réservé | |
| ASYMMETRIC | | réservé | non réservé | |
| AT | non réservé | réservé | réservé | réservé |
| ATOMIC | | réservé | non réservé | |
| ATTRIBUTE | | non réservé | | |
| ATTRIBUTES | | non réservé | | |
| AUTHORIZATION | réservé (peut être une fonction) | réservé | réservé | réservé |

| | | | | |
|---|---|---|---|---|
| AVG | | réservé | non réservé | réservé |
| BACKWARD | non réservé | | | |
| BEFORE | non réservé | non réservé | réservé | |
| BEGIN | non réservé | réservé | réservé | réservé |
| BERNOULLI | | non réservé | | |
| BETWEEN | réservé (peut être une fonction) | réservé | non réservé | réservé |
| BIGINT | non réservé (ne peut pas être une fonction ou un type) | réservé | | |
| BINARY | réservé (peut être une fonction) | réservé | réservé | |
| BIT | non réservé (ne peut pas être une fonction ou un type) | | réservé | réservé |
| BITVAR | | | non réservé | |
| BIT_LENGTH | | | non réservé | réservé |
| BLOB | | réservé | réservé | |
| BOOLEAN | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | |
| BOTH | réservé | réservé | réservé | réservé |
| BREADTH | | non réservé | réservé | |
| BY | non réservé | réservé | réservé | réservé |
| C | | non réservé | non réservé | non réservé |
| CACHE | non réservé | | | |
| CALL | | réservé | réservé | |
| CALLED | non réservé | réservé | non réservé | |
| CARDINALITY | | réservé | non réservé | |
| CASCADE | non réservé | non réservé | réservé | réservé |
| CASCADED | | réservé | réservé | réservé |
| CASE | réservé | réservé | réservé | réservé |
| CAST | réservé | réservé | réservé | réservé |
| CATALOG | | non réservé | réservé | réservé |

| | | | | |
|---|---|---|---|---|
| CATALOG_NAME | | non réservé | non réservé | non réservé |
| CEIL | | réservé | | |
| CEILING | | réservé | | |
| CHAIN | non réservé | non réservé | non réservé | |
| CHAR | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| CHARACTER | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| CHARACTERISTICS | non réservé | non réservé | | |
| CHARACTERS | | non réservé | | |
| CHARACTER_LENGTH | | réservé | non réservé | réservé |
| CHARACTER_SET_CATALOG | | non réservé | non réservé | non réservé |
| CHARACTER_SET_NAME | | non réservé | non réservé | non réservé |
| CHARACTER_SET_SCHEMA | | non réservé | non réservé | non réservé |
| CHAR_LENGTH | | réservé | non réservé | réservé |
| CHECK | réservé | réservé | réservé | réservé |
| CHECKED | | | non réservé | |
| CHECKPOINT | non réservé | | | |
| CLASS | non réservé | | réservé | |
| CLASS_ORIGIN | | non réservé | non réservé | non réservé |
| CLOB | | réservé | réservé | |
| CLOSE | non réservé | réservé | réservé | réservé |
| CLUSTER | non réservé | | | |
| COALESCE | non réservé (ne peut pas être une fonction ou un type) | réservé | non réservé | réservé |
| COBOL | | non réservé | non réservé | non réservé |
| COLLATE | réservé | réservé | réservé | réservé |
| COLLATION | | non réservé | réservé | réservé |
| COLLATION_CATALOG | | | | |

| | | | | |
|---|---|---|---|---|
| | | non réservé | non réservé | non réservé |
| COLLATION_NAME | | non réservé | non réservé | non réservé |
| COLLATION_SCHEMA | | non réservé | non réservé | non réservé |
| COLLECT | | réservé | | |
| COLUMN | réservé | réservé | réservé | réservé |
| COLUMN_NAME | | non réservé | non réservé | non réservé |
| COMMAND_FUNCTION | | non réservé | non réservé | non réservé |
| COMMAND_FUNCTION_CODE | | non réservé | non réservé | |
| COMMENT | non réservé | | | |
| COMMIT | non réservé | réservé | réservé | réservé |
| COMMITTED | non réservé | non réservé | non réservé | non réservé |
| COMPLETION | | | réservé | |
| CONDITION | | réservé | | |
| CONDITION_NUMBER | | non réservé | non réservé | non réservé |
| CONNECT | | réservé | réservé | réservé |
| CONNECTION | | non réservé | réservé | réservé |
| CONNECTION_NAME | | non réservé | non réservé | non réservé |
| CONSTRAINT | réservé | réservé | réservé | réservé |
| CONSTRAINTS | non réservé | non réservé | réservé | réservé |
| CONSTRAINT_CATALOG | | non réservé | non réservé | non réservé |
| CONSTRAINT_NAME | | non réservé | non réservé | non réservé |
| CONSTRAINT_SCHEMA | | non réservé | non réservé | non réservé |
| CONSTRUCTOR | | non réservé | réservé | |
| CONTAINS | | non réservé | non réservé | |
| CONTINUE | | non réservé | réservé | réservé |
| CONVERSION | non réservé | | | |
| CONVERT | non réservé (ne peut pas être une fonction ou un | réservé | non réservé | réservé |

| | | | | |
|---|---|---|---|---|
| | type) | | | |
| COPY | non réservé | | | |
| CORR | | réservé | | |
| CORRESPONDING | | réservé | réservé | réservé |
| COUNT | | réservé | non réservé | réservé |
| COVAR_POP | | réservé | | |
| COVAR_SAMP | | réservé | | |
| CREATE | réservé | réservé | réservé | réservé |
| CREATEDB | non réservé | | | |
| CREATEUSER | non réservé | | | |
| CROSS | réservé (peut être une fonction) | réservé | réservé | réservé |
| CSV | non réservé | | | |
| CUBE | | réservé | réservé | |
| CUME_DIST | | réservé | | |
| CURRENT | | réservé | réservé | réservé |
| CURRENT_DATE | réservé | réservé | réservé | réservé |
| CURRENT_DEFAULT_TRANSFORM_GROUP | | réservé | | |
| CURRENT_PATH | | réservé | réservé | |
| CURRENT_ROLE | | réservé | réservé | |
| CURRENT_TIME | réservé | réservé | réservé | réservé |
| CURRENT_TIMESTAMP | réservé | réservé | réservé | réservé |
| CURRENT_TRANSFORM_GROUP_FOR_TYPE | | réservé | | |
| CURRENT_USER | réservé | réservé | réservé | réservé |
| CURSOR | non réservé | réservé | réservé | réservé |
| CURSOR_NAME | | non réservé | non réservé | non réservé |
| CYCLE | non réservé | réservé | réservé | |
| DATA | | non réservé | réservé | non réservé |
| DATABASE | non réservé | | | |
| DATE | | réservé | réservé | réservé |
| DATETIME_INTERVAL_CODE | | non réservé | non réservé | non réservé |
| DATETIME_INTERVAL_PRECISION | | non réservé | non réservé | non réservé |
| DAY | non réservé | réservé | réservé | réservé |
| DEALLOCATE | non réservé | réservé | réservé | réservé |
| DEC | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| DECIMAL | | réservé | réservé | réservé |

| | | | | |
|---|---|---|---|---|
| | non réservé (ne peut pas être une fonction ou un type) | | | |
| DECLARE | non réservé | réservé | réservé | réservé |
| DEFAULT | réservé | réservé | réservé | réservé |
| DEFAULTS | non réservé | non réservé | | |
| DEFERRABLE | réservé | non réservé | réservé | réservé |
| DEFERRED | non réservé | non réservé | réservé | réservé |
| DEFINED | | non réservé | non réservé | |
| DEFINER | non réservé | non réservé | non réservé | |
| DEGREE | | non réservé | | |
| DELETE | non réservé | réservé | réservé | réservé |
| DELIMITER | non réservé | | | |
| DELIMITERS | non réservé | | | |
| DENSE_RANK | | réservé | | |
| DEPTH | | non réservé | réservé | |
| DEREF | | réservé | réservé | |
| DERIVED | | non réservé | | |
| DESC | réservé | non réservé | réservé | réservé |
| DESCRIBE | | réservé | réservé | réservé |
| DESCRIPTOR | | non réservé | réservé | réservé |
| DESTROY | | | réservé | |
| DESTRUCTOR | | | réservé | |
| DETERMINISTIC | | réservé | réservé | |
| DIAGNOSTICS | | non réservé | réservé | réservé |
| DICTIONARY | | | réservé | |
| DISCONNECT | | réservé | réservé | réservé |
| DISPATCH | | non réservé | non réservé | |
| DISTINCT | réservé | réservé | réservé | réservé |
| DO | réservé | | | |
| DOMAIN | non réservé | non réservé | réservé | réservé |

| | | | | |
|---|---|---|---|---|
| DOUBLE | non réservé | réservé | réservé | réservé |
| DROP | non réservé | réservé | réservé | réservé |
| DYNAMIC | | réservé | réservé | |
| DYNAMIC_FUNCTION | | non réservé | non réservé | non réservé |
| DYNAMIC_FUNCTION_CODE | | non réservé | non réservé | |
| EACH | non réservé | réservé | réservé | |
| ELEMENT | | réservé | | |
| ELSE | réservé | réservé | réservé | réservé |
| ENCODING | non réservé | | | |
| ENCRYPTED | non réservé | | | |
| END | réservé | réservé | réservé | réservé |
| END-EXEC | | réservé | réservé | réservé |
| EQUALS | | non réservé | réservé | |
| ESCAPE | non réservé | réservé | réservé | réservé |
| EVERY | | réservé | réservé | |
| EXCEPT | réservé | réservé | réservé | réservé |
| EXCEPTION | | non réservé | réservé | réservé |
| EXCLUDE | | non réservé | | |
| EXCLUDING | non réservé | non réservé | | |
| EXCLUSIVE | non réservé | | | |
| EXEC | | réservé | réservé | réservé |
| EXECUTE | non réservé | réservé | réservé | réservé |
| EXISTING | | | non réservé | |
| EXISTS | non réservé (ne peut pas être une fonction ou un type) | réservé | non réservé | réservé |
| EXP | | réservé | | |
| EXPLAIN | non réservé | | | |
| EXTERNAL | non réservé | réservé | réservé | réservé |
| EXTRACT | non réservé (ne peut pas être une fonction ou un type) | réservé | non réservé | réservé |
| FALSE | réservé | réservé | réservé | réservé |
| FETCH | non réservé | réservé | réservé | réservé |
| FILTER | | réservé | | |
| FINAL | | non réservé | non réservé | |

| | | | | |
|---|---|---|---|---|
| FIRST | non réservé | non réservé | réservé | réservé |
| FLOAT | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| FLOOR | | réservé | | |
| FOLLOWING | | non réservé | | |
| FOR | réservé | réservé | réservé | réservé |
| FORCE | non réservé | | | |
| FOREIGN | réservé | réservé | réservé | réservé |
| FORTRAN | | non réservé | non réservé | non réservé |
| FORWARD | non réservé | | | |
| FOUND | | non réservé | réservé | réservé |
| FREE | | réservé | réservé | |
| FREEZE | réservé (peut être une fonction) | | | |
| FROM | réservé | réservé | réservé | réservé |
| FULL | réservé (peut être une fonction) | réservé | réservé | réservé |
| FUNCTION | non réservé | réservé | réservé | |
| FUSION | | réservé | | |
| G | | non réservé | non réservé | |
| GENERAL | | non réservé | réservé | |
| GENERATED | | non réservé | non réservé | |
| GET | | réservé | réservé | réservé |
| GLOBAL | non réservé | réservé | réservé | réservé |
| GO | | non réservé | réservé | réservé |
| GOTO | | non réservé | réservé | réservé |
| GRANT | réservé | réservé | réservé | réservé |
| GRANTED | | non réservé | non réservé | |
| GROUP | réservé | réservé | réservé | réservé |
| GROUPING | | réservé | réservé | |
| HANDLER | non réservé | | | |
| HAVING | réservé | réservé | réservé | réservé |
| HIERARCHY | | | | |

| | | | | |
|---|---|---|---|---|
| | | non réservé | non réservé | |
| HOLD | non réservé | réservé | non réservé | |
| HOST | | | réservé | |
| HOUR | non réservé | réservé | réservé | réservé |
| IDENTITY | | réservé | réservé | réservé |
| IGNORE | | | réservé | |
| ILIKE | réservé (peut être une fonction) | | | |
| IMMEDIATE | non réservé | non réservé | réservé | réservé |
| IMMUTABLE | non réservé | | | |
| IMPLEMENTATION | | non réservé | non réservé | |
| IMPLICIT | non réservé | | | |
| IN | réservé | réservé | réservé | réservé |
| INCLUDING | non réservé | non réservé | | |
| INCREMENT | non réservé | non réservé | | |
| INDEX | non réservé | | | |
| INDICATOR | | réservé | réservé | réservé |
| INFIX | | | non réservé | |
| INHERITS | non réservé | | | |
| INITIALIZE | | | réservé | |
| INITIALLY | réservé | non réservé | réservé | réservé |
| INNER | réservé (peut être une fonction) | réservé | réservé | réservé |
| INOUT | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | |
| INPUT | non réservé | non réservé | réservé | réservé |
| INSENSITIVE | non réservé | réservé | non réservé | réservé |
| INSERT | non réservé | réservé | réservé | réservé |
| INSTANCE | | non réservé | non réservé | |
| INSTANTIABLE | | non réservé | non réservé | |
| INSTEAD | non réservé | | | |

| | | | | |
|---|---|---|---|---|
| INT | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| INTEGER | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| INTERSECT | réservé | réservé | réservé | réservé |
| INTERSECTION | | réservé | | |
| INTERVAL | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| INTO | réservé | réservé | réservé | réservé |
| INVOKER | non réservé | non réservé | non réservé | |
| IS | réservé (peut être une fonction) | réservé | réservé | réservé |
| ISNULL | réservé (peut être une fonction) | | | |
| ISOLATION | non réservé | non réservé | réservé | réservé |
| ITERATE | | | réservé | |
| JOIN | réservé (peut être une fonction) | réservé | réservé | réservé |
| K | | non réservé | non réservé | |
| KEY | non réservé | non réservé | réservé | réservé |
| KEY_MEMBER | | non réservé | non réservé | |
| KEY_TYPE | | non réservé | non réservé | |
| LANCOMPILER | non réservé | | | |
| LANGUAGE | non réservé | réservé | réservé | réservé |
| LARGE | non réservé | réservé | réservé | |
| LAST | non réservé | non réservé | réservé | réservé |
| LATERAL | | réservé | réservé | |
| LEADING | réservé | réservé | réservé | réservé |
| LEFT | réservé (peut être une fonction) | réservé | réservé | réservé |
| LENGTH | | non réservé | non réservé | non réservé |
| LESS | | | réservé | |
| LEVEL | non réservé | non réservé | réservé | réservé |

| | | | | |
|---|---|---|---|---|
| LIKE | réservé (peut être une fonction) | réservé | réservé | réservé |
| LIMIT | réservé | | réservé | |
| LISTEN | non réservé | | | |
| LN | | réservé | | |
| LOAD | non réservé | | | |
| LOCAL | non réservé | réservé | réservé | réservé |
| LOCALTIME | réservé | réservé | réservé | |
| LOCALTIMESTAMP | réservé | réservé | réservé | |
| LOCATION | non réservé | | | |
| LOCATOR | | non réservé | réservé | |
| LOCK | non réservé | | | |
| LOWER | | réservé | non réservé | réservé |
| M | | non réservé | non réservé | |
| MAP | | non réservé | réservé | |
| MATCH | non réservé | réservé | réservé | réservé |
| MATCHED | | non réservé | | |
| MAX | | réservé | non réservé | réservé |
| MAXVALUE | non réservé | non réservé | | |
| MEMBER | | réservé | | |
| MERGE | | réservé | | |
| MESSAGE_LENGTH | | non réservé | non réservé | non réservé |
| MESSAGE_OCTET_LENGTH | | non réservé | non réservé | non réservé |
| MESSAGE_TEXT | | non réservé | non réservé | non réservé |
| METHOD | | réservé | non réservé | |
| MIN | | réservé | non réservé | réservé |
| MINUTE | non réservé | réservé | réservé | réservé |
| MINVALUE | non réservé | non réservé | | |
| MOD | | réservé | non réservé | |
| MODE | non réservé | | | |

| | | | | |
|---|---|---|---|---|
| MODIFIES | | réservé | réservé | |
| MODIFY | | | réservé | |
| MODULE | | réservé | réservé | réservé |
| MONTH | non réservé | réservé | réservé | réservé |
| MORE | | non réservé | non réservé | non réservé |
| MOVE | non réservé | | | |
| MULTISET | | réservé | | |
| MUMPS | | non réservé | non réservé | non réservé |
| NAME | | non réservé | non réservé | non réservé |
| NAMES | non réservé | non réservé | réservé | réservé |
| NATIONAL | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| NATURAL | réservé (peut être une fonction) | réservé | réservé | réservé |
| NCHAR | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| NCLOB | | réservé | réservé | |
| NESTING | | non réservé | | |
| NEW | réservé | réservé | réservé | |
| NEXT | non réservé | non réservé | réservé | réservé |
| NO | non réservé | réservé | réservé | réservé |
| NOCREATEDB | non réservé | | | |
| NOCREATEUSER | non réservé | | | |
| NONE | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | |
| NORMALIZE | | réservé | | |
| NORMALIZED | | non réservé | | |
| NOT | réservé | réservé | réservé | réservé |
| NOTHING | non réservé | | | |
| NOTIFY | non réservé | | | |
| NOTNULL | réservé (peut être une fonction) | | | |
| NOWAIT | non réservé | | | |
| NULL | réservé | réservé | réservé | réservé |

| | | | | |
|---|---|---|---|---|
| NULLABLE | | non réservé | non réservé | non réservé |
| NULLIF | non réservé (ne peut pas être une fonction ou un type) | réservé | non réservé | réservé |
| NULLS | | non réservé | | |
| NUMBER | | non réservé | non réservé | non réservé |
| NUMERIC | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| OBJECT | non réservé | non réservé | réservé | |
| OCTETS | | non réservé | | |
| OCTET_LENGTH | | réservé | non réservé | réservé |
| OF | non réservé | réservé | réservé | réservé |
| OFF | réservé | | réservé | |
| OFFSET | réservé | | | |
| OIDS | non réservé | | | |
| OLD | réservé | réservé | réservé | |
| ON | réservé | réservé | réservé | réservé |
| ONLY | réservé | réservé | réservé | réservé |
| OPEN | | réservé | réservé | réservé |
| OPERATION | | | réservé | |
| OPERATOR | non réservé | | | |
| OPTION | non réservé | non réservé | réservé | réservé |
| OPTIONS | | non réservé | non réservé | |
| OR | réservé | réservé | réservé | réservé |
| ORDER | réservé | réservé | réservé | réservé |
| ORDERING | | non réservé | | |
| ORDINALITY | | non réservé | réservé | |
| OTHERS | | non réservé | | |
| OUT | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | |
| OUTER | réservé (peut être une fonction) | réservé | réservé | réservé |

| | | | | |
|---|---|---|---|---|
| OUTPUT | | non réservé | réservé | réservé |
| OVER | | réservé | | |
| OVERLAPS | réservé (peut être une fonction) | réservé | non réservé | réservé |
| OVERLAY | non réservé (ne peut pas être une fonction ou un type) | réservé | non réservé | |
| OVERRIDING | | non réservé | non réservé | |
| OWNER | non réservé | | | |
| PAD | | non réservé | réservé | réservé |
| PARAMETER | | réservé | réservé | |
| PARAMETERS | | | réservé | |
| PARAMETER_MODE | | non réservé | non réservé | |
| PARAMETER_NAME | | non réservé | non réservé | |
| PARAMETER_ORDINAL_POSITION | | non réservé | non réservé | |
| PARAMETER_SPECIFIC_CATALOG | | non réservé | non réservé | |
| PARAMETER_SPECIFIC_NAME | | non réservé | non réservé | |
| PARAMETER_SPECIFIC_SCHEMA | | non réservé | non réservé | |
| PARTIAL | non réservé | non réservé | réservé | réservé |
| PARTITION | | réservé | | |
| PASCAL | | non réservé | non réservé | non réservé |
| PASSWORD | non réservé | | | |
| PATH | | non réservé | réservé | |
| PERCENTILE_CONT | | réservé | | |
| PERCENTILE_DISC | | réservé | | |
| PERCENT_RANK | | réservé | | |
| PLACING | réservé | non réservé | | |
| PLI | | non réservé | non réservé | non réservé |
| POSITION | non réservé (ne peut pas être une fonction ou un type) | réservé | non réservé | réservé |

| | | | | |
|---|---|---|---|---|
| POSTFIX | | | réservé | |
| POWER | | réservé | | |
| PRECEDING | | non réservé | | |
| PRECISION | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| PREFIX | | | réservé | |
| PREORDER | | | réservé | |
| PREPARE | non réservé | réservé | réservé | réservé |
| PRESERVE | non réservé | non réservé | réservé | réservé |
| PRIMARY | réservé | réservé | réservé | réservé |
| PRIOR | non réservé | non réservé | réservé | réservé |
| PRIVILEGES | non réservé | non réservé | réservé | réservé |
| PROCEDURAL | non réservé | | | |
| PROCEDURE | non réservé | réservé | réservé | réservé |
| PUBLIC | | non réservé | réservé | réservé |
| QUOTE | non réservé | | | |
| RANGE | | réservé | | |
| RANK | | réservé | | |
| READ | non réservé | non réservé | réservé | réservé |
| READS | | réservé | réservé | |
| REAL | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| RECHECK | non réservé | | | |
| RECURSIVE | | réservé | réservé | |
| REF | | réservé | réservé | |
| REFERENCES | réservé | réservé | réservé | réservé |
| REFERENCING | | réservé | réservé | |
| REGR_AVGX | | réservé | | |
| REGR_AVGY | | réservé | | |
| REGR_COUNT | | réservé | | |
| REGR_INTERCEPT | | réservé | | |
| REGR_R2 | | réservé | | |
| REGR_SLOPE | | réservé | | |
| REGR_SXX | | réservé | | |
| REGR_SXY | | réservé | | |

| | | | | |
|---|---|---|---|---|
| REGR_SYY | | réservé | | |
| REINDEX | non réservé | | | |
| RELATIVE | non réservé | non réservé | réservé | réservé |
| RELEASE | non réservé | réservé | | |
| RENAME | non réservé | | | |
| REPEATABLE | non réservé | non réservé | non réservé | non réservé |
| REPLACE | non réservé | | | |
| RESET | non réservé | | | |
| RESTART | non réservé | non réservé | | |
| RESTRICT | non réservé | non réservé | réservé | réservé |
| RESULT | | réservé | réservé | |
| RETURN | | réservé | réservé | |
| RETURNED_CARDINALITY | | non réservé | | |
| RETURNED_LENGTH | | non réservé | non réservé | non réservé |
| RETURNED_OCTET_LENGTH | | non réservé | non réservé | non réservé |
| RETURNED_SQLSTATE | | non réservé | non réservé | non réservé |
| RETURNS | non réservé | réservé | réservé | |
| REVOKE | non réservé | réservé | réservé | réservé |
| RIGHT | réservé (peut être une fonction) | réservé | réservé | réservé |
| ROLE | | non réservé | réservé | |
| ROLLBACK | non réservé | réservé | réservé | réservé |
| ROLLUP | | réservé | réservé | |
| ROUTINE | | non réservé | réservé | |
| ROUTINE_CATALOG | | non réservé | non réservé | |
| ROUTINE_NAME | | non réservé | non réservé | |
| ROUTINE_SCHEMA | | non réservé | non réservé | |
| ROW | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | |
| ROWS | non réservé | réservé | réservé | réservé |

| | | | | |
|---|---|---|---|---|
| ROW_COUNT | | non réservé | non réservé | non réservé |
| ROW_NUMBER | | réservé | | |
| RULE | non réservé | | | |
| SAVEPOINT | non réservé | réservé | réservé | |
| SCALE | | non réservé | non réservé | non réservé |
| SCHEMA | non réservé | non réservé | réservé | réservé |
| SCHEMA_NAME | | non réservé | non réservé | non réservé |
| SCOPE | | réservé | réservé | |
| SCOPE_CATALOG | | non réservé | | |
| SCOPE_NAME | | non réservé | | |
| SCOPE_SCHEMA | | non réservé | | |
| SCROLL | non réservé | réservé | réservé | réservé |
| SEARCH | | réservé | réservé | |
| SECOND | non réservé | réservé | réservé | réservé |
| SECTION | | non réservé | réservé | réservé |
| SECURITY | non réservé | non réservé | non réservé | |
| SELECT | réservé | réservé | réservé | réservé |
| SELF | | non réservé | non réservé | |
| SENSITIVE | | réservé | non réservé | |
| SEQUENCE | non réservé | non réservé | réservé | |
| SERIALIZABLE | non réservé | non réservé | non réservé | non réservé |
| SERVER_NAME | | non réservé | non réservé | non réservé |
| SESSION | non réservé | non réservé | réservé | réservé |
| SESSION_USER | réservé | réservé | réservé | réservé |
| SET | non réservé | réservé | réservé | réservé |
| SETOF | non réservé (ne peut pas être une fonction ou un type) | | | |
| SETS | | non réservé | réservé | |

| | | | | |
|---|---|---|---|---|
| SHARE | non réservé | | | |
| SHOW | non réservé | | | |
| SIMILAR | réservé (peut être une fonction) | réservé | non réservé | |
| SIMPLE | non réservé | non réservé | non réservé | |
| SIZE | | non réservé | réservé | réservé |
| SMALLINT | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| SOME | réservé | réservé | réservé | réservé |
| SOURCE | | non réservé | non réservé | |
| SPACE | | non réservé | réservé | réservé |
| SPECIFIC | | réservé | réservé | |
| SPECIFICTYPE | | réservé | réservé | |
| SPECIFIC_NAME | | non réservé | non réservé | |
| SQL | | réservé | réservé | réservé |
| SQLCODE | | | | réservé |
| SQLERROR | | | | réservé |
| SQLEXCEPTION | | réservé | réservé | |
| SQLSTATE | | réservé | réservé | réservé |
| SQLWARNING | | réservé | réservé | |
| SQRT | | réservé | | |
| STABLE | non réservé | | | |
| START | non réservé | réservé | réservé | |
| STATE | | non réservé | réservé | |
| STATEMENT | non réservé | non réservé | réservé | |
| STATIC | | réservé | réservé | |
| STATISTICS | non réservé | | | |
| STDDEV_POP | | réservé | | |
| STDDEV_SAMP | | réservé | | |
| STDIN | non réservé | | | |
| STDOUT | non réservé | | | |
| STORAGE | non réservé | | | |
| STRICT | non réservé | | | |
| STRUCTURE | | non réservé | réservé | |
| STYLE | | | | |

| | | | | |
|---|---|---|---|---|
| | | non réservé | non réservé | |
| SUBCLASS_ORIGIN | | non réservé | non réservé | non réservé |
| SUBLIST | | | non réservé | |
| SUBMULTISET | | réservé | | |
| SUBSTRING | non réservé (ne peut pas être une fonction ou un type) | réservé | non réservé | réservé |
| SUM | | réservé | non réservé | réservé |
| SYMMETRIC | | réservé | non réservé | |
| SYSID | non réservé | | | |
| SYSTEM | | réservé | non réservé | |
| SYSTEM_USER | | réservé | réservé | réservé |
| TABLE | réservé | réservé | réservé | réservé |
| TABLESAMPLE | | réservé | | |
| TABLESPACE | non réservé | | | |
| TABLE_NAME | | non réservé | non réservé | non réservé |
| TEMP | non réservé | | | |
| TEMPLATE | non réservé | | | |
| TEMPORARY | non réservé | non réservé | réservé | réservé |
| TERMINATE | | | réservé | |
| THAN | | | réservé | |
| THEN | réservé | réservé | réservé | réservé |
| TIES | | non réservé | | |
| TIME | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| TIMESTAMP | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| TIMEZONE_HOUR | | réservé | réservé | réservé |
| TIMEZONE_MINUTE | | réservé | réservé | réservé |
| TO | réservé | réservé | réservé | réservé |
| TOAST | non réservé | | | |
| TOP_LEVEL_COUNT | | non réservé | | |

| | | | | |
|---|---|---|---|---|
| TRAILING | réservé | réservé | réservé | réservé |
| TRANSACTION | non réservé | non réservé | réservé | réservé |
| TRANSACTIONS_COMMITTED | | non réservé | non réservé | |
| TRANSACTIONS_ROLLED_BACK | | non réservé | non réservé | |
| TRANSACTION_ACTIVE | | non réservé | non réservé | |
| TRANSFORM | | non réservé | non réservé | |
| TRANSFORMS | | non réservé | non réservé | |
| TRANSLATE | | réservé | non réservé | réservé |
| TRANSLATION | | réservé | réservé | réservé |
| TREAT | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | |
| TRIGGER | non réservé | réservé | réservé | |
| TRIGGER_CATALOG | | non réservé | non réservé | |
| TRIGGER_NAME | | non réservé | non réservé | |
| TRIGGER_SCHEMA | | non réservé | non réservé | |
| TRIM | non réservé (ne peut pas être une fonction ou un type) | réservé | non réservé | réservé |
| TRUE | réservé | réservé | réservé | réservé |
| TRUNCATE | non réservé | | | |
| TRUSTED | non réservé | | | |
| TYPE | non réservé | non réservé | non réservé | non réservé |
| UESCAPE | | réservé | | |
| UNBOUNDED | | non réservé | | |
| UNCOMMITTED | non réservé | non réservé | non réservé | non réservé |
| UNDER | | non réservé | réservé | |
| UNENCRYPTED | non réservé | | | |
| UNION | réservé | réservé | réservé | réservé |
| UNIQUE | réservé | réservé | réservé | réservé |
| UNKNOWN | non réservé | réservé | réservé | réservé |

| | | | | |
|---|---|---|---|---|
| UNLISTEN | non réservé | | | |
| UNNAMED | | non réservé | non réservé | non réservé |
| UNNEST | | réservé | réservé | |
| UNTIL | non réservé | | | |
| UPDATE | non réservé | réservé | réservé | réservé |
| UPPER | | réservé | non réservé | réservé |
| USAGE | non réservé | non réservé | réservé | réservé |
| USER | réservé | réservé | réservé | réservé |
| USER_DEFINED_TYPE_CATALOG | | non réservé | non réservé | |
| USER_DEFINED_TYPE_CODE | | non réservé | | |
| USER_DEFINED_TYPE_NAME | | non réservé | non réservé | |
| USER_DEFINED_TYPE_SCHEMA | | non réservé | non réservé | |
| USING | réservé | réservé | réservé | réservé |
| VACUUM | non réservé | | | |
| VALID | non réservé | | | |
| VALIDATOR | non réservé | | | |
| VALUE | | réservé | réservé | réservé |
| VALUES | non réservé | réservé | réservé | réservé |
| VARCHAR | non réservé (ne peut pas être une fonction ou un type) | réservé | réservé | réservé |
| VARIABLE | | | réservé | |
| VARYING | non réservé | réservé | réservé | réservé |
| VAR_POP | | réservé | | |
| VAR_SAMP | | réservé | | |
| VERBOSE | réservé (peut être une fonction) | | | |
| VIEW | non réservé | non réservé | réservé | réservé |
| VOLATILE | non réservé | | | |
| WHEN | réservé | réservé | réservé | réservé |
| WHENEVER | | réservé | réservé | réservé |
| WHERE | réservé | réservé | réservé | réservé |
| WIDTH_BUCKET | | réservé | | |
| WINDOW | | réservé | | |
| WITH | non réservé | réservé | réservé | réservé |
| WITHIN | | réservé | | |

| | | | | |
|---|---|---|---|---|
| WITHOUT | non réservé | réservé | réservé | |
| WORK | non réservé | non réservé | réservé | réservé |
| WRITE | non réservé | non réservé | réservé | réservé |
| YEAR | non réservé | | réservé | réservé |
| ZONE | non réservé | non réservé | réservé | réservé |

# Annexe D. Compatibilité SQL

Cette section tente de mettre en évidence dans quelle mesure PostgreSQL est conforme au standard SQL actuel. L'information qui suit n'est pas une indication complète de compatibilité mais présente les thèmes principaux avec autant de détails que possible, c'est−à−dire en restant raisonnable et utile pour l'utilisateur.

Le nom complet du standard SQL est ISO/IEC 9075 << Database Language SQL >>. Une version revue du standard est publiée de temps en temps dont la plus récente date de la fin 2003. Cette version est référencée en temps que ISO/IEC 9075:2003, ou de manière informelle comme SQL:2003. Les versions précédentes à celle−ci étaient SQL:1999 et SQL−92. Chaque version remplace la précédente, donc toute indication de support des versions précédentes n'a aucun mérite. Le développement de PostgreSQL tend à se conformer à la dernière version officielle dans la mesure où celle−ci ne viendrait pas s'opposer aux fonctionnalités traditionnelles ou au bon sens. Le projet PostgreSQL n'était pas représenté dans le groupe de travail ISO/IEC 9075 Working Group lors de la préparation de SQL:2003. Malgré cela, un grand nombre des fonctionnalités requises par SQL:2003 sont déjà supportées, bien que, parfois, avec une syntaxe ou une fonction légèrement différente. Une meilleure compatibilité est attendue pour les prochaines versions.

SQL−92 définit trois ensembles de fonctionnalité pour le test de compatibilité : base, intermédiaire, complète. La majorité des systèmes de gestion de bases de données se disant compatibles au standard SQL sont en général conformes au niveau des bases en vertu du fait que l'ensemble des fonctionnalités des niveaux intermédiaire et complet sont soit trop volumineux soit en conflit avec des comportements précédemment implantés.

Avec SQL99, le standard SQL a défini un vaste ensemble de fonctionnalités individuelles à la place des trois niveaux de fonctionnalités définis dans SQL−92 et dont l'efficacité est discutable. Une grande partie représente les fonctionnalités << centrales >> que chaque implémentation de SQL doit fournir. Les fonctionnalités restantes sont purement optionnelles. Certaines de ces fonctionnalités optionnelles sont regroupées au sein de << paquetages >>. Une implémentation peut ainsi se dire conforme à certains paquetages, se déclarant ainsi conforme à des groupes particuliers de fonctions.

Le standard SQL:2003 est aussi divisée en un certain nombre de parties. Chacune est connu par un pseudonyme. Notez que ces parties ne sont pas numérotées dans l'ordre.

- ISO/IEC 9075−1 Framework (SQL/Framework)
- ISO/IEC 9075−2 Foundation (SQL/Foundation)
- ISO/IEC 9075−3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075−4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075−9 Management of External Data (SQL/MED)
- ISO/IEC 9075−10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075−11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075−13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075−14 XML−related specifications (SQL/XML)

PostgreSQL couvre les parties 1, 2 et 11. La partie 3 est similaire à l'interface ODBC et la partie 4 est similaire au langage de programmation PL/pgSQL mais une compatibilité exacte n'est pas spécifiquement voulue ou vérifiée dans chaque cas.

PostgreSQL supporte la plupart des fonctionnalités majeures de SQL:2003. Des 164 fonctionnalités requises pour une compatibilité Core, PostgreSQL est conforme sur au moins 150. De plus, il existe une longue liste de fonctionnalités supplémentaires supportées. Il est important de noter qu'au moment de l'écriture de ce texte,

aucune version actuelle d'un quelconque système de gestion de bases de données n'indique une compatibilité totale au Core de SQL:2003.

Dans les deux sections suivantes, nous fournissons une liste de ces fonctionnalités supportées par PostgreSQL suivies de la liste des fonctionnalités définies dans SQL:2003 qui ne sont pas encore prises en compte. Ces deux listes sont approximatives : certains détails mineurs peuvent ne pas être compatibles et certaines grandes parties d'une fonctionnalité non supportée peuvent avoir été implantées. Vous pourrez trouver des informations plus précises sur ce qui fonctionne ou ne fonctionne pas dans les différents chapitres principaux de la documentation.

> **Note :** Les codes de fonctionnalité contenant un tiret sont des sous−fonctionnalités. Le cas échéant, si une sous−fonction n'est pas supportée, la fonctionnalité de niveau supérieur sera définie comme non supportée même si d'autres sous−fonctions lui appartenant sont supportées.

# D.1. Fonctionnalités supportées

| Identifiant | Paquetage | Description | Commentaire |
|---|---|---|---|
| B012 | Cœur | Intégration de C | |
| B021 | | SQL direct | |
| E011 | Cœur | Types de données numériques | |
| E011−01 | Cœur | INTEGER and SMALLINT data types | |
| E011−02 | Cœur | REAL, DOUBLE PRECISON, and FLOAT data types | |
| E011−03 | Cœur | DECIMAL and NUMERIC data types | |
| E011−04 | Cœur | Arithmetic operators | |
| E011−05 | Cœur | Numeric comparison | |
| E011−06 | Cœur | Implicit casting among the numeric data types | |
| E021 | Cœur | Character data types | |
| E021−01 | Cœur | CHARACTER data type | |
| E021−02 | Cœur | CHARACTER VARYING data type | |
| E021−03 | Cœur | Character literals | |
| E021−04 | Cœur | CHARACTER_LENGTH function | |
| E021−05 | Cœur | OCTET_LENGTH function | |
| E021−06 | Cœur | SUBSTRING function | |
| E021−07 | Cœur | Character concatenation | |
| E021−08 | Cœur | UPPER and LOWER functions | |
| E021−09 | Cœur | TRIM function | |
| E021−10 | Cœur | Implicit casting among the character data types | |
| E021−11 | Cœur | POSITION function | |
| E021−12 | Cœur | Character comparison | |
| E031 | Cœur | Identifiers | |
| E031−01 | Cœur | Delimited identifiers | |
| E031−02 | Cœur | Lower case identifiers | |

| | | | |
|---|---|---|---|
| E031−03 | Cœur | Trailing underscore | |
| E051 | Cœur | Basic query specification | |
| E051−01 | Cœur | SELECT DISTINCT | |
| E051−02 | Cœur | GROUP BY clause | |
| E051−04 | Cœur | GROUP BY can contain columns not in <select list> | |
| E051−05 | Cœur | Select list items can be renamed | AS is required |
| E051−06 | Cœur | HAVING clause | |
| E051−07 | Cœur | Qualified * in select list | |
| E051−08 | Cœur | Correlation names in the FROM clause | |
| E051−09 | Cœur | Rename columns in the FROM clause | |
| E061 | Cœur | Basic predicates and search conditions | |
| E061−01 | Cœur | Comparison predicate | |
| E061−02 | Cœur | BETWEEN predicate | |
| E061−03 | Cœur | IN predicate with list of values | |
| E061−04 | Cœur | LIKE predicate | |
| E061−05 | Cœur | LIKE predicate ESCAPE clause | |
| E061−06 | Cœur | NULL predicate | |
| E061−07 | Cœur | Quantified comparison predicate | |
| E061−08 | Cœur | EXISTS predicate | |
| E061−09 | Cœur | Subqueries in comparison predicate | |
| E061−11 | Cœur | Subqueries in IN predicate | |
| E061−12 | Cœur | Subqueries in quantified comparison predicate | |
| E061−13 | Cœur | Correlated subqueries | |
| E061−14 | Cœur | Search condition | |
| E071 | Cœur | Basic query expressions | |
| E071−01 | Cœur | UNION DISTINCT table operator | |
| E071−02 | Cœur | UNION ALL table operator | |
| E071−03 | Cœur | EXCEPT DISTINCT table operator | |
| E071−05 | Cœur | Columns combined via table operators need not have exactly the same data type | |
| E071−06 | Cœur | Table operators in subqueries | |
| E081−01 | Cœur | SELECT privilege | |
| E081−02 | Cœur | DELETE privilege | |
| E081−03 | Cœur | INSERT privilege at the table level | |
| E081−04 | Cœur | UPDATE privilege at the table level | |
| E081−06 | Cœur | REFERENCES privilege at the table level | |
| E081−08 | Cœur | WITH GRANT OPTION | |
| E091 | Cœur | Set functions | |
| E091−01 | Cœur | AVG | |
| E091−02 | Cœur | COUNT | |
| E091−03 | Cœur | MAX | |
| E091−04 | Cœur | MIN | |

| | | | |
|---|---|---|---|
| E091−05 | Cœur | SUM | |
| E091−06 | Cœur | ALL quantifier | |
| E091−07 | Cœur | DISTINCT quantifier | |
| E101 | Cœur | Basic data manipulation | |
| E101−01 | Cœur | INSERT statement | |
| E101−03 | Cœur | Searched UPDATE statement | |
| E101−04 | Cœur | Searched DELETE statement | |
| E111 | Cœur | Single row SELECT statement | |
| E121−01 | Cœur | DECLARE CURSOR | |
| E121−02 | Cœur | ORDER BY columns need not be in select list | |
| E121−03 | Cœur | Value expressions in ORDER BY clause | |
| E121−08 | Cœur | CLOSE statement | |
| E121−10 | Cœur | FETCH statement implicit NEXT | |
| E121−17 | Cœur | WITH HOLD cursors | |
| E131 | Cœur | Null value support (NULLs in lieu of values) | |
| E141 | Cœur | Basic integrity constraints | |
| E141−01 | Cœur | NOT NULL constraints | |
| E141−02 | Cœur | UNIQUE constraints of NOT NULL columns | |
| E141−03 | Cœur | PRIMARY KEY constraints | |
| E141−04 | Cœur | Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action | |
| E141−06 | Cœur | CHECK constraints | |
| E141−07 | Cœur | Column defaults | |
| E141−08 | Cœur | NOT NULL inferred on PRIMARY KEY | |
| E141−10 | Cœur | Names in a foreign key can be specified in any order | |
| E151 | Cœur | Transaction support | |
| E151−01 | Cœur | COMMIT statement | |
| E151−02 | Cœur | ROLLBACK statement | |
| E152 | Cœur | Basic SET TRANSACTION statement | |
| E152−01 | Cœur | SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause | |
| E152−02 | Cœur | SET TRANSACTION statement: READ ONLY and READ WRITE clauses | |
| E161 | Cœur | SQL comments using leading double minus | |
| F021 | Cœur | Basic information schema | |
| F021−01 | Cœur | COLUMNS view | |
| F021−02 | Cœur | TABLES view | |
| F021−03 | Cœur | VIEWS view | |
| F021−04 | Cœur | TABLE_CONSTRAINTS view | |
| F021−05 | Cœur | REFERENTIAL_CONSTRAINTS view | |
| F021−06 | Cœur | CHECK_CONSTRAINTS view | |

| | | | |
|---|---|---|---|
| F031 | Cœur | Basic schema manipulation | |
| F031−01 | Cœur | CREATE TABLE statement to create persistent base tables | |
| F031−02 | Cœur | CREATE VIEW statement | |
| F031−03 | Cœur | GRANT statement | |
| F031−04 | Cœur | ALTER TABLE statement: ADD COLUMN clause | |
| F031−13 | Cœur | DROP TABLE statement: RESTRICT clause | |
| F031−16 | Cœur | DROP VIEW statement: RESTRICT clause | |
| F031−19 | Cœur | REVOKE statement: RESTRICT clause | |
| F032 | | CASCADE drop behavior | |
| F033 | | ALTER TABLE statement: DROP COLUMN clause | |
| F034 | | Extended REVOKE statement | |
| F034−01 | | REVOKE statement performed by other than the owner of a schema object | |
| F034−02 | | REVOKE statement: GRANT OPTION FOR clause | |
| F034−03 | | REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION | |
| F041 | Cœur | Basic joined table | |
| F041−01 | Cœur | Inner join (but not necessarily the INNER keyword) | |
| F041−02 | Cœur | INNER keyword | |
| F041−03 | Cœur | LEFT OUTER JOIN | |
| F041−04 | Cœur | RIGHT OUTER JOIN | |
| F041−05 | Cœur | Outer joins can be nested | |
| F041−07 | Cœur | The inner table in a left or right outer join can also be used in an inner join | |
| F041−08 | Cœur | All comparison operators are supported (rather than just =) | |
| F051 | Cœur | Basic date and time | |
| F051−01 | Cœur | DATE data type (including support of DATE literal) | |
| F051−02 | Cœur | TIME data type (including support of TIME literal) with fractional seconds precision of at least 0 | |
| F051−03 | Cœur | TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6 | |
| F051−04 | Cœur | Comparison predicate on DATE, TIME, and TIMESTAMP data types | |
| F051−05 | Cœur | Explicit CAST between datetime types and character types | |
| F051−06 | Cœur | CURRENT_DATE | |
| F051−07 | Cœur | LOCALTIME | |
| F051−08 | Cœur | LOCALTIMESTAMP | |
| F052 | Enhanced datetime facilities | Intervals and datetime arithmetic | |
| F081 | Cœur | UNION and EXCEPT in views | |

| | | | |
|---|---|---|---|
| F111−02 | | READ COMMITTED isolation level | |
| F131 | Cœur | Grouped operations | |
| F131−01 | Cœur | WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views | |
| F131−02 | Cœur | Multiple tables supported in queries with grouped views | |
| F131−03 | Cœur | Set functions supported in queries with grouped views | |
| F131−04 | Cœur | Subqueries with GROUP BY and HAVING clauses and grouped views | |
| F131−05 | Cœur | Single row SELECT with GROUP BY and HAVING clauses and grouped views | |
| F171 | | Multiple schemas per user | |
| F191 | Enhanced integrity management | Referential delete actions | |
| F201 | Cœur | CAST function | |
| F221 | Cœur | Explicit defaults | |
| F222 | | INSERT statement: DEFAULT VALUES clause | |
| F231 | | Privilege Tables | |
| F231−01 | | TABLE_PRIVILEGES view | |
| F231−02 | | COLUMN_PRIVILEGES view | |
| F231−03 | | USAGE_PRIVILEGES view | |
| F251 | | Domain support | |
| F261 | Cœur | CASE expression | |
| F261−01 | Cœur | Simple CASE | |
| F261−02 | Cœur | Searched CASE | |
| F261−03 | Cœur | NULLIF | |
| F261−04 | Cœur | COALESCE | |
| F271 | | Compound character literals | |
| F281 | | LIKE enhancements | |
| F302 | OLAP facilities | INTERSECT table operator | |
| F302−01 | OLAP facilities | INTERSECT DISTINCT table operator | |
| F302−02 | OLAP facilities | INTERSECT ALL table operator | |
| F304 | OLAP facilities | EXCEPT ALL table operator | |
| F311−01 | Cœur | CREATE SCHEMA | |
| F311−02 | Cœur | CREATE TABLE for persistent base tables | |
| F311−03 | Cœur | CREATE VIEW | |
| F311−05 | Cœur | GRANT statement | |
| F321 | | User authorization | |
| F361 | | Subprogram support | |
| F381 | | Extended schema manipulation | |
| F381−01 | | ALTER TABLE statement: ALTER COLUMN clause | |
| F381−02 | | ALTER TABLE statement: ADD CONSTRAINT clause | |
| F381−03 | | ALTER TABLE statement: DROP CONSTRAINT clause | |

| | | | |
|---|---|---|---|
| F391 | | Long identifiers | |
| F401 | OLAP facilities | Extended joined table | |
| F401−01 | OLAP facilities | NATURAL JOIN | |
| F401−02 | OLAP facilities | FULL OUTER JOIN | |
| F401−03 | OLAP facilities | UNION JOIN | |
| F401−04 | OLAP facilities | CROSS JOIN | |
| F411 | Enhanced datetime facilities | Time zone specification | |
| F421 | | National character | |
| F431 | | Read−only scrollable cursors | |
| F431−01 | | FETCH with explicit NEXT | |
| F431−02 | | FETCH FIRST | |
| F431−03 | | FETCH LAST | |
| F431−04 | | FETCH PRIOR | |
| F431−05 | | FETCH ABSOLUTE | |
| F431−06 | | FETCH RELATIVE | |
| F441 | | Extended set function support | |
| F471 | Cœur | Scalar subquery values | |
| F481 | Cœur | Expanded NULL predicate | |
| F491 | Enhanced integrity management | Constraint management | |
| F501 | Cœur | Features and conformance views | |
| F501−01 | Cœur | SQL_FEATURES view | |
| F501−02 | Cœur | SQL_SIZING view | |
| F501−03 | Cœur | SQL_LANGUAGES view | |
| F502 | | Enhanced documentation tables | |
| F502−01 | | SQL_SIZING_PROFILES view | |
| F502−02 | | SQL_IMPLEMENTATION_INFO view | |
| F502−03 | | SQL_PACKAGES view | |
| F511 | | BIT data type | |
| F531 | | Temporary tables | |
| F555 | Enhanced datetime facilities | Enhanced seconds precision | |
| F561 | | Full value expressions | |
| F571 | | Truth value tests | |
| F591 | OLAP facilities | Derived tables | |
| F611 | | Indicator data types | |
| F651 | | Catalog name qualifiers | |
| F701 | Enhanced integrity management | Referential update actions | |
| F711 | | ALTER domain | |
| F761 | | Session management | |

| | | | |
|---|---|---|---|
| F771 | | Connection management | |
| F781 | | Self−referencing operations | |
| F791 | | Insensitive cursors | |
| F801 | | Full set function | |
| S071 | Enhanced object support | SQL paths in function and type name resolution | |
| S111 | Enhanced object support | ONLY in query expressions | |
| S211 | Enhanced object support, SQL/MM support | User−defined cast functions | |
| T031 | | BOOLEAN data type | |
| T141 | | SIMILAR predicate | |
| T151 | | DISTINCT predicate | |
| T171 | | LIKE clause in table definition | |
| T191 | Enhanced integrity management | Referential action RESTRICT | |
| T201 | Enhanced integrity management | Comparable data types for referential constraints | |
| T211−01 | Enhanced integrity management, Active database | Triggers activated on UPDATE, INSERT, or DELETE of one base table | |
| T211−02 | Enhanced integrity management, Active database | BEFORE triggers | |
| T211−03 | Enhanced integrity management, Active database | AFTER triggers | |
| T211−04 | Enhanced integrity management, Active database | FOR EACH ROW triggers | |
| T211−07 | Enhanced integrity management, Active database | TRIGGER privilege | |
| T212 | Enhanced integrity management | Enhanced trigger capability | |
| T231 | | SENSITIVE cursors | |
| T241 | | START TRANSACTION statement | |
| T312 | | OVERLAY function | |
| T321−01 | Cœur | User−defined functions with no overloading | |
| T321−03 | Cœur | Function invocation | |
| T321−06 | Cœur | ROUTINES view | |
| T321−07 | Cœur | PARAMETERS view | |
| T322 | PSM, SQL/MM support | Overloading of SQL−invoked functions and procedures | |

| | | | |
|---|---|---|---|
| T323 | | Explicit security for external routines | |
| T351 | | Bracketed SQL comments (/*...*/ comments) | |
| T441 | | ABS and MOD functions | |
| T501 | | Enhanced EXISTS predicate | |
| T551 | | Optional key words for default syntax | |
| T581 | | Regular expression substring function | |
| T591 | | UNIQUE constraints of possibly NULL columns | |

# D.2. Fonctionnalités non supportées

Les fonctionnalités suivantes définies dans SQL:2003 ne sont pas implantées dans cette version de PostgreSQL. Dans certains cas, des fonctionnalités similaires sont disponibles.

| Identifiant | Paquetage | Description | Commentaire |
|---|---|---|---|
| B011 | Cœur | Ada embarqué | |
| B013 | Cœur | COBOL embarqué | |
| B014 | Cœur | Fortran embarqué | |
| B015 | Cœur | MUMPS embarqué | |
| B016 | Cœur | Pascal embarqué | |
| B017 | Cœur | PL/I embarqué | |
| B031 | | SQL dynamique de base | |
| B032 | | SQL dynamique étendu | |
| B032−01 | | Instruction <describe input> | |
| B041 | | Extensions aux déclarations des exceptions du SQL embarqué | |
| B051 | | Droits d'exécution améliorés | |
| E081 | Cœur | Privilèges de base | |
| E081−05 | Cœur | Privilège UPDATE au niveau de la colonne | |
| E081−07 | Cœur | Privilège REFERENCES au niveau de la colonne | |
| E121 | Cœur | Support du curseur basique | |
| E121−04 | Cœur | Instruction OPEN | |
| E121−06 | Cœur | Instruction UPDATE positionnée | |
| E121−07 | Cœur | Instruction DELETE positionnée | |
| E153 | Cœur | Requêtes composées de sous−requêtes, avec mise à jour possible | |
| E171 | Cœur | Support de SQLSTATE | |
| E182 | Cœur | Langage de module | |
| F111 | | Niveaux d'isolation autres que SERIALIZABLE | |
| F111−01 | | Niveau d'isolation READ UNCOMMITTED | |
| F111−03 | | Niveau d'isolation REPEATABLE READ | |

| | | | |
|---|---|---|---|
| F121 | | Gestion basique des diagnostiques | |
| F121−01 | | Instruction GET DIAGNOSTICS | |
| F121−02 | | Instruction SET TRANSACTION : clause DIAGNOSTICS SIZE | |
| F181 | | Support des modules multiples | |
| F291 | | Prédicat UNIQUE | |
| F301 | | CORRESPONDING dans les expressions de requêtes | |
| F311 | Cœur | Instruction de définition de schéma | |
| F311−04 | Cœur | CREATE VIEW : WITH CHECK OPTION | |
| F341 | | Tables d'utilisation | |
| F451 | | Définition des ensembles de caractères | |
| F461 | | Ensembles de caractères nommés | |
| F521 | Gestion améliorée de l'intégrité | Affectations | |
| F641 | Fonctionnalités OLAP | Constructeurs de ligne et de table | |
| F661 | | Tables simples | |
| F671 | Gestion avancée de l'intégrité | Sous−requête dans CHECK | Laissé de côté intentionnellement |
| F691 | | Collation et traduction | |
| F721 | | Contraintes pouvant être différées | Seulement pour les clés étrangères |
| F731 | | Privilège INSERT sur les colonnes | |
| F741 | | Types MATCH référentiels | Pas encore de correspondance partielle |
| F751 | | Améliorations du CHECK sur les vues | |
| F811 | | Positionnement de drapeaux étendu | |
| F812 | Cœur | Positionnement basique de drapeaux | |
| F813 | | Positionnement étendu de drapeaux étendus pour << Core SQL Flagging >> et << Catalog Lookup >> seulement | |
| F821 | | Références des tables locales | |
| F831 | | Mise à jour d'un curseur complet | |
| F831−01 | | Curseurs navigables modifiables | |
| F831−02 | | Curseurs ordonnés modifiables | |
| S011 | Cœur | Types de données distincts | |
| S011−01 | Cœur | Vue USER_DEFINED_TYPES | |
| S023 | Support des objets basiques, support de SQL/MM | Types structurés de base | |
| S024 | Support des objets avancés, support de SQL/MM | Types structurés avancés | |
| S041 | | Types de références de base | |

| | | | |
|---|---|---|---|
| | Support des objets basiques | | |
| S043 | Support des objets avancés | Types de références avancés | |
| S051 | Support des objets basiques | Create table of type | |
| S081 | Support des objets avancés | Sous−tables | |
| S091 | Support de SQL/MM | Support des tableaux basiques | Les tableaux PostgreSQL sont différents |
| S091−01 | Support de SQL/MM | Tableaux de type de données intégrés | |
| S091−02 | Support de SQL/MM | Tableaux de types distincts | |
| S091−03 | Support de SQL/MM | Expressions de tableaux | |
| S092 | Support de SQL/MM | Tableaux de types définis par l'utilisateur | |
| S094 | | Tableaux de types reference | |
| S151 | Support des objets basiques | Prédicat de type | |
| S161 | Support des objets avancés | Traitement des sous−types | |
| S201 | | Routines SQL sur les tableaux | |
| S201−01 | | Paramètres de type tableau | |
| S201−02 | | Tableau comme type de résultat d'une fonction | |
| S231 | Support des objets avancés | Pointeurs des types structurés | |
| S232 | | Pointeurs de tableaux | |
| S241 | Support des objets avancés | Fonctions de transformation | |
| S251 | | Tris définis par l'utilisateur | |
| S261 | | Méthode de type spécifique | |
| T011 | | Timestamp dans le schéma d'informations | |
| T041 | Support des objets basiques | Support du type de données LOB basique | |
| T041−01 | Support des objets basiques | Type de données BLOB | |
| T041−02 | Support des objets basiques | Type de données CLOB | |
| T041−03 | Support des objets basiques | Fonctions POSITION, LENGTH, LOWER, TRIM, UPPER et SUBSTRING pour les types de données LOB | |
| T041−04 | Support des objets basiques | Concaténation de types de données LOB | |
| T041−05 | Support des objets basiques | Limite des LOB : non préhensible | |
| T042 | | Support du type de données LOB étendu | |

| | | | |
|---|---|---|---|
| T051 | | Types de ligne | |
| T111 | | Jointure, union et colonnes modifiables | |
| T121 | | WITH (sans RECURSIVE) dans l'expression d'une requête | |
| T131 | | Requête récursive | |
| T211 | Gestion avancée de l'intégrité, base de données active | Fonctionnalités basiques des déclencheurs | |
| T211−05 | Gestion avancée de l'intégrité, base de données active | Possibilité de spécifier une condition de recherche qui doit être vraie avant que le déclencheur ne soit invoqué | |
| T211−06 | Gestion avancée de l'intégrité, base de données active | Support de règles en exécution pour l'interaction entre les déclencheurs et les contraintes | |
| T211−08 | Gestion avancée de l'intégrité, base de données active | Déclencheurs multiples pour le même événement exécutés dans l'ordre de leur création | Laissé de côté intentionnellement |
| T251 | | Instruction SET TRANSACTION : option LOCAL | |
| T261 | | Transactions chaînées | |
| T271 | | Points de sauvegarde | |
| T281 | | Privilège SELECT avec granularité des colonnes | |
| T301 | | Dépendances fonctionnelles | |
| T321 | Cœur | Routines de base SQL | |
| T321−02 | Cœur | Procédures stockées définies par l'utilisateur sans surchargement | |
| T321−04 | Cœur | Instruction CALL | |
| T321−05 | Cœur | Instruction RETURN | |
| T331 | | Rôles basiques | |
| T332 | | Rôles étendus | |
| T401 | | INSERT dans un curseur | |
| T411 | | Instruction UPDATE : option SET ROW | |
| T431 | Fonctionnalités OLAP | Opérations CUBE et ROLLUP | |
| T461 | | Prédicat BETWEEN symétrique | |
| T471 | | Ensemble de résultats comme valeur de retour | |
| T491 | | Table dérivée LATERAL | |
| T511 | | Comptage des transactions | |
| T541 | | Références de tables modifiables | |
| T561 | | Pointeurs rémanents | |
| T571 | | Fonctions externes appelées via SQL renvoyant un tableau | |
| T601 | | Références locales de curseur | |

# Annexe E. Notes de version

## E.1. Version 8.0.5

**Date de sortie :** 2005−12−12

Cette version contient quelques corrections sur la 8.0.4.

### E.1.1. Migration vers la version 8.0.5

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une version 8.0.X. Néanmoins, si vous mettez à jour à partir d'une version antérieure à la 8.0.3, voir les notes de sortie de la 8.0.3.

### E.1.2. Modifications

- Correction d'un cas extrême dans la gestion des traces des transactions

  Il existait une petite possibilité pour laquelle une opération d'entrée/sortie pouvait être initiée pour la mauvaise page, amenant à un échec Assert ou à une corruption de données.
- Correction des problèmes du processus d'écriture en tâche de fond après avoir récupéré des erreurs (Tom)

  Le processus d'écriture en tâche de fond was found to leak buffer pins after write errors. Bien que non fatal en soi, ceci pourrait amener des blocages mystérieux sur des commandes VACUUM ultérieures.
- Empêche un échec si le client envoie un message de protocole Bind quand la transaction en cours est déjà annulée
- Corrections sur `/contrib/ltree` (Teodor)
- Corrections de compilation sur AIX et HPUX (Tom)
- Tente de nouveau les lectures et écritures de fichiers après une erreur NO_SYSTEM_RESOURCES sous Windows (Qingqing Zhou)
- Correction d'un échec intermittent quand `log_line_prefix` inclut `%i`
- Correction du problème de performance de psql avec les scripts longs sur Windows (Merlin Moncure)
- Correction de mises à jour manquantes du fichier `pg_group`
- Correction d'une vieille erreur sur les jointures externes

  Ce bogue a quelque fois causé une fausse erreur << RIGHT JOIN is only supported with merge−joinable join conditions >>.
- Retardait l'initialisation du fuseau horaire jusqu'à la création du fichier `postmaster.pid`

  Ceci évite la confusion des scripts de démarrage qui s'attendent à voir apparaître rapidement le fichier pid.
- Empêchement d'un arrêt brutal dans pg_autovacuum quand une table a été supprimée
- Correction de problèmes avec des références de lignes entières (`foo.*`) pour des résultats de sous−requêtes

# E.2. Version 8.0.4

**Date de sortie :** 2005−10−04

Cette version contient quelques corrections de la 8.0.3.

## E.2.1. Migration vers la version 8.0.4

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une version 8.0.X. Néanmoins, si vous mettez à jour à partir d'une version antérieure à la 8.0.3, voir les notes de sortie de la 8.0.3.

## E.2.2. Modifications

- Fix error that allowed `VACUUM` to remove `ctid` chains too soon, and add more checking in code that follows `ctid` links

  This fixes a long−standing problem that could cause crashes in very rare circumstances.
- Fix `CHAR()` to properly pad spaces to the specified length when using a multiple−byte character set (Yoshiyuki Asaba)

  In prior releases, the padding of `CHAR()` was incorrect because it only padded to the specified number of bytes without considering how many characters were stored.
- Force a checkpoint before committing `CREATE DATABASE`

  This should fix recent reports of << index is not a btree >> failures when a crash occurs shortly after `CREATE DATABASE`.
- Fix the sense of the test for read−only transaction in `COPY`

  The code formerly prohibited `COPY TO`, where it should prohibit `COPY FROM`.
- Handle consecutive embedded newlines in `COPY` CSV−mode input
- Fix `date_trunc(week)` for dates near year end
- Fix planning problem with outer−join ON clauses that reference only the inner−side relation
- Further fixes for `x FULL JOIN y ON true` corner cases
- Fix overenthusiastic optimization of `x IN (SELECT DISTINCT ...)` and related cases
- Fix mis−planning of queries with small `LIMIT` values due to poorly thought out << fuzzy >> cost comparison
- Make `array_in` and `array_recv` more paranoid about validating their OID parameter
- Fix missing rows in queries like `UPDATE a=... WHERE a...` with GiST index on column `a`
- Improve robustness of datetime parsing
- Improve checking for partially−written WAL pages
- Improve robustness of signal handling when SSL is enabled
- Improve MIPS and M68K spinlock code
- Don't try to open more than `max_files_per_process` files during postmaster startup
- Various memory leakage fixes
- Various portability improvements
- Update timezone data files
- Improve handling of DLL load failures on Windows
- Improve random−number generation on Windows
- Make `psql -f filename` return a nonzero exit code when opening the file fails

- Change pg_dump to handle inherited check constraints more reliably
- Fix password prompting in pg_restore on Windows
- Fix PL/PgSQL to handle `var := var` correctly when the variable is of pass−by−reference type
- Fix PL/Perl `%_SHARED` so it's actually shared
- Fix `contrib/pg_autovacuum` to allow sleep intervals over 2000 sec
- Update `contrib/tsearch2` to use current Snowball code

# E.3. Version 8.0.3

**Date de sortie :** 2005−05−09

Cette version contient quelques corrections sur la 8.0.2, dont des correctifs sur des trous de sécurité.

## E.3.1. Migration vers la version 8.0.3

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une version 8.0.X. Néanmoins, c'est une façon possible de gérer deux problèmes de sécurité significatifs, qui ont été trouvé dans le contenu initial des catalogues systèmes des versions 8.0.X. Une séquence sauvegarde/initdb/restauration utilisant le initdb de la 8.0.3 corrigera automatiquement ces problèmes.

Le problème de sécurité le plus important est que les fonctions de conversion du codage des ensembles de caractères peuvent être appelées à partir de commandes SQL par des utilisateurs non pribilégiés alors que les fonctions n'ont pas été conçues pour un tel usage et ne sont pas sécurisées contre des choix rusés des arguments. Le correctif implique de modifier la liste déclarée des arguments de ces fonctions pour qu'elles ne soient plus appelées à partir des commandes SQL. (Ceci n'affecte pas leur utilisation normale par la machinerie de conversion du codage.)

Le problème le moins important est que le module `contrib/tsearch2` crée plusieurs fonctions déclarant par erreur renvoyées `internal` alors qu'elles n'acceptent pas les arguments `internal`. Ceci casse la sûreté des types pour toutes les fonctions utilisant des arguments `internal`.

Il est fortement recommandé que toutes les installations corrigent ces erreurs, soit par un initdb soit en suivant les procédures de réparation données ci−dessous. Les erreurs permettent au moins à des utilisateurs non privilégiés de la base de données d'arrêter brutalement leur processus serveur et pourraient permettre à des utilisateurs non privilégiés de gagner les privilèges d'un superutilisateur.

Si vous souhaitez ne pas lancer d'initdb, exécutez la même procédure de réparation manuelle que celle montrée dans les notes de la version 7.4.8.

## E.3.2. Modifications

- Modification de la signature de la fonction de codage pour empêcher toute mauvaise utilisation
- Modification de `contrib/tsearch2` pour éviter une utilisation non sécurisée des résultats `INTERNAL` de la fonction
- Prévention contre un second paramètre incorrect pour `record_out`
- Réparation d'une ancienne condition qui permettait à une transaction d'être vue comme validée pour certains buts (par exemple SELECT FOR UPDATE) légèrement plus tôt que dans les autres buts

Ceci est un bogue extrêmement sérieux car il pourrait amener à des incohérences apparents des données et visibles brièvement par les applications.

- Réparation d'une condition entre extension de relation et VACUUM

  Théoriquement, ceci pourrait avoir causé des pertes d'une page de données tout juste insérées, bien que la probabilité d'un tel scénario semble extrêmement faible. Il n'existe pas de cas connus où cela a provoqué plus qu'un échec d'Assert.
- Correction des comparaisons des valeurs `TIME WITH TIME ZONE`

  Le code de comparaison était mauvais dans le cas où le commutateur de configuration `--enable-integer-datetimes` avait été utilisé. NOTE : si vous avez un index sur une colonne `TIME WITH TIME ZONE`, il sera nécessaire de le `REINDEX`er après avoir installé cette mise à jour parce que ce correctif corrige l'ordre de tri des valeurs de colonnes.
- Correction de `EXTRACT(EPOCH)` pour les valeurs de type `TIME WITH TIME ZONE`
- Correction d'un mauvais affichage des secondes fractionnelles négatives dans les valeurs `INTERVAL`

  Cette erreur est seulement survenue quand l'option de configuration `--enable-integer-datetimes` avait été utilisée.
- Correction de pg_dump pour qu'il sauvegarde correctement les noms des déclencheurs contenant `%` (Neil)
- Encore plus de correctifs 64 bits pour `contrib/intagg`
- Empêche une optimisation incorrecte des fonctions renvoyant `RECORD`
- Empêche un arrêt brutal sur `COALESCE(NULL,NULL)`
- Correction du makefile de Borland pour libpq
- Correction de `contrib/btree_gist` pour le type `timetz` (Teodor)
- Fait que `pg_ctl` vérifie le PID trouvé dans `postmaster.pid` pour voir s'il s'agit toujours d'un processus en vie
- Correction des problèmes de `pg_dump`/`pg_restore` causés par l'ajout de tampons horaires dans les sauvegardes
- Correction de l'interaction entre les curseurs et le lancement de déclencheurs déferrés lors de validation de transaction
- Correction d'une perte mémoire dans les fonctions SQL renvoyant des types de données passés par référence

# E.4. Release 8.0.2

**Date de sortie :** 2005−04−07

Cette version contient quelques corrections de la 8.0.1.

## E.4.1. Migration vers la version 8.0.2

Une sauvegarde/restauration n'est pas requise pour ceux utilisant les versions 8.0.*. Cette version met à jour le numéro de version majeure des bibliothèques PostgreSQL, donc il pourrait être nécessaire de re−lier certaines applications utilisateurs si elles n'arrivent pas à trouver la bonne bibliothèque partagée.

## E.4.2. Modifications

- Incrémentation du numéro de version majeure pour toutes les bibliothèques d'interface (Bruce)

  Ceci aurait dû être fait depuis la 8.0.0. C'est requis pour que les versions 7.4.X des applications clientes de PostgreSQL, comme psql, puissent être utilisées sur la même machine que les applications 8.0.X. Ceci pourrait nécessiter une nouvelle édition des liens des applications clientes utilisant ces bibliothèques.
- Ajout du paramètre `wal_sync_method`, uniquement sous Windows, pour`fsync_writethrough` (Magnus, Bruce)

  Ce paramètre fait que PostgreSQL écrit les WAL via le cache d'écriture du disque. Ce comportement était appelé auparavant `fsync` mais a été renommé car il agit complètement différemment de `fsync` sur les autres plateformes.
- Active la valeur `open_datasync` du paramètre `wal_sync_method` sur Windows et le prend par défaut sur cette plateforme (Magnus, Bruce)

  Comme la valeur par défaut n'est plus `fsync_writethrough`, une perte de données est possible lors d'une coupure de courant si le disque a activé le cache en écriture. Pour désactiver ce cache sur Windows, à partir du Device Manager, choisissez les propriétés du disque, puis `Policies`.
- Le nouvel algorithme de gestion du cache 2Q remplace ARC (Tom)

  Ceci a été accompli pour éviter un brevet logiciel en attente sur ARC. Le code 2Q pourrait être un peu plus lent qu'ARC pour certains charges de travail. Un meilleur algorithme de gestion de cache apparaîtra pour la version 8.1.
- Ajustements du planificateur pour améliorer le comportement sur les tables nouvellement créées (Tom)
- Autorise plpgsql d'affecter un élément initialement `NULL` à un tableau (Tom)

  Auparavant, le tableau resterait `NULL` mais maintenant, il devient un tableau à un seul élément. Le moteur SQL principal a été modifié pour gérer un `UPDATE` d'une valeur de tableau NULL de cette façon dans la version 8.0 mais le cas similaire dans plpgsql a été oublié.
- Conversion de `\r\n` et `\r` en `\n` dans le corps des fonctions en plpython (Michael Fuhr)

  Ceci empêche les erreurs de syntaxe quand le code plpython a été écrit sur un client Windows ou Mac.
- Autorise les curseurs SPI pour gérer les commandes outils qui renvoient des lignes comme `EXPLAIN` (Tom)
- Correction de l'échec de `CLUSTER` après `ALTER TABLE SET WITHOUT OIDS` (Tom)
- Réduction de l'utilisation de la mémoire d'`ALTER TABLE ADD COLUMN` (Neil)
- Correction d'`ALTER LANGUAGE RENAME` (Tom)
- Documentation des options `register` et `unregister` de pg_ctl spécifiques à Windows (Magnus)
- S'assure que les opérations effectuées lors de l'arrêt du moteur sont comptées par le récupérateur de statistiques

  Ceci devrait résoudre les rapports de pg_autovacuum ne nettoyant pas assez fréquemment les catalogues systèmes — il n'était pas tenu au courant des suppressions de catalogues causées par la suppression de table temporaire lors de l'arrêt du moteur.
- Modification de la valeur par défaut, sous Windows, pour le paramètre de configuration `log_destination` par `eventlog` (Magnus)

Par défaut, un serveur fonctionnant sous Windows enverra maintenant la sortie des traces au gestionnaire des traces de Windows plutôt que sur la sortie standard des erreurs.
- Rend l'authentification Kerberos fonctionnel sur Windows (Magnus)
- Autorise `ALTER DATABASE RENAME` pour les superutilisateurs qui n'ont pas le droit CREATEDB (Tom)
- Modification des entrées WAL pour les commandes `CREATE` et `DROP DATABASE` pour ne pas spécifier les chemins absolus (Tom)

  Ceci permet la récupération à un instant sur une machine différente avec des emplacements de bases de données pouvant être différents. Notez que `CREATE TABLESPACE` est toujours hasardeux dans certaines situations.
- Correction d'un arrêt brutal suivant l'arrêt d'un serveur avec une transaction ouverte qui a créé une table et ouvert un curseur sur elle (Tom)
- Correction de `array_map()` pour qu'il puisse appeler des fonctions de langages de procédures (Tom)
- Plusieurs corrections pour `contrib/tsearch2` et `contrib/btree_gist` (Teodor)
- Correction de quelques arrêts brutaux de certaines fonctions de `contrib/pgcrypto` concernant certaines plateformes (Marko Kreen)
- Correction de `contrib/intagg` pour les plateformes 64 bits (Tom)
- Correction de bogues d'ecpg dans l'analyse d'instructions `CREATE` (Michael)
- Contournement d'un bogue de gcc sur powerpc et amd64 posant des problèmes sur ecpg (Christof Petig)
- N'utilise pas les versions `upper()`, `lower()` et `initcap()` prenant compte de la locale utilisée lorsque la locale est `C` (Bruce)

  Ceci permet à ces fonctions de fonctionner sur les plateformes qui génèrent des erreurs pour les données non 7 bits lorsque la locale est `C`.
- Correction de `quote_ident()` pour mettre entre guillemets les noms qui correspondent à des mots clés (Tom)
- Correction de `to_date()` pour correspondre raisonnablement quand les champs `CC` et `YY` sont utilisées tous les deux (Karel)
- Empêche `to_char(interval)` d'échouer quand lui est passée une intervalle de zéro mois (Tom)
- Correction du mauvais mois renvoyé par `date_trunc('week')` (Bruce)

  `date_trunc('week')` a renvoyé la mauvaise année pour les quelques premiers jours de Janvier pour certaines années.
- Utilise la longueur par défaut du masque pour la classe `D` d'adresses pour les types de données `INET` (Tom)

## E.5. Release 8.0.1

**Date de sortie :** 2005−01−31

Cette version contient une grande variété de corrections provenant de la version 8.0.0, incluant plusieurs problèmes relatifs sur la sécurité.

### E.5.1. Migration vers la version 8.0.1

Une sauvegarde/restauration n'est pas requis pour ceux utilisant la version 8.0.0.

### E.5.2. Modifications

- Interdiction de `LOAD` aux utilisateurs standards

  Sur les plateformes qui exécutent automatiquement les fonctions d'initialisation d'une bibliothèque partagée (ceci inclut au moins Windows et les Unix basés sur ELF), `LOAD` peut être utilisé pour que le serveur exécute un code arbitraire. Merci à NGS Software pour nous l'avoir indiqué.
- Vérification comme le créateur d'une fonction d'agrégat a le droit d'exécutr les fonctions de transition spécifiées

  Cet aperçu est rendu possible pour dépasser un déni de droit sur EXECUTE pour une fonction.
- Correctif de sécurité et de problèmes pour les 64 bits dans contrib/intagg
- Ajout du marquage STRICT nécessaire pour certaines fonctions de contrib functions (Kris Jurka)
- Évite l'écrasement de tampon quand la déclaration de curseur de plpgsql a trop de paramètres (Neil)
- Fait que `ALTER TABLE ADD COLUMN` force les contraintes de domaines dans tous les cas
- Correction d'une erreur de planification pour des jointures externes FULL et RIGHT

  Le résultat de la jointure a été pris par erreur pour être trié de la même façon que l'entrée gauche. This could not only deliver mis−sorted output to the user, but in case of nested merge joins could give outright wrong answers.
- Amélioration de la plantification des requêtes d'agrégats groupées
- `ROLLBACK TO` *`point_de_sauvegarde`* ferme les curseurs créés depuis le point de sauvegarde
- Correction de la taille inadéquate de la pile du serveur sur Windows
- Évite SHGetSpecialFolderPath() sur Windows (Magnus)
- Correction de quelques problèmes lors de l'exécution de pg_autovacuum en tant que service Windows (Dave Page)
- Corrections de plusieurs bogues mineurs dans pg_dump/pg_restore
- Correction d'une erreur de segmentation dans ecpg avec les structures nommées utilisées dans les typedefs (Michael)

## E.6. Release 8.0

**Release date :** 2005−01−19

### E.6.1. Overview

Major changes in this release:

Microsoft Windows Native Server
: This is the first PostgreSQL release to run natively on Microsoft Windows® as a server. It can run as a Windows service. This release supports NT−based Windows releases like Windows 2000, Windows XP, and Windows 2003. Older releases like Windows 95, Windows 98, and Windows ME are not supported because these operating systems do not have the infrastructure to support PostgreSQL. A separate installer project has been created to ease installation on Windows — see http://www.postgresql.org/ftp/win32/.

Although tested throughout our release cycle, the Windows port does not have the benefit of years of use in production environments that PostgreSQL has on Unix platforms. Therefore it should be treated with the same level of caution as you would a new product.

Previous releases required the Unix emulation toolkit Cygwin in order to run the server on Windows operating systems. PostgreSQL has supported native clients on Windows for many years.

Savepoints
: Savepoints allow specific parts of a transaction to be aborted without affecting the remainder of the transaction. Prior releases had no such capability; there was no way to recover from a statement failure within a transaction except by aborting the whole transaction. This feature is valuable for application writers who require error recovery within a complex transaction.

Point−In−Time Recovery
: In previous releases there was no way to recover from disk drive failure except to restore from a previous backup or use a standby replication server. Point−in−time recovery allows continuous backup of the server. You can recover either to the point of failure or to some transaction in the past.

Tablespaces
: Tablespaces allow administrators to select different file systems for storage of individual tables, indexes, and databases. This improves performance and control over disk space usage. Prior releases used initlocation and manual symlink management for such tasks.

Improved Buffer Management, `CHECKPOINT`, `VACUUM`
: This release has a more intelligent buffer replacement strategy, which will make better use of available shared buffers and improve performance. The performance impact of vacuum and checkpoints is also lessened.

Change Column Types
: A column's data type can now be changed with `ALTER TABLE`.

New Perl Server−Side Language
: A new version of the plperl server−side language now supports a persistent shared storage area, triggers, returning records and arrays of records, and SPI calls to access the database.

Comma−separated−value (CSV) support in `COPY`
: `COPY` can now read and write comma−separated−value files. It has the flexibility to interpret non−standard quoting and separation characters too.

## E.6.2. Migration to version 8.0

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- In `READ COMMITTED` serialization mode, volatile functions now see the results of concurrent transactions committed up to the beginning of each statement within the function, rather than up to the beginning of the interactive command that called the function.
- Functions declared `STABLE` or `IMMUTABLE` always use the snapshot of the calling query, and therefore do not see the effects of actions taken after the calling query starts, whether in their own transaction or other transactions. Such a function must be read−only, too, meaning that it cannot use any SQL commands other than `SELECT`.
- Non−deferred `AFTER` triggers are now fired immediately after completion of the triggering query, rather than upon finishing the current interactive command. This makes a différence when the triggering query occurred within a function: the trigger is invoked before the function proceeds to its next operation.

- Server configuration parameters `virtual_host` and `tcpip_socket` have been replaced with a more general parameter `listen_addresses`. Also, the server now listens on `localhost` by default, which eliminates the need for the `-i` postmaster switch in many scenarios.
- Server configuration parameters `SortMem` and `VacuumMem` have been renamed to `work_mem` and `maintenance_work_mem` to better reflect their use. The original names are still supported in `SET` and `SHOW`.
- Server configuration parameters `log_pid`, `log_timestamp`, and `log_source_port` have been replaced with a more general parameter `log_line_prefix`.
- Server configuration parameter `syslog` has been replaced with a more logical `log_destination` variable to control the log output destination.
- Server configuration parameter `log_statement` has been changed so it can selectively log just database modification or data définition statements. Server configuration parameter `log_duration` now prints only when `log_statement` prints the query.
- Server configuration parameter `max_expr_depth` parameter has been replaced with `max_stack_depth` which measures the physical stack size rather than the expression nesting depth. This helps prevent session termination due to stack overflow caused by recursive functions.
- The `length()` function no longer counts trailing spaces in `CHAR(n)` values.
- Casting an integer to `BIT(N)` selects the rightmost N bits of the integer, not the leftmost N bits as before.
- Updating an element or slice of a NULL array value now produces a non−NULL array result, namely an array containing just the assigned−to positions.
- Syntax checking of array input values has been tightened up considerably. Junk that was previously allowed in odd places with odd results now causes an error. Empty−string element values must now be written as `""`, rather than writing nothing. Also changed behavior with respect to whitespace surrounding array elements: trailing whitespace is now ignored, for symmetry with leading whitespace (which has always been ignored).
- Overflow in integer arithmetic operations is now detected and reported as an error.
- The arithmetic operators associated with the single−byte `"char"` data type have been removed.
- The `extract()` function (also called `date_part`) now returns the proper year for BC dates. It previously returned one less than the correct year. The function now also returns the proper values for millennium and century.
- `CIDR` values now must have their non−masked bits be zero. For exemple, we no longer allow `204.248.199.1/31` as a `CIDR` value. Such values should never have been accepted by PostgreSQL and will now be rejected.
- `EXECUTE` now returns a completion tag that matches the executed statement.
- psql's `\copy` command now reads or writes to the query's `stdin/stdout`, rather than psql's `stdin/stdout`. The previous behavior can be accessed via new `pstdin`/`pstdout` parameters.
- The JDBC client interface has been removed from the core distribution, and is now hosted at http://jdbc.postgresql.org.
- The Tcl client interface has also been removed. There are several Tcl interfaces now hosted at http://gborg.postgresql.org.
- The server now uses its own time zone database, rather than the one supplied by the operating system. This will provide consistent behavior across all platforms. In most cases, there should be little noticeable différence in time zone behavior, except that the time zone names used by `SET`/`SHOW TimeZone` may be different from what your platform provides.
- Configure's threading option no longer requires users to run tests or edit configuration files; threading options are now detected automatically.
- Now that tablespaces have been implemented, initlocation has been removed.
- The API for user−defined GiST indexes has been changed. The Union and PickSplit methods are now passed a pointer to a special `GistEntryVector` structure, rather than a `bytea`.

## E.6.3. Deprecated Features

Some aspects of PostgreSQL's behavior have been determined to be suboptimal. For the sake of backward compatibility these have not been removed in 8.0, but they are considered deprecated and will be removed in the next major release.

- The 8.1 release will remove the function `to_char()` for intervals.
- The server now warns of empty strings passed to `oid`/`float4`/`float8` data types, but continues to interpret them as zeroes as before. In the next major release, empty strings will be considered invalid input for these data types.
- By default, tables in PostgreSQL 8.0 and earlier are created with `OID`s. In the next release, this will *not* be the case: to create a table that contains `OID`s, the `WITH OIDS` clause must be specified or the `default_with_oids` configuration parameter must be set. Users are encouraged to explicitly specify `WITH OIDS` if their tables require OID for compatibility with future releases of PostgreSQL.

## E.6.4. Changes

Below you will find a detailed account of the changes between release 8.0 and the previous major release.

### E.6.4.1. Performance Improvements

- Support cross−data−type index usage (Tom)

  Before this change, many queries would not use an index if the data types did not match exactly. This improvement makes index usage more intuitive and consistent.
- New buffer replacement strategy that improves caching (Jan)

  Prior releases used a least−recently−used (LRU) cache to keep recently referenced pages in memory. The LRU algorithm did not consider the number of times a specific cache entry was accessed, so large table scans could force out useful cache pages. The new cache algorithm uses four separate lists to track most recently used and most frequently used cache pages and dynamically optimize their replacement based on the work load. This should lead to much more efficient use of the shared buffer cache. Administrators who have tested shared buffer sizes in the past should retest with this new cache replacement policy.
- Add subprocess to write dirty buffers periodically to reduce checkpoint writes (Jan)

  In previous releases, the checkpoint process, which runs every few minutes, would write all dirty buffers to the operating system's buffer cache then flush all dirty operating system buffers to disk. This resulted in a periodic spike in disk usage that often hurt performance. The new code uses a background writer to trickle disk writes at a steady pace so checkpoints have far fewer dirty pages to write to disk. Also, the new code does not issue a global `sync()` call, but instead `fsync()`s just the files written since the last checkpoint. This should improve performance and minimize degradation during checkpoints.
- Add ability to prolong vacuum to reduce performance impact (Jan)

  On busy systems, `VACUUM` performs many I/O requests which can hurt performance for other users. This release allows you to slow down `VACUUM` to reduce its impact on other users, though this increases the total duration of `VACUUM`.
- Improve B−tree index performance for duplicate keys (Dmitry Tkach, Tom)

This improves the way indexes are scanned when many duplicate values exist in the index.

- Use dynamically−generated table size estimates while planning (Tom)

  Formerly the planner estimated table sizes using the values seen by the last `VACUUM` or `ANALYZE`, both as to physical table size (number of pages) and number of rows. Now, the current physical table size is obtained from the kernel, and the number of rows is estimated by multiplying the table size by the row density (rows per page) seen by the last `VACUUM` or `ANALYZE`. This should produce more reliable estimates in cases where the table size has changed significantly since the last housekeeping command.
- Improved index usage with `OR` clauses (Tom)

  This allows the optimizer to use indexes in statements with many OR clauses that would not have been indexed in the past. It can also use multi−column indexes where the first column is specified and the second column is part of an `OR` clause.
- Improve matching of partial index clauses (Tom)

  The server is now smarter about using partial indexes in queries involving complex `WHERE` clauses.
- Improve performance of the GEQO optimizer (Tom)

  The GEQO optimizer is used to plan queries involving many tables (by default, twelve or more). This release speeds up the way queries are analyzed to decrease time spent in optimization.
- Miscellaneous optimizer improvements

  There is not room here to list all the minor improvements made, but numerous special cases work better than in prior releases.
- Improve lookup speed for C functions (Tom)

  This release uses a hash table to lookup information for dynamically loaded C functions. This improves their speed so they perform nearly as quickly as functions that are built into the server executable.
- Add type−specific `ANALYZE` statistics capability (Mark Cave−Ayland)

  This feature allows more flexibility in generating statistics for non−standard data types.
- `ANALYZE` now collects statistics for expression indexes (Tom)

  Expression indexes (also called functional indexes) allow users to index not just columns but the results of expressions and function calls. With this release, the optimizer can gather and use statistics about the contents of expression indexes. This will greatly improve the quality of planning for queries in which an expression index is relevant.
- New two−stage sampling method for `ANALYZE` (Manfred Koizar)

  This gives better statistics when the density of valid rows is very different in different regions of a table.
- Speed up `TRUNCATE` (Tom)

  This buys back some of the performance loss observed in 7.4, while still keeping `TRUNCATE` transaction−safe.

### E.6.4.2. Server Changes

- Add WAL file archiving and point−in−time recovery (Simon Riggs)
- Add tablespaces so admins can control disk layout (Gavin)
- Add a built−in log rotation program (Andreas Pflug)

  It is now possible to log server messages conveniently without relying on either syslog or an external log rotation program.
- Add new read−only server configuration parameters to show server compile−time settings: `block_size`, `integer_datetimes`, `max_function_args`, `max_identifier_length`, `max_index_keys` (Joe)
- Make quoting of `sameuser`, `samegroup`, and `all` remove special meaning of these terms in `pg_hba.conf` (Andrew)
- Use clearer IPv6 name `::1/128` for `localhost` in default `pg_hba.conf` (Andrew)
- Use CIDR format in `pg_hba.conf` exemples (Andrew)
- Rename server configuration parameters `SortMem` and `VacuumMem` to `work_mem` and `maintenance_work_mem` (Old names still supported) (Tom)

  This change was made to clarify that bulk operations such as index and foreign key creation use `maintenance_work_mem`, while `work_mem` is for workspaces used during query exécution.
- Allow logging of session disconnections using server configuration `log_disconnections` (Andrew)
- Add new server configuration parameter `log_line_prefix` to allow control of information emitted in each log line (Andrew)

  Available information includes user name, database name, remote IP address, and session start time.
- Remove server configuration parameters `log_pid`, `log_timestamp`, `log_source_port`; functionality superseded by `log_line_prefix` (Andrew)
- Replace the `virtual_host` and `tcpip_socket` parameters with a unified `listen_addresses` parameter (Andrew, Tom)

  `virtual_host` could only specify a single IP address to listen on. `listen_addresses` allows multiple addresses to be specified.
- Listen on localhost by default, which eliminates the need for the `-i` postmaster switch in many scenarios (Andrew)

  Listening on localhost (`127.0.0.1`) opens no new security holes but allows configurations like Windows and JDBC, which do not support local sockets, to work without special adjustments.
- Remove `syslog` server configuration parameter, and add more logical `log_destination` variable to control log output location (Magnus)
- Change server configuration parameter `log_statement` to take values `all`, `mod`, `ddl`, or `none` to select which queries are logged (Bruce)

  This allows administrators to log only data définition changes or only data modification statements.
- Some logging−related configuration parameters could formerly be adjusted by ordinary users, but only in the << more verbose >> direction. They are now treated more strictly: only superusers can set them. However, a superuser may use `ALTER USER` to provide per−user settings of these values for non−superusers. Also, it is now possible for superusers to set values of superuser−only configuration parameters via `PGOPTIONS`.
- Allow configuration files to be placed outside the data directory (mlw)

By default, configuration files are kept in the cluster's top directory. With this addition, configuration files can be placed outside the data directory, easing administration.

- Plan prepared queries only when first executed so constants can be used for statistics (Oliver Jowett)

  Prepared statements plan queries once and execute them many times. While prepared queries avoid the overhead of re−planning on each use, the quality of the plan suffers from not knowing the exact parameters to be used in the query. In this release, planning of unnamed prepared statements is delayed until the first exécution, and the actual parameter values of that exécution are used as optimization hints. This allows use of out−of−line parameter passing without incurring a performance penalty.
- Allow `DECLARE CURSOR` to take parameters (Oliver Jowett)

  It is now useful to issue `DECLARE CURSOR` in a `Parse` message with parameters. The parameter values sent at `Bind` time will be substituted into the exécution of the cursor's query.
- Fix hash joins and aggregates of `inet` and `cidr` data types (Tom)

  Release 7.4 handled hashing of mixed `inet` and `cidr` values incorrectly. (This bug did not exist in prior releases because they wouldn't try to hash either data type.)
- Make `log_duration` print only when `log_statement` prints the query (Ed L.)

---

### E.6.4.3. Query Changes

- Add savepoints (nested transactions) (Alvaro)
- Unsupported isolation levels are now accepted and promoted to the nearest supported level (Peter)

  The SQL specification states that if a database doesn't support a specific isolation level, it should use the next more restrictive level. This change complies with that recommendation.
- Allow `BEGIN WORK` to specify transaction isolation levels like `START TRANSACTION` does (Bruce)
- Fix table permission checking for cases in which rules generate a query type different from the originally submitted query (Tom)
- Implement dollar quoting to simplify single−quote usage (Andrew, Tom, David Fetter)

  In previous releases, because single quotes had to be used to quote a function's body, the use of single quotes inside the function text required use of two single quotes or other error−prone notations. With this release we add the ability to use "dollar quoting" to quote a block of text. The ability to use different quoting delimiters at different nesting levels greatly simplifies the task of quoting correctly, especially in complex functions. Dollar quoting can be used anywhere quoted text is needed.
- Make `CASE val WHEN compval1 THEN ...` evaluate `val` only once (Tom)

  `CASE` no longer evaluates the tested expression multiple times. This has benefits when the expression is complex or is volatile.
- Test `HAVING` before computing target list of an aggregate query (Tom)

  Fixes improper failure of cases such as `SELECT SUM(win)/SUM(lose) ... GROUP BY ... HAVING SUM(lose) > 0`. This should work but formerly could fail with divide−by−zero.
- Replace `max_expr_depth` parameter with `max_stack_depth` parameter, measured in kilobytes of stack size (Tom)

  This gives us a fairly bulletproof defense against crashing due to runaway recursive functions. Instead of measuring the depth of expression nesting, we now directly measure the size of the exécution stack.

- Allow arbitrary row expressions (Tom)

  This release allows SQL expressions to contain arbitrary composite types, that is, row values. It also allows functions to more easily take rows as arguments and return row values.
- Allow LIKE/ILIKE to be used as the operator in row and subselect comparisons (Fabien Coelho)
- Avoid locale−specific case conversion of basic ASCII letters in identifiers and keywords (Tom)

  This solves the << Turkish problem >> with mangling of words containing I and i. Folding of characters outside the 7−bit−ASCII set is still locale−aware.
- Improve syntax error reporting (Fabien, Tom)

  Syntax error reports are more useful than before.
- Change EXECUTE to return a completion tag matching the executed statement (Kris Jurka)

  Previous releases return an EXECUTE tag for any EXECUTE call. In this release, the tag returned will reflect the command executed.
- Avoid emitting NATURAL CROSS JOIN in rule listings (Tom)

  Such a clause makes no logical sense, but in some cases the rule decompiler formerly produced this syntax.

---

### E.6.4.4. Object Manipulation Changes

- Add COMMENT ON for casts, conversions, languages, operator classes, and large objects (Christopher)
- Add new server configuration parameter default_with_oids to control whether tables are created with OIDs by default (Neil)

  This allows administrators to control whether CREATE TABLE commands create tables with or without OID columns by default. (Note: the current factory default setting for default_with_oids is TRUE, but the default will become FALSE in future releases.)
- Add WITH / WITHOUT OIDS clause to CREATE TABLE AS (Neil)
- Allow ALTER TABLE DROP COLUMN to drop an OID column (ALTER TABLE SET WITHOUT OIDS still works) (Tom)
- Allow composite types as table columns (Tom)
- Allow ALTER ... ADD COLUMN with defaults and NOT NULL constraints; works per SQL spec (Rod)

  It is now possible for ADD COLUMN to create a column that is not initially filled with NULLs, but with a specified default value.
- Add ALTER COLUMN TYPE to change column's type (Rod)

  It is now possible to alter a column's data type without dropping and re−adding the column.
- Allow multiple ALTER actions in a single ALTER TABLE command (Rod)

  This is particularly useful for ALTER commands that rewrite the table (which include ALTER COLUMN TYPE and ADD COLUMN with a default). By grouping ALTER commands together, the table need be rewritten only once.
- Allow ALTER TABLE to add SERIAL columns (Tom)

  This falls out from the new capability of specifying defaults for new columns.

- Allow changing the owners of aggregates, conversions, databases, functions, operators, operator classes, schemas, types, and tablespaces (Christopher, Euler Taveira de Oliveira)

  Previously this required modifying the system tables directly.
- Allow temporary object creation to be limited to `SECURITY DEFINER` functions (Sean Chittenden)
- Add `ALTER TABLE ... SET WITHOUT CLUSTER` (Christopher)

  Prior to this release, there was no way to clear an auto−cluster specification except to modify the system tables.
- Constraint/Index/`SERIAL` names are now *`table_column_type`* with numbers appended to guarantee uniqueness within the schema (Tom)

  The SQL specification states that such names should be unique within a schema.
- Add `pg_get_serial_sequence()` to return a `SERIAL` column's sequence name (Christopher)

  This allows automated scripts to reliably find the `SERIAL` sequence name.
- Warn when primary/foreign key data type mismatch requires costly lookup
- New `ALTER INDEX` command to allow moving of indexes between tablespaces (Gavin)
- Make `ALTER TABLE OWNER` change dépendent sequence ownership too (Alvaro)

---

### E.6.4.5. Utility Command Changes

- Allow `CREATE SCHEMA` to create triggers, indexes, and sequences (Neil)
- Add `ALSO` keyword to `CREATE RULE` (Fabien Coelho)

  This allows `ALSO` to be added to rule creation to contrast it with `INSTEAD` rules.
- Add `NOWAIT` option to `LOCK` (Tatsuo)

  This allows the `LOCK` command to fail if it would have to wait for the requested lock.
- Allow `COPY` to read and write comma−separated−value (CSV) files (Andrew, Bruce)
- Generate error if the `COPY` delimiter and NULL string conflict (Bruce)
- `GRANT`/`REVOKE` behavior follows the SQL spec more closely
- Avoid locking conflict between `CREATE INDEX` and `CHECKPOINT` (Tom)

  In 7.3 and 7.4, a long−running B−tree index build could block concurrent `CHECKPOINT`s from completing, thereby causing WAL bloat because the WAL log could not be recycled.
- Database−wide `ANALYZE` does not hold locks across tables (Tom)

  This reduces the potential for deadlocks against other backends that want exclusive locks on tables. To get the benefit of this change, do not execute database−wide `ANALYZE` inside a transaction block (`BEGIN` block); it must be able to commit and start a new transaction for each table.
- `REINDEX` does not exclusively lock the index's parent table anymore

  The index itself is still exclusively locked, but readers of the table can continue if they are not using the particular index being rebuilt.
- Erase MD5 user passwords when a user is renamed (Bruce)

  PostgreSQL uses the user name as salt when encrypting passwords via MD5. When a user's name is changed, the salt will no longer match the stored MD5 password, so the stored password becomes useless. In this release a notice is generated and the password is cleared. A new password must then be assigned if the user is to be able to log in with a password.

- New pg_ctl `kill` option for Windows (Andrew)

  Windows does not have a `kill` command to send signals to backends so this capability was added to pg_ctl.
- Information schema improvements
- Add `--pwfile` option to initdb so the initial password can be set by GUI tools (Magnus)
- Detect locale/encoding mismatch in initdb (Peter)
- Add `register` command to pg_ctl to register Windows operating system service (Dave Page)

### E.6.4.6. Data Type and Function Changes

- More complete support for composite types (row types) (Tom)

  Composite values can be used in many places where only scalar values worked before.
- Reject non−rectangular array values as erroneous (Joe)

  Formerly, `array_in` would silently build a surprising result.
- Overflow in integer arithmetic operations is now detected (Tom)
- The arithmetic operators associated with the single−byte `"char"` data type have been removed.

  Formerly, the parser would select these operators in many situations where an << unable to select an operator >> error would be more appropriate, such as `NULL * NULL`. If you actually want to do arithmetic on a `"char"` column, you can cast it to integer explicitly.
- Syntax checking of array input values considerably tightened up (Joe)

  Junk that was previously allowed in odd places with odd results now causes an `ERROR`, for exemple, non−whitespace after the closing right brace.
- Empty−string array element values must now be written as `""`, rather than writing nothing (Joe)

  Formerly, both ways of writing an empty−string element value were allowed, but now a quoted empty string is required. The case where nothing at all appears will probably be considered to be a NULL element value in some future release.
- Array element trailing whitespace is now ignored (Joe)

  Formerly leading whitespace was ignored, but trailing whitespace between an element value and the delimiter or right brace was significant. Now trailing whitespace is also ignored.
- Emit array values with explicit array bounds when lower bound is not one (Joe)
- Accept `YYYY-monthname-DD` as a date string (Tom)
- Make `netmask` and `hostmask` functions return maximum−length mask length (Tom)
- Change factorial function to return `numeric` (Gavin)

  Returning `numeric` allows the factorial function to work for a wider range of input values.
- `to_char`/`to_date()` date conversion improvements (Kurt Roeckx, Fabien Coelho)
- Make `length()` disregard trailing spaces in `CHAR(n)` (Gavin)

  This change was made to improve consistency: trailing spaces are semantically insignificant in `CHAR(n)` data, so they should not be counted by `length()`.
- Warn about empty string being passed to `OID`/`float4`/`float8` data types (Neil)

  8.1 will throw an error instead.
- Allow leading or trailing whitespace in `int2`/`int4`/`int8`/`float4`/`float8` input routines (Neil)

- Better support for IEEE `Infinity` and `NaN` values in `float4`/`float8` (Neil)

  These should now work on all platforms that support IEEE−compliant floating point arithmetic.
- Add `week` option to `date_trunc()` (Robert Creager)
- Fix `to_char` for `1 BC` (previously it returned `1 AD`) (Bruce)
- Fix `date_part(year)` for BC dates (previously it returned one less than the correct year) (Bruce)
- Fix `date_part()` to return the proper millennium and century (Fabien Coelho)

  In previous versions, the century and millennium results had a wrong number and started in the wrong year, as compared to standard reckoning of such things.
- Add `ceiling()` as an alias for `ceil()`, and `power()` as an alias for `pow()` for standards compliance (Neil)
- Change `ln()`, `log()`, `power()`, and `sqrt()` to emit the correct `SQLSTATE` error codes for certain error conditions, as specified by SQL:2003 (Neil)
- Add `width_bucket()` function as defined by SQL:2003 (Neil)
- Add `generate_series()` functions to simplify working with numeric sets (Joe)
- Fix `upper/lower/initcap()` functions to work with multibyte encodings (Tom)
- Add boolean and bitwise integer `AND`/`OR` aggregates (Fabien Coelho)
- New session information functions to return network addresses for client and server (Sean Chittenden)
- Add function to determine the area of a closed path (Sean Chittenden)
- Add function to send cancel request to other backends (Magnus)
- Add `interval` plus `datetime` operators (Tom)

  The reverse ordering, `datetime` plus `interval`, was already supported, but both are required by the SQL standard.
- Casting an integer to `BIT(N)` selects the rightmost N bits of the integer (Tom)

  In prior releases, the leftmost N bits were selected, but this was deemed unhelpful, not to mention inconsistent with casting from bit to int.
- Require `CIDR` values to have all non−masked bits be zero (Kevin Brintnall)

---

### E.6.4.7. Server−Side Language Changes

- In `READ COMMITTED` serialization mode, volatile functions now see the results of concurrent transactions committed up to the beginning of each statement within the function, rather than up to the beginning of the interactive command that called the function.
- Functions declared `STABLE` or `IMMUTABLE` always use the snapshot of the calling query, and therefore do not see the effects of actions taken after the calling query starts, whether in their own transaction or other transactions. Such a function must be read−only, too, meaning that it cannot use any SQL commands other than `SELECT`. There is a considerable performance gain from declaring a function `STABLE` or `IMMUTABLE` rather than `VOLATILE`.
- Non−deferred `AFTER` triggers are now fired immediately after completion of the triggering query, rather than upon finishing the current interactive command. This makes a différence when the triggering query occurred within a function: the trigger is invoked before the function proceeds to its next operation. For exemple, if a function inserts a new row into a table, any non−deferred foreign key checks occur before proceeding with the function.
- Allow function parameters to be declared with names (Dennis Bjorklund)

  This allows better documentation of functions. Whether the names actually do anything dépends on the specific function language being used.

- Allow PL/pgSQL parameter names to be referenced in the function (Dennis Bjorklund)

  This basically creates an automatic alias for each named parameter.
- Do minimal syntax checking of PL/pgSQL functions at creation time (Tom)

  This allows us to catch simple syntax errors sooner.
- More support for composite types (row and record variables) in PL/pgSQL

  For exemple, it now works to pass a rowtype variable to another function as a single variable.
- Default values for PL/pgSQL variables can now reference previously declared variables
- Improve parsing of PL/pgSQL FOR loops (Tom)

  Parsing is now driven by presence of `".."` rather than data type of `FOR` variable. This makes no différence for correct functions, but should result in more understandable error messages when a mistake is made.
- Major overhaul of PL/Perl server−side language (Command Prompt, Andrew Dunstan)
- In PL/Tcl, SPI commands are now run in subtransactions. If an error occurs, the subtransaction is cleaned up and the error is reported as an ordinary Tcl error, which can be trapped with `catch`. Formerly, it was not possible to catch such errors.
- Accept `ELSEIF` in PL/pgSQL (Neil)

  Previously PL/pgSQL only allowed `ELSIF`, but many people are accustomed to spelling this keyword `ELSEIF`.

---

#### E.6.4.8. psql Changes

- Improve psql information display about database objects (Christopher)
- Allow psql to display group membership in `\du` and `\dg` (Markus Bertheau)
- Prevent psql `\dn` from showing temporary schemas (Bruce)
- Allow psql to handle tilde user expansion for file names (Zach Irmen)
- Allow psql to display fancy prompts, including color, via readline (Reece Hart, Chet Ramey)
- Make psql `\copy` match `COPY` command syntax fully (Tom)
- Show the location of syntax errors (Fabien Coelho, Tom)
- Add `CLUSTER` information to psql `\d` display (Bruce)
- Change psql `\copy stdin/stdout` to read from command input/output (Bruce)
- Add `pstdin`/`pstdout` to read from psql's `stdin`/`stdout` (Mark Feit)
- Add global psql configuration file, `psqlrc.sample` (Bruce)

  This allows a central file where global psql startup commands can be stored.
- Have psql `\d+` indicate if the table has an `OID` column (Neil)
- On Windows, use binary mode in psql when reading files so control−Z is not seen as end−of−file
- Have `\dn+` show permissions and description for schemas (Dennis Bjorklund)
- Improve tab completion support (Stefan Kaltenbrunn, Greg Sabino Mullane)
- Allow boolean settings to be set using upper or lower case (Michael Paesold)

---

#### E.6.4.9. pg_dump Changes

- Use dépendency information to improve the reliability of pg_dump (Tom)

  This should solve the longstanding problems with related objects sometimes being dumped in the

wrong order.
- Have pg_dump output objects in alphabetical order if possible (Tom)

  This should make it easier to identify changes between dump files.
- Allow pg_restore to ignore some SQL errors (Fabien Coelho)

  This makes pg_restore's behavior similar to the results of feeding a pg_dump output script to psql. In most cases, ignoring errors and plowing ahead is the most useful thing to do. Also added was a pg_restore option to give the old behavior of exiting on an error.
- pg_restore `-l` display now includes objects' schema names
- New begin/end markers in pg_dump text output (Bruce)
- Add start/stop times for pg_dump/pg_dumpall in verbose mode (Bruce)
- Allow most pg_dump options in pg_dumpall (Christopher)
- Have pg_dump use `ALTER OWNER` rather than `SET SESSION AUTHORIZATION` by default (Christopher)

### E.6.4.10. libpq Changes

- Make libpq's `SIGPIPE` handling thread–safe (Bruce)
- Add `PQmbdsplen()` which returns the display length of a character (Tatsuo)
- Add thread locking to SSL and Kerberos connections (Manfred Spraul)
- Allow `PQoidValue()`, `PQcmdTuples()`, and `PQoidStatus()` to work on `EXECUTE` commands (Neil)
- Add `PQserverVersion()` to provide more convenient access to the server version number (Greg Sabino Mullane)
- Add `PQprepare/PQsendPrepared()` functions to support preparing statements without necessarily specifying the data types of their parameters (Abhijit Menon–Sen)
- Many ECPG improvements, including `SET DESCRIPTOR` (Michael)

### E.6.4.11. Source Code Changes

- Allow the database server to run natively on Windows (Claudio, Magnus, Andrew)
- Shell script commands converted to C versions for Windows support (Andrew)
- Create an extension makefile framework (Fabien Coelho, Peter)

  This simplifies the task of building extensions outside the original source tree.
- Support relocatable installations (Bruce)

  Directory paths for installed files (such as the `/share` directory) are now computed relative to the actual location of the executables, so that an installation tree can be moved to another place without reconfiguring and rebuilding.
- Use `--with-docdir` to choose installation location of documentation; also allow `--infodir` (Peter)
- Add `--without-docdir` to prevent installation of documentation (Peter)
- Upgrade to DocBook V4.2 SGML (Peter)
- New `PostgreSQL` CVS tag (Marc)

  This was done to make it easier for organizations to manage their own copies of the PostgreSQL CVS repository. File version stamps from the master repository will not get munged by checking into or out of a copied repository.

- Clarify locking code (Manfred Koizar)
- Buffer manager cleanup (Neil)
- Decouple platform tests from CPU spinlock code (Bruce, Tom)
- Add inlined test−and−set code on PA−RISC for gcc (ViSolve, Tom)
- Improve i386 spinlock code (Manfred Spraul)
- Clean up spinlock assembly code to avoid warnings from newer gcc releases (Tom)
- Remove JDBC from source tree; now a separate project
- Remove the libpgtcl client interface; now a separate project
- More accurately estimate memory and file descriptor usage (Tom)
- Improvements to the Mac OS X startup scripts (Ray A.)
- New `fsync()` test program (Bruce)
- Major documentation improvements (Neil, Peter)
- Remove pg_encoding; not needed anymore
- Remove pg_id; not needed anymore
- Remove initlocation; not needed anymore
- Auto−detect thread flags (no more manual testing) (Bruce)
- Use Olson's public domain timezone library (Magnus)
- With threading enabled, use thread flags on Unixware for backend executables too (Bruce)

  Unixware can not mix threaded and non−threaded object files in the same executable, so everything must be compiled as threaded.
- psql now uses a flex−generated lexical analyzer to process command strings
- Reimplement the linked list data structure used throughout the backend (Neil)

  This improves performance by allowing list append and length operations to be more efficient.
- Allow dynamically loaded modules to create their own server configuration parameters (Thomas Hallgren)
- New Brazilian version of FAQ (Euler Taveira de Oliveira)
- Add French FAQ (Guillaume Lelarge)
- New pgevent for Windows logging
- Make libpq and ECPG build as proper shared libraries on OS X (Tom)

### E.6.4.12. Contrib Changes

- Overhaul of `contrib/dblink` (Joe)
- `contrib/dbmirror` improvements (Steven Singer)
- New `contrib/xml2` (John Gray, Torchbox)
- Updated `contrib/mysql`
- New version of `contrib/btree_gist` (Teodor)
- New `contrib/trgm`, trigram matching for PostgreSQL (Teodor)
- Many `contrib/tsearch2` improvements (Teodor)
- Add double metaphone to `contrib/fuzzystrmatch` (Andrew)
- Allow `contrib/pg_autovacuum` to run as a Windows service (Dave Page)
- Add functions to `contrib/dbsize` (Andreas Pflug)
- Removed `contrib/pg_logger`: obsoleted by integrated logging subprocess
- Removed `contrib/rserv`: obsoleted by various separate projects

# E.7. Version 7.4.10

**Date de sortie :** 2005−12−12

Cette version contient quelques corrections sur la 7.4.9.

## E.7.1. Migration vers la version 7.4.10

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une version 7.4.X. Néanmoins, si vous mettez à jour à partir d'une version antérieure à la 7.4.8, voir les notes de sortie de la 7.4.8.

## E.7.2. Modifications

- Correction d'un cas extrême dans la gestion des traces des transactions

  Il existait une petite possibilité pour laquelle une opération d'entrée/sortie pouvait être initiée pour la mauvaise page, amenant à un échec Assert ou à une corruption de données.
- Empêche un échec si le client envoie un message de protocole Bind quand la transaction en cours est déjà annulée
- Corrections sur `/contrib/ltree` (Teodor)
- Corrections de compilation sur AIX et HPUX (Tom)
- Correction d'une vieille erreur sur les jointures externes

  Ce bogue a quelque fois causé une fausse erreur << RIGHT JOIN is only supported with merge−joinable join conditions >>.
- Empêchement d'un arrêt brutal dans pg_autovacuum quand une table a été supprimée

# E.8. Version 7.4.9

**Date de sortie :** 2005−10−04

Cette version contient quelques corrections de la 7.4.8.

## E.8.1. Migration à partir de la version 7.4.9

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une version 7.4.X. Néanmoins, si vous mettez à jour à partir d'une version antérieure à la 7.4.8, voir les notes de sortie de la 7.4.8.

## E.8.2. Modifications

- Correction d'une erreur qui permettait au `VACUUM` de supprimer les chaînes `ctid` trop tôt, et ajout de vérification supplémentaire dans le code qui suit les liens `ctid`

  Ceci corrige un vieux problème qui pourrait causer des arrêt brutaux dans de rares circonstances.
- Correction de `CHAR()` pour ajouter le nom d'espaces spécifié lors de l'utilisation d'un ensemble de caractères multi−octets (Yoshiyuki Asaba)

Dans les versions précédentes, l'ajout pour `CHAR()` était incorrect car il ajoutait seulement le nombre d'octets spécifiés sans prendre en considération la façon dont les caractères étaient stockés.

- Correction du sens du test pour les transactions en lecture seule dans `COPY`

  Le code empêchait `COPY TO` alors qu'il aurait dû interdire `COPY FROM`.
- Correction du problème de planification avec les clauses ON des jointures externes qui référencent seulement la relation du côté interne
- Autres corrections sur les cas particuliers du `x FULL JOIN y ON true`
- Rend `array_in` et `array_recv` plus paranoïaque sur la validation de leur paramètre OID
- Correction des lignes manquantes dans les requêtes du type `UPDATE a=... WHERE a...` avec un index GiST sur la colonne `a`
- Amélioration de la stabilité de l'analyse des valeurs du type datetime
- Amélioration de la vérification pour les pages WAL partiellement écrites
- Amélioration de la solidité de la gestion des signaux lorsque SSL est activé
- Empêche l'ouverture de plus de `max_files_per_process` fichiers lors du démarrage de postmaster
- Plusieurs corrections de perte mémoire
- Quelques améliorations sur la portabilité
- Correction de PL/PgSQL pour gérer correctement `var := var` quand la variable est passée par référence
- Mise à jour de `contrib/tsearch2` pour utiliser le code Snowball actuel

# E.9. Version 7.4.8

**Date de sortie :** 2005−05−09

Cette version contient quelques corrections sur la 7.4.7, dont des correctifs sur des trous de sécurité.

## E.9.1. Migration vers la version 7.4.8

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une version 7.4.X. Néanmoins, c'est une façon possible de gérer deux problèmes de sécurité significatifs, qui ont été trouvé dans le contenu initial des catalogues systèmes des versions 7.4.X. Une séquence sauvegarde/initdb/restauration utilisant le initdb de la 7.4.8 corrigera automatiquement ces problèmes.

Le problème de sécurité le plus important est que les fonctions de conversion du codage des ensembles de caractères peuvent être appelées à partir de commandes SQL par des utilisateurs non pribilégiés alors que les fonctions n'ont pas été conçues pour un tel usage et ne sont pas sécurisées contre des choix rusés des arguments. Le correctif implique de modifier la liste déclarée des arguments de ces fonctions pour qu'elles ne soient plus appelées à partir des commandes SQL. (Ceci n'affecte pas leur utilisation normale par la machinerie de conversion du codage.)

Le problème le moins important est que le module `contrib/tsearch2` crée plusieurs fonctions déclarant par erreur renvoyées `internal` alors qu'elles n'acceptent pas les arguments `internal`. Ceci casse la sûreté des types pour toutes les fonctions utilisant des arguments `internal`.

Il est fortement recommandé que toutes les installations corrigent ces erreurs, soit par un initdb soit en suivant les procédures de réparation données ci−dessous. Les erreurs permettent au moins à des utilisateurs non privilégiés de la base de données d'arrêter brutalement leur processus serveur et pourraient permettre à des utilisateurs non privilégiés de gagner les privilèges d'un superutilisateur.

Si vous souhaitez ne pas lancer d'initdb, exécutez la procédure suivante à la place en tant que superutilisateur de la base de données :

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
      AND proargtypes[2] = 'cstring'::regtype;
-- La commande devrait rapporter qu'elle a mis à jour 90 lignes ;
-- si ce n'est pas le cas, annulez (rollback) et investiguez au lieu de valider !
COMMIT;
```

Ensuite, si vous avez installé `contrib/tsearch2`, faites

```
BEGIN;
UPDATE pg_proc SET proargtypes[0] = 'internal'::regtype
WHERE oid IN (
    'dex_init(text)'::regprocedure,
    'snb_en_init(text)'::regprocedure,
    'snb_ru_init(text)'::regprocedure,
    'spell_init(text)'::regprocedure,
    'syn_init(text)'::regprocedure
);
-- La commande devrait rapporter qu'elle a mis à jour 90 lignes ;
-- si ce n'est pas le cas, annulez (rollback) et investiguez au lieu de valider !
COMMIT;
```

Si cette commande échoue avec un message comme << function "dex_init(text)" does not exist >>, alors soit `tsearch2` n'est pas installé dans cette base de données, soit vous l'avez déjà mise à jour.

Les procédures ci–dessus doivent être exécutées pour *chaque* base de données d'une installation, ceci incluant `template1` et devant idéalement inclure aussi `template0`. Si vous ne corrigez pas les bases de données modèles, alors toute base de données créée par la suite contiendra les mêmes erreurs. `template1` peut être corrigé de la même façon que les autres bases de données alors que `template0` requiert des étapes supplémentaires. Tout d'abord, à partir de n'importe quel base de données

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Ensuite, connectez–vous à `template0` et effectuez les opérations de réparation. Faites

```
-- gelez de nouveau template0:
VACUUM FREEZE;
-- et protégez-contre toute nouvelle modification :
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

## E.9.2. Changes

- Modification de la signature de la fonction de codage pour empêcher toute mauvaise utilisation
- Modification de `contrib/tsearch2` pour éviter une utilisation non sécurisée des résultats `INTERNAL` de la fonction
- Réparation d'une ancienne condition qui permettait à une transaction d'être vue comme validée pour certains buts (par exemple SELECT FOR UPDATE) légèrement plus tôt que dans les autres buts

  Ceci est un bogue extrêmement sérieux car il pourrait amener à des incohérences apparents des données et visibles brièvement par les applications.
- Réparation d'une condition entre extension de relation et VACUUM

Théoriquement, ceci pourrait avoir causé des pertes d'une page de données tout juste insérées, bien que la probabilité d'un tel scénario semble extrêmement faible. Il n'existe pas de cas connus où cela a provoqué plus qu'un échec d'Assert.

- Correction des comparaisons des valeurs `TIME WITH TIME ZONE`

  Le code de comparaison était mauvais dans le cas où le commutateur de configuration `--enable-integer-datetimes` avait été utilisé. NOTE : si vous avez un index sur une colonne `TIME WITH TIME ZONE`, il sera nécessaire de le `REINDEX`er après avoir installé cette mise à jour parce que ce correctif corrige l'ordre de tri des valeurs de colonnes.
- Correction de `EXTRACT(EPOCH)` pour les valeurs de type `TIME WITH TIME ZONE`
- Correction d'un mauvais affichage des secondes fractionnelles négatives dans les valeurs `INTERVAL`

  Cette erreur est seulement survenue quand l'option de configuration `--enable-integer-datetimes` avait été utilisée.
- S'assure que les opérations effectuées pendant l'arrêt du serveur sont comptées par le collecteur de statistiques

  Ceci doit avoir résolu les rapports concernant pg_autovacuum, comme quoi il ne lançait pas assez fréquemment VACUUM sur les catalogues systèmes — il n'a pas été question de suppressions des catalogues causées par la suppression de tables temporaires lors de l'arrêt du serveur.
- Vérification supplémentaires de dépassement de tampon dans plpgsql (Neil)
- Correction de pg_dump pour qu'il sauvegarde correctement les noms des déclencheurs contenant `%` (Neil)
- Correction de `contrib/pgcrypto` pour les nouvelles constructions d'OpenSSL (Marko Kreen)
- Encore plus de correctifs 64 bits pour `contrib/intagg`
- Empêche une optimisation incorrecte des fonctions renvoyant `RECORD`
- Empêche `to_char(interval)` d'arrêter brutalement le serveur (dump core) pour les formats relatifs au mois
- Empêche un arrêt brutal sur `COALESCE(NULL,NULL)`
- Correction de `array_map` pour appeler correctement les fonctions PL
- Correction de la vérification des droits dans `ALTER DATABASE RENAME`
- Correction de `ALTER LANGUAGE RENAME`
- Make `RemoveFromWaitQueue` clean up after itself

  Ceci corrige une erreur de gestion du verrour qui pourrait seulement être visible si une transaction a été dégagée de l'attente d'un verrour (typiquement dans le cas d'une requête annulée) et que le détenteur du verrou le relâche dans une fenêtre très petite.
- Correction d'un problème avec un paramètre non typé apparaissant dans `INSERT ... SELECT`
- Correction d'un échec de `CLUSTER` après `ALTER TABLE SET WITHOUT OIDS`

# E.10. Version 7.4.7

**Date de sortie :** 2005−01−31

Cette version contient une variété de corrections de la version 7.4.6, incluant des correctifs de sécurité.

### E.10.1. Migration vers la version 7.4.7

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une 7.4.X.

### E.10.2. Modifications

- Interdit `LOAD` aux utilisateurs standards

  Sur les plateformes qui exécuteront automatiquement les fonctions d'initialisation d'une bibliothèque partagée (ceci inclut au moins Windows et les Unix basés sur ELF), `LOAD` est utilisable pour faire exécuter un code arbitraire par le serveur. Merci à NGS Software pour cette information.
- Vérifie que le créateur d'une fonction d'agrégat a le droit d'exécuter les fonctions de transition spécifiées

  Ce problème rendait possible le contournement de l'interdiction du droit EXECUTE sur une fonction.
- Correction de problèmes de sécurité et de problèmes sur les 64 bits dans contrib/intagg
- Ajout du marquage STRICT nécessaire à quelques fonctions dans contrib (Kris Jurka)
- Évite un dépassement de tampon lorsque la déclaration du curseur plpgsql dispose trop de paramètres (Neil)
- Correction d'erreurs de plannification pour les jointures complètes et externes à droite

  Le résultat de la jointure était faussement supposé trié de la même façon que l'entrée gauche. Ceci pouvait délivrer une sortie mal triée à l'utilisateur mais, dans le cas de jointures d'assemblage imbriquées, pouvait donner de mauvaises réponses.
- Correction de plperl pour les guillemets dans les champs
- Correction de l'affichage des intervalles négatifs pour les styles de date SQL et GERMAN
- Fait que age(timestamptz) calcule à partir du fuseau horaire et non pas de GMT

## E.11. Sortie 7.4.6

**Date de sortie :** 2004−10−22

Cette version contient plusieurs correctifs de la version 7.4.5.

### E.11.1. Migration vers la version 7.4.6

Une sauvegarde restauration n'est pas requise pour ceux utilisant une version 7.4.X.

### E.11.2. Modifications

- Réparation d'un échec possible pour mettre à jour les bits d'astuce sur disque

  Sous quelques rares circonstances, ceci pourrait mener à des échecs << could not access transaction status >>, qui se qualifient comme bogue pouvant entraîner des pertes de données.
- Assurance que la jointure externe hachée ne manque pas de lignes

  Les jointures gauches très importantes pourraient échouer à afficher les lignes du côté gauche sans correspondance en donnant seulement la distribution droite des données.

- Interdit l'exécution de pg_ctl en tant que root

  Ceci est fait pour évacuer tout risque de problèmes de sécurité.
- Évite l'utilisation de fichiers temporaires dans `/tmp` dans `make_oidjoins_check`

  Ceci a été rapportée comme un problème de sécurité bien qu'il est de peu d'importance car il n'y a aucune raison pour que les utilisateurs autres que les développeurs utilisent ce script de toute façon.
- Empêche l'arrêt forcé du moteur à partir de la ré−émission du résultat de la commande précédente

  Dans de rares cas, un client pourrait penser que sa dernière commande a réussi alors qu'elle a réellement été annulée par l'arrêt forcé de la base de données.
- Réparation d'un bogue dans `pg_stat_get_backend_idset`

  Cela pouvait amener un mauvais comportement dans quelques vues de statistiques du système.
- Correction d'une petite perte mémoire dans postmaster
- Correction du bogue << expected both swapped tables to have TOAST tables >>

  Ceci pouvait arriver dans les cas tels que CLUSTER après ALTER TABLE DROP COLUMN.
- Empêche `pg_ctl restart` d'ajouter `-D` plusieurs fois
- Correction du problème de valeurs NULL dans les index GiST
- `::` n'est plus interprété comme une variable dans une instruction de préparation ECPG

# E.12. Sortie 7.4.5

**Date de sortie :** 2004−08−18

Cette version contient la correction d'un bogue sérieux de la version 7.4.4.

## E.12.1. Migration vers la version 7.4.5

Une sauvegarde restauration n'est pas requise pour ceux utilisant une version 7.4.X.

## E.12.2. Modifications

- Réparation d'un arrêt brutal possible lors d'insertions concurrentes dans l'index B−tree

  Ce correctif s'occupe d'un cas ras dans les insertions concurrentes dans un index B−tree, pouvant résulter en un PANIC du serveur. Aucun dommage permanent ne devrait en résulter mais cela vaut néanmoins une pré−version. Ce bogue n'existe pas dans les versions antérieures 7.4.

# E.13. Sortie 7.4.4

**Date de sortie :** 2004−08−16

Cette version contient quelques correctifs sur la 7.4.3.

### E.13.1. Migration vers la version 7.4.4

Une sauvegarde restauration n'est pas requise pour ceux utilisant une version 7.4.X.

### E.13.2. Modifications

- Empêche une perte possible de transactions validées lors d'un arrêt brutal

  À cause d'un verrouillage insuffisant entre la validation des transactions et des points de vérification, il était possible pour les transactions validées juste avant le point de vérification perdu, complètement ou partiellement, le plus récent après un arrêt brutal de la base de données et un redémarrage. Ceci est un bogue sérieux qui existait depuis PostgreSQL 7.1.
- Vérification de la restriction HAVING avant d'évaluer une liste de résultat d'un plan d'agrégat
- Évite un arrêt brutal lorsque l'identifiant de l'utilisateur courant est supprimé
- Correction des tableaux croisés hachés pour les tableaux sans ligne (Joe)
- Force la mise à jour du cache après renommage d'une colonne dans une clé étrangère
- L'affichage propre des requêtes UNION fonctionne correctement
- Fait que psql gère les retours chariot `\r\n` proprement dans COPY IN
- pg_dump gèrait les ACL avec des options grant non correctement
- Correction du support des threads pour OS X et Solaris
- Mise à jour du pilote JDBC (construction 215) avec différents correctifs
- Corrections pour ECPG
- Corrections des traductions (contributeurs variés)

## E.14. Sortie 7.4.3

**Date de sortie :** 2004−06−14

Cette version contient quelques correctifs pour la 7.4.2.

### E.14.1. Migration vers la version 7.4.3

Une sauvegarde restauration n'est pas requise pour ceux utilisant une version 7.4.X.

### E.14.2. Modifications

- Correction de pertes mémoire temporaires lors de l'utilisation d'agrégats non hachés (Tom)
- Correctifs pour ECPG, incluant quelques−uns pour la compatibilité avec Informix (Michael)
- Corrections pour compiler avec l'activation des threads, en particulier pour Solaris (Bruce)
- Correction d'une erreur dans la fin de COPY IN lors de l'utilisation de l'ancien protocole réseau (ljb)
- Plusieurs corrections importantes dans pg_autovacuum, incluant des correctifs pour les grosses tables, les OID non signés, la stabilité, les tables temporaires et le mode de débogage (Matthew T. O'Connor)
- Correction d'un problème lors de la lecture de sauvegardes au format tar sur NetBSD et BSD/OS (Bruce)
- Quelques corrections sur JDBC
- Correction de ALTER SEQUENCE RESTART lorsque last_value vaut la valeur de réinitialisation (Tom)
- Réparation de l'échec après recalcul des sous−sélections imbriquées (Tom)

- Correction de problèmes avec les expressions non constantes dans LIMIT/OFFSET
- Support de FULL JOIN sans clause de jointure, tels que X FULL JOIN Y ON TRUE (Tom)
- Correction d'un autre bogue sur les tables sans colonnes (Tom)
- Amélioration de la gestion des identifieurs non qualifiés dans les clauses GROUP BY des sous−sélections (Tom)

  Les alias de liste de sélection à l'intérieur de sous−sélections prendront maintenant la précédence sur les noms des niveaux de requêtes externes.
- Ne génère pas de << NATURAL CROSS JOIN >> lors de la décompilation de règles (Tom)
- Ajout de vérification pour la taille de champ invalide dans le COPY binaire (Tom)

  Ceci corrige une faille de sécurité difficile à exploiter.
- Évite les conflits de verrouillage entre `ANALYZE` et `LISTEN`/`NOTIFY`
- Nombreuses mises à jour de traduction (plusieurs contributeurs)

# E.15. Sortie 7.4.2

**Date de sortie :** 2004−03−08

Cette version contient quelques correctifs pour la 7.4.1.

## E.15.1. Migration vers la version 7.4.2

Une sauvegarde restauration n'est pas requise pour ceux utilisant une version 7.4.X. Néanmoins, c'est la méthode conseillé car plus simple pour l'incorporation de la correction sur deux erreurs trouvées dans le contenu initial des catalogues systèmes de la 7.4. Une séquence sauvegarde/initdb/restauration utilisant l'initdb de la 7.4.2 corrigera automatiquement ces problèmes.

La plus sévère des deux erreurs est que le type de données `anyarray` a un mauvais label d'alignement ; ceci est un problème parce que le catalogue système `pg_statistic` utilise des colonnes `anyarray`. Ce mauvais label peut causer des mauvaises estimations du planificateur et même des arrêts brutaux lorsque des requêtes de planification impliquent des clauses `WHERE` sur des colonnes doublement alignées doubles (telles que `float8` et `timestamp`). Il est fortement recommandé que toutes les installations réparent cette erreur, soit par initdb soit en suivant la procédure de réparation manuelle donnée ci−dessous.

L'erreur moindre est que la vue système `pg_settings` devrait être marquée comme ayant un accès public en mise à jour pour permettre l'utilisation de `UPDATE pg_settings` comme substitut pour `SET`. Ceci peut être corrigé soit par initdb soit manuellement mais il n'est pas nécessaire de le corriger sauf si vous voulez utiliser `UPDATE pg_settings`.

Si vous ne souhaitez pas lancer un initdb, la procédure suivante corrigera `pg_statistic`. En tant que superutilisateur de la base de données, faites :

```
-- efface les anciennes données de pg_statistic :
DELETE FROM pg_statistic;
VACUUM pg_statistic;
-- ceci devrait mettre à jour 1 ligne :
UPDATE pg_type SET typalign = 'd' WHERE oid = 2277;
-- ceci devrait mettre à jour 6 lignes :
UPDATE pg_attribute SET attalign = 'd' WHERE atttypid = 2277;
--
```

```
-- À ce moment, vous DEVEZ lancer un nouveau moteur pour éviter un arrêt brutal
--
-- repopulate pg_statistic:
ANALYZE;
```

Ceci peut se faire sur une base de données en réel mais attention au fait que tous les serveurs de la base de données modifiée doivent être relancés avant qu'il ne soit sain de repeupler `pg_statistic`.

Pour corriger l'erreur `pg_settings`, faites simplement :

```
GRANT SELECT, UPDATE ON pg_settings TO PUBLIC;
```

Les procédures ci−dessus doivent être exécutées dans *chaque* base de données d'une installation, ceci incluant `template1`, et idéalement `template0`. Si vous ne corrigez pas les bases de données modèles, alors toute nouvelle base de données contiendra les mêmes erreurs. `template1` peut être corrigé de la même façon que toute autre base de données, mais corriger `template0` requiert quelques étapes supplémentaires. Tout d'abord, à partir de n'importe quelle session de base de données

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

nsuite, connectez−vous à `template0` et exécutez les procédures de réparation suivante. Enfin, faites

```
-- re-gèle template0:
VACUUM FREEZE;
-- et la protège contre toute modifications futures :
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

## E.15.2. Modifications

La version 7.4.2 incorpore toutes les corrections de la version 7.3.6, ainsi que les suivantes :

- Correction du bogue d'alignement de `pg_statistics` qui pouvait provoquer un arrêt brutal de l'optimiseur

  Voir ci−dessus pour les détails sur ce problème.
- Autorise les utilisateurs standards à mettre à jour `pg_settings`
- Correction de quelques bogues de l'optimiseur, la plupart amenant des erreurs << variable not found in subplan target lists >> errors (NdT : variable non trouvée dans les listes cibles du sous−plan
- Évite les échecs dû au manque de mémoire lors du la ncement de plusieurs gros parcours d'index
- Correction d'un problème multioctets pouvant amener des erreurs du type << out of memory >> (manque de mémoire) lors d'une opération `COPY IN`
- Correction de problèmes avec `SELECT INTO` / `CREATE TABLE AS` à partir de tables sans OID
- Correction de problèmes avec les tests de régression `alter_table` lors de tests en parallèle
- Correction de problèmes survenant à la limite du nombre de fichiers ouverts, notamment sur OS X (Tom)
- Correction partielle pour les problèmes de la locale turque

  initdb réussira maintenant avec la locale turque, mais il reste quelques inconvénients associés avec le problème `i/I`.
- Fait que pg_dump initialise le codage du client lors d'une restauration
- Plusieurs autres corrections de pg_dump
- Autorise ecpg à utiliser de nouveau des mots clés C comme noms de colonnes (Michael)

- Ajout de `WHENEVER NOT_FOUND` à ecpg pour `SELECT/INSERT/UPDATE/DELETE` (Michael)
- Correction d'un arrêt brutal d'ecpg pour les requêtes appelant des fonctions renvoyant des ensembles (Michael)
- Plusieurs autres corrections d'ecpg (Michael)
- Corrections pour le compilateur Borland
- Amélioration de la construction avec les threads (Bruce)
- Plusieurs autres corrections pour la construction
- Plusieurs corrections sur JDBC

# E.16. Sortie 7.4.1

**Date de sortie :** 2003−12−22

Cette version contient plusieurs correctifs de la 7.4.

## E.16.1. Migration vers la version 7.4.1

Une sauvegarde/restauration n'est *pas* requise pour ceux utilisant la version 7.4.

Si vous voulez installer les correctifs dans le schéma d'information, vous avez besoin de le recharger dans la base de données. Ceci est accompli soit en initialisant un nouveau groupe de bases avec `initdb` soit en lançant la séquence suivante de commandes SQL dans chaque base de données (incluant idéalement `template1`) en tant que superutilisateur avec psql, après installation de la nouvelle version :

```
DROP SCHEMA information_schema CASCADE;
\i /usr/local/pgsql/share/information_schema.sql
```

Substituez votre chemin d'installation dans la deuxième commande.

## E.16.2. Modifications

- Correction d'un bogue dans l'analyse de `CREATE SCHEMA` avec ECPG (Michael)
- Correction d'une erreur de compilation quand `--enable-thread-safety` et `--with-perl` sont utilisés ensemble (Peter)
- Correction des sous−requêtes utilisant des jointures de découpage (Tom)

  Certaines sous−requêtes qui utilisaient des jointures de découpage pouvaient s'arrêter brutalement à cause de structures mal partagées.
- Correction d'un problème de compactage de la carte des espaces libres (Tom)

  Ceci corrige un bogue où le compactage de la carte des espaces libres pouvait entraîner un arrêt du serveur de la base de données.
- Correction pour la construction de libpq à partir du compilateur Borland (Bruce)
- Correction de `netmask()` et de `hostmask()` pour renvoyer le masklen de longueur maximale (Tom)

  Correction de ces fonctions pour renvoyer des valeurs cohérentes avec les versions pre−7.4..
- Plusieurs corrections de `contrib/pg_autovacuum`

Les correctifs incluent une mauvaise initialisation des variables, un vacuum manquant après un `TRUNCATE` et un dépassement du calcul de durée pour les vacuums longs.
- Autorise la compilation de `contrib/cube` sous Cygwin (Jason Tishler)
- Correction de l'utilisation par Solaris du fichier mot de passe lorsqu'aucun mot de passe n'est défini (Tom)

  Correction de l'arrêt brutal sur Solaris causé par l'utilisation de tout type d'authentification par mot de passe lorsqu'aucun mot de passe n'est défini.
- Correction de JDBC pour les problèmes de threads, autres corrections
- Correction sur les recherches dans les index `bytea` (Joe)
- Correction du schéma d'information pour les types de données bit (Peter)
- Force zero_damaged_pages à être actif lors de la récupération à partir des WAL
- Empêche quelques cas obscures de << variable not in subplan target lists >>
- Rend `PQescapeBytea` et `byteaout` cohérent l'un avec l'autre (Joe)
- Échappe la sortie `bytea` pour les octets > 0x7e (Joe)

  Si plusieurs codages clients sont utilisés pour la sortie et l'entrée `bytea`, il est possible de corrompre les valeurs `bytea` en modifiant les codages. Cette correction échappe tous les octets qui pourraient être affectés.
- Ajout des appels manquants à `SPI_finish()` à la fonction `get_tuple_of_interest()` de dblink (Joe)
- Nouvelle FAQ tchèque
- Correction de la vue du schéma d'informations `constraint_column_usage` pour les clés étrangères (Peter)
- Corrections sur ECPG (Michael)
- Correction d'un bogue avec plusieurs sous−requêtes `IN` et des jointures dans les sous−requêtes (Tom)
- Autorise le fonctionnement de `COUNT('x')` (Tom)
- Installation des fichiers d'inclusion d'ECPG pour la compatibilité Informix dans un répertoire séparé (Peter)

  Certains noms des fichiers d'inclusion d'ECPG pour la compatibilité Informix entrent en conflit avec les fichiers d'inclusion du système d'exploitation. En les installant dans leur propre répertoire, les conflits de noms ont été réduits.
- Correction d'une perte mémoire sur SSL (Neil)

  Cette version corrige un bogue de la 7.4 où SSL ne libérait pas toute la mémoire qu'il avait alloué.
- Empêche l'utilisation du nom de service par `pg_service.conf` comme nom de la base par défaut (Bruce)
- Correction de l'authentification locale par ident sur FreeBSD (Tom)

# E.17. Sortie 7.4

**Date de sortie :** 2003−11−17

## E.17.1. Aperçu

Modifications majeures dans cette version :

Les sous−requêtes `IN`/`NOT IN` sont maintenant bien plus efficaces

Dans les versions précédentes, les sous−requêtes IN/NOT IN ont été jointes par la requête supérieure en parcourant séquentiellement la sous−requête pour trouver une correspondance. Le code de la 7.4 utilise les mêmes techniques sophistiquées utilisées par les jointures ordinaires et est donc bien plus rapide. Un IN sera maintenant aussi rapide ou plus rapide que la sous−requête EXISTS équivalente ; ceci inverse l'information conventionnelle qui s'appliquait aux versions précédentes.

Amélioration du traitement du GROUP BY en utilisant les << hash buckets >>

Dans les précédentes versions, les lignes à grouper étaient d'abord triées. Le code de la 7.4 traite le GROUP BY sans tri en accumulant les résultats dans une table de découpage avec une entrée par groupe. Néanmoins, il utilisera toujours la technique du tri si la table de découpage est estimée trop importante pour être contenue dans sort_mem.

Nouvelle fonctionnalité de jointure par découpage sur des clés multiples

Dans les précédentes versions, les jointures par découpage pouvaient uniquement survenir sur des clés uniques. Cette version ajoute les jointures par découpage sur plusieurs colonnes.

Les requêtes utilisant la syntaxe JOIN explicite sont maintenant mieux optimisées

Les précédentes versions évaluaient les requêtes en utilisant uniquement la syntaxe JOIN explicite dans l'ordre fournie par la syntaxe. La version 7.4 autorise l'optimisation complète de ces requêtes, ce qui signifie que l'optimiseur considère tous les ordres possibles de jointure et choisit le plus efficace. Néanmoins, les jointures externes doivent toujours suivre l'ordre déclaré.

Code des expressions rationnelles plus rapide et plus puissant

Le module entier des expressions rationnelles a été remplacé avec une nouvelle version de Henry Spencer, écrit originellement pour Tcl. Le code améliore énormément les performances et supporte plusieurs types d'expressions rationnelles.

Fonctions en ligne pour les fonctions SQL simples

Les fonctions SQL simples peuvent maintenant être en ligne en incluant le SQL dans la requête principale. Ceci améliore les performances en éliminant la surcharge par appel. Ceci signifie que les fonctions SQL simples se comportent maintenant comme des macros.

Support complet des connexions IPv6 et des données de type adresse IPv6

Les versions précédentes autorisaient seulement les connexions IPv4 et les types de données IP supportaient seulement les adresses IPv4. Cette version ajoute le support complet de IPv6 dans ces deux domaines.

Amélioration majeure dans les performances et la fiabilité de SSL

Plusieurs personnes très au courant de l'API SSL ont revu notre code SSL pour améliorer la négotiation de clé SSL et la récupération en cas d'erreur.

Le << free space map >> réutilise efficacement les pages d'index, et quelques autres améliorations des espaces libres

Dans les précédentes versions, les pages d'index B−tree qui étaient laissées vides à cause de lignes supprimées pouvaient seulement être ré−utilisées par les lignes de valeurs d'index similaires aux lignes originellement indexées sur cette page. Avec la version 7.4, VACUUM enregistre les pages d'index vide et permet leur ré−utilisation pour toute ligne d'index future.

Schéma d'information au standard SQL

Le schéma d'information fournit une façon standardisée et stable d'accéder aux informations sur les objets du schéma définis dans une base de données.

Les curseurs se conforment mieux au standard SQL

Les commandes FETCH et MOVE ont été revues pour mieux se conformer au standard SQL.

Les curseurs peuvent exister en dehors des transactions

Ces curseurs sont aussi appelés des curseurs détenables.

Nouveau protocole client/serveur

Le nouveau protocole ajoute des codes d'erreur, plus d'informations sur le statut, un démarrage plus rapide, un meilleur support des transmissions de données binaires, des valeurs de paramètres séparées à partir des commandes SQL, des instructions préparées disponibles au niveau du protocole et une récupération plus propre pour les échecs lors d'un COPY. L'ancien protocole est toujours supporté par

le serveur et les clients.

Les applications libpq et ECPG sont maintenant totalement compatibles avec les threads
: Alors que les précédentes versions de libpq supportaient déjà les threads, cette version améliore la compatibilité en corrigeant certains codes, non compatibles, utilisés lors du lancement de la connexion à la base de données. L'option `--enable-thread-safety` de `configure` doit être utilisé pour activer cette fonctionnalité.

Nouvelle version de l'indexage complet de texte
: Une nouvelle suite d'indexage complet de texte est disponible dans `contrib/tsearch2`.

Nouvel outil autovacuum
: Le nouvel outil autovacuum dans `contrib/autovacuum` surveille les tables de statistiques de la base de données sur les activités `INSERT`/`UPDATE`/`DELETE` et lance automatiquement un `VACUUM` quand cela est nécessaire.

La gestion des tableaux a été améliorée et déplacée dans le cœur du serveur
: Beaucoup de limitations des tableaux ont été supprimées et les tableaux se comportent beaucoup plus comme des types de données complètement supportés.

## E.17.2. Migration vers la version 7.4

Une sauvegarde/restauration grâce à pg_dump est requise pour ceux souhaitant migrer leurs données d'une version précédente.

Observez les incompatibilités suivantes :

- Le paramétrage autocommit, côté serveur, a été supprimé et réimplémenté dans les applications clients et dans les langages. L'autocommit posait beaucoup trop de problèmes avec les langages et applications qui voulaient contrôler leur propre comportement de validation automatique, donc celui−ci a été supprimé du serveur et ajouté aux API clientes individuelles si appropriées.
- Le contenu des messages d'erreur a un peu changé dans cette version. Un effort significatif a été fourni pour rendre ces messages plus consistants et plus orientés vers l'utilisateur. Si vos applications essaient de détecter des conditions d'erreur en analysant le message d'erreur, vous êtes fortement encouragé à utiliser le nouveau code d'erreur à la place.
- Les jointures internes utilisant la syntaxe `JOIN` explicite pourraient se comporter différemment parce qu'elles sont mieux optimisées.
- Un certain nombre de paramètres de configuration serveur ont été renommés pour plus de clarté, principalement ceux liés aux traces.
- `FETCH 0` ou `MOVE 0` ne font maintenant plus rien. Dans les précédentes versions, `FETCH 0` récupérerait toutes les lignes restantes et `MOVE 0` se déplacerait à la fin du curseur.
- `FETCH` and `MOVE` renvoient maintenant le nombre réel de lignes récupérées/déplacées ou 0 si au début ou à la fin du curseur. Les versions précédentes renverraient le nombre de lignes passé à la commande, pas le nombre de lignes réellement récupérées ou déplacées.
- `COPY` traite maintenant les fichiers processus qui utilisent les séquences retour chariot ou retour chariot/fin de ligne. Les retours chariots ou fin de ligne littérals ne sont plus acceptés dans les données ; utilisez à la place `\r` et `\n`.
- Les espaces suivants sont maintenant supprimés lors de la conversion du type `char(`*n*`)` aux types `varchar(`*n*`)` et `text`. C'est ce que la plupart des gens s'attendaient à voir arriver.
- Le type de données `float(`*p*`)` mesure maintenant *p* en chiffres binaires, et non pas en chiffres décimals. Le nouveau comportement suit le standard SQL.
- Les valeurs de dates ambigues doivent correspondre à l'ordre spécifié par le paramétrage `datestyle`. Dans les précédentes versions, la spécification de la date `10/20/03` était interprétée comme une date d'octobre même si `datestyle` spécifiait la date en premier. La 7.4 envoie une

erreur si une spécification de date est invalide pour le paramétrage courant de `datestyle`.
- Les fonctions `oidrand`, `oidsrand` et `userfntest` ont été supprimées. Ces fonctions n'étaient plus utiles.
- Les littéraux de chaîne spécifiant des valeurs date/heure, tels que `'now'` ou `'today'`, ne fonctionneront plus comme attendus dans les expressions par défaut de colonnes ; ils ne ajouteront maintenant par défaut l'heure de création de la table par défaut, pas le moment de l'ajout. À la place, les fonctions `now()`, `current_timestamp` ou `current_date` devraient être utilisées.

  Dans les précédentes versions, il y avait un code spécial pour que les chaînes telles que `'now'` soient interprétées au moment de l'`INSERT` et non pas au moment de la création de la table mais ce contournement ne couvrait pas tous les cas. La version 7.4 requiert maintenant que les valeurs par défaut soient définies proprement en utilisant des fonctions comme `now()` ou `current_timestamp`. Elles fonctionneront dans toutes les situations.
- Le signe dollar (`$`) n'est plus permis dans les noms d'opérateurs. À la place, cela peut être un caractère dans les identifieurs, à condition qu'il ne soit pas le premier. Ceci a été fait pour améliorer la compatibilité avec les autres systèmes de bases de données et pour éviter les problèmes de syntaxe lorsque des emplacements de paramètres (`$`*n*) sont écrits adjacents à ceux des opérateurs.

## E.17.3. Modifications

Vous trouverez ci−dessous un détail des modifications entre la version 7.4 et la précédente version majeure.

### E.17.3.1. Modifications des opérations du serveur

- Autorisation des connexions IPv6 au serveur (Nigel Kukard, Johan Jordaan, Bruce, Tom, Kurt Roeckx, Andrew Dunstan)
- Correction de SSL pour gérer les erreurs proprement (Nathan Mueller)

  Dans les précédentes versions, certains rapports d'erreur de l'API SSL n'étaient pas gérés correctement. Cette version corrige ces problèmes.
- Amélioration des performances et de la sécurité du protocole SSL (Sean Chittenden)

  La renégotiation de clé SSL arrivait trop fréquemment, impliquant des performances SSL pauvres. De plus, la gestion de la clé initiale a été améliorée.
- Affiche d'informations sur les verrous lorsqu'un blocage (deadlock) est détecté (Tom)

  Ceci permet un débogage plus facile de ce type de situation.
- Mise à jour des heures de modification des socket s `/tmp` pour éviter leur suppression (Tom)

  Ceci devrait aider à empêcher le nettoyage du répertoire `/tmp` par des scripts d'administration.
- Activation de PAM pour Mac OS X (Aaron Hillegass)
- Index B−tree complètement compatibles avec les WAL (Tom)

  Dans les précédentes versions, dans certains cas rares, un arrêt brutal du serveur pouvait causer la corruption d'index B−tree. Cette version supprime ces quelques cas rares.
- Permet le compactage des index B−tree et la réutilisation des pages vides (Tom)
- Corrige la recherche incohérente des index lors de la division de la première page racine (Tom)

  Dans les précédentes versions, lorsqu'un index sur une page se divisait en deux pages, il y avait une brève période où une autre session de base de données pouvait ne pas voir une entrée d'index. Cette

version corrige ce cas rare d'échecs.
- Améliore la logique d'allocation de l'espace libre (Tom)
- Préserve les informations sur l'espace libre entre les différents redémarrages du serveur (Tom)

  Dans les précédentes versions, la carte de l'espace libre n'était pas sauvegardée quand le postmaster était arrêté, donc les serveurs nouvellement démarrés n'avaient pas d'informations sur l'espace libre. Cette version sauvegarde la carte de l'espace libre et la recharge quand le serveur est relancé.
- Ajoute le temps de lancement dans `pg_stat_activity` (Neil)
- Nouveau code pour détecter les pages disque corrompues ; efface avec `zero_damaged_pages` (Tom)
- Nouveau protocole client/serveur : plus rapide, plus de limite sur la longueur du nom de l'utilisateur, permet une sortie propre de la commande `COPY` (Tom)
- Ajoute un statut de transaction, un ID de table, un ID de colonne au client/serveur protocol (Tom)
- Ajoute des entrées/sorties binaires au protocole client/serveur (Tom)
- Supprime le paramétrage serveur autocommit ; déplacez dans les applications clientes (Tom)
- Changement des messages d'erreurs, codes d'erreurs et trois niveaux de détails d'erreurs (Tom, Joe, Peter)

---

### E.17.3.2. Améliorations des performances

- Ajout du découpage pour les agrégats `GROUP BY` (Tom)
- Jointures à boucles imbriquées plus intelligentes sur les index multicolonnes (Tom)
- Autorisation des jointures découpées à clés multiples (Tom)
- Amélioration du << constant folding >> (Tom)
- Ajout de la capacité de mettre en ligne les fonctions SQL simples (Tom)
- Réduction de l'utilisation de la mémoire pour les requêtes utilisant des fonctions complexes (Tom)

  Dans les précédentes versions, les fonctions renvoyant la mémoire allouée ne la libéraient pas tant que la requête n'était pas terminée. Cette version permet la libréation de la mémoire allouée par les fonctions lorsque l'appel de celle–ci se termine, réduisant la mémoire totale utilisée par les fonctions.
- Amélioration des performances de l'optimiseur GEQO (Tom)

  Cette version corrige quelques inefficacités dans la façon dont l'optimiseur GEQO gère les chemins de requête potentiels.
- Autorise la gestion des `IN`/`NOT IN` via des tables de découpages (Tom)
- Amélioration des performances de `NOT IN (`*`subquery`*`)` (Tom)
- Permet que la plupart des sous–requêtes `IN` soient traitées comme des jointures (Tom)
- Les opérations de correspondance de modèles peuvent utiliser des index quelque soit la locale (Peter)

  Il n'existe aucune méthode pourque des locales non ASCII utilisent les index standards pour des comparaisons `LIKE`. Cette version ajoute un moyen de créer un index spécial pour `LIKE`.
- Permet au postmaster de précharger les bibliothèques en utilisant `preload_libraries` (Joe)

  Pour les bibliothèques dynamiques requérant un long moment pour se charger, cette option est disponible pour que la bibliothèque soit pré–chargée par postmaster et héritée par toutes les sessions de la base de données.
- Améliore le calcul des coûts par l'optimiseur, particulièrement pour les sous–requêtes (Tom)
- Évite le tri lors la sous–requête `ORDER BY` correspond à la requête de haut niveau (Tom)
- Déduit que `WHERE a.x = b.y AND b.y = 42` signifie aussi que `a.x = 42` (Tom)
- Autorise les jointures de découpage (hash/merge) sur des jointures complexes (Tom)
- Autorise les jointures de découpage sur un plus grand nombre de types de données (Tom)

- Autorise l'optimisation des jointures internes explicites, désactivez avec `join_collapse_limit` (Tom)
- Ajoute le paramètre `from_collapse_limit` pour contrôler la conversion des sous−requêtes en jointures (Tom)
- Utilise le code plus rapide et plus puissant des expressions rationnelles provenant de Tcl (Henry Spencer, Tom)
- Utilise les ensembles de relation dans l'optimiseur (Tom)
- Améliore le temps de lancement de la connexion (Tom)

  Le nouveau protocole client/serveur requiert moins de paquets réseau pour commencer une session de la base de données.
- Améliore les performances des déclencheurs/contraintes (Stephan)
- Améliore la rapidité de `col IN (const, const, const, ...)` (Tom)
- Corrige les index de découpage, qui pourraient être cassés dans de rares cas (Tom)
- Améliore la concurrence et la rapidité des index de découpage (Tom)

  Les versions précédentes souffraient d'une pauvre performance pour les index de découpage, particulièrement dans les situations avec de nombreux clients. Cette version corrige ceci. Le groupe de développement est intéressé par des rapports de comparaison entre les performances d'un index B−tree et d'un index de découpage.
- Alignement des tampons partagés sur des limites sur 32 bits pour des amélioration de la rapidité de copie (Manfred Spraul)

  Certains CPU réalisent des copies de données plus rapidement si les adresses sont alignées sur 32 bits.
- Type de données `numeric` réimplémenté pour de meilleurs performances (Tom)

  `numeric` était auparavant stocké en base 100. Le nouveau code utilise la base 10000 pour des performances significativement meilleures.

---

### E.17.3.3. Modification de la configuration du serveur

- Renommage du paramètre serveur `server_min_messages` et `log_min_messages` (Bruce)

  Ceci a été fait pour que la plupart des paramètres contrôlant les traces serveur commencent avec `log_`.
- Renommage de `show_*_stats` en `log_*_stats` (Bruce)
- Renommage de `show_source_port` en `log_source_port` (Bruce)
- Renommage de `hostname_lookup` en `log_hostname` (Bruce)
- Ajout de `checkpoint_warning` pour avertir en cas de points de vérification trop fréquent (Bruce)

  Dans les versions précédentes, il était difficile de déterminer si un point de vérification arrivait trop fréquemment. Cette fonctionnalité ajoute un avertissement aux traces du serveur lorsqu'un nombre excessif de points de vérification survient.
- Nouveaux paramètres du serveur en lecture seule con cernant la localisation (Tom)
- Modifie les messages des traces du serveur en `DEBUG` plutôt que `LOG` (Bruce)
- Empêche les variables de traces du serveur d'être désactivées par des utilisateurs standards (Bruce)

  Ceci est une fonctionnalité de sécurité pour que les utilisateurs standards ne puissent pas désactivés les traces activées par l'administrateur.
- `log_min_messages`/`client_min_messages` contrôle maintenant la sortie `debug_*` (Bruce)

Ceci centralise les informations de débogages du client de façon à ce que toutes les sorties de débogages puissent être envoyées soit au client soit aux traces du serveur.

- Ajoute le support du serveur Rendezvous de Mac OS X (Chris Campbell)

  Ceci permet aux hôtes Mac OS X de chercher les serveurs PostgreSQL disponibles sur le réseau.
- Ajoute la possibilité d'afficher seulement les instructions lentes en utilisant `log_min_duration_statement` (Christopher)

  Ceci est une fonctionnalité de débogage souvent demandée qui permet aux administrateurs de voir seulement les requêtes lentes dans les traces du serveur.
- Autorise `pg_hba.conf` à accepter des masques réseau au format CIDR (Andrew Dunstan)

  Ceci permet aux administrateurs d'assembler l'adresse IP de l'hôte et les champs de masques dans un seul champ CIDR dans `pg_hba.conf`.
- Nouveau paramètre `is_superuser` en lecture seule (Tom)
- Nouveau paramètre `log_error_verbosity` pour contrôler le détail des erreurs (Tom)

  Ceci fonctionne avec la nouvelle version des rapports d'erreurs pour fournir des informations supplémentaires comme des astuces, de noms de fichiers et des numéros de lignes.
- `postgres --describe-config` affiche maintenant les variables de configuration du serveur (Aizaz Ahmed, Peter)

  Cette option est utile pour les outils d'administration qui ont besoin de connaître les noms des variables de coniguration et leur minimum, maximum, valeur par défaut et descriptionss.
- Ajoute de nouvelles colonnes dans `pg_settings` : `context`, `type`, `source`, `min_val`, `max_val` (Joe)
- Par défaut, `shared_buffers` vaut 1000 et `max_connections` 100 si possible (Tom)

  Les anciennes versions avaient pour valeur par défaut 64 tampons partagés pour que PostgreSQL puisse démarrer même sur de très vieux systèmes. Cette version teste la mémoire partagée autorisée par la plateforme et sélectionne des valeurs par défaut plus raisonnable si possible. Bien sûr, les utilisateurs sont toujours encouragés à évaluer leur charge de ressources en accord avec `shared_buffers`.
- Nouveau type d'enregistrement dans `pg_hba.conf` `hostnossl` pour empêcher les connexions SSL (Jon Jensen)

  Dans les précédentes versions, il n'existait pas de moyens d'empêcher les connexions SSL si le client et le serveur supportaient SSL. Cette option le permet.
- Nouveau déplacement du paramètre `geqo_random_seed` (Tom)
- Ajout du paramètre serveur `regex_flavor` pour contrôler le traitement des expressions rationnelles (Tom)
- Fait que `pg_ctl` gère mieux les ports non standards (Greg)

---

### E.17.3.4. Modifications sur les requêtes

- Nouveau schéma d'informations au standard SQL (Peter)
- Ajoute des transactions en lecture seule (Peter)
- Affichage du nom de la clé et de la valeur dans les messages de violation de clé étrangère (Dmitry Tkach)
- Permet aux utilisateurs de voir leur propre requêtes dans `pg_stat_activity` (Kevin Brown)

Dans les versions précédentes, seul le superutilisateur pouvait voir les chaînes de requêtes en utilisant `pg_stat_activity`. Maintenant, les utilisateurs ordinaires peuvent voir leurs propres requêtes.

- Corrige les agrégats dans les sous−requêtes pour correspondre au standard SQL (Tom)

  Le standard SQL indique qu'une fonction d'agrégat apparaissant dans une sous−requête appartient à la requête externe si son argument contient seulement des variables de la requête externe. Les versions précédentes de PostgreSQL ne gèrent pas ce point correctement.
- Ajout d'une option pour empêcher l'ajout automatique de tables référencées dans la requête (Nigel J. Andrews)

  Par défaut, les tables mentionnées dans la requête sont automatiquement ajoutées dans la clause `FROM` s'ils n'y sont pas déjà. Ceci est compatible avec le comportement historique de POSTGRES mais est contraire au standard SQL. Cette option permet de sélectionner un comportement compatible avec le standard.
- Autorise `UPDATE ... SET col = DEFAULT` (Rod)

  Ceci permet à `UPDATE` d'initialiser une colonne à sa valeur déclarée par défaut.
- Autorise l'utilisation d'expressions dans `LIMIT`/`OFFSET` (Tom)

  Dans les versions précédentes, `LIMIT`/`OFFSET` pouvaient seulement être des constantes, pas des expressions.
- Implémentation de `CREATE TABLE AS EXECUTE` (Neil, Peter)

### E.17.3.5. Object Manipulation Changes

- Rend la grammaire de `CREATE SEQUENCE` plus conforme à SQL:2003 (Neil)
- Ajout des déclencheurs au niveau instruction (Neil)

  Bien que ceci permet à un déclencheur d'être lancé à la fin d'une instruction, il ne permet pas au déclencheur d'accéder à toutes les lignes modifiées par l'instruction. Cette possibilité est planifiée pour une future version.
- Ajout de contraintes de vérifications pour les domaines (Rod)

  Ceci améliore grandement l'utilité des domaines en les permettant d'utiliser les contraintes de vérification.
- Ajout d'`ALTER DOMAIN` (Rod)

  Ceci permet la manipulation de domaines existants.
- Correction de plusieurs bogues pour les tables comprenant zéro colonne (Tom)

  PostgreSQL supporte les tables sans colonne. Ceci corrige plusieurs bogues survenant lors de l'utilisation de telles tables.
- Fait que `ALTER TABLE ... ADD PRIMARY KEY` ajoute une contrainte non NULL (Rod)

  Dans les précédentes versions, `ALTER TABLE ... ADD PRIMARY` aurait ajouter un index unique mais pas de contrainte non NULL. Ceci est corrigé dans cette version.
- Ajout de `ALTER TABLE ... WITHOUT OIDS` (Rod)

  Ceci permet de contrôler le fait qu'il y ait ou non une colonne OID pour les nouvelles lignes. Ceci est surtout utile pour sauvegarder de l'espace disque.

- Ajout de `ALTER SEQUENCE` pour modifier les valeurs minimum, maximum, l'incrément, le cache et le cycle (Rod)
- Ajout de `ALTER TABLE ... CLUSTER ON` (Alvaro Herrera)

  Cette commande est utilisée par `pg_dump` pour enregistrer la colonne de groupement pour chaque table précédemment groupée. Cette information est utilisée pour regrouper toutes les tables précédemment groupées.
- Amélioration de la conversion automatique de type pour les domaines (Rod, Tom)
- Permet les signes dollar dans les identifieurs sauf en premier caractère (Tom)
- Interdit les signes dollar dans les noms d'opérateur, pour que `x=$1` fonctionne (Tom)
- Autorise la copie du schéma de la table en utilisant `LIKE` *`soustable`*, aussi une fonctionnalité SQL:2003 `INCLUDING DEFAULTS` (Rod)
- Ajout de la clause `WITH GRANT OPTION` pour `GRANT` (Peter)

  Ceci permet à `GRANT` de donner aux autres utilisateurs la capacité de données des privilèges sur un objet.

---

### E.17.3.6. Modifications des commandes utilitaires

- Ajout de la clause `ON COMMIT` à `CREATE TABLE` pour les tables temporaires (Gavin)

  Ceci permet à une table d'être supprimée ou d'avoir toutes ces lignes supprimées lors de la validation d'une transaction.
- Permet aux curseurs d'agir en dehors des transactions en utilisant `WITH HOLD` (Neil)

  Dans les précédentes versions, les curseurs étaient supprimés à la fin de la transaction qui les avait créé. Les curseurs peuvent maintenant être créés avec l'option `WITH HOLD` qui leur permet de continuer à être utilisé après que la transaction qui les a créé soit validée.
- `FETCH 0` et `MOVE 0` ne font maintenant plus rien (Bruce)

  Dans les précédentes versions, `FETCH 0` récupérait toutes les lignes restantes et `MOVE 0` se déplaçait à la fin du curseur.
- Fait que `FETCH` et `MOVE` renvoient le nombre de lignes récupérées/déplacées ou zéro si au début ou à la fin du curseur, suivant le standard SQL (Bruce)

  Dans les versions précédentes, le nombre de lignes renvoyé par `FETCH` et `MOVE` ne reflétaient pas exactement le nombre de lignes traitées.
- Gestion propre de `SCROLL` avec les curseurs ou renvoi d'une erreur (Neil)

  Permettre un accès aléatoire (à la fois en avant et en arrière) aux différents types de requêtes ne peut pas se faire sans quelques travaux supplémentaires. Si `SCROLL` est spécifié lors de la création du curseur, ce travail supplémentaire a été réalisé. De plus, si le curseur a été créé avec `NO SCROLL`, aucun accès aléatoire ne sera autorisé.
- Implémente les options compatibles SQL `FIRST`, `LAST`, `ABSOLUTE` *`n`*, `RELATIVE` *`n`* pour `FETCH` et `MOVE` (Tom)
- Autorise `EXPLAIN` sur `DECLARE CURSOR` (Tom)
- Permet à `CLUSTER` d'utiliser les index marqués comme pré−groupés par défaut (Alvaro Herrera)
- Permet à `CLUSTER` de grouper toutes les tables (Alvaro Herrera)

  Ceci permet à toutes les tables précédemment groupées dans une base de données d'être regroupées avec une seule commande.

- Empêche `CLUSTER` sur les index partiels (Tom)
- Autorise les retours à la ligne DOS et Mac dans les fichiers `COPY` (Bruce)
- Interdit les retours chariot comme valeur de données, les antislash−retour−chariot et `\r` sont toujours autorisés (Bruce)
- Modifications sur `COPY` (binaire, `\.`) (Tom)
- Récupération propre après un échec de `COPY` (Tom)
- Empêche les possibles pertes de mémoire de `COPY` (Tom)
- Fait que `TRUNCATE` soit sain lors de transactions (Rod)

  `TRUNCATE` peut maintenant être utilisé à l'intérieur d'une transaction. Si la transaction est annulée, les modifications faites par `TRUNCATE` sont automatiquement annulées.
- Permet la préparation et le lien de commandes utilitaires comme `FETCH` et `EXPLAIN` (Tom)
- Ajout de `EXPLAIN EXECUTE` (Neil)
- Amélioration des performances de `VACUUM` sur les index en réduisant le trafic des WAL (Tom)
- Les index fonctionnels ont été généralisés en des index sur des expressions (Tom)

  Dans les versions précédentes, les index fonctionnels supportaient seulement une simple fonction appliquée à un ou plusieurs noms de commandes. Cette version accepte tout type d'expression scalaire.
- Fait que `SHOW TRANSACTION ISOLATION` corresponde à l'entrée de `SET TRANSACTION ISOLATION` (Tom)
- Fait que `COMMENT ON DATABASE` sur les bases de données non locales génère un message d'avertissement plutôt qu'une erreur (Rod)

  Les commentaires de bases de données sont stockés dans les tables locales de la base de données. Donc, les commentaires sur une base de données doivent être stockés dans chaque base de données.
- Améliore la fiabilité de `LISTEN`/`NOTIFY` (Tom)
- Permet à `REINDEX` de réindexer fiablement les index de catalogues système non partagés (Tom)

  Ceci permet aux tables système d'être réindexées sans avoir besoin d'une session en solo, qui était nécessaire dans les versions précédentes. Les seules tables qui requièrent maintenant une session solo sont les tables système global `pg_database`, `pg_shadow` et `pg_group`.

### E.17.3.7. Modifications sur les types de données et sur les fonctions

- Nouveau paramètre serveur `extra_float_digits` pour contrôler la précision d'affichage des nombres à virgule glottante (Pedro Ferreira, Tom)

  Ceci contrôle la précision qui causait des problèmes lors des tests de régression.
- Autorisation de `+1300` comme spécificateur numérique de fuseau horaire, pour FJST (Tom)
- Suppression des fonctions `oidrand`, `oidsrand` et `userfntest` rarement utilisées (Neil)
- Ajout de la fonction `md5()` sur le serveur principal, auparavant déjà dans `contrib/pgcrypto` (Joe)

  Une fonction MD5 était fréquemment demandée. Pour des possibilités plus complexes de cryptage, utilisez `contrib/pgcrypto`.
- Augmentation de l'étendue des données de `timestamp` (John Cochran)
- Modification de `EXTRACT(EPOCH FROM timestamp)` pour que `timestamp without time zone` soit supposé être dans le fuseau local, et non pas en GMT (Tom)
- Récupération des divisions par zéro au cas où le système d'exploitation ne le permet pas (Tom)
- Modification du type de données `numeric` interne en base 10000 (Tom)

- Nouvelle fonction `hostmask()` (Greg Wickham)
- Corrections pour `to_char()` et `to_timestamp()` (Karel)
- Permet aux fonctions de prendre tout type de données en argument et de renvoyer aussi tout type, en utilisant `anyelement` et `anyarray` (Joe)

  Ceci permet la création de fonctions qui peuvent fonctionner avec tout type de données.
- Les tableaux peuvent maintenant être spécifiés comme `ARRAY[1,2,3]`, `ARRAY[['a','b'],['c','d']]` ou `ARRAY[ARRAY[ARRAY[2]]]` (Joe)
- Autorise une comparaison correcte des tableaux, ceci incluant le support de `ORDER BY` et `DISTINCT` (Joe)
- Autorise les index sur les colonnes de type tableau (Joe)
- Autorise la concaténation de tableaux avec `||` (Joe)
- Autorise la qualification `WHERE` *`expr op`* `ANY/SOME/ALL (`*`array_expr`*`)` (Joe)

  Ceci permet aux tableaux de se comporter comme une liste de valeurs dans des requêtes comme `SELECT * FROM tab WHERE col IN (array_val)`.
- Nouvelles fonctions de tableaux `array_append`, `array_cat`, `array_lower`, `array_prepend`, `array_to_string`, `array_upper`, `string_to_array` (Joe)
- Autorise l'utilisation de fonctions polymorphiques par les agrégats définis par l'utilisateur (Joe)
- Permet les affectations dans des tableaux vides (Joe)
- Autorise 60 dans la partie secondes des champs de type `time`, `timestamp` et `interval` (Tom)

  Les valeurs à soixante secondes sont nécesssaires pour les << leap seconds >>.
- Permet au type de données `cidr` d'être converti en `text` (Tom)
- Interdit les noms de fuseaux horaires invalides dans SET TIMEZONE (Tom)
- Supprime les espaces à la fin lorsque une donnée de type `char` est convertie en type `varchar` ou `text` (Tom)
- Fait que `float(`*`p`*`)` mesure la précision *`p`* en chiffres binaires, et non pas en chiffres décimaux (Tom)
- Ajoute le support IPv6 aux types de données `inet` et `cidr` (Michael Graff)
- Ajoute la fonction `family()` pour rapporter si l'adresse fait partie de la famille IPv4 ou IPv6 (Michael Graff)
- Fait que `SHOW datestyle` génère une sortie similaire à celle utilisée par `SET datestyle` (Tom)
- Fait que `EXTRACT(TIMEZONE)` et `SET/SHOW TIME ZONE` suivent la convention SQL pour le signe des décalages de zones horaires, c'est−à−dire que le positif est l'est à partir d'UTC (Tom)
- Correction de `date_trunc('quarter', ...)` (Böjthe Zoltán)

  Les anciennes versions renvoyaient une valeur incorrecte pour cet appel de fonction.
- Fait que `initcap()` soit plus compatible avec Oracle (Mike Nolan)

  `initcap()` met en majuscule toute lettre après un caractère non alphanumérique, plutôt qu'après un espace blanc.
- Autorise seulement l'ordre du champ `datestyle` pour les valeurs de données ne faisant pas partie du format ISO−8601 (Greg)
- Ajout de nouvelles valeurs pour `datestyle`, `MDY`, `DMY` et `YMD` pour initialiser l'ordre du champ en entrée ; honore `US` et `European` pour une compatibilité ascendante (Tom)
- Les chaînes littérales comme `'now'` ou `'today'` ne fonctionneront plus comme valeur par défaut d'une colonne. À la place, utilisez des fonctions comme `now()`, `current_timestamp`. (modification requise pour les instructions préparées) (Tom)
- Traitement de NaN comme plus important que tout autre valeur dans `min()`/`max()` (Tom)

NaN était déjà trié après les valeurs numériques ordinaires dans la plupart des cas, mais `min()` et `max()` ne le faisaient pas correctement.
- Empêche l'intervalle de supprimer l'affichage de `:00`
- Nouvelles fonctions `pg_get_triggerdef(prettyprint)` et `pg_constraint_is_visible() (Christopher)`
- Permet la spécification de l'heure avec `040506` ou `0405` (Tom)
- L'ordre de la date en entrée doit être `YYYY-MM-DD` (avec une année sur quatre chiffres) ou correspondre à `datestyle`
- Fait que `pg_get_constraintdef` supporte les contraintes uniques, de clés primaires et de vérification de contraintes (Christopher)

### E.17.3.8. Server–Side Language Changes

- Empêche PL/pgSQL de s'arrêter brutalement lorsque `RETURN NEXT` est utilisé avec une variable d'enregistrement ne correspondant à aucune ligne (Tom)
- Fait que l'interface `spi_execute` de PL/Python gère les valeurs NULL correctement (Andrew Bosma)
- Autorise la déclaration des variables de type composé dans PL/pgSQL sans `%ROWTYPE` (Tom)
- Corrige la fonction `_quote()` de PL/Python pour gérer les gros entiers
- Modification de PL/Python en langage sans confiance, maintenant appelé `plpythonu` (Kevin Jacobs, Tom)

  Le langage Python ne support plus un environnement d'exécution restreint, donc la version de confiant de PL/Python a été supprimée. Si cette situation évolue, une version de PL/python pouvant être utilisé par des utilisateurs ordinaires sera préparée.
- Autorise les fonctions polymorphiques dans PL/pgSQL (Joe, Tom)
- Autorise les fonctions SQL polymorphiques (Joe)
- Amélioration du mécanisme de cache des fonctions compilées dans PL/pgSQL avec un support complet du polymorphisme (Joe)
- Ajout du nouveau paramètre `$0` dans PL/pgSQL, représentant le type de retour réel de la fonction (Joe)
- Autorise PL/Tcl et PL/Python à utiliser le même déclencheur sur des tables multiples (Tom)
- Correction de `spi_prepare` de PL/Tcl pour qu'il accepte les noms de types qualifiés complètement dans la liste des types des paramètres (Jan)

### E.17.3.9. Modifications sur psql

- Ajout de `\pset pager always` pour utiliser en permanence le pagineur (Greg)

  Ceci force l'utilisation du pagineur même si le nombre de lignes est plus petit que la hauteur de l'écran. Ceci est valable pour les lignes de données ne tenant pas sur une ligne de l'écran.
- Amélioration de la complétion par tabulation (Rod, Ross Reedstrom, Ian Barwick)
- Réordonnancement de l'aide `\?` en la groupant (Harald Armin Massa, Bruce)
- Ajout des commandes antislash pour liste les schémas, fonctions de conversions et conversions (Christopher)
- `\encoding` est modifié pour se baser sur le paramètre serveur `client_encoding` (Tom)

  Dans les versions précédentes, `\encoding` n'était pas tenu au courant des changements de codage fait en utilisant `SET client_encoding`.
- Sauvegarde du tampon de l'éditeur dans l'historique de readline (Ross)

Lorsque \e est utilisé pour éditer une requête, le résultat est sauvegardé dans l'historique de readline pour récupération en utilisant la flèche vers le haut.
- Amélioration de l'affichage de \d (Christopher)
- Amélioration du mode HTML pour être plus conforme au standard (Greg)
- Nouvelle fonction \set AUTOCOMMIT off (Tom)

  Ceci prend la place du paramètre serveur autocommit qui a été supprimé.
- Nouveau \set VERBOSITY pour contrôler le détail des erreurs (Tom)

  Ceci contrôle le niveau du nouveau détail des rapports.
- Nouvelle invite de séquence d'échappement %x pour afficher l'état de la transaction (Tom)
- Les options longues pour psql sont maintenant disponibles sur toutes les plateformes

### E.17.3.10. Modifications de pg_dump

- Multiples corrections de pg_dump, incluant le format tar et les gros objets
- Autorise la sauvegarde de schémas spécifiques par pg_dump (Neil)
- Fait que pg_dump préserve les caractéristiques de stockage des colonnes (Christopher)

  Ceci préserve l'information ALTER TABLE ... SET STORAGE.
- Fait que pg_dump préserve les caractéristiques CLUSTER (Christopher)
- Fait que pg_dumpall utilise GRANT/REVOKE pour sauvegarder les droits au niveau de la base de données (Tom)
- Autorise pg_dumpall à supporter les options -a, -s, -x de pg_dump (Tom)
- Empêche pg_dump de mettre en minuscule les identifieurs spécifiés sur la ligne de commande (Tom)
- Les options --use-set-session-authorization et --no-reconnect de pg_dump ne font maintenant plus rien, toutes les sauvegardes utilisent SET SESSION AUTHORIZATION

  pg_dump ne se reconnecte plus pour changer d'utilisateurs mais utilise à la place SET SESSION AUTHORIZATION. Ceci réduira le nombre de demandes de mot de passe lors des restaurations.
- Les options longues pour pg_dump sont maintenant disponibles sur toutes les plateformes

  PostgreSQL inclut maintenant ses propres routines de traitement des options longues.

### E.17.3.11. Modifications de libpq

- Ajout de la fonction PQfreemem pour libérer la mémoire de Windows, suggéré pour le NOTIFY (Bruce)

  Windows requiert que la mémoire allouée dans une bibliothèque soit libérée par une fonction de la même bibliothèque, donc free() ne fonctionne pas pour libérer la mémoire allouée par libpq. PQfreemem est la bonne façon de libérer la mémoire libpq, spécialement sur Windows, et est aussi recommandée sur les autres plateformes.
- Fonctionnalité de service de document, et ajout d'un fichier d'exemple (Bruce)

  Ceci permet aux clients de rechercher une information de connexion dans un ficher central sur la machine client.
- Fait que PQsetdbLogin a les mêmes valeurs par défaut que PQconnectdb (Tom)
- Autorise libpq à échouer proprement lorsque les ensembles de résultats sont trop importants (Tom)
- Amélioration de la performance de la fonction PQunescapeBytea (Ben Lamb)

- Autorise une libpq totalement compatible avec les threads avec l'option `--enable-thread-safety` de `configure` (Lee Kindness, Philip Yarra)
- Autorise la fonction `pqInternalNotice` à accepter les chaînes de format et leurs arguments au lieu d'un simple message préformaté (Tom, Sean Chittenden)
- Contrôle la négotiation SSL avec les valeurs `sslmode` `disable`, `allow`, `prefer` et `require` (Jon Jensen)
- Autorise les nouveaux codes d'erreur et les niveaux de texte (Tom)
- Autorise l'accès à la table et la colonne sous−jacente du résultat d'une requête (Tom)

  Ceci est utile pour les applications de création de requêtes qui veulent connaître la table sous−jacente et les noms de colonnes spécifiées avec un ensemble de résultats spécifique.
- Autorise l'accès au statut de la transaction en cours (Tom)
- Ajoute la capacité de passer des données binaires directement au serveur (Tom)
- Ajout des fonctions `PQexecPrepared` et `PQsendQueryPrepared` réalisant un bind/execute des précédentes instructions préparées (Tom)

#### E.17.3.12. Modification de JDBC

- Autorise `setNull` sur des ensembles de résultats en lecture/écriture
- Autorise `executeBatch` sur une instruction préparée (Barry)
- Support des connexions SSL (Barry)
- Gestion des noms de schéma dans les ensembles de résultats (Paul Sorenson)
- Ajout du support de refcursor (Nic Ferrier)

#### E.17.3.13. Diverses modifications dans l'interface

- Empêche les possibles pertes de mémoire ou << core dump >> lors d'un arrêt de libpgtcl (Tom)
- Ajout de la compatibilité Informix pour ECPG (Michael)

  Ceci permet à ECPG de traiter des programmes C embarqués qui étaient écrit en utilisant certaines extensions Informix.
- Ajout du type `decimal` à ECPG sur une longueur fixe, pour Informix (Michael)
- Autorise les programmes SQL embarqués en mode compatible avec les threads après utilisation de l'option `--enable-thread-safety` de `configure` (Lee Kindness, Bruce)

  Ceci permet à de nombreux threads d'accéder à la base de données en même temps.
- Déplacement du client Python PyGreSQL sur http://www.pygresql.org (Marc)

#### E.17.3.14. Modifications du code source

- Empêche le besoin de fichiers de résultats de régression séparés par plateforme (Tom)
- Amélioration de la primitive de verrou sur PPC (Reinhard Max)
- Nouvelle fonction `palloc0` pour allouer et effacer la mémoire (Bruce)
- Correction du code de verrou pour le CPU s390x (64−bit) (Tom)
- Autorise OpenBSD à utiliser les crédentiels ident locales (William Ahern)
- Fait que les arbres de plans de requêtes sont en lecture seule pour l'exécuteur (Tom)
- Ajout du script de démarrage pour Darwin (David Wheeler)
- Permet à libpq de compiler avec le compilateur C++ de Borland (Lester Godwin, Karl Waclawek)
- Utilisation de notre propre version de `getopt_long()` si nécessaire (Peter)
- Conversion des scripts d'administration en C (Peter)

- Bison >= 1.85 est maintenant requis pour construire la grammaire PostgreSQL, s'il s'agit d'une construction à partir du CVS
- Assemblage de la documentation en un seul livre (Peter)
- Ajout de fonctions compatibles Windows (Bruce)
- Permet aux interfaces clients d'être compilées sous MinGW (Bruce)
- Nouvelle fonction `ereport()` pour les rapports d'erreur (Tom)
- Support du compilateur Intel sur Linux (Peter)
- Amélioration des scripts de démarrage Linux (Slawomir Sudnik, Darko Prenosil)
- Ajout du support pour AMD Opteron et Itanium (Jeffrey W. Baker, Bruce)
- Suppression de l'option `--enable-recode` de `configure`

  Ce n'était plus nécessaire maintenant que nous avons `CREATE CONVERSION`.
- Génère une erreur de compilation si le code spinlock n'est pas trouvé (Bruce)

  Les plateformes sans code spinlock échoueront maintenant à la compilation plutôt que silencieusement lors de l'utilisation de sémaphores. Cet échec peut être désactivé avec une nouvelle option `configure`.

### E.17.3.15. Modifications dans contrib

- Modification de la licence de dbmirror par celle de BSD
- Amélioration de earthdistance (Bruno Wolff III)
- Amélioration de la portabilité de pgcrypto (Marko Kreen)
- Suppression du crash en xml (John Gray, Michael Richards)
- Mise à jour d'oracle
- Mise à jour de mysql
- Mise à jour de cube (Bruno Wolff III)
- Mise à jour d'earthdistance pour utiliser cube (Bruno Wolff III)
- Mise à jour de btree_gist (Oleg)
- Nouveau module de recherche en texte complet tsearch2 (Oleg, Teodor)
- Ajout d'une fonction inter tableau basé sur le découpage dans tablefuncs (Joe)
- Ajout d'une colonne serial pour trier les familles `connectby()` dans tablefuncs (Nabil Sayegh,Joe)
- Ajout des connexions nommées persistantes dans dblink (Shridhar Daithanka)
- Le nouveau pg_autovacuum permet des `VACUUM` automatique (Matthew T. O'Connor)
- Fait que pgbench honore les variables d'environnement `PGHOST`, `PGPORT`, `PGUSER` (Tatsuo)
- Amélioration d'intarray (Teodor Sigaev)
- Amélioration de pgstattuple (Rod)
- Correction d'un bogue dans `metaphone()` (contrib fuzzystrmatch)
- Amélioration d'adddépend (Rod)
- Mise à jour de spi/timetravel (Böjthe Zoltán)
- Correction de l'option `-s` de dbase et amélioration de la gestion des non ASCII (Thomas Behr, Márcio Smiderle)
- Suppression du module tableau car les fonctionnalités sont maintenant incluent par défaut (Joe)

## E.18. Version 7.3.12

**Date de sortie :** 2005-12-12

Cette version contient quelques corrections sur la 7.3.11.

### E.18.1. Migration vers la version 7.3.12

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une version 7.3.X. Néanmoins, si vous mettez à jour à partir d'une version antérieure à la 7.3.10, voir les notes de sortie de la 7.3.10.

### E.18.2. Modifications

- Correction d'un cas extrême dans la gestion des traces des transactions

  Il existait une petite possibilité pour laquelle une opération d'entrée/sortie pouvait être initiée pour la mauvaise page, amenant à un échec Assert ou à une corruption de données.
- Corrections sur `/contrib/ltree` (Teodor)
- Correction d'une vieille erreur sur les jointures externes

  Ce bogue a quelque fois causé une fausse erreur << RIGHT JOIN is only supported with merge−joinable join conditions >>.
- Empêchement d'un arrêt brutal dans pg_autovacuum quand une table a été supprimée

## E.19. Version 7.3.11

**Date de sortie :** 2005−10−04

Cette version contient quelques correctifs par rapport à la 7.3.10.

### E.19.1. Migration vers la version 7.3.11

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une version 7.3.X. Néanmoins, si vous mettez à jour à partir d'une version antérieure à la 7.3.10, voir les notes de sortie de la 7.3.10.

### E.19.2. Modifications

- Correction d'une erreur qui permettait au `VACUUM` de supprimer les chaînes `ctid` trop tôt, et ajout de vérification supplémentaire dans le code qui suit les liens `ctid`

  Ceci corrige un vieux problème qui pourrait causer des arrêt brutaux dans de rares circonstances.
- Correction de `CHAR()` pour ajouter le nom d'espaces spécifié lors de l'utilisation d'un ensemble de caractères multi−octets (Yoshiyuki Asaba)

  Dans les versions précédentes, l'ajout pour `CHAR()` était incorrect car il ajoutait seulement le nombre d'octets spécifiés sans prendre en considération la façon dont les caractères étaient stockés.
- Correction des lignes manquantes dans les requêtes du type `UPDATE a=... WHERE a...` avec un index GiST sur la colonne `a`
- Amélioration de la vérification pour les pages WAL partiellement écrites
- Amélioration de la solidité de la gestion des signaux lorsque SSL est activé
- Plusieurs corrections de perte mémoire
- Quelques améliorations sur la portabilité
- Correction de PL/PgSQL pour gérer correctement `var := var` quand la variable est passée par référence

# E.20. Version 7.3.10

**Date de sortie :** 2005−05−09

Cette version contient quelques corrections sur la 7.3.9, dont des correctifs sur des trous de sécurité.

## E.20.1. Migration vers la version 7.3.10

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une version 7.3.X. Néanmoins, c'est une façon possible de gérer un problème de sécurité significatif, qui a été trouvé dans le contenu initial des catalogues systèmes des versions 7.3.X. Une séquence sauvegarde/initdb/restauration utilisant le initdb de la 7.3.10 corrigera automatiquement ces problèmes.

Le problème de sécurité le plus important est que les fonctions de conversion du codage des ensembles de caractères peuvent être appelées à partir de commandes SQL par des utilisateurs non pribilégiés alors que les fonctions n'ont pas été conçues pour un tel usage et ne sont pas sécurisées contre des choix rusés des arguments. Le correctif implique de modifier la liste déclarée des arguments de ces fonctions pour qu'elles ne soient plus appelées à partir des commandes SQL. (Ceci n'affecte pas leur utilisation normale par la machinerie de conversion du codage.) Il est fortement recommandé que toutes les installations corrigent ces erreurs, soit par un initdb soit en suivant les procédures de réparation données ci−dessous. Les erreurs permettent au moins à des utilisateurs non privilégiés de la base de données d'arrêter brutalement leur processus serveur et pourraient permettre à des utilisateurs non privilégiés de gagner les privilèges d'un superutilisateur.

Si vous souhaitez ne pas lancer d'initdb, exécutez la procédure suivante à la place en tant que superutilisateur de la base de données :

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
      AND proargtypes[2] = 'cstring'::regtype;
-- La commande devrait rapporter qu'elle a mis à jour 90 lignes ;
-- si ce n'est pas le cas, annulez (rollback) et investiguez au lieu de valider !
COMMIT;
```

Les procédures ci−dessus doivent être exécutées pour *chaque* base de données d'une installation, ceci incluant `template1` et devant idéalement inclure aussi `template0`. Si vous ne corrigez pas les bases de données modèles, alors toute base de données créée par la suite contiendra les mêmes erreurs. `template1` peut être corrigé de la même façon que les autres bases de données alors que `template0` requiert des étapes supplémentaires. Tout d'abord, à partir de n'importe quel base de données

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Ensuite, connectez−vous à `template0` et effectuez les opérations de réparation. Faites

```
-- gelez de nouveau template0 :
VACUUM FREEZE;
-- et protégez-contre toute nouvelle modification :
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

## E.20.2. Modifications

- Modification de la signature de la fonction de codage pour empêcher toute mauvaise utilisation
- Réparation d'une ancienne condition qui permettait à une transaction d'être vue comme validée pour certains buts (par exemple SELECT FOR UPDATE) légèrement plus tôt que dans les autres buts

  Ceci est un bogue extrêmement sérieux car il pourrait amener à des incohérences apparents des données et visibles brièvement par les applications.
- Réparation d'une condition entre extension de relation et VACUUM

  Théoriquement, ceci pourrait avoir causé des pertes d'une page de données tout juste insérées, bien que la probabilité d'un tel scénario semble extrêmement faible. Il n'existe pas de cas connus où cela a provoqué plus qu'un échec d'Assert.
- Correction des comparaisons des valeurs `TIME WITH TIME ZONE`

  Le code de comparaison était mauvais dans le cas où le commutateur de configuration `--enable-integer-datetimes` avait été utilisé. NOTE : si vous avez un index sur une colonne `TIME WITH TIME ZONE`, il sera nécessaire de le `REINDEX`er après avoir installé cette mise à jour parce que ce correctif corrige l'ordre de tri des valeurs de colonnes.
- Correction de `EXTRACT(EPOCH)` pour les valeurs de type `TIME WITH TIME ZONE`
- Correction d'un mauvais affichage des secondes fractionnelles négatives dans les valeurs `INTERVAL`

  Cette erreur est seulement survenue quand l'option de configuration `--enable-integer-datetimes` avait été utilisée.
- Vérification supplémentaires de dépassement de tampon dans plpgsql (Neil)
- Correction de pg_dump pour qu'il sauvegarde correctement les noms des déclencheurs contenant `%` (Neil)
- Correction de `contrib/pgcrypto` pour les nouvelles constructions d'OpenSSL (Marko Kreen)
- Encore plus de correctifs 64 bits pour `contrib/intagg`
- Empêche une optimisation incorrecte des fonctions renvoyant `RECORD`

# E.21. Sortie 7.3.9

**Date de sortie :** 2005−01−31

Cette version contient une grande variété de correctifs pour la 7.3.8, incluant certains spécifiques à des questions de sécurité.

## E.21.1. Migration vers la version 7.3.9

Une sauvegarde/restauration n'est pas requise pour ceux qui utilisent une version 7.3.X.

## E.21.2. Modifications

- Interdit l'utilisation de `LOAD` aux utilisateurs standards

  Sur les plateformes qui exécuteront automatiquement les fonctions d'initialisation d'une bibliothèque partagée (ceci inclut au moins Windows et les Unix basés sur ELF), `LOAD` est utilisable pour faire exécuter un code arbitraire par le serveur. Merci à NGS Software pour cette information.

- Vérifie que le créateur d'une fonction d'agrégat a le droit d'exécuter les fonctions de transition spécifiées

  Ce problème rendait possible le contournement de l'interdiction du droit EXECUTE sur une fonction.
- Correction de problèmes de sécurité et de problèmes sur les 64 bits dans contrib/intagg
- Ajout du marquage STRICT nécessaire à quelques fonctions dans contrib (Kris Jurka)
- Évite un dépassement de tampon lorsque la déclaration du curseur plpgsql dispose trop de paramètres (Neil)
- Correction d'erreurs de plannification pour les jointures complètes et externes à droite

  Le résultat de la jointure était faussement supposé trié de la même façon que l'entrée gauche. Ceci pouvait délivrer une sortie mal triée à l'utilisateur mais, dans le cas de jointures d'assemblage imbriquées, pouvait donner de mauvaises réponses.
- Correction de plperl pour les guillemets dans les champs
- Correction de l'affichage des intervalles négatifs pour les styles de date SQL et GERMAN
- Fait que age(timestamptz) calcule à partir du fuseau horaire et non pas de GMT

# E.22. Sortie 7.3.8

**Date de sortie :** 2004−10−22

Cette version contient plusieurs corrections pour la version 7.3.7.

## E.22.1. Migration vers la version 7.3.8

Une sauvegarde/restauration n'est pas requise pour ceux qui utilisent une version 7.3.X.

## E.22.2. Modifications

- Réparation d'un échec possible pour mettre à jour les bits d'astuce sur disque

  Sous quelques rares circonstances, ceci pourrait mener à des échecs << could not access transaction status >>, qui se qualifient comme bogue pouvant entraîner des pertes de données.
- Assurance que la jointure externe hachée ne manque pas de lignes

  Les jointures gauches très importantes pourraient échouer à afficher les lignes du côté gauche sans correspondance en donnant seulement la distribution droite des données.
- Interdit l'exécution de pg_ctl en tant que root

  Ceci est fait pour évacuer tout risque de problèmes de sécurité.
- Évite l'utilisation de fichiers temporaires dans `/tmp` dans `make_oidjoins_check`

  Ceci a été rapportée comme un problème de sécurité bien qu'il est de peu d'importance car il n'y a aucune raison pour que les utilisateurs autres que les développeurs utilisent ce script de toute façon.

# E.23. Sortie 7.3.7

**Date de sortie :** 2004−08−16

Cette version contient une correction critique de la version 7.3.6 et quelques éléments mineurs.

## E.23.1. Migration vers la version 7.3.7

Une sauvegarde/restauration n'est pas requise pour ceux qui utilisent une version 7.3.X.

## E.23.2. Modifications

- Empêche une perte possible de transactions validées lors d'un arrêt brutal

  À cause d'un verrouillage insuffisant entre la validation des transactions et des points de vérification, il était possible pour les transactions validées juste avant le point de vérification perdu, complètement ou partiellement, le plus récent après un arrêt brutal de la base de données et un redémarrage. Ceci est un bogue sérieux qui existait depuis PostgreSQL 7.1.
- Suppression du traitement de mots asymétriques dans tsearch (Teodor)
- Qualifie les noms de fonction proprement avec le schéma lors de la sauvegarde d'un CAST

# E.24. Sortie 7.3.6

**Date de sortie :** 2004−03−02

Cette version contient plusieurs correction de la version 7.3.5.

## E.24.1. Migration vers la version 7.3.6

Une sauvegarde/restauration n'est *pas* requise pour ceux qui utilisent une version 7.3.X.

## E.24.2. Modifications

- Revert erroneous changes in rule permissions checking

  A patch applied in 7.3.3 to fix a corner case in rule permissions checks turns out to have disabled rule−related permissions checks in many not−so−corner cases. This would for exemple allow users to insert into views they weren't supposed to have permission to insert into. We have therefore reverted the 7.3.3 patch. The original bug will be fixed in 8.0.
- Repair incorrect order of operations in GetNewTransactionId()

  This bug could result in failure under out−of−disk−space conditions, including inability to restart even after disk space is freed.
- Ensure configure selects −fno−strict−aliasing even when an external value for CFLAGS is supplied

  On some platforms, building with −fstrict−aliasing causes bugs.
- Make pg_restore handle 64−bit off_t correctly

This bug prevented proper restoration from archive files exceeding 4Gb.
- Make contrib/dblink not assume that local and remote type OID match (Joe)
- Quote connectby()'s start_with argument properly (Joe)
- Don't crash when a rowtype argument to a plpgsql function is NULL
- Avoid generating invalid character encoding sequences in corner cases when planning LIKE operations
- Ensure text_position() cannot scan past end of source string in multibyte cases (Korea PostgreSQL Users' Group)
- Fix index optimization and selectivity estimates for LIKE operations on bytea columns (Joe)

# E.25. Sortie 7.3.5

**Date de sortie :** 2003−12−03

Cette version dispose de quelques corrections de la version 7.3.4.

## E.25.1. Migration vers la version 7.3.5

Une sauvegarde/restauration n'est *pas* requise pour ceux qui utilisent une version 7.3.X.

## E.25.2. Modifications

- Force zero_damaged_pages to be on during recovery from WAL
- Prevent some obscure cases of << variable not in subplan target lists >>
- Force stats processes to detach from shared memory, ensuring cleaner shutdown
- Make PQescapeBytea and byteaout consistent with each other (Joe)
- Added missing SPI_finish() calls to dblink's get_tuple_of_interest() (Joe)
- Fix for possible foreign key violation when rule rewrites INSERT (Jan)
- Support qualified type names in PL/Tcl's spi_prepare command (Jan)
- Make pg_dump handle a procedural language handler located in pg_catalog
- Make pg_dump handle cases where a custom opclass is in another schema
- Make pg_dump dump binary−compatible casts correctly (Jan)
- Fix insertion of expressions containing subqueries into rule bodies
- Fix incorrect argument processing in clusterdb script (Anand Ranganathan)
- Fix problems with dropped columns in plpython triggers
- Repair problems with to_char() reading past end of its input string (Karel)
- Fix GB18030 mapping errors (Tatsuo)
- Fix several problems with SSL error handling and asynchronous SSL I/O
- Remove ability to bind a list of values to a single parameter in JDBC (prevents possible SQL−injection attacks)
- Fix some errors in HAVE_INT64_TIMESTAMP code paths
- Fix corner case for btree search in parallel with first root page split

# E.26. Sortie 7.3.4

**Date de sortie :** 2003−07−24

Cette version comporte plusieurs corrections sur la version 7.3.3.

## E.26.1. Migration vers la version 7.3.4

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux disposant d'une 7.3.*.

## E.26.2. Modifications

- Réparation des problèmes de conversion timestamp vers date avant 2000
- Empêche la rare probabilité d'un échec de lancement du serveur (Tom)
- Correction de bogues dans la conversion interval vers time (Tom)
- Ajout des noms de contraintes dans quelques endroits de pg_dump (Rod)
- Amélioration des performances de fonctions avec beaucoup de paramètres (Tom)
- Correction du dépassement de tampon de to_ascii() (Tom)
- Empêche que la restauration des commentaires de la base de données envoie une erreur (Tom)
- Coutournement du strxfrm() bogué présent dans quelques versions de Solaris (Tom)
- Réalise un échappement propre des chaînes setObject() de jdbc pour améliorer la sécurité (Barry)

# E.27. Sortie 7.3.3

**Date de sortie :** 2003−05−22

Cette version comporte plusieurs corrections sur la version 7.3.2.

## E.27.1. Migration vers la version 7.3.3

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 7.3.*.

## E.27.2. Modifications

- Réparation de calculs quelque fois incorrect de StartUpID après un crash
- Évite des lenteurs avec un grand nombre de déclencheurs déferrés dans une transaction (Stephan)
- Ne verrouille pas la ligne référencée quand `UPDATE` ne modifie pas la valeur de la clé étrangère (Jan)
- Utilisation de `-fPIC`, et non pas `-fpic`, sur Sparc (Tom Callaway)
- Réparation du manque de connaissance des schémas dans contrib/reindexdb
- Correction d'une erreur dans contrib/intarray pour le tableau résultat à zéro élément (Teodor)
- S'assure que le script createuser quitte avec control−C (Oliver)
- Correction d'erreurs quand le type d'une colonne supprimée a lui−même été supprimé
- `CHECKPOINT` ne cause pas de panique sur la base de données sur à un échec sur des étapes non critiques
- Accepte 60 dans les champs secondes des valeurs de type timestamp, time, interval
- Lancement d'une note, pas d'une erreur, si la précision de `TIMESTAMP`, `TIME` ou `INTERVAL` est trop large
- Correction de la fonction de conversion `abstime-to-time` (la correction n'est pas appliquée sauf si vous lancez initdb)
- Correction de l'entrée pg_proc pour `timestampt_izone` (le correctif n'est pas appliqué sauf si vous lancez initdb)
- Fait que `EXTRACT(EPOCH FROM timestamp without time zone)` traite l'entrée comme une heure locale

- 'now'::timestamptz donne la mauvaise réponse si le fuseau horaire a changé précédemment lors de la transaction
- Le HAVE_INT64_TIMESTAMP pour time with timezone surcharge son entrée
- Accepte GLOBAL TEMP/TEMPORARY comme synonyme de TEMPORARY
- Évite un mauvais de la vérification des droits du schéma dans les déclencheurs de clés étrangères
- Correction de bogues dans les déclencheurs de clés étrangères pour l'action SET DEFAULT
- Correction d'une mauvaise vérification de << time−qual >> dans la récupération des lignes pour UPDATE et DELETE triggers
- Les clauses de clés étrangères étaient analysées mais ignorées dans ALTER TABLE ADD COLUMN
- Correction du problème du script createlang dans les cas où la fonction de gestion existe déjà
- Correction du mauvais comportement pour les tables sans colonne dans pg_dump, COPY, ANALYZE, ainsi qu'à d'autres emplacements
- Correction du mauvais comportement dans func_error() sur les noms de types contenant '%'
- Correction du mauvais comportement de replace() sur les chaînes contenant '%'
- Échec lorsque des expressions rationnelles contiennent certains caractères multi−octets
- Tient bien compte des NULL dans plus de cas pour les estimations de taille de jointures
- Évite un conflit avec la définition du système de la fonction ou macro isblank()
- Correction de l'échec pour la conversion de valeurs de type point en EUC_TW (Tatsuo)
- Correction d'une récupération d'une erreur à partir des appels SSL_read/SSL_write
- Don't do early constant−folding of type coercion expressions
- Valide les champs des en−têtes de page immédiatement après les avoir lu dans chaque page
- Réparation d'une mauvaise vérification pour les variables non groupées dans des jointures non nommées
- Correction d'un dépassement de tampon dans to_ascii (Guido Notari)
- Corrections de contrib/ltree (Teodor)
- Correction d'un arrêt brutal (<< core dump >>) dans la détections de verrous bloqués sur les machines où char est non signé
- Évite de se trouver sans tampons dans des parcours d'index multiples (bogue introduit dans la 7.3)
- Correction des fonctions d'estimation de la sélectivité du planficateur pour gérer correctement les domaines
- Corrige le bogue d'allocation mémoire de dbmirror (Steven Singer)
- Empêche les boucles infinies dans ln(numeric) à cause d'une erreur d'arrondi
- GROUP BY en plein confusion s'il y a plusieurs éléments GROUP BY identiques
- Correction d'un mauvais plan lorsque des UPDATE/DELETE héritées référencent une autre table héritée
- Empêche le groupement sur des index incomplets (partiel ou stockant uniquement les non NULL)
- Demande d'arrêt du service au bon moment s'il arrive au moment du lancement
- Correction des liens gauches dans les index temporaires (pourrait faire oublier des entrées à des parcours inverses)
- Correction d'une gestion incorrecte du paramétrage client_encoding dans postgresql.conf (Tatsuo)
- Correction d'un échec pour répondre à pg_ctl stop −m fast une fois que Async_NotifyHandler est lancé
- Correction de SPI dans les cas où la règle contient plusieurs instructions du même type
- Correction d'un problème avec la vérification pour un mauvais type de droit d'accès dans la requête de la règle
- Correction d'un problème avec EXCEPT dans CREATE RULE
- Empêche des problèmes lors de la suppression de tables temporaires avec les colonnes de type serial
- Correction de l'échec replace_vars_with_subplan_refs dans les vues complexes
- Correction de la lenteur des expressions rationnelles dans les codes à un seul octet (Tatsuo)
- Permet la qualification de noms de type dans CREATE CAST et DROP CAST
- Accepte SETOF type[], qui avait été écrit auparavant SETOF _type

- Correction d'un arrêt brutal (<< core dump >>) pg_dump dans certains cas pour les langages de procédures
- Force le style de date ISO dans la sortie pg_dump pour des raisons de portabilité (Oliver)
- pg_dump échouait lors de la gestion d'une erreur renvoyée par `lo_read` (Oleg Drokin)
- pg_dumpall échouait avec les groupes sans membres (Nick Eskelinen)
- pg_dumpall échouait lors de la reconnaissance de l'option
- pg_restore échouait lors de la restauration des blobs si −X disable−triggers est spécifié
- Réparation de la perte mémoire entre fonctions dans plpgsql
- La commande `elog` de pltcl s'arrêtait brutalement si des mauvais paramètres étaient soumis (Ian Harding)
- plpython utilisait de mauvaises valeurs de `atttypmod` (Brad McLean)
- Correction d'une mauvaise mise entre guillemets des valeurs booléennes dans l'interface Python (D'Arcy)
- Ajout de la méthode `addDataType()` pour l'interface PGConnection de JDBC
- Correction de nombreux problèmes avec les ensembles de résultats en lecture/écriture pour JDBC (Shawn Green)
- Correction de nombreux problèmes avec DatabaseMetaData pour JDBC (Kris Jurka, Peter Royal)
- Correction d'un problème avec l'analyse des ACL de table dans JDBC
- Meilleur message d'erreur pour les problèmes de conversions des ensembles de caractères dans JDBC

# E.28. Sortie 7.3.2

**Date de sortie :** 2003−02−04

Cette version comporte plusieurs corrections sur la version 7.3.1.

## E.28.1. Migration vers la version 7.3.2

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 7.3.*.

## E.28.2. Modifications

- Restauration de la création de la colonne OID dans CREATE TABLE AS / SELECT INTO
- Correction de l'arrêt brutal pg_dump lors de la sauvegarde des vues possédant des commentaires
- Sauvegarde proprement les contraintes DEFERRABLE/INITIALLY DEFERRED
- Correction d'UPDATE lors la numérotation des colonnes de tables enfants diffère de celle du parent
- Augmentation de la valeur par défaut de max_fsm_relations
- Correction d'un problème lors de la récupération en sens inverse ddans un curseur pour une requête à une seule ligne
- Fait que la récupération (<< fetch >>) inverse fonctionne correctement avec le curseur sur une requête SELECT DISTINCT
- Correction des problèmes lors du chargement de fichiers pg_dump contenant une utilisation de contrib/lo
- Correction d'un problème avec les noms utilisateurs entièrement en chiffres
- Correction d'une probable perte de mémoire et d'un arrêt brutal lors de la déconnexion de libpgtcl
- Fait que la commande spi_execute de plpython gère correctement les NULL (Andrew Bosma)
- Ajuste une erreur plpython rapportant que ses tests de régression ont encore réussi
- Fonctionne avec bison 1.875
- Bonne gestion des noms comprenant des majuscule et des minus cules dans %type de plpgsql (Neil)

- Corrige un arrêt brutal (<< core dump >>) dans pltcl lors de l'exécution d'une requête réécrite par une règle
- Réparation des dépassements d'indices de tableau (suivant le rapport de Yichen Xie)
- Réduction de MAX_TIME_PRECISION, passant de 13 à 10 dans le cas des nombres à virgule flottante
- Gère correctement la casse des noms de variables suivant la configuration de la base de données et de l'utilisateur
- Correction de l'arrêt brutal (<< coredump >>) dans RETURN NEXT de plpgsql lorsqu'un SELECT into record ne renvoie aucune ligne
- Correction d'une utilisation obsolète de pg_type.typprtlen dans l'interface client python
- Bonne gestions des secondes à fraction dans le type timestamp du pilote JDBC
- Amélioration des performances de getImportedKeys() dans JDBC
- Fait que les liens symboliques de la bibliothèque partagée fonctionnent en standard dans HPUX (Giles)
- Réparation d'un comportement d'arrondi inconsistent pour timestamp, time, interval
- Correction sur la négotiation SSL (Nathan Mueller)
- Fait que la fonctionnalité ~/.pgpass de libpq fonctionne lors de la connexion avec PQconnectDB
- Mise à jour de my2pg, ora2pg
- Mise à jour de traduction
- Ajout de conversions entre les types lo et oid dans contrib/lo
- Le code fastpath vérifie maintenant les droits d'appels de fonctions

# E.29. Sortie 7.3.1

**Date de sortie :** 2002−12−18

Cette version comporte plusieurs corrections sur la version 7.3.

## E.29.1. Migration vers la version 7.3.1

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 7.3. Néanmoins, il devrait être noté que la principale bibliothèque d'interface de PostgreSQL, libpq, a un nouveau numéro majeur de version pour cette sortie, ce qui pourrait nécessiter la recompilation de code client dans certains cas.

## E.29.2. Modifications

- Correction d'un arrêt brutal (<< core dump >>) de COPY TO lorsque le codage client/serveur ne correspond pas (Tom)
- Autorise pg_dump à fonctionner avec les serveurs pré−7.2 (Philip)
- Corrections de contrib/adddépend (Tom)
- Correction d'un problème avec la suppression de paramétrages par utilisateur ou par base de données (Tom)
- Correction de contrib/vacuumlo (Tom)
- Autorise le cryptage 'password' même lorsque pg_shadow contient des mots de passe MD5 (Bruce)
- Correction de contrib/dbmirror (Steven Singer)
- Corrections sur l'optimiseur (Tom)
- Corrections de contrib/tsearch (Teodor Sigaev, Magnus)
- Permet aux noms de locale d'être saisis avec des majuscules et minuscules mélangées (Nicolai Tufar)
- Incrémente le numéro de version majeure de la bibliothèque libpq (Bruce)

- Corrections des rapports d'erreur de pg_hba.conf (Bruce, Neil)
- Ajout de SCO Openserver 5.0.4 comme plateforme supporté (Bruce)
- Empêche EXPLAIN d'arrêter brutalement le serveur (Tom)
- Corrections sur SSL (Nathan Mueller)
- Empêche la création de colonnes composites via ALTER TABLE (Tom)

# E.30. Sortie 7.3

**Date de sortie :** 2002−11−27

## E.30.1. Aperçu

Modifications majeures dans cette version :

Schémas
: Les schémas permettent aux utilisateurs de créer des objets dans des espaces de noms séparés, donc deux personnes ou applications peuvent avoir des tables de même nom. Il existe aussi un schéma public pour les tables partagées. La création de table/index peut être restreinte en supprimant les droits du le schéma public.

Suppression de colonne
: PostgreSQL supporte maintenant la fonctionnalité `ALTER TABLE ... DROP COLUMN`.

Fonctions de tables
: Les fonctions renvoyant plusieurs lignes et/ou plusieurs colonnes sont maintenant plus facile à utiliser qu'auparavant. Vous pouvez appeler une << fonction de table >> dans la clause `FROM` d'un `SELECT`, traitant sa sortie comme une table. De plus, les fonctions PL/pgSQL peuvent maintenant renvoyer des ensembles.

Requêtes préparées
: PostgreSQL supporte maintenant les requêtes préparées pour des performances améliorées.

Récupération des dépendances
: PostgreSQL enregistre maintenant les dépendances de objets, ce qui permet des améliorations dans de nombreux domaines. Les instructions `DROP` prennent maintenant soit `CASCADE` soit `RESTRICT` pour contrôler si les objets dépendant sont aussi supprimés.

Droits
: Les fonctions et langages de procédures disposent maintenant de droits et les fonctions peuvent être définies pour être exécutées suivant les droits de leur créateur.

Internationalisation
: Le support du multioctet et des locales est maintenant activé en permanence.

Traces
: Une variété d'options de traces a été améliorée.

Interfaces
: Un grand nombre d'interfaces a été déplacé dans http://gborg.postgresql.org où ils peuvent être développés et mis à disposition indépendamment.

Fonctions/Identifieurs
: Par défaut, les fonctions prennent maintenant jusqu'à 32 paramètres et les identifieurs peuvent être d'une taille maximum de 63 octets. De plus, `OPAQUE` est maintenant obsolète : il existe des << pseudo−types de données >> pour représenter chacune des significations précédentes de `OPAQUE` dans l'argument et les types de résultat de la fonction.

## E.30.2. Migration vers la version 7.3

Une sauvegarde/restauration utilisant pg_dump est requise pour ceux souhaitant migrer leur données à partir d'une version précédente. S i votre application examine les catalogues systèmes, des modifications supplémentaires seront requises à cause de l'introduction des schémas dans la version 7.3 ; pour plus d'informations, voir : http://developer.postgresql.org/~momjian/upgrade_tips_7.3.

Observez les incompatibilités suivantes :

- Les clients avant la 6.3 ne sont plus supportés.
- `pg_hba.conf` a maintenant une colonne pour le nom de l'utilisateur et pour des fonctionnalités supplémentaires. Les fichiers existants doivent être ajustés.
- Quelques paramètres de trace de `postgresql.conf` ont été renommés.
- `LIMIT #,#` a été désactivé ; utilisez `LIMIT # OFFSET #`.
- Les instructions `INSERT` avec des listes de colonnes doivent spécifier une valeur pour chaque colonne spécifiée. Par exemple, `INSERT INTO tab (col1, col2) VALUES ('val1')` est maintenant non valide. Il est toujours permis de fournir moins de colonnes qu'attendu si l'`INSERT` n'a pas de liste de colonnes.
- Les colonnes de type `serial` ne sont plus automatiquement `UNIQUE` ; du coup, un index ne sera pas automatiquement créé.
- Une commande `SET` à l'intérieur d'une transaction est maintenant annulée
- `COPY` ne considère plus que les colonnes de fin manquantes doivent être NULL. Toutes les colonnes doivent être spécifiées. (Néanmoins, vous pouvez avoir un effet similaire en spécifiant une liste de colonnes dans la commande `COPY`.)
- Le type de données `timestamp` est maintenant équivalent à `timestamp without time zone`, au lieu de `timestamp with time zone`.
- Les bases de données, avant la version 7.3 et chargées dans cette version, n'auront pas les dépendances du nouvel objet pour les colonnes de type `serial`, les contraintes uniques et les clés étrangères. Voir le répertoire `contrib/adddépend/` pour une description détaillée et un script qui ajoute ces dépendances.
- Une chaîne vide (`''`) n'est plus permise comme entrée dans un champ d'entier. Auparavant, c'était silencieusement interprété comme 0.

---

## E.30.3. Modifications

### E.30.3.1. Opérations du serveur

- Ajout de la vue pg_locks pour afficher les verrous (Neil)
- Corrections de sécurité pour l'allocation en mémoire de la négotiation des mots de passe (Neil)
- Suppression du support pour le protocole version 0 FE/BE (PostgreSQL 6.2 et précédent) (Tom)
- Réservation des quelques derniers emplacements du serveur pour les superutilisateurs, ajout du paramètre superuser_reserved_connections pour contrôler ceci (Nigel J. Andrews)

---

### E.30.3.2. Performance

- Amélioration du lancement en appelant localtime() seulement une fois (Tom)
- Mise en cache des informations du catalogue système dans des fichiers plats pour un lancement plus rapide (Tom)
- Amélioration du cache pour les informations des index (Tom)
- Améliorations de l'optimiseur (Tom, Fernando Nasser)

- Les caches catalog stockent maintenant les recherches échoués (Tom)
- Amélioration des fonctions de découpage (Neil)
- Amélioration de la performance de la mise en jeton des requêtes et de la gestion du réseau (Peter)
- Amélioration de la rapidité pour la restauration des objets larges (Mario Weilguni)
- Marquage des entrées d'index expirés lors de la première re chercher, sauvant ainsi quelques récupérations d'en−tête (Tom)
- Évite un remplissage de bitmap NULL excessif (Manfred Koizar)
- Ajout du qsort() pour Solaris sous licence BSD, pour les performances (Bruce)
- Réduction du coût par ligne des quatre octets (Manfred Koizar)
- Correction d'un bogue dans l'optimiseur GEQO (Neil Conway)
- Utiliser WITHOUT OID fait réellement gagner quatre octets par ligne (Manfred Koizar)
- Ajout de la variable default_statistics_target pour spécifier des cibles ANALYZE (Neil)
- Utilisation du tampon cache local pour les tables temporaires pour qu'il n'y ait pas de surcharge des WAL (Tom)
- Amélioration des performances de la carte des espaces libres sur de grandes tables (Stephen Marshall, Tom)
- Amélioration de la concurrence des écritures dans les WAL (Tom)

---

#### E.30.3.3. Droits

- Ajout de droits sur les fonctions et langages de procédure (Peter)
- Ajout de OWNER à CREATE DATABASE pour que les superutilisateurs puissent créer des bases de données pour les utilisateurs non privilégiés (Gavin Sherry, Tom)
- Ajout des nouveaux bits de droits EXECUTE et USAGE (Tom)
- Ajout de SET SESSION AUTHORIZATION DEFAULT et RESET SESSION AUTHORIZATION (Tom)
- Permet que les fonctions soient exécutées avec les droits du propriétaire de la fonction (Peter)

---

#### E.30.3.4. Configuration du serveur

- Les messages de traces du serveur sont maintenant marqués avec LOG, et non pas DEBUG (Bruce)
- Ajout d'une colonne utilisateur à pg_hba.conf (Bruce)
- log_connections affiche deux lignes dans le journal des traces (Tom)
- Suppression de debug_level dans postgresql.conf, maintenant server_min_messages (Bruce)
- Nouvelle commande ALTER DATABASE/USER ... SET pour l'initialisation de la base de données par utilisateur (Peter)
- Nouveaux paramètres server_min_messages et client_min_messages pour contrôler quels messages sont envoyés au journal des traces du serveur ou aux applications clientes (Bruce)
- Permet à pg_hba.conf de spécifier des listes d'utilisateurs /bases de données séparées par des virgules, des noms de groupe avant le signe + en suffixe et des noms de fichiers avec un @ en préfixe (Bruce)
- Suppression de la capacité d'un deuxième fichier de mot de passe et de l'outil pg_password (Bruce)
- Ajout d'une variable db_user_namespace pour les noms utilisateurs locaux à la base de données (Bruce)
- Amélioration de SSL (Bear Giles)
- Les mots de passe sont cryptés par défaut (Bruce)
- Autorise la réinitialisation de pg_statistics par l'appel de pg_stat_reset() (Christopher)
- Ajout du paramètre log_duration (Bruce)
- Renommage de debug_print_query en log_statement (Bruce)
- Renommage de show_query_stats en show_statement_stats (Bruce)
- Ajout de param log_min_error_statement pour afficher les commandes à tracer en cas d'erreur

(Gavin)

### E.30.3.5. Requêtes

- Les curseurs ne sont pas sensibles, ce qui signifie que leur contenu ne change pas (Tom)
- Désactivation de la syntaxe LIMIT #,# ; maintenant seul LIMIT # OFFSET # est supporté (Bruce)
- Augmentation de la taille de l'identifiant à 63 (Neil, Bruce)
- Corrections d'UNION pour assembler trois ou plus de trois colonnes de taille différente (Tom)
- Ajout du mot clé DEFAULT pour INSERT, par exemple INSERT ... (..., DEFAULT, ...) (Rod)
- Les vues peuvent avoir des valeurs par défaut en utilisant ALTER COLUMN ... SET DEFAULT (Neil)
- Échec en insertion (INSERT) des des listes de colonnes ne co rrespondant pas à toutes les valeurs de colonnes, par exemple INSERT INTO tab (col1, col2) VALUES ('val1'); (Rod)
- Correction pour les alias de jointure (Tom)
- Correction des FULL OUTER JOIN (Tom)
- Amélioration des rapports d'identifiant invalide et d'emplacement (Tom, Gavin)
- Correction d'OPEN cursor(args) (Tom)
- Les 'ctid' peuvent être utilisées dans une vue et currtid(viewname) (Hiroshi)
- Correction pour CREATE TABLE AS avec UNION (Tom)
- Amélioration de la syntaxe SQL99 (Thomas)
- Ajout de la variable statement_timeout pour annuler des requêtes (Bruce)
- Autorisation des requêtes préparées avec PREPARE/EXECUTE (Neil)
- FOR UPDATE peut apparaître après LIMIT/OFFSET (Bruce)
- Ajout d'une variable autocommit (Tom, David Van Wie)

### E.30.3.6. Manipulation d'objets

- Rend optionnel les signes d'égalité dans CREATE DATABASE (Gavin Sherry)
- Fait que ALTER TABLE OWNER change aussi le propriétaire de l'index (Neil)
- Le nouveau ALTER TABLE nomtable ALTER COLUMN nom_colonne SET STORAGE contrôle le stockage de TOAST, la compression (John Gray)
- Ajout du support du schéma, CREATE/DROP SCHEMA (Tom)
- Création d'un schéma pour les tables temporaires (Tom)
- Ajout de la variable search_path pour la recherche de schémas (Tom)
- Ajout de ALTER TABLE SET/DROP NOT NULL (Christopher)
- Nouveaux niveaux de volatilité CREATE FUNCTION (Tom)
- Rend les noms de règles uniques seulement par table (Tom)
- Ajout d'une clause 'ON tablename' à DROP RULE et à COMMENT ON RULE (Tom)
- Ajout de ALTER TRIGGER RENAME (Joe)
- Nouvelles fonctions current_schema() et current_schemas() (Tom)
- Autorise les fonctions à renvoyer plusieurs lignes (fonctions de table) (Joe)
- Rend WITH optionnel dans CREATE DATABASE pour rester consistant (Bruce)
- Ajout de la conservation des dépendances d'objets (Rod, Tom)
- Ajout de RESTRICT/CASCADE aux commandes DROP (Rod)
- Ajout de ALTER TABLE DROP pour les cas sans CHECK CONSTRAINT (Rod)
- Destruction automatique de la séquence lors du DROP d'une table avec SERIAL (Rod)
- Empêche la suppression d'une colonne si celle−ci est utilisée par une clé étrangère (Rod)
- Supprime automatiquement les contraintes/fonctions lorsqu'un objet est supprimé (Rod)
- Ajout de CREATE/DROP OPERATOR CLASS (Bill Studenmund, Tom)
- Ajout de ALTER TABLE DROP COLUMN (Christopher, Tom, Hiroshi)

- Empêche les colonnes héritées d'être supprimées ou renommées (Alvaro Herrera)
- Correction des contraintes de clés étrangères pour ne pas avoir d'erreurs sur les états intermédiaires de la base de données (Stephan)
- Propagation du renommage de colonne ou de table pour les contraintes de clés étrangères
- Ajout de CREATE OR REPLACE VIEW (Gavin, Neil, Tom)
- Ajout de CREATE OR REPLACE RULE (Gavin, Neil, Tom)
- Fait que les règles s'exécutent dans l'ordre alphabétique, renvoyant des valeurs plus prévisibles (Tom)
- Les déclencheurs sont maintenant lancés dans l'ordre alphabétique (Tom)
- Ajout de /contrib/adddépend pour gérer les dépendances d'objets pre−7.3 (Rod)
- Permet une meilleure conversion lors de l'insertion et de la mise à jour des valeurs (Tom)

---

#### E.30.3.7. Commandes utilitaires

- Fait que COPY TO affiche les retours chariots et retours à la ligne avec \r et \n (Tom)
- Permet à DELIMITER dans COPY FROM d'être propre sur huit bits (Tatsuo)
- Fait que pg_dump utilise ALTER TABLE ADD PRIMARY KEY pour des raisons de performance (Neil)
- Désactivation des crochets dans les règles à plusieurs instructions (Bruce)
- Désactivation de l'appel du VACUUM à l'intérieur d'une fonction (Bruce)
- Autorise dropdb et d'autres scripts à utiliser des identifieurs avec des espaces (Bruce)
- Restreint les modifications de commentaire de la base de données à la base de données courante
- Autorise les commentaires sur les opérateurs, indépendent de la fonction sous−jacente (Rod)
- Annule les commandes SET dans les transactions annulées (Tom)
- EXPLAIN s'affiche maintenant comme une requête (Tom)
- Affichage des expressions de condition et tri des clés dans EXPLAIN (Tom)
- Ajout de 'SET LOCAL var = valeur' pour initialiser les variables pour une seule transaction (Tom)
- Autorise le lancement d'ANALYZE dans une transaction (Bruce)
- Amélioration de la syntaxe de COPY en utilisant les nouvelles clauses WITH, conservant ainsi la compatibilité ascendante (Bruce)
- Correction de pg_dump pour afficher de façon consistante les balises dans les sauvegardes non ASCII (Bruce)
- Fait que les contraintes de clés étrangères dans le fichier de sauvegarde (Rod)
- Ajout de COMMENT ON CONSTRAINT (Rod)
- Autorise COPY TO/FROM à spécifier les noms de colonnes (Brent Verner)
- Sauvegarde les contraintes UNIQUE et PRIMARY KEY comme ALTER TABLE (Rod)
- SHOW s'affiche comme le résultat d'une requête (Joe)
- Génération d'un échec sur les lignes courtes de COPY plutôt que de rajouter des NULL (Neil)
- Correction de CLUSTER pour préserver tous les attributs des tables (Alvaro Herrera)
- Nouvelle table pg_settings pour visualiser/modifier les paramétrages GUC (Joe)
- Ajout de la mise entre guillemets intelligente, amélioration sur la portabilité de la sortie de pg_dump (Peter)
- Sauvegarde les colonnes serial comme SERIAL (Tom)
- Activation du support des gros fichiers, >2 Go pour pg_dump (Peter, Philip Warner, Bruce)
- Interdit TRUNCATE sur les tables qui sont impliquées dans des contraintes référentielles (Rod)
- Fait que TRUNCATE tronque aussi automatiquement la table toast de la relation (Tom)
- Ajout de l'outil clusterdb qui groupera automatiquement une base de données entière en se basant sur les opérations CLUSTER précédentes (Alvaro Herrera)
- Revue de pg_dumpall (Peter)
- Autorise le REINDEX des tables TOAST (Tom)
- Implémentation de START TRANSACTION, suivant SQL99 (Neil)

- Correction des rares corruptions d'index lorsqu'une division de page affecte la suppression en masse (Tom)
- Correction de l'héritage sur ALTER TABLE ... ADD COLUMN (Alvaro Herrera)

### E.30.3.8. Types de données et fonctions

- Correction pour que factorial(0) renvoie 1 (Bruce)
- Améliorations des types date/time/timezone (Thomas)
- Correction pour l'extraction d'un morceau de tableau (Tom)
- Correction de extract/date_part pour rapporter les bonnes microsecondes sur des valeurs de type timestamp (Tatsuo)
- Autorise text_substr() et bytea_substr() à lire des valeurs TOAST de façon plus efficace (John Gray)
- Ajout du support du domaine (Rod)
- Fait que WITHOUT TIME ZONE est la valeur par défaut des types de données TIMESTAMP et TIME data (Thomas)
- Autorise un autre schéma de stockage des entiers sur 64 bits pour les types date/time en utilisant −−enable−integer−datetimes dans configure (Thomas)
- Fait que timezone(timestamptz) renvoie timestamp plutôt que string (Thomas)
- Autorise les secondes fractionnées dans les types date/time pour les dates datant de l'an 1 avant JC (Thomas)
- Limite les types de données timestamp à six places décimales pour la précision (Thomas)
- Modification des fonctions de conversion timezone de timetz() à timezone() (Thomas)
- Ajout des variables de configuration datestyle et timezone (Tom)
- Ajout d'OVERLAY(), qui autorise la substitution d'une sous−chaîne dans une chaîne (Thomas)
- Ajout de SIMILAR TO (Thomas, Tom)
- Ajout d'un SUBSTRING(chaîne FROM modele FOR échappement) utilisant les expressions rationnelles (Thomas)
- Ajout des fonctions LOCALTIME et LOCALTIMESTAMP (Thomas)
- Ajout des types composés nommés utilisant CREATE TYPE nom_type AS (column) (Joe)
- Autorise la définition d'un type composé dans la clause alias d'une table (Joe)
- Ajout d'une nouvelle API pour simplifier la création de fonctions de table en langage C (Joe)
- Supprime les parenthèses vides compatible ODBC des appels de fonctions SQL99 si elles ne sont pas compatibles avec le standard (Thomas)
- Le type de données macaddr accepte 12 chiffres hexadécimaux sans séparateurs (Mike Wyer)
- Ajout de CREATE/DROP CAST (Peter)
- Ajout de l'opérateur IS DISTINCT FROM (Thomas)
- Ajout de la fonction SQL99 TREAT(), synonyme de CAST() (Thomas)
- Ajout de pg_backend_pid() pour afficher le pid du serveur (Bruce)
- Ajout du prédicat IS OF / IS NOT OF (Thomas)
- Autorise les constantes de type chaîne de bits sans une longueur définie (Thomas)
- Autorise la conversion entre les entiers à huit octets et les chaînes de bit (Thomas)
- Implémente la conversion littérale en hexadécimale vers un littéral de type chaîne de bits (Thomas)
- Autorise l'apparition des fonctions de table dans la clause FROM (Joe)
- Augmente le nombre maximum de paramètres de fonctions à 32 (Bruce)
- Ne crée plus automatiquement un index pour les colonnes SERIAL (Tom)
- Ajout de current_database() (Rod)
- Correction de cash_words() pour ne pas avoir de dépassement de tampons (Tom)
- Ajout des fonctions replace(), split_part(), to_hex() (Joe)
- Correction de LIKE pour bytea en argument du côté droit (Joe)
- Empêche les arrêt brutaux causés par SELECT cash_out(2) (Tom)
- Correction de to_char(1,'FM999.99') pour qu'elle renvoie un point (Karel)

- Correction des fonctions trigger/type/language renvoyant OPAQUE pour qu'elles renvoient un type correct (Tom)

#### E.30.3.9. Internationalisation

- Ajout de codages supplémentaires : Koréen (JOHAB), Thai (WIN874), Vietnamien (TCVN), Arabe (WIN1256), Chinois Simplifié (GBK), Koréen (UHC) (Eiji Tokuya)
- Activation par défaut du support de la locale (Peter)
- Ajout des variables locale (Peter)
- Échappement des octets >= 0x7f pour le multioctet dans PQescapeBytea/PQunescapeBytea (Tatsuo)
- Ajout de la prise en compte de la locale dans les classes de caractères des expressions rationnelles
- Activation par défaut du support multioctets (Tatsuo)
- Ajout du support multioctets GB18030 (Bill Huang)
- Ajout de CREATE/DROP CONVERSION, permettant les locales chargeables (Tatsuo, Kaori)
- Ajout de la table pg_conversion (Tatsuo)
- Ajout de la fonction SQL99 CONVERT() (Tatsuo)
- pg_dumpall, pg_controldata et pg_resetxlog sont maintenant au courant de la langue nationale (Peter)
- Nouvelles traductions et mises à jour

#### E.30.3.10. Langages du côté serveur

- Autorise les fonctions récursives SQL (Peter)
- Modification de la construction de PL/Tcl pour utiliser le compilateur configuré et Makefile.shlib (Peter)
- Refonte de la variable FOUND PL/pgSQL pour être encore plus compatible avec Oracle (Neil, Tom)
- Autorise PL/pgSQL à gérer les identifieurs entre guillemets (Tom)
- Autorisation des fonctions PL/pgSQL renvoyant des ensembles de résultats (Neil)
- PL/pgSQL connait les schémas (Joe)
- Suppression de quelques pertes mémoires (Nigel J. Andrews, Tom)

#### E.30.3.11. psql

- Ne met plus en minuscule les noms de bases de données indiqués avec \connect pour une compatibilité avec la 7.2.0 (Tom)
- Ajout de \timing pour chronométrer les requêtes utilisateur (Greg Sabino Mullane)
- Fait que \d affiche des informations sur les index (Greg Sabino Mullane)
- Le nouveau \dD affiche les domaines (Jonathan Eisler)
- Permet à psql d'afficher les règles sur les vues (Paul ?)
- Correction pour la substitution des variables avec psql (Tom)
- Autorise l'affichage de la structure des tables temporaires avec \d (Tom)
- Autorise l'affichage les clés étrangères avec \d (Rod)
- Correction de \? pour honorer l'afficheur page par page avec \pset (Bruce)
- Fait que psql rapporte son numéro de version au lancement (Tom)
- Autorise la spécification des noms de colonnes avec \copy (Tom)

#### E.30.3.12. libpq

- Ajout de ~/.pgpass pour stocker la combinaison hôte /mot de passe de l'utilisateur (Alvaro Herrera)
- Ajout de la fonction PQunescapeBytea() à libpq (Patrick Welche)
- Correction pour l'envoi de grandes requêtes sur des connexions non bloquantes (Bernhard Herzog)

- Correction pour l'utilisation de timers par libpq sur Win9X (David Ford)
- Autorisation pour que le notify de libpq gère des serveurs avec des identifieurs de longueurs différentes (Tom)
- Ajout de PQescapeString() et PQescapeBytea() dans libpq pour Windows (Bruce)
- Correction pour SSL avec les connexions non bloquantes (Jack Bates)
- Ajout d'un paramètre de déconnexion automatique au bout d'un certain temps sans activité (Denis A Ustimenko)

#### E.30.3.13. JDBC

- JDBC se compile avec le JDK 1.4 (Dave)
- Ajout du support JDBC 3 (Barry)
- Autorise JDBC à configurer le niveau des traces en ajoutant ?loglevel=X à l'URL de connexion (Barry)
- Ajout du message Driver.info() qui affiche le numéro de version (Barry)
- Ajout des ensembles de résultats en lecture/écriture (Raghu Nidagal, Dave)
- Ajout du support pour les instructions appellables (Paul Bethe)
- Ajout de la possibilité d'annulation d'une requête
- Ajout du rafraîchissement d'une ligne (Dave)
- Correction du cryptage MD5 pour gérer les serveurs multibyte (Jun Kawai)
- Ajout du support pour les instructions préparées (Barry)

#### E.30.3.14. Interfaces diverses

- Correction d'un bogue ECPG concernant les nombres en octal avec les guillemets simples (Michael)
- Déplacement de src/interfaces/libpgeasy dans http://gborg.postgresql.org (Marc, Bruce)
- Amélioration de l'interface Python (Elliot Lee, Andrew Johnson, Greg Copeland)
- Ajout de l'événement de fermeture de la connexion dans libpgtcl (Gerhard Hintermayer)
- Déplacement de src/interfaces/libpq++ dans http://gborg.postgresql.org (Marc, Bruce)
- Déplacement de src/interfaces/odbc dans http://gborg.postgresql.org (Marc)
- Déplacement de src/interfaces/libpgeasy dans http://gborg.postgresql.org (Marc, Bruce)
- Déplacement de src/interfaces/perl5 dans http://gborg.postgresql.org (Marc, Bruce)
- Suppression de src/bin/pgaccess des sources, maintenant sur http://www.pgaccess.org (Bruce)
- Ajout de la commande pg_on_connection_loss à libpgtcl (Gerhard Hintermayer, Tom)

#### E.30.3.15. Code source

- Correction pour un make parallèle (Peter)
- Corrections spécifiques pour AIX lors de l'édition des liens avec Tcl (Andreas Zeugswetter)
- Permet à PL/Perl d'être construit avec Cygwin (Jason Tishler)
- Amélioration des compilations MIPS (Peter, Oliver Elphick)
- Requiert Autoconf version 2.53 (Peter)
- Requiert readline et zlib par défaut dans configure (Peter)
- Permet l'utilisation de << Intimate Shared Memory >> (ISM) par Solaris pour des raisons de performances (Scott Brunza, P.J. Josh Rovero)
- Activation permanente de syslog dans la compilation, suppression de l'option −−enable−syslog (Tatsuo)
- Activation permanente du multi−octets dans la compilation, suppression de l'option −−enable−multibyte (Tatsuo)

- Activation permanente de la locale dans la compilation, suppression de l'option −−enable−locale (Peter)
- Correction pour la création des DLL Win9x (Magnus Naeslund)
- Correction pour l'utilisation de link() par le code WAL sur Windows, BeOS (Jason Tishler)
- Ajout de sys/types.h à c.h, suppression à partir des fichiers principaux (Peter, Bruce)
- Correction d'un blocage sur les machines SMP sous AIX (Tomoyuki Niijima)
- Correction d'un autre blocage SMP sur AIX (Tomoyuki Niijima)
- Correction de la gestion de dates avant 1970 sur les nouvelles bibliothèques glibc (Tom)
- Correction du verrour sur les PowerPC SMP (Tom)
- Empêche gcc −ffast−math d'être utilisé (Peter, Tom)
- Bison >= 1.50 est maintenant requis pour les constructions développeur
- Le support Kerberos 5 se construit maintenant avec Heimdal (Peter)
- Ajout de l'appendice dans le guide utilisateur qui liste toutes les fonctionnalités SQL (Thomas)
- Amélioration de l'édition des liens des modules chargeables pour utiliser RTLD_NOW (Tom)
- Nouveaux niveaux d'erreurs WARNING, INFO, LOG, DEBUG[1−5] (Bruce)
- Le nouveau répertoire src/port contient les fonctions remplacées de la libc (Peter, Bruce)
- Nouveau catalogue système pg_namespace pour les schémas (Tom)
- Ajout de pg_class.relnamespace pour les schémas (Tom)
- Ajout de pg_type.typnamespace pour les schémas (Tom)
- Ajout de pg_proc.pronamespace pour les schémas (Tom)
- Restructuration des agrégats pour disposer des entrées de pg_proc (Tom)
- Les relations système ont maintenant leur propre espace de noms, le test pg_* n'est plus requis (Fernando Nasser)
- Renommage des noms d'index TOAST pour être *_index au lieu de *_idx (Neil)
- Ajout des espaces de noms pour les opérateurs et les classes d'opérateurs (Tom)
- Ajout de vérifications supplémentaires pour le fichier de contrôle du serveur (Thomas)
- Nouvelle FAQ polonaise (Marcin Mazurek)
- Ajout du support des sémaphores Posix (Tom)
- Document nécessaire pour reindex (Bruce)
- Renommage de quelques identifieurs internes pour simplifier la compilation Windows (Jan, Katherine Ward)
- Ajout de documentation sur l'espace de disque (Bruce)
- Suppression de KSQO de GUC (Bruce)
- Correction d'une perte mémoire dans rtree (Kenneth Been)
- Modification de quelques messages d'erreur pour la cohérence (Bruce)
- Suppression des colonnes de tables système non utilisées (Peter)
- Colonnes système NOT NULL lorsqu'appropriées (Tom)
- Nettoyage de l'utilisation de sprintf à la place de snprintf() (Neil, Jukka Holappa)
- Suppression d'OPAQUE et création de sous−types spécifiques (Tom)
- Nettoyages de la gestion interne des tableaux (Joe, Tom)
- Interdiction de pg_atoi('') (Bruce)
- Suppression du paramètres wal_files car les fichiers WAL sont maintenant recyclés (Bruce)
- Ajout des numéros de version dans les pages d'en−tête (Tom)

#### E.30.3.16. Contrib

- Autorise les tableaux d'inet dans /contrib/array (Neil)
- Corrections pour GiST (Teodor Sigaev, Neil)
- Mise à jour de /contrib/mysql
- Ajout de /contrib/dbsize pour afficher la taille des tables sans vacuum (Peter)
- Ajout de /contrib/intagg, routines d'agrégat des entiers (mlw)

- Amélioration de /contrib/oid2name (Neil, Bruce)
- Amélioration de /contrib/tsearch (Oleg, Teodor Sigaev)
- Nettoyage de /contrib/rserver (Alexey V. Borzov)
- Mise à jour de l'outil /contrib/oracle conversion (Gilles Darold)
- Mise à jour de /contrib/dblink (Joe)
- Amélioration des options supportées par /contrib/vacuumlo (Mario Weilguni)
- Amélioration de /contrib/intarray (Oleg, Teodor Sigaev, Andrey Oktyabrski)
- Ajout de l'outil /contrib/reindexdb (Shaun Thomas)
- Ajout de l'indexage dans /contrib/isbn_issn (Dan Weston)
- Ajout de /contrib/dbmirror (Steven Singer)
- Amélioration de /contrib/pgbench (Neil)
- Ajout des exemples de fonctions de table /contrib/tablefunc (Joe)
- Ajout du type de données pour les structures en arbre /contrib/ltree (Teodor Sigaev, Oleg Bartunov)
- Déplacement de /contrib/pg_controldata, pg_resetxlog dans le répertoire principal des sources (Bruce)
- Correction à /contrib/cube (Bruno Wolff)
- Amélioration de /contrib/fulltextindex (Christopher)

# E.31. Version 7.2.8

**Date de sortie :** 2005−05−09

Cette version contient quelques corrections sur la 7.2.7, dont un correctif sur un trou de sécurité.

## E.31.1. Migration vers la version 7.2.8

Une sauvegarde/restauration n'est pas requise pour ceux utilisant la version 7.2.X.

## E.31.2. Modifications

- Réparation d'une ancienne condition qui permettait à une transaction d'être vue comme validée pour certains buts (par exemple SELECT FOR UPDATE) légèrement plus tôt que dans les autres buts

  Ceci est un bogue extrêmement sérieux car il pourrait amener à des incohérences apparents des données et visibles brièvement par les applications.
- Réparation d'une condition entre extension de relation et VACUUM

  Théoriquement, ceci pourrait avoir causé des pertes d'une page de données tout juste insérées, bien que la probabilité d'un tel scénario semble extrêmement faible. Il n'existe pas de cas connus où cela a provoqué plus qu'un échec d'Assert.
- Correction des comparaisons des valeurs `TIME WITH TIME ZONE`

  Le code de comparaison était mauvais dans le cas où le commutateur de configuration `--enable-integer-datetimes` avait été utilisé. NOTE : si vous avez un index sur une colonne `TIME WITH TIME ZONE`, il sera nécessaire de le `REINDEX`er après avoir installé cette mise à jour parce que ce correctif corrige l'ordre de tri des valeurs de colonnes.
- Correction de `EXTRACT(EPOCH)` pour les valeurs de type `TIME WITH TIME ZONE`
- Vérification supplémentaires de dépassement de tampon dans plpgsql (Neil)
- Correction de pg_dump pour qu'il sauvegarde correctement les noms des index et déclencheurs contenant `%` (Neil)

- Empêche `to_char(interval)` d'arrêter brutalement le serveur (dump core) pour les formats relatifs au mois
- Correction de `contrib/pgcrypto` pour les nouvelles constructions d'OpenSSL (Marko Kreen)

# E.32. Sortie 7.2.7

**Date de sortie :** 2005−01−31

Cette version contient quelques correctifs de la version 7.2.6, certains portant sur des problèmes de sécurité.

## E.32.1. Migration vers la version 7.2.7

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une 7.2.X.

## E.32.2. Modifications

- Interdit `LOAD` aux utilisateurs standards

  Sur les plateformes qui exécuteront automatiquement les fonctions d'initialisation d'une bibliothèque partagée (ceci inclut au moins Windows et les Unix basés sur ELF), `LOAD` est utilisable pour faire exécuter un code arbitraire par le serveur. Merci à NGS Software pour cette information.
- Ajout du marquage STRICT nécessaire à quelques fonctions dans contrib (Kris Jurka)
- Évite un dépassement de tampon lorsque la déclaration du curseur plpgsql dispose trop de paramètres (Neil)
- Correction d'erreurs de plannification pour les jointures complètes et externes à droite

  Le résultat de la jointure était faussement supposé trié de la même façon que l'entrée gauche. Ceci pouvait délivrer une sortie mal triée à l'utilisateur mais, dans le cas de jointures d'assemblage imbriquées, pouvait donner de mauvaises réponses.
- Correction de l'affichage des intervalles négatifs pour les styles de date SQL et GERMAN

# E.33. Sortie 7.2.6

**Date de sortie :** 2004−10−22

Cette version contient quelques corrections de la version 7.2.5.

## E.33.1. Migration vers la version 7.2.6

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une 7.2.X.

## E.33.2. Modifications

- Réparation d'un échec possible pour mettre à jour les bits d'astuce sur disque

  Sous quelques rares circonstances, ceci pourrait mener à des échecs << could not access transaction status >>, qui se qualifient comme bogue pouvant entraîner des pertes de données.
- Assurance que la jointure externe hachée ne manque pas de lignes

Les jointures gauches très importantes pourraient échouer à afficher les lignes du côté gauche sans correspondance en donnant seulement la distribution droite des données.
- Interdit l'exécution de pg_ctl en tant que root

  Ceci est fait pour évacuer tout risque de problèmes de sécurité.
- Évite l'utilisation de fichiers temporaires dans `/tmp` dans `make_oidjoins_check`

  Ceci a été rapportée comme un problème de sécurité bien qu'il est de peu d'importance car il n'y a aucune raison pour que les utilisateurs autres que les développeurs utilisent ce script de toute façon.
- Mise à jour vers les nouvelles versions de Bison

# E.34. Sortie 7.2.5

**Date de sortie :** 2004−08−16

Cette version contient quelques corrections de la version 7.2.4.

## E.34.1. Migration vers la version 7.2.5

Une sauvegarde/restauration n'est pas requise pour ceux utilisant une 7.2.X.

## E.34.2. Modifications

- Réparation d'un arrêt brutal possible lors d'insertions concurrentes dans l'index B−tree

  Ce correctif s'occupe d'un cas ras dans les insertions concurrentes dans un index B−tree, pouvant résulter en un PANIC du serveur. Aucun dommage permanent ne devrait en résulter mais cela vaut néanmoins une pré−version. Ce bogue n'existe pas dans les versions antérieures 7.1.
- Correction d'un cas particulier dans la recherche en parallèle dans des arbres balancés (btree) avec une page racine divisée
- Correction d'un dépassement de tampon dans `to_ascii` (Guido Notari)
- Correction d'un arrêt brutal lors de la détection d'un verrou mortel sur les machines dont le type char est non signé
- Correction d'un échec pour répondre à `pg_ctl stop -m fast` après le lancement d'un Async_NotifyHandler
- Correction de pertes mémoire dans pg_dump
- Évite les conflits avec la définition système de la fonction ou macro `isblank()`

# E.35. Sortie 7.2.4

**Date de sortie :** 2003−01−30

Cette version comporte plusieurs corrections sur la version 7.2.3, incluant des correctifs pour empêcher des possibles pertes de données.

### E.35.1. Migration vers la version 7.2.4

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 7.2.*.

### E.35.2. Modifications

- Correction de quelques cas supplémentaires d'erreurs << No one parent tuple was found >> lors du VACUUM
- Empêche VACUUM d'être appelé à l'intérieur d'une fonction (Bruce)
- S'assure que les mises à jour de pg_clog sont synchronisées sur disque avant de marquer le point de vérification comme terminé
- Empêche le dépassement d'entier lors de grosses jointures de découpage
- Fait que les commandes GROUP fonctionnent lorsque pg_group.grolist est assez large pour être mis en << TOAST >>
- Correction d'erreurs dans les tables datetime ; quelques noms de fuseau horaire n'étaient pas reconnus
- Correction de quelques dépassements d'entier dans circle_poly(), path_encode(), path_add() (Neil)
- Réparation d'erreurs de logique dans lseg_eq(), lseg_ne(), lseg_center()

## E.36. Sortie 7.2.3

**Date de sortie :** 2002−10−01

Cette version comporte plusieurs corrections sur la version 7.2.2, incluant des correctifs pour empêcher des possibles pertes de données.

### E.36.1. Migration vers la version 7.2.3

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 7.2.*.

### E.36.2. Modifications

- Empêche la perte possible de traces de transaction compressées (Tom)
- Empêche les non superutilisateurs d'augmenter les informations de vacuum les plus récentes (Tom)
- Gestion des dates antérieures à 1970 dans les nouvelles versions de glibc (Tom)
- Correction de blocage possible lors de l'arrêt du serveur
- Empêche des blocages de spinlock sur les machines SMP PPC (Tomoyuki Niijima)
- Correction de pg_dump pour sauvegarder correctement le FULL JOIN USING (Tom)

## E.37. Sortie 7.2.2

**Date de sortie :** 2002−08−23

Cette version comporte plusieurs corrections sur la version 7.2.1.

### E.37.1. Migration vers la version 7.2.2

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant version 7.2.*.

### E.37.2. Modifications

- Autorise EXECUTE de "CREATE TABLE AS ... SELECT" en PL/pgSQL (Tom)
- Correction pour la réinitialisation de l'id des traces des transactions compressées (Tom)
- Correction de PQescapeBytea/PQunescapeBytea pour qu'ils gèrent les octets supérieurs à 0x7f (Tatsuo)
- Correction pour que l'arrêt brutal de psql et pg_dump lors de l'appel d'options longues inexistantes (Tatsuo)
- Correction d'un arrêt brutal lors de l'appel d'opérateurs de géométriques (Tom)
- Autorise l'appel de OPEN cursor(args) (Tom)
- Correction de la construction de l'index rtree_gist (Teodor)
- Correction de la sauvegarde des agrégats définis par l'utilisateur (Tom)
- Corrections pour contrib/intarray (Oleg)
- Corrections des requêtes complexes UNION/EXCEPT/INTERSECT utilisant des parenthèses (Tom)
- Correction de pg_convert (Tatsuo)
- Correction pour l'arrêt brutal avec des chaînes DATA longues (Thomas, Neil)
- Corrections pour repeat(), lpad(), rpad() et les chaînes longues (Neil)

## E.38. Sortie 7.2.1

**Date de sortie :** 2002−03−21

Cette version comporte plusieurs corrections sur la version 7.2.

### E.38.1. Migration vers la version 7.2.1

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 7.2.

### E.38.2. Modifications

- S'assure que les compteurs de séquence ne reviennent pas en arrière après un arrêt brutal (Tom)
- Correction des touches pour la conversion kanji de pgaccess (Tatsuo)
- Amélioration de l'optimiseur (Tom)
- Amélioration des entrées/sorties (Tom)
- Nouvelle FAQ russe
- Correction de compilation pour le AuthBlockSig manquant (Heiko)
- Fuseaux horaires supplémentaires et correction des fuseaux horaires (Thomas)
- Permet à \connect dans psql de gérer les noms de base de données et d'utilisateurs possédant des minuscules et des majuscules (Tom)
- Renvoie le bon OID sur la complétion des commandes même a vec des règles ON INSERT (Tom)
- Autorise COPY FROM à utiliser des délimiteurs sur 8 bits (Tatsuo)
- Correction d'un bogue dans extract/date_part pour les millisecondes/microsecondes (Tatsuo)
- Amélioration de la gestion de plusieurs UNION avec différentes longueurs (Tom)
- Amélioration de contrib/btree_gist (Teodor Sigaev)

- Amélioration du dictionnaire de contrib/tsearch, voir README.tsearch pour une étape d'installation supplémentaire (Thomas T. Thai, Teodor Sigaev)
- Correction sur la gestion des indices de tableau (Tom)
- Autorise le lancement de EXECUTE sur "CREATE TABLE AS ... SELECT" en PL/pgSQL (Tom)

# E.39. Sortie 7.2

**Date de sortie :** 2002−02−04

## E.39.1. Aperçu

Cette version améliore PostgreSQL pour une utilisation avec des applications gèrant de gros volumes.

Modifications majeures dans cette version :

VACUUM
: L'opération VACUUM ne verrouille plus les tables, permettant du coup un accès à l'utilisateur normal pendant le vacuum. Une nouvelle commande `VACUUM FULL` fait le vacuum ancien style en verrouillant la table et en diminuant la copie disque de la table.

Transactions
: Il n'y a plus de problèmes avec les installations dépassant quatre milliards de transactions.

OID
: Les OID sont maintenant optionnelles. Les utilisateurs peuvent maintenant créer des tables sans OID dans le cas où l'utilisation d'OID est excessif.

Optimiseur
: Le système calcule maintenant des statistiques de colonnes lors d'`ANALYZE`, permettant de meilleurs choix pour l'optimiseur.

Sécurité
: Une nouvelle option de cryptage MD5 permet un stockage plus sécurisé et le transfert des mots de passe. Une nouvelle option d'authentification par socket de domaine Unix est disponible sur les systèmes Linux et BSD.

Statistiques
: Les administrateurs peuvent utiliser le nouveau module de statistiques d'accès aux tables pour obtenir des informations détaillées sur l'utilisation des tables et index.

Internationalisation
: Les messages des programmes et des bibliothèques peuvent maintenant être affichés dans plusieurs langues.

## E.39.2. Migration vers la version 7.2

Une sauvegarde/restauration avec `pg_dump` est requis pour ceux qui souhaitent migrer leur données d'une version précédente.

Observez les incompatibilités suivantes :

- La sémantique de la commande `VACUUM` a changé dans cette version. Vous devez mettre à jour vos procédures de maintenance en adéquation.
- Dans cette version, les comparaisons utilisant `= NULL` retourneront toujours faux (ou NULL, plus précisément). Les anciennes versions transformaient automatiquement cette syntaxe en `IS NULL`.

L'ancien comportement peut être réactivé en utilisant un paramètre de `postgresql.conf`.
- Les fichiers de configuration `pg_hba.conf` et `pg_ident.conf` sont maintenant rechargés après réception d'un signal SIGHUP, pas avec chaque connexion.
- La fonction `octet_length()` renvoie maintenant la taille des données non compressées.
- La valeur de type date/time `'current'` n'est plus disponible. Vous aurez besoin de réécrire vos applications.
- Les fonctions `timestamp()`, `time()` et `interval()` ne sont plus disponibles. À la place, utilisez `timestamp 'string'` ou `CAST`.

La syntaxe `SELECT ... LIMIT #,#` sera supprimée dans la prochaine version. Vous devez modifier vos requêtes pour utiliser des clauses LIMIT et OFFSET séparées, c'est−à−dire `LIMIT 10 OFFSET 20`.

## E.39.3. Modifications

### E.39.3.1. Server Operation

- Création de fichiers temporaires dans un répertoire séparé (Bruce)
- Suppression des fichiers temporaires orphelins au lancement de postmaster (Bruce)
- Ajout d'index uniques à quelques tables système (Tom)
- Réorganisation des opérateurs de tables système (Oleg Bartunov, Teodor Sigaev, Tom)
- Renommage de pg_log en pg_clog (Tom)
- Activation de SIGTERM, SIGQUIT pour tuer les serveurs (Jan)
- Suppression de la limite du nombre de serveurs au moment de la compilation (Tom)
- Meilleure nettoyage pour les échecs de ressources sémaphores (Tatsuo, Tom)
- Permet un meilleur retour au début des ID de transaction (Tom)
- Suppression des OID pour certaines tables système (Tom)
- Suppression de la vérification d'erreur "violation de la modification des données ayant lancé le déclencheur" (Tom)
- Création du portail SPI des plans préparés/sauvegardés (Jan)
- Les fonctions de colonnes SPI fonctionnent avec les colonnes systèmes (Tom)
- Amélioration de la compression des grandes valeurs (Tom)
- Collecteur de statistiques pour les accès aux tables et index (Jan)
- Tronque les noms de séquences très longs en une valeur raisonnable (Tom)
- Mesure les temps de transaction en millisecondes (Thomas)
- Correction des parcours séquentiels de TID (Hiroshi)
- L'ID du superutilisateur est maintenant fixé à 1 (Peter E)
- Nouvelle option "reload" de pg_ctl (Tom)

### E.39.3.2. Performance

- Amélioration de l'optimiseur (Tom)
- Nouvelle statistique dans la colonne histogramme pour l'optimiseur (Tom)
- Réutilisation des fichiers de trace en pré−écriture (WAL) plutôt que de les annuler (Tom)
- Amélioration du cache (Tom)
- Amélioration de l'optimiseur avec IS NULL, IS NOT NULL (Tom)
- Amélioration du gestionnaire de verrous pour réduire le temps de conservation d'un verrou (Tom)
- Conservation des entrées relcache pour les fonctions de support d'accès par index (Tom)
- Permet une meilleure sélectivité avec NaN et les infinies dans NUMERIC (Tom)
- Amélioration des performances de R−tree (Kenneth Been)
- Division des B−tree plus efficace (Tom)

#### E.39.3.3. Droits

- Modification des droits UPDATE, DELETE pour qu'ils soient bien distincts (Peter E)
- Nouveaux droits REFERENCES, TRIGGER (Peter E)
- Autorise GRANT/REVOKE vers/à partir de plus d'un utilisateur à la fois (Peter E)
- Nouvelle fonction has_table_privilege() (Joe Conway)
- Autorise les utilisateurs standards à lancer un vacuum sur la base de données (Tom)
- Nouvelle commande SET SESSION AUTHORIZATION (Peter E)
- Correction d'un bogue dans les modifications de droits sur les tables nouvellement créées (Tom)
- Interdit l'accès de pg_statistic aux utilisateurs normaux, ajout de vues accessibles aux utilisateurs (Tom)

#### E.39.3.4. Authentification de l'utilisateur

- Lance un autre postmaster avant de réaliser l'authentification pour empêcher les blocages (Peter E)
- Ajout de l'authentification ident sur les sockets de domaine Unix pour Linux, *BSD (Helge Bahmann, Oliver Elphick, Teodor Sigaev, Bruce)
- Ajout d'une méthode d'authentification par mot de passe utilisant le cryptage MD5 (Bruce)
- Autorise le cryptage des mots de passe stockés utilisant MD5 (Bruce)
- Authentification PAM (Dominic J. Eidson)
- Chargement de pg_hba.conf et pg_ident.conf uniquement au lancement et après un SIGHUP (Bruce)

#### E.39.3.5. Configuration du serveur

- Interprétation de quelques abréviations de fuseaux horaires en tant qu'australien plutôt qu'américain du nord, configurable à l'exécution (Bruce)
- Nouveau paramètre pour initialiser le niveau d'isolation par défaut d'une transaction (Peter E)
- Nouveau paramètre pour activer la conversion de "expr = NULL" en "expr IS NULL", désactivé par défaut (Peter E)
- Nouveau paramètre pour contrôler l'utilisation de la mémoire par VACUUM (Tom)
- Nouveau paramètre pour initialiser le délai autorisé pour terminer l'authentification du client (Tom)
- Nouveau paramètre pour configurer le nombre maximum de fichiers ouverts (Tom)

#### E.39.3.6. Requêtes

- Les instructions ajoutées par des règles INSERT s'exécutent maintenant après l'INSERT (Jan)
- Empêche les noms de relation mal formés dans la liste cible (Bruce)
- Les NULL sont maintenant triés après les valeurs normales dans un ORDER BY (Tom)
- Nouveaux tests booléens IS UNKNOWN, IS NOT UNKNOWN (Tom)
- Nouveau mode de verrou SHARE UPDATE EXCLUSIVE (Tom)
- Nouvelle commande EXPLAIN ANALYZE affichant les temps d'exécution et les nombres de lignes (Martijn van Oosterhout)
- Correction d'un problème avec LIMIT et les sous−requêtes (Tom)
- Correction de LIMIT, DISTINCT ON placés dans des sous−requêtes (Tom)
- Correction d'EXCEPT/INTERSECT imbriqués (Tom)

#### E.39.3.7. Manipulation de schémas

- Correction de SERIAL pour les tables temporaires (Bruce)
- Autorise les séquences temporaires (Bruce)
- Les séquences utilisent maintenant int8 en interne (Tom)
- Le nouveau SERIAL8 crée des colonnes int8 avec des séquences, par défaut il s'agit de séquences SERIAL4 (Tom)
- Rend les OID optionnel en utilisant WITHOUT OIDS (Tom)
- Ajout de la synatxe %TYPE pour CREATE TYPE (Ian Lance Taylor)
- Ajout de ALTER TABLE / DROP CONSTRAINT pour les contraintes CHECK (Christopher Kings−Lynne)
- Nouveau CREATE OR REPLACE FUNCTION pour modifier la fonction existante (préservant la fonction OID) (Gavin Sherry)
- Ajout d'ALTER TABLE / ADD [ UNIQUE | PRIMARY ] (Christopher Kings−Lynne)
- Autorise le renommage de colonne dans les vues
- Fait que ALTER TABLE / RENAME COLUMN met à jour les noms de colonnes des index (Brent Verner)
- Correction pour ALTER TABLE / ADD CONSTRAINT ... CHECK avec les tables héritées (Stephan Szabo)
- ALTER TABLE RENAME met à jour correctement les arguments du déclencheurs de clé étrangère (Brent Verner)
- DROP AGGREGATE et COMMENT ON AGGREGATE acceptent maintenant un aggtype (Tom)
- Ajout de la conversion automatique du type de données en retour pour les fonctions SQL (Tom)
- Permet aux index GiST de gérer les NULL et les index à plusieurs clés (Oleg Bartunov, Teodor Sigaev, Tom)
- Activation des index partiels (Martijn van Oosterhout)

---

#### E.39.3.8. Commandes outils

- Ajout de RESET ALL, SHOW ALL (Marko Kreen)
- CREATE/ALTER USER/GROUP accepte les options dans n'importe quel ordre (Vince)
- Ajout de la fonctionnalité LOCK A, B, C (Neil Padgett)
- Nouvelle option ENCRYPTED/UNENCRYPTED pour CREATE/ALTER USER (Bruce)
- Le nouveau VACUUM léger ne verrouille pas la table ; l'ancienne sémantique est disponible en tant que VACUUM FULL (Tom)
- Désactivation de COPY TO/FROM sur les vues (Bruce)
- La chaîne COPY DELIMITERS doit avoir exactement un caractère (Tom)
- Le message d'avertissement du VACUUM pour le nombre de lignes d'index plus petit que l'en−tête apparaît maintenant si approprié (Martijn van Oosterhout)
- Correction des vérifications de droits pour CREATE INDEX (Tom)
- Interdit l'utilisation inappropriée de CREATE/DROP INDEX/TRIGGER/VIEW (Tom)

---

#### E.39.3.9. Types de données et fonctions

- SUM(), AVG(), COUNT() utilisent maintenant int8 en interne pour des questions de rapidité (Tom)
- Ajout de convert(), convert2() (Tatsuo)
- Nouvelle fonction bit_length() (Peter E)
- Fait que le "n" dans CHAR(n)/VARCHAR(n) représente le nombre de lettres, pas le nombre d'octets (Tatsuo)
- CHAR(), VARCHAR() rejetent maintenant les chaînes trop longues (Peter E)

- BIT VARYING rejete maintenant les chaînes de bits trop longues (Peter E)
- BIT rejete maintenant les chaînes de bits ne correspondant pas à la taille déclarée (Peter E)
- Fonctions de conversions texte pour INET, CIDR (Alex Pilosov)
- Opérateurs << et <<= indexables pour INET, CIDR (Alex Pilosov)
- Bytea \### requiert maintenant un numéro valide à trois chiffres octals
- Amélioration des comparaisons de Bytea, supporte maintenant =, <>, >, >=, < et <=
- Bytea supporte maintenant les index B−tree
- Bytea supporte maintenant LIKE, LIKE...ESCAPE, NOT LIKE, NOT LIKE...ESCAPE
- Bytea supporte maintenant la concaténation
- Nouvelles fonctions bytea : position, substring, trim, btrim et length
- Nouveau mode de fonction pour encode(), "escaped", convertit les bytea échappés de façon minimale vers/à partir du texte
- Ajout de pg_database_encoding_max_length() (Tatsuo)
- Ajout de la fonction pg_client_encoding() (Tatsuo)
- now() renvoie l'heure avec une précision à la milliseconde (Thomas)
- Nouveau type de données TIMESTAMP WITHOUT TIMEZONE (Thomas)
- Ajout de la spécification ISO date/time avec "T", yyyy−mm−ddThh:mm:ss (Thomas)
- Nouvelles fonctions de comparaisons des xid/int (Hiroshi)
- Ajout de la précision vers les types de données TIME, TIMESTAMP et INTERVAL (Thomas)
- Modification de la ligique de coercion des types pour tenter en premier lieu les fonctions compatibles binairement (Tom)
- Nouvelle fonction encode() installée par défaut (Marko Kreen)
- Amélioration des fonctions de conversion to_*() (Karel Zak)
- Optimisation de LIKE/ILIKE lors de l'utilisation de codage sur un seul octet (Tatsuo)
- Nouvelles fonctions dans contrib/pgcrypto : crypt(), hmac(), encrypt(), gen_salt() (Marko Kreen)
- Bonne description de la fonction translate() (Bruce)
- Ajout de l'argument INTERVAL pour SET TIME ZONE (Thomas)
- Ajout de la syntaxe INTERVAL YEAR TO MONTH (etc.) (Thomas)
- Optimisation des fonctions length lors de l'utilisation de codage sur un seul octet (Tatsuo)
- Corrections de path_inter, path_distance, path_length, dist_ppath pour gérer les chemins fermés (Curtis Barrett, Tom)
- octet_length(text) renvoie maintenant une longueur non compressée (Tatsuo, Bruce)
- Gestion du nom complet "July" dans les littéraux date/time (Greg Sabino Mullane)
- Quelques appels de fonction datatype() sont maintenant évalués différemment
- Ajout du support pour les spécifications d'heure julienne et ISO (Thomas)

#### E.39.3.10. Internationalisation

- Support de la langue nationale dans psql, pg_dump, libpq et le serveur (Peter E)
- Traductions des messages en chinois (simplifié, traditionnel), tchèque, français, allemand, hongrois, russe, suèdois (Peter E, Serguei A. Mokhov, Karel Zak, Weiping He, Zhenbang Wei, Kovacs Zoltan)
- Rend trim, ltrim, rtrim, btrim, lpad, rpad, translate conscient du multi−octets (Tatsuo)
- Ajout du support de LATIN5,6,7,8,9,10 (Tatsuo)
- Ajout du support de ISO 8859−5,6,7,8 (Tatsuo)
- Le bon LATIN5 signifie ISO−8859−9, non pas ISO−8859−5 (Tatsuo)
- Rend mic2ascii() conscient du non ASCII (Tatsuo)
- Rejète les séquences de caractères multi−octets invalides (Tatsuo)

#### E.39.3.11. PL/pgSQL

- Utilise maintenant les portails pour les boucles SELECT, permettant d'énormes ensembles de résultats (Jan)
- Support de CURSOR et de REFCURSOR (Jan)
- Peut maintenant renvoyer des curseurs ouverts (Jan)
- Ajout de ELSEIF (Klaus Reger)
- Amélioration des rapports d'erreurs PL/pgSQL, incluant l'emplacement de l'erreur (Tom)
- Autorisation des mots clés IS ou FOR dans la déclaration de curseurs pour compatibilité (Bruce)
- Correction de SELECT ... FOR UPDATE (Tom)
- Correction de PERFORM lorsqu'il renvoie plusieurs lignes (Tom)
- PL/pgSQL utilise le code de coercion de type du serveur (Tom)
- Correction d'une perte mémoire (Jan, Tom)
- Rend optionnel le point virgule en fin (Tom)

---

#### E.39.3.12. PL/Perl

- Nouveau PL/Perl sans confiance du serveur (Alex Pilosov)
- PL/Perl est maintenant construit sur quelques plateformes même si libperl n'est pas une bibliothèque partagée (Peter E)

---

#### E.39.3.13. PL/Tcl

- Rapporte maintenant errorInfo (Vsevolod Lobko)
- Ajout de la fonction spi_lastoid (bob@redivi.com)

---

#### E.39.3.14. PL/Python

- ... est nouveau (Andrew Bosma)

---

#### E.39.3.15. psql

- \d affiche les index dans un groupe unique et primaire (Christopher Kings−Lynne)
- Autorise les points virgules à la fin dans les commandes antislashs (Greg Sabino Mullane)
- Lecture du mot de passe à partir de /dev/tty si possible
- Force une nouvelle invite du mot de passe lors d'un changement d'utilisateur et de base de données (Tatsuo, Tom)
- Formate le bon nombre de colonnes pour Unicode (Patrice)

---

#### E.39.3.16. libpq

- Nouvelle fonction PQescapeString() pour échapper les guillemets dans les chaînes de commande (Florian Weimer)
- Nouvelle fonction PQescapeBytea() échappe les chaînes binaires à utiliser comme des littéraux de chaînes SQL

---

#### E.39.3.17. JDBC

- Renvoie l'OID d'INSERT (Ken K)
- Gestion d'un plus grand nombre de types de données (Ken K)
- Gestion des guillemets simples et des retours chariots dans les chaînes (Ken K)
- Gestion des variables NULL (Ken K)
- Correction pour la gestion des fuseaux horaires (Barry Lind)
- Amélioration du support de Druid
- Autorisation des caractères à huit bits avec un serveur non compatible avec le multi−octets (Barry Lind)
- Support des types BIT, BINARY (Ned Wolpert)
- Réduction de l'utilisation de la mémoire (Michael Stephens, Dave Cramer)
- Mise à jour de DatabaseMetaData (Peter E)
- Ajout de DatabaseMetaData.getCatalogs() (Peter E)
- Corrections du codage (Anders Bengtsson)
- Méthodes get/setCatalog (Jason Davies)
- DatabaseMetaData.getColumns() renvoie maintenant les valeurs par défaut des colonnes (Jason Davies)
- Amélioration de la performance de DatabaseMetaData.getColumns() (Jeroen van Vianen)
- Rassemblement de JDBC1 et JDBC2 (Anders Bengtsson)
- Améliorations des performances des transactions (Barry Lind)
- Corrections sur les tableaux (Greg Zoller)
- Ajout de la sérialisation
- Correction du traitement en lots (Rene Pijlman)
- Réorganisation de la méthode ExecSQL (Anders Bengtsson)
- Corrections de getColumn() (Jeroen van Vianen)
- Correction de la fonction isWriteable() (Rene Pijlman)
- Amélioration du passage des tests de conformance JDBC2 (Rene Pijlman)
- Ajout du type bytea (Barry Lind)
- Ajout de isNullable() (Rene Pijlman)
- Corrections pour la suite de tests JDBC date/time (Liam Stewart)
- Correction sur SELECT 'id' AS xxx FROM table (Dave Cramer)
- Correction de DatabaseMetaData pour afficher la bonne précision (Mark Lillywhite)
- Nouvelles clés getImported/getExported (Jason Davies)
- Support des mots de passe cryptés MD5 (Jeremy Wohl)
- Correction pour réellement utiliser le cache des types (Ned Wolpert)

---

#### E.39.3.18. ODBC

- Suppression de la limite sur la taille des requêtes (Hiroshi)
- Suppression de la limite sur la taille du champ texte (Hiroshi)
- Correction de SQLPrimaryKeys en mode multi−octets (Hiroshi)
- Autorisation des appels de procédure ODBC (Hiroshi)
- Amélioration de la gestion des booléens (Aidan Mountford)
- La plupart des options de configuration sont maintenant configurables via le DSN (Hiroshi)
- Corrections des performances du multi−octets (Hiroshi)
- Autorise l'utilisation du pilote avec iODBC ou unixODBC (Peter E)
- Support du cryptage des mots de passe MD5 (Bruce)
- Ajout de plusieurs fonctions de compatibilité dans odbc.sql (Peter E)

---

#### E.39.3.19. ECPG

- Implémentation de EXECUTE ... INTO (Christof Petig)
- Support du descripteur de plusieurs lignes (c'est−à−dire CARDINALITY) (Christof Petig)
- Correction pour les paramètres de GRANT (Lee Kindness)
- Correction d'un bogue sur INITIALLY DEFERRED
- Plusieurs corrections de bogues (Michael, Christof Petig)
- Allocation automatique pour les tableaux de variable (int *ind_p=NULL)
- Allocation automatique pour les tableaux de chaînes (char **foo_pp=NULL)
- Correction de ECPGfree_auto_mem
- Tous les noms de fonction avec un lien externe sont maintenant préfixés par ECPG
- Corrections pour les tableaux de structures (Michael)

---

#### E.39.3.20. Interfaces diverses

- Correction de fetchone() pour Python (Gerhard Haring)
- Utilisation d'UTF, Unicode dans Tcl lorsqu'approprié (Vsevolod Lobko, Reinhard Max)
- Ajout de COPY TO/FROM dans Tcl (ljb)
- Empêche la sortie des classes d'opérateurs d'index par défaut dans pg_dump (Tom)
- Correction d'une perte mémoire dans libpgeasy (Bruce)

---

#### E.39.3.21. Construction et installation

- Corrections sur configure, le chargeur dynamique et les bibliothèques partagées (Peter E)
- Corrections dans le port QNX 4 (Bernd Tegge)
- Corrections dans les ports Cygwin et Windows (Jason Tishler, Gerhard Haring, Dmitry Yurtaev, Darko Prenosil, Mikhail Terekhov)
- Corrections pour les échecs de communication par socket sous Windows (Magnus, Mikhail Terekhov)
- Correction de compilation pour Hurd (Oliver Elphick)
- Corrections sur BeOS (Cyril Velter)
- Suppression de configure −−enable−unicode−conversion, activé maintenant par le multi−octets (Tatsuo)
- Corrections pour AIX (Tatsuo, Andreas)
- Correction du make parallèle (Peter E)
- Installation des pages man du langage SQL dans des répertoires spécifiques aux OS (Peter E)
- Renommage de config.h en pg_config.h (Peter E)
- Réorganisation de l'emplacement d'installation des fichiers d'en−tête (Peter E)

---

#### E.39.3.22. Code source

- Suppression de SEP_CHAR (Bruce)
- Nouvelles accroches GUC (Tom)
- Assemblage de GUC et de la gestion de la ligne de commandes (Marko Kreen)
- Suppression de EXTEND INDEX (Martijn van Oosterhout, Tom)
- Nouvel outil pgjindent pour indenter le code java (Bruce)
- Suppression de la définition de true/false lors d'une compilation sous C++ (Leandro Fanzone, Tom)
- Corrections à pgindent (Bruce, Tom)
- Remplacement de strcasecmp() avec strcmp() lorsque cela est approprié (Peter E)
- Amélioration de la portabilité dynahash (Tom)

- Ajout de l'utilisation de 'volatile' dans les structures spinlock
- Amélioration de la logique de gestion des signaux (Tom)

#### E.39.3.23. Contrib

- Nouveau contrib/rtree_gist (Oleg Bartunov, Teodor Sigaev)
- Nouveau contrib/tsearch, indexage de textes complets (Oleg, Teodor Sigaev)
- Ajout de contrib/dblink pour l'accès distant aux bases de données (Joe Conway)
- Outil de conversion Oracle contrib/ora2pg (Gilles Darold)
- Outil de conversion XML contrib/xml (John Gray)
- Corrections sur contrib/fulltextindex (Christopher Kings−Lynne)
- Nouveau contrib/fuzzystrmatch avec levenshtein et métaphone, rassemblement de soundex (Joe Conway)
- Ajout des requêtes booléennes dans contrib/intarray, des recherches binaires, et correctifs (Oleg Bartunov)
- Nouvel outil pg_upgrade (Bruce)
- Ajout de nouvelles options à pg_resetxlog (Bruce, Tom)

## E.40. Sortie 7.1.3

**Date de sortie :** 2001−08−15

### E.40.1. Migration vers la version 7.1.3

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant 7.1.X.

### E.40.2. Modifications

```
Remove unused WAL segements of large transactions (Tom)
Multiaction rule fix (Tom)
PL/pgSQL memory allocation fix (Jan)
VACUUM buffer fix (Tom)
Regression test fixes (Tom)
pg_dump fixes for GRANT/REVOKE/comments on views, user-defined types (Tom)
Fix subselects with DISTINCT ON or LIMIT (Tom)
BeOS fix
Disable COPY TO/FROM a view (Tom)
Cygwin build (Jason Tishler)
```

## E.41. Sortie 7.1.2

**Date de sortie :** 2001−05−11

Il n'y a qu'une seule correction depuis la 7.1.1.

### E.41.1. Migration vers la version 7.1.2

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant 7.1.X.

### E.41.2. Modifications

```
Correction des SELECT qui ne renvoient aucune ligne dans PL/pgSQL
Correction pour l'arrêt brutal suite à l'utilisation d'un antislash dans psql
Correction du droit sur l'intégrité référentielles
Corrections sur l'optimiseur
Nettoyages de pg_dump
```

## E.42. Sortie 7.1.1

**Date de sortie :** 2001−05−05

Cette version comporte plusieurs corrections sur la version 7.1.

### E.42.1. Migration vers la version 7.1.1

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 7.1.

### E.42.2. Modifications

```
Correction pour l'opérateur numérique MODULO (Tom)
Corrections sur pg_dump (Philip)
pg_dump peut sauvegarder des bases de données d'une version 7.0 (Philip)
Corrections sur readline 4.2 (Peter E)
Corrections sur JOIN (Tom)
Corrections sur AIX, MSWIN, VAX, N32K (Tom)
Corrections sur Multibytes (Tom)
Corrections sur Unicode (Tatsuo)
Amélioration de l'optimiseur (Tom)
Correction pour les lignes complètes dans les fonctions (Tom)
Correction pour pg_ctl et les chaînes d'options comprenant des espaces (Peter E)
Corrections sur ODBC (Hiroshi)
EXTRACT prend maintenant prendre un argument de type chaîne (Thomas)
Corrections sur Python (Darcy)
```

## E.43. Sortie 7.1

**Date de sortie :** 2001−04−13

Cette version s'occupe principalement de la suppression des limitations qui ont existé dans le code de PostgreSQL depuis de nombreuses années.

Modifications majeures dans cette version :

Write−ahead Log (WAL)

Pour maintenant la cohérence dans le cas d'un arrêt brutal du système d'exploitation, les précédentes versions de PostgreSQL forçaient les modifications des données sur disque avant chaque validation de transaction. Avec les WAL, seul un fichier de traces doit être mis sur disque, ce qui améliore grandement les performances. Si vous utilisiez –F dans les précédentes versions pour désactiver les envois sur le disque, vous pourriez considérer l'arrêt de l'utilisation de cette option.

TOAST

TOAST – Les précédentes versions avaient une limite de longueur de la ligne, typiquement 8 Ko – 32 Ko. Cette limite rendait difficile le stockage de champs de texte longs. Avec TOAST, les longues lignes de longueur indéfinie peuvent être stockées avec de bonnes performances.

Jointures externes

Nous supportons maintenant les jointures externes. Le contournement UNION/NOT IN pour les jointures externes n'est plus nécessaire. Nous utilisons la syntaxe SQL92 des jointures externes.

Gestionnaire de fonctions

Le précédent gestionnaire de fonctions C ne gérait pas correctement les valeurs NULL, pas plus qu'il ne supportait les CPU 64 bits (Alpha). Le nouveau gestionnaire de fonctions le fait. Vous pouvez continuer à utiliser vos anciennes fonctions personnalisées mais vous pourriez avoir besoin de les réécrire dans le futur pour utiliser la nouvelle interface d'appel du gestionnaire de fonctions.

Requêtes complexes

Un grand nombre des requêtes complexes, qui n'étaient pas supportées dans les précédentes versions, fonctionnent maintenant. Beaucoup de combinaisons de vues, d'agrégats, UNION, LIMIT, de curseurs, de sous–requêtes et de tables héritées fonctionnent maintenant sans soucis. Les tables héritées sont maintenant accédées par défaut. Les sous–requêtes dans FROM sont aussi supportées.

## E.43.1. Migration vers la version 7.1

Une sauvegarde/restauration utilisant pg_dump est requis pour ceux souhaitant migrer leur données des versions précédentes.

## E.43.2. Modifications

```
Bug Fixes
---------
Many multibyte/Unicode/locale fixes (Tatsuo and others)
More reliable ALTER TABLE RENAME (Tom)
Kerberos V fixes (David Wragg)
Fix for INSERT INTO...SELECT where targetlist has subqueries (Tom)
Prompt nom_utilisateur/password on standard error (Bruce)
Large objects inv_read/inv_write fixes (Tom)
Fixes for to_char(), to_date(), to_ascii(), and to_timestamp() (Karel,
    Daniel Baldoni)
Prevent query expressions from leaking memory (Tom)
Allow UPDATE of arrays elements (Tom)
Wake up lock waiters during cancel (Hiroshi)
Fix rare cursor crash when using hash join (Tom)
Fix for DROP TABLE/INDEX in rolled-back transaction (Hiroshi)
Fix psql crash from \l+ if MULTIBYTE enabled (Peter E)
Fix truncation of rule names during CREATE VIEW (Ross Reedstrom)
Fix PL/perl (Alex Kapranoff)
Disallow LOCK on views (Mark Hollomon)
Disallow INSERT/UPDATE/DELETE on views (Mark Hollomon)
Disallow DROP RULE, CREATE INDEX, TRUNCATE on views (Mark Hollomon)
Allow PL/pgSQL accept non-ASCII identifiers (Tatsuo)
Allow views to proper handle GROUP BY, aggregates, DISTINCT (Tom)
Fix rare failure with TRUNCATE command (Tom)
Allow UNION/INTERSECT/EXCEPT to be used with ALL, subqueries, views,
```

```
    DISTINCT, ORDER BY, SELECT...INTO (Tom)
Fix parser failures during aborted transactions (Tom)
Allow temporary relations to properly clean up indexes (Bruce)
Fix VACUUM problem with moving rows in same page (Tom)
Modify pg_dump to better handle user-defined items in template1 (Philip)
Allow LIMIT in VIEW (Tom)
Require cursor FETCH to honor LIMIT (Tom)
Allow PRIMARY/FOREIGN Key définitions on inherited columns (Stephan)
Allow ORDER BY, LIMIT in subqueries (Tom)
Allow UNION in CREATE RULE (Tom)
Make ALTER/DROP TABLE rollback-able (Vadim, Tom)
Store initdb collation in pg_control so collation cannot be changed (Tom)
Fix INSERT...SELECT with rules (Tom)
Fix FOR UPDATE inside views and subselects (Tom)
Fix OVERLAPS operators conform to SQL92 spec regarding NULLs (Tom)
Fix lpad() and rpad() to handle length less than input string (Tom)
Fix use of NOTIFY in some rules (Tom)
Overhaul btree code (Tom)
Fix NOT NULL use in Pl/pgSQL variables (Tom)
Overhaul GIST code (Oleg)
Fix CLUSTER to preserve constraints and column default (Tom)
Improved deadlock detection handling (Tom)
Allow multiple SERIAL columns in a table (Tom)
Prevent occasional index corruption (Vadim)

Enhancements
------------
Add OUTER JOINs (Tom)
Function manager overhaul (Tom)
Allow ALTER TABLE RENAME on indexes (Tom)
Improve CLUSTER (Tom)
Improve ps status display for more platforms (Peter E, Marc)
Improve CREATE FUNCTION failure message (Ross)
JDBC improvements (Peter, Travis Bauer, Christopher Cain, William Webber,
    Gunnar)
Grand Unified Configuration scheme/GUC.  Many options can now be set in
    data/postgresql.conf, postmaster/postgres flags, or SET commands (Peter E)
Improved handling of file descriptor cache (Tom)
New warning code about auto-created table alias entries (Bruce)
Overhaul initdb process (Tom, Peter E)
Overhaul of inherited tables; inherited tables now accessed by default;
   new ONLY key word prevents it (Chris Bitmead, Tom)
ODBC cleanups/improvements (Nick Gorham, Stephan Szabo, Zoltan Kovacs,
    Michael Fork)
Allow renaming of temp tables (Tom)
Overhaul memory manager contexts (Tom)
pg_dumpall uses CREATE USER or CREATE GROUP rather using COPY (Peter E)
Overhaul pg_dump (Philip Warner)
Allow pg_hba.conf secondary password file to specify only nom_utilisateur (Peter E)
Allow TEMPORARY or TEMP key word when creating temporary tables (Bruce)
New memory leak checker (Karel)
New SET SESSION CHARACTERISTICS (Thomas)
Allow nested block comments (Thomas)
Add WITHOUT TIME ZONE type qualifier (Thomas)
New ALTER TABLE ADD CONSTRAINT (Stephan)
Use NUMERIC accumulators for INTEGER aggregates (Tom)
Overhaul aggregate code (Tom)
New VARIANCE and STDDEV() aggregates
Improve dépendency ordering of pg_dump (Philip)
New pg_restore command (Philip)
New pg_dump tar output option (Philip)
New pg_dump of large objects  (Philip)
```

```
New ESCAPE option to LIKE (Thomas)
New case-insensitive LIKE - ILIKE (Thomas)
Allow functional indexes to use binary-compatible type (Tom)
Allow SQL functions to be used in more contexts (Tom)
New pg_config utility (Peter E)
New PL/pgSQL EXECUTE command which allows dynamic SQL and utility statements
    (Jan)
New PL/pgSQL GET DIAGNOSTICS statement for SPI value access (Jan)
New quote_identifiers() and quote_literal() functions (Jan)
New ALTER TABLE table OWNER TO user command (Mark Hollomon)
Allow subselects in FROM, i.e. FROM (SELECT ...) [AS] alias (Tom)
Update PyGreSQL to version 3.1 (D'Arcy)
Store tables as files named by OID (Vadim)
New SQL function setval(seq,val,bool) for use in pg_dump (Philip)
Require DROP VIEW to remove views, no DROP TABLE (Mark)
Allow DROP VIEW view1, view2 (Mark)
Allow multiple objects in DROP INDEX, DROP RULE, and DROP TYPE (Tom)
Allow automatic conversion to/from Unicode (Tatsuo, Eiji)
New /contrib/pgcrypto hashing functions (Marko Kreen)
New pg_dumpall --globals-only option (Peter E)
New CHECKPOINT command for WAL which creates new WAL log file (Vadim)
New AT TIME ZONE syntax (Thomas)
Allow location of Unix domain socket to be configurable (David J. MacKenzie)
Allow postmaster to listen on a specific IP address (David J. MacKenzie)
Allow socket path name to be specified in hostname by using leading slash
    (David J. MacKenzie)
Allow CREATE DATABASE to specify template database (Tom)
New utility to convert MySQL schema dumps to SQL92 and PostgreSQL (Thomas)
New /contrib/rserv replication toolkit (Vadim)
New file format for COPY BINARY (Tom)
New /contrib/oid2name to map numeric files to table names (B Palmer)
New "idle in transaction" ps status message (Marc)
Update to pgaccess 0.98.7 (Constantin Teodorescu)
pg_ctl now defaults to -w (wait) on shutdown, new -l (log) option
Add rudimentary dépendency checking to pg_dump (Philip)

Types
-----
Fix INET/CIDR type ordering and add new functions (Tom)
Make OID behave as an unsigned type (Tom)
Allow BIGINT as synonym for INT8 (Peter E)
New int2 and int8 comparison operators (Tom)
New BIT and BIT VARYING types (Adriaan Joubert, Tom, Peter E)
CHAR() no longer faster than VARCHAR() because of TOAST (Tom)
New GIST seg/cube exemples (Gene Selkov)
Improved round(numeric) handling (Tom)
Fix CIDR output formatting (Tom)
New CIDR abbrev() function (Tom)

Performance
-----------
Write-Ahead Log (WAL) to provide crash recovery with less performance
    overhead (Vadim)
ANALYZE stage of VACUUM no longer exclusively locks table (Bruce)
Reduced file seeks (Denis Perchine)
Improve BTREE code for duplicate keys (Tom)
Store all large objects in a single table (Denis Perchine, Tom)
Improve memory allocation performance (Karel, Tom)

Source Code
-----------
New function manager call conventions (Tom)
```

```
SGI portability fixes (David Kaelbling)
New configure --enable-syslog option (Peter E)
New BSDI README (Bruce)
configure script moved to top level, not /src (Peter E)
Makefile/configuration/compilation overhaul (Peter E)
New configure --with-python option (Peter E)
Solaris cleanups (Peter E)
Overhaul /contrib Makefiles (Karel)
New OpenSSL configuration option (Magnus, Peter E)
AIX fixes (Andreas)
QNX fixes (Maurizio)
New heap_open(), heap_openr() API (Tom)
Remove colon and semi-colon operators (Thomas)
New pg_class.relkind value for views (Mark Hollomon)
Rename ichar() to chr() (Karel)
New documentation for btrim(), ascii(), chr(), repeat() (Karel)
Fixes for NT/Cygwin (Pete Forman)
AIX port fixes (Andreas)
New BeOS port (David Reid, Cyril Velter)
Add proofreader's changes to docs (Addison-Wesley, Bruce)
New Alpha spinlock code (Adriaan Joubert, Compaq)
UnixWare port overhaul (Peter E)
New Darwin/MacOS X port (Peter Bierman, Bruce Hartzler)
New FreeBSD Alpha port (Alfred)
Overhaul shared memory segments (Tom)
Add IBM S/390 support (Neale Ferguson)
Moved macmanuf to /contrib (Larry Rosenman)
Syslog improvements (Larry Rosenman)
New template0 database that contains no user additions (Tom)
New /contrib/cube and /contrib/seg GIST sample code (Gene Selkov)
Allow NetBSD's libedit instead of readline (Peter)
Improved assembly language source code format (Bruce)
New contrib/pg_logger
New --template option to createdb
New contrib/pg_control utility (Oliver)
New FreeBSD tools ipc_check, start-scripts/freebsd
```

# E.44. Sortie 7.0.3

**Date de sortie :** 2000−11−11

Cette version comporte plusieurs corrections sur la version 7.0.2.

## E.44.1. Migration vers la version 7.0.3

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant une version 7.0.*.

## E.44.2. Modifications

```
Corrections sur Jdbc (Peter)
Corrections sur les objets larges (Tom)
Correction d'un trou dans COPY WITH OIDS (Tom)
Correction du parcours d'index inverse (Tom)
Correction de SELECT ... FOR UPDATE pour qu'il vérifie les clés dupliquées
(Hiroshi)
Ajout de --enable-syslog à configure (Marc)
```

Correction de l'annulation d'une transaction à la fin du serveur dans de rares cas (Tom)
Correction sur \l+ de psql lorsque le multi-octets est activé (Tatsuo)
Autorise PL/pgSQL à accepter les identifieurs non ascii (Tatsuo)
Fait que vacuum vide toujours les tampons (Tom)
Correction pour autoriser l'annulation lors de l'attente d'un verrou (Hiroshi)
Correction d'un problème d'allocation mémoire dans le code d'authentification de l'utilisateur (Tom)
Suppression d'une utilisation boguée de int4out() (Tom)
Corrections pour de nombreuses sous-requêtes dans COALESCE ou BETWEEN (Tom)
Corrections pour les déclencheurs en échec sur un en-tête ouvert dans certains cas (Jeroen van Vianen)
Correction pour la sélectivité erronée des différences (Tom)
Correction pour l'utilisation erronée de strcmp() (Tom)
Correction du bogue où le gestionnaire de stockage accède à des éléments en dehors de la fin du fichier (Tom)
Correction pour inclure le message errno du noyau dans tous les messages elog de smgr (Tom)
Correction pour le chemin '.' ne faisant pas partie du PATH au moment de la construction (SL Baur)
Correction pour l'erreur de manque de descripteurs de fichiers (Tom)
Correction pour sauvegarder le drapeau 'iscachable' des fonctions dans pg_dump (Tom)
Correction pour les sous-requêtes dans la liste des cibles du nœud Append (Tom)
Correction pour les plans de jointures assemblées (Tom)
Correction de l'échec de TRUNCATE sur les relations avec index (Tom)
Évite le redémarrage de toute la base de données sur une erreur d'écriture (Hiroshi)
Correction de nodeMaterial pour honorer chgParam en recalculant sa sortie (Tom)
Correction du problème de VACUUM lors du déplacement de la chaîne de mise à jour des versions de lignes lorsque la source et la destination d'une version de ligne se trouve sur la même page (Tom)
Correction de  user.c CommandCounterIncrement (Tom)
Correction des problèmes de limite AM/PM dans to_char() (Karel Zak)
Correction de la gestion de l'agrégat TIME (Tom)
Correction de to_char() pour éviter la génération d'un codedump sur les entrées NULL (Tom)
Correction d'un tampon (Tom)
Correction pour les insertion/copie de chaînes multi-octets plus longues dans des types de données char() (Tatsuo)
Correction sur les arrêt brutaux du moteur, à l'annulation (Tom)

# E.45. Sortie 7.0.2

**Date de sortie :** 2000-06-05

C'est un nouveau paquetage de 7.0.1 avec une documentation ajoutée.

## E.45.1. Migration vers la version 7.0.2

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 7.*.

### E.45.2. Modifications

```
Ajout de la documentation dans l'archive tar.
```

## E.46. Sortie 7.0.1

**Date de sortie :** 2000−06−01

Ceci est une version nettoyée de la 7.0.

### E.46.1. Migration vers la version 7.0.1

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 7.0.

### E.46.2. Modifications

```
Correction de nombreux problèmes avec CLUSTER (Tom)
Autorise le fonctionnement de ALTER TABLE RENAME avec les index (Tom)
Correction de plpgsql pour gérer datetime->timestamp et timespan->interval
(Bruce)
Nouvelle option --with-setproctitle de configure pour utiliser setproctitle()
(Marc, Bruce)
Correction l'erreur de décalage d'un dans ResultSet à partir de la 6.5.3 et
ultérieur.
Corrections sur ResultSet de jdbc (Joseph Shraibman)
Configuration de l'optimiseur (Tom)
Correction de la création d'un utilisateur pour pgaccess
Correction pour les échecs d'UNLISTEN
Corrections sur IRIX (David Kaelbling)
Corrections sur QNX (Andreas Kardos)
Réduction du niveau de verrou pour COPY IN (Tom)
Modification de libpqeasy pour utiliser les paramètres de style PQconnectdb()
(Bruce)
Correction de pg_dump pour gérer les index sur les OID (Tom)
Correction d'une petite perte mémoire (Tom)
Correction de createdb/dropdb pour Solaris (Tatsuo)
Correction pour les connexions non bloquantes (Alfred Perlstein)
Correction pour une récupération sale après des échecs de RENAME TABLE (Tom)
Copie de pg_ident.conf.sample dans le répertoire /lib à l'installation (Bruce)
Ajout du support de SJIS UDC (NEC selection IBM kanji) (Eiji Tokuya)
Correction des messages syslog trop longs (Tatsuo)
Correction d'un problème avec les index trop longs entre guillemets (Tom)
Correction de ResultSet.getTimestamp() pour JDBC (Gregory Krasnow & Floyd
Marinescu)
Modifications d'ecpg (Michael)
```

## E.47. Sortie 7.0

**Date de sortie :** 2000−05−08

Cette version contient de nombreuses améliorations dans beaucoup de domaines, démontrant l'acroissement continu de PostgreSQL. Il y a plus d'améliorations et de corrections dans la 7.0 que dans n'importe quelle

version précédente. Les développeurs ont confiance dans le fait qu'il s'agit de la meilleure version à ce jour ; nous avons fait de notre mieux pour sortir des versions solides et celle−ci ne fait pas exception.

Modifications majeures dans cette version :

Clés étrangères
: Les clés étrangères sont maintenant implémentées avec l'exception des clés étrangères PARTIAL MATCH. Beaucoup d'utilisateurs ont demandé cette fonctionnalité et nous sommes heureux de leur offrir.

Revue de l'optimiseur
: En continuant sur un travail commencé un an auparavant, l'optimiseur a été amélioré, permettant la sélection de meilleurs plans de requêtes et de performance plus rapide avec une utilisation moindre de la mémoire.

Mise à jour de psql
: psql, notre terminal interactif, a été mis à jour avec une variété de nouvelles fonctionnalités. Voir la page de manuel de psql pour les détails.

Syntaxe de jointure
: La syntaxe de jointure SQL92 est maintenant supportée, bien qu'il ne s'agisse que des `INNER JOIN` actuellement. `JOIN`, `NATURAL JOIN`, `JOIN/USING` et `JOIN/ON` sont disponibles ainsi que les noms de corrélation des colonnes.

## E.47.1. Migration vers la version 7.0

Une sauvegarde/restauration utilisant pg_dump est requis pour ceux souhaitant migrer leur données des versions précédentes de PostgreSQL. Pour ceux mettant à jour à partir de 6.5.*, vous pourriez utiliser à la place pg_upgrade pour mettre à jour cette version ; néanmoins, une installation complète avec sauvegarde/restauration est toujours la méthode la plus robuste pour les mises à jour.

Les problèmes d'interface et de compatibilité à considérer pour cette nouvelle version incluent :

- Les types date/time `datetime` et `timespan` ont été remplacé par les types définis par SQL92 `timestamp` et `interval`. Bien qu'il y ait eu quelques efforts pour faciliter la transition en permettant à PostgreSQL de reconnaitre les noms de type obsolètes et de les traduire dans les noms des nouveaux types, ce mécanisme n'est pas complètement transparent pour votre application existante.
- L'optimiseur a été substantiellement amélioré dans le domaine de l'estimation du coût d'une requête. Dans certains cas, ceci résultera en des temps de requêtes moindres car l'optimiseur fera un meilleur choix de plan de requêtes. Néanmoins, sur un petit nombre de cas, impliquant habituellement des distributions pathologiques de données, vos temps de requêtes pourraient augmenter. Si vous gérez de gros volumes de données, vous pourriez vouloir vérifier vos requêtes au niveau de leur performance.
- Les interfaces JDBC et ODBC ont été mises à jour et étendues.
- La fonction de chaîne `CHAR_LENGTH` est maintenant une fonction native. Les versions précédentes traduisaient ceci en un appel à `LENGTH`, ce qui pouvait résulter en une ambiguïté avec les autres types implémentant `LENGTH` tels que les types géométriques.

## E.47.2. Modifications

```
Bug Fixes
---------
Prevent function calls exceeding maximum number of arguments (Tom)
Improve CASE construct (Tom)
```

```
Fix SELECT coalesce(f1,0) FROM int4_tbl GROUP BY f1 (Tom)
Fix SELECT sentence.words[0] FROM sentence GROUP BY sentence.words[0] (Tom)
Fix GROUP BY scan bogue (Tom)
Improvements in SQL grammar processing (Tom)
Fix for views involved in INSERT ... SELECT ... (Tom)
Fix for SELECT a/2, a/2 FROM test_missing_target GROUP BY a/2 (Tom)
Fix for subselects in INSERT ... SELECT (Tom)
Prevent INSERT ... SELECT ... ORDER BY (Tom)
Fixes for relations greater than 2GB, including vacuum
Improve propagating system table changes to other backends (Tom)
Improve propagating user table changes to other backends (Tom)
Fix handling of temp tables in complex situations (Bruce, Tom)
Allow table locking at table open, improving concurrent reliability (Tom)
Properly quote sequence names in pg_dump (Ross J. Reedstrom)
Prevent DROP DATABASE while others accessing
Prevent any rows from being returned by GROUP BY if no rows processed (Tom)
Fix SELECT COUNT(1) FROM table WHERE ...' if no rows matching WHERE (Tom)
Fix pg_upgrade so it works for MVCC (Tom)
Fix for SELECT ... WHERE x IN (SELECT ... HAVING SUM(x) > 1) (Tom)
Fix for "f1 datetime DEFAULT 'now'"  (Tom)
Fix problems with CURRENT_DATE used in DEFAULT (Tom)
Allow comment-only lines, and ;;; lines too. (Tom)
Improve recovery after failed disk writes, disk full (Hiroshi)
Fix cases where table is mentioned in FROM but not joined (Tom)
Allow HAVING clause without aggregate functions (Tom)
Fix for "--" comment and no trailing newline, as seen in perl interface
Improve pg_dump failure error reports (Bruce)
Allow sorts and hashes to exceed 2GB file sizes (Tom)
Fix for pg_dump dumping of inherited rules (Tom)
Fix for NULL handling comparisons (Tom)
Fix inconsistent state caused by failed CREATE/DROP commands (Hiroshi)
Fix for dbname with dash
Prevent DROP INDEX from interfering with other backends (Tom)
Fix file descriptor leak in verify_password()
Fix for "Unable to identify an operator =$" problem
Fix ODBC so no segfault if CommLog and Debogue enabled (Dirk Niggemann)
Fix for recursive exit call (Massimo)
Fix for extra-long timezones (Jeroen van Vianen)
Make pg_dump preserve primary key information (Peter E)
Prevent databases with single quotes (Peter E)
Prevent DROP DATABASE inside  transaction (Peter E)
ecpg memory leak fixes (Stephen Birch)
Fix for SELECT NULL::text, SELECT int4fac(NULL) and SELECT 2 + (NULL) (Tom)
Y2K timestamp fix (Massimo)
Fix for VACUUM 'HEAP_MOVED_IN was not expected' errors (Tom)
Fix for views with tables/columns containing spaces  (Tom)
Prevent privileges on indexes (Peter E)
Fix for spinlock stuck problem when error is generated (Hiroshi)
Fix ipcclean on Linux
Fix handling of NULL constraint conditions (Tom)
Fix memory leak in odbc driver (Nick Gorham)
Fix for privilege check on UNION tables (Tom)
Fix to allow SELECT 'a' LIKE 'a' (Tom)
Fix for SELECT 1 + NULL (Tom)
Fixes to CHAR
Fix log() on numeric type (Tom)
Deprecate ':' and ';' operators
Allow vacuum of temporary tables
Disallow inherited columns with the same name as new columns
Recover or force failure when disk space is exhausted (Hiroshi)
Fix INSERT INTO ... SELECT with AS columns matching result columns
Fix INSERT ... SELECT ... GROUP BY groups by target columns not source columns
```

```
(Tom)
Fix CREATE TABLE test (a char(5) DEFAULT text '', b int4) with INSERT (Tom)
Fix UNION with LIMIT
Fix CREATE TABLE x AS SELECT 1 UNION SELECT 2
Fix CREATE TABLE test(col char(2) DEFAULT user)
Fix mismatched types in CREATE TABLE ... DEFAULT
Fix SELECT * FROM pg_class where oid in (0,-1)
Fix SELECT COUNT('asdf') FROM pg_class WHERE oid=12
Prevent user who can create databases can modifying pg_database table (Peter E)
Fix btree to give a useful elog when key > 1/2 (page - overhead) (Tom)
Fix INSERT of 0.0 into DECIMAL(4,4) field (Tom)

Enhancements
------------
New CLI interface include file sqlcli.h, based on SQL3/SQL98
Remove all limits on query length, row length limit still exists (Tom)
Update jdbc protocol to 2.0 (Jens Glaser <jens@jens.de>)
Add TRUNCATE command to quickly truncate relation (Mike Mascari)
Fix to give super user and createdb user proper update catalog rights (Peter E)
Allow ecpg bool variables to have NULL values (Christof)
Issue ecpg error if NULL value for variable with no NULL indicator (Christof)
Allow ^C to cancel COPY command (Massimo)
Add SET FSYNC and SHOW PG_OPTIONS commands(Massimo)
Function name overloading for dynamically-loaded C functions (Frankpitt)
Add CmdTuples() to libpq++(Vince)
New CREATE CONSTRAINT TRIGGER and SET CONSTRAINTS commands(Jan)
Allow CREATE FUNCTION/WITH clause to be used for all language types
configure --enable-debug adds -g (Peter E)
configure --disable-debogue removes -g (Peter E)
Allow more complex default expressions (Tom)
First real FOREIGN KEY constraint trigger functionality (Jan)
Add FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)
Add FOREIGN KEY ... MATCH <unspecified> referential actions (Don Baccus)
Allow WHERE restriction on ctid (physical heap location) (Hiroshi)
Move pginterface from contrib to interface directory, rename to pgeasy (Bruce)
Change pgeasy connectdb() parameter ordering (Bruce)
Require SELECT DISTINCT target list to have all ORDER BY columns (Tom)
Add Oracle's COMMENT ON command (Mike Mascari <mascarim@yahoo.com>)
libpq's PQsetNoticeProcessor function now returns previous hook(Peter E)
Prevent PQsetNoticeProcessor from being set to NULL (Peter E)
Make USING in COPY optional (Bruce)
Allow subselects in the target list (Tom)
Allow subselects on the left side of comparison operators (Tom)
New parallel regression test (Jan)
Change backend-side COPY to write files with permissions 644 not 666 (Tom)
Force permissions on PGDATA directory to be secure, even if it exists (Tom)
Added psql LASTOID variable to return last inserted oid (Peter E)
Allow concurrent vacuum and remove pg_vlock vacuum lock file (Tom)
Add privilege check for vacuum (Peter E)
New libpq functions to allow asynchronous connections: PQconnectStart(),
   PQconnectPoll(), PQresetStart(), PQresetPoll(), PQsetenvStart(),
   PQsetenvPoll(), PQsetenvAbort (Ewan Mellor)
New libpq PQsetenv() function (Ewan Mellor)
create/alter user extension (Peter E)
New postmaster.pid and postmaster.opts under $PGDATA (Tatsuo)
New scripts for create/drop user/db (Peter E)
Major psql overhaul (Peter E)
Add const to libpq interface (Peter E)
New libpq function PQoidValue (Peter E)
Show specific non-aggregate causing problem with GROUP BY (Tom)
Make changes to pg_shadow recreate pg_pwd file (Peter E)
Add aggregate(DISTINCT ...) (Tom)
```

```
Allow flag to control COPY input/output of NULLs (Peter E)
Make postgres user have a password by default (Peter E)
Add CREATE/ALTER/DROP GROUP (Peter E)
All administration scripts now support --long options (Peter E, Karel)
Vacuumdb script now supports --all option (Peter E)
ecpg new portable FETCH syntax
Add ecpg EXEC SQL IFDEF, EXEC SQL IFNDEF, EXEC SQL ELSE, EXEC SQL ELIF
        and EXEC SQL ENDIF directives
Add pg_ctl script to control backend start-up (Tatsuo)
Add postmaster.opts.default file to store start-up flags (Tatsuo)
Allow --with-mb=SQL_ASCII
Increase maximum number of index keys to 16 (Bruce)
Increase maximum number of function arguments to 16 (Bruce)
Allow configuration of maximum number of index keys and arguments (Bruce)
Allow unprivileged users to change their passwords (Peter E)
Password authentication enabled; required for new users (Peter E)
Disallow dropping a user who owns a database (Peter E)
Change initdb option --with-mb to --enable-multibyte
Add option for initdb to prompts for superuser password (Peter E)
Allow complex type casts like col::numeric(9,2) and col::int2::float8 (Tom)
Updated user interfaces on initdb, initlocation, pg_dump, ipcclean (Peter E)
New pg_char_to_encoding() and pg_encoding_to_char() functions (Tatsuo)
libpq non-blocking mode (Alfred Perlstein)
Improve conversion of types in casts that don't specify a length
New plperl internal programming language (Mark Hollomon)
Allow COPY IN to read file that do not end with a newline (Tom)
Indicate when long identifiers are truncated (Tom)
Allow aggregates to use type equivalency (Peter E)
Add Oracle's to_char(), to_date(), to_datetime(), to_timestamp(), to_number()
        conversion functions (Karel Zak <zakkr@zf.jcu.cz>)
Add SELECT DISTINCT ON (expr [, expr ...]) targetlist ... (Tom)
Check to be sure ORDER BY is compatible with the DISTINCT operation (Tom)
Add NUMERIC and int8 types to ODBC
Improve EXPLAIN results for Append, Group, Agg, Unique (Tom)
Add ALTER TABLE ... ADD FOREIGN KEY (Stephan Szabo)
Allow SELECT .. FOR UPDATE in PL/pgSQL (Hiroshi)
Enable backward sequential scan even after reaching EOF (Hiroshi)
Add btree indexing of boolean values, >= and <= (Don Baccus)
Print current line number when COPY FROM fails (Massimo)
Recognize POSIX time zone e.g. "PST+8" and "GMT-8" (Thomas)
Add DEC as synonym for DECIMAL (Thomas)
Add SESSION_USER as SQL92 key word, same as CURRENT_USER (Thomas)
Implement SQL92 column aliases (aka correlation names) (Thomas)
Implement SQL92 join syntax (Thomas)
Make INTERVAL reserved word allowed as a column identifier (Thomas)
Implement REINDEX command (Hiroshi)
Accept ALL in aggregate function SUM(ALL col) (Tom)
Prevent GROUP BY from using column aliases (Tom)
New psql \encoding option (Tatsuo)
Allow PQrequestCancel() to terminate when in waiting-for-lock state (Hiroshi)
Allow negation of a negative number in all cases
Add ecpg descriptors (Christof, Michael)
Allow CREATE VIEW v AS SELECT f1::char(8) FROM tbl
Allow casts with length, like foo::char(8)
New libpq functions PQsetClientEncoding(), PQclientEncoding() (Tatsuo)
Add support for SJIS user defined characters (Tatsuo)
Larger views/rules supported
Make libpq's PQconndefaults() thread-safe (Tom)
Disable // as comment to be ANSI conforming, should use -- (Tom)
Allow column aliases on views CREATE VIEW name (collist)
Fixes for views with subqueries (Tom)
Allow UPDATE table SET fld = (SELECT ...) (Tom)
```

```
SET command options no longer require quotes
Update pgaccess to 0.98.6
New SET SEED command
New pg_options.sample file
New SET FSYNC command (Massimo)
Allow pg_descriptions when creating tables
Allow pg_descriptions when creating types, columns, and functions
Allow psql \copy to allow delimiters (Peter E)
Allow psql to print NULLs as distinct from "" [NULL] (Peter E)

Types
-----
Many array fixes (Tom)
Allow bare column names to be subscripted as arrays (Tom)
Improve type casting of int and float constants (Tom)
Cleanups for int8 inputs, range checking, and type conversion (Tom)
Fix for SELECT timespan('21:11:26'::time) (Tom)
netmask('x.x.x.x/0') is 255.255.255.255 instead of 0.0.0.0 (Oleg Sharoiko)
Add btree index on NUMERIC (Jan)
Perl fix for large objects containing NUL characters (Douglas Thomson)
ODBC fix for for large objects (free)
Fix indexing of cidr data type
Fix for Ethernet MAC addresses (macaddr type) comparisons
Fix for date/time types when overflows happened in computations (Tom)
Allow array on int8 (Peter E)
Fix for rounding/overflow of NUMERIC type, like NUMERIC(4,4) (Tom)
Allow NUMERIC arrays
Fix bogues in NUMERIC ceil() and floor() functions (Tom)
Make char_length()/octet_length including trailing blanks (Tom)
Made abstime/reltime use int4 instead of time_t (Peter E)
New lztext data type for compressed text fields
Revise code to handle coercion of int and float constants (Tom)
Start at new code to implement a BIT and BIT VARYING type (Adriaan Joubert)
NUMERIC now accepts scientific notation (Tom)
NUMERIC to int4 rounds (Tom)
Convert float4/8 to NUMERIC properly (Tom)
Allow type conversion with NUMERIC (Thomas)
Make ISO date style (2000-02-16 09:33) the default (Thomas)
Add NATIONAL CHAR [ VARYING ] (Thomas)
Allow NUMERIC round and trunc to accept negative scales (Tom)
New TIME WITH TIME ZONE type (Thomas)
Add MAX()/MIN() on time type (Thomas)
Add abs(), mod(), fac() for int8 (Thomas)
Rename functions to round(), sqrt(), cbrt(), pow() for float8 (Thomas)
Add transcendental math functions (e.g. sin(), acos()) for float8 (Thomas)
Add exp() and ln() for NUMERIC type
Rename NUMERIC power() to pow() (Thomas)
Improved TRANSLATE() function (Edwin Ramirez, Tom)
Allow X=-Y operators  (Tom)
Allow SELECT float8(COUNT(*))/(SELECT COUNT(*) FROM t) FROM t GROUP BY f1; (Tom)
Allow LOCALE to use indexes in regular expression searches (Tom)
Allow creation of functional indexes to use default types

Performance
-----------
Prevent exponential space consumption with many AND's and OR's (Tom)
Collect attribute selectivity values for system columns (Tom)
Reduce memory usage of aggregates (Tom)
Fix for LIKE optimization to use indexes with multibyte encodings (Tom)
Fix r-tree index optimizer selectivity (Thomas)
Improve optimizer selectivity computations and functions (Tom)
Optimize btree searching for cases where many equal keys exist (Tom)
```

Enable fast LIKE index processing only if index present (Tom)
Re-use free space on index pages with duplicates (Tom)
Improve hash join processing (Tom)
Prevent descending sort if result is already sorted(Hiroshi)
Allow commuting of index scan query qualifications (Tom)
Prefer index scans in cases where ORDER BY/GROUP BY is required (Tom)
Allocate large memory requests in fix-sized chunks for performance (Tom)
Fix vacuum's performance by reducing memory allocation requests (Tom)
Implement constant-expression simplification (Bernard Frankpitt, Tom)
Use secondary columns to be used to determine start of index scan (Hiroshi)
Prevent quadruple use of disk space when doing internal sorting (Tom)
Faster sorting by calling fewer functions (Tom)
Create system indexes to match all system caches (Bruce, Hiroshi)
Make system caches use system indexes (Bruce)
Make all system indexes unique (Bruce)
Improve pg_statistics management for VACUUM speed improvement (Tom)
Flush backend cache less frequently (Tom, Hiroshi)
COPY now reuses previous memory allocation, improving performance (Tom)
Improve optimization cost estimation (Tom)
Improve optimizer estimate of range queries x > lowbound AND x < highbound
(Tom)
Use DNF instead of CNF where appropriate (Tom, Taral)
Further cleanup for OR-of-AND WHERE-clauses (Tom)
Make use of index in OR clauses (x = 1 AND y = 2) OR (x = 2 AND y = 4) (Tom)
Smarter optimizer computations for random index page access (Tom)
New SET variable to control optimizer costs (Tom)
Optimizer queries based on LIMIT, OFFSET, and EXISTS qualifications (Tom)
Reduce optimizer internal housekeeping of join paths for speedup (Tom)
Major subquery speedup (Tom)
Fewer fsync writes when fsync is not disabled (Tom)
Improved LIKE optimizer estimates (Tom)
Prevent fsync in SELECT-only queries (Vadim)
Make index creation use psort code, because it is now faster (Tom)
Allow creation of sort temp tables > 1 Gig

Source Tree Changes
-------------------
Fix for linux PPC compile
New generic expression-tree-walker subroutine (Tom)
Change form() to varargform() to prevent portability problems
Improved range checking for large integers on Alphas
Clean up #include in /include directory (Bruce)
Add scripts for checking includes (Bruce)
Remove un-needed #include's from *.c files (Bruce)
Change #include's to use <> and "" as appropriate (Bruce)
Enable Windows compilation of libpq
Alpha spinlock fix from Uncle George <gatgul@voicenet.com>
Overhaul of optimizer data structures (Tom)
Fix to cygipc library (Yutaka Tanida)
Allow pgsql to work on newer Cygwin snapshots (Dan)
New catalog version number (Tom)
Add Linux ARM
Rename heap_replace to heap_update
Update for QNX (Dr. Andreas Kardos)
New platform-specific regression handling (Tom)
Rename oid8 -> oidvector and int28 -> int2vector (Bruce)
Included all yacc and lex files into the distribution (Peter E.)
Remove lextest, no longer needed (Peter E)
Fix for libpq and psql on Windows (Magnus)
Internally change datetime and timespan into timestamp and interval (Thomas)
Fix for plpgsql on BSD/OS
Add SQL_ASCII test case to the regression test (Tatsuo)

```
configure --with-mb now deprecated (Tatsuo)
NT fixes
NetBSD fixes (Johnny C. Lam <lamj@stat.cmu.edu>)
Fixes for Alpha compiles
New multibyte encodings
```

# E.48. Sortie 6.5.3

**Date de sortie :** 1999−10−13

This is basically a cleanup release for 6.5.2. We have added a new PgAccess that was missing in 6.5.2, and installed an NT−specific fix.

## E.48.1. Migration vers la version 6.5.3

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 6.5.*.

## E.48.2. Modifications

```
Updated version of pgaccess 0.98
NT-specific patch
Fix dumping rules on inherited tables
```

# E.49. Sortie 6.5.2

**Date de sortie :** 1999−09−15

This is basically a cleanup release for 6.5.1. We have fixed a variety of problems reported by 6.5.1 users.

## E.49.1. Migration vers la version 6.5.2

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 6.5.*.

## E.49.2. Modifications

```
subselect+CASE fixes(Tom)
Add SHLIB_LINK setting for solaris_i386 and solaris_sparc ports(Daren Sefcik)
Fixes for CASE in WHERE join clauses(Tom)
Fix BTScan abort(Tom)
Repair the check for redundant UNIQUE and PRIMARY KEY indexes(Thomas)
Improve it so that it checks for multicolumn constraints(Thomas)
Fix for Windows making problem with MB enabled(Hiroki Kataoka)
Allow BSD yacc and bison to compile pl code(Bruce)
Fix SET NAMES working
int8 fixes(Thomas)
Fix vacuum's memory consumption(Hiroshi,Tatsuo)
Reduce the total memory consumption of vacuum(Tom)
Fix for timestamp(datetime)
Rule deparsing boguefixes(Tom)
Fix quoting problems in mkMakefile.tcldefs.sh.in and
```

```
mkMakefile.tkdefs.sh.in(Tom)
This is to re-use space on index pages freed by vacuum(Vadim)
document -x for pg_dump(Bruce)
Fix for unary operators in rule deparser(Tom)
Comment out FileUnlink of excess segments during mdtruncate()(Tom)
IRIX linking fix from Yu Cao >yucao@falcon.kla-tencor.com<
Repair logic error in LIKE: should not return LIKE_ABORT
   when reach end of pattern before end of text(Tom)
Repair incorrect cleanup of heap memory allocation during transaction abort(Tom)
Updated version of pgaccess 0.98
```

# E.50. Sortie 6.5.1

**Date de sortie :** 1999−07−15

This is basically a cleanup release for 6.5. We have fixed a variety of problems reported by 6.5 users.

## E.50.1. Migration vers la version 6.5.1

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 6.5.

## E.50.2. Modifications

```
Add NT README file
Portability fixes for linux_ppc, IRIX, linux_alpha, OpenBSD, alpha
Remove QUERY_LIMIT, use SELECT...LIMIT
Fix for EXPLAIN on inheritance(Tom)
Patch to allow vacuum on multisegment tables(Hiroshi)
R-Tree optimizer selectivity fix(Tom)
ACL file descriptor leak fix(Atsushi Ogawa)
New expresssion subtree code(Tom)
Avoid disk writes for read-only transactions(Vadim)
Fix for removal of temp tables if last transaction was aborted(Bruce)
Fix to prevent too large row from being created(Bruce)
plpgsql fixes
Allow port numbers 32k - 64k(Bruce)
Add ^ precidence(Bruce)
Rename sort files called pg_temp to pg_sorttemp(Bruce)
Fix for microseconds in time values(Tom)
Tutorial source cleanup
New linux_m68k port
Fix for sorting of NULL's in some cases(Tom)
Shared library dépendencies fixed (Tom)
Fixed glitches affecting GROUP BY in subselects(Tom)
Fix some compiler warnings (Tomoaki Nishiyama)
Add Win1250 (Czech) support (Pavel Behal)
```

# E.51. Sortie 6.5

**Date de sortie :** 1999−06−09

This release marks a major step in the development team's mastery of the source code we inherited from Berkeley. You will see we are now easily adding major features, thanks to the increasing size and experience

of our world−wide development team.

Here is a brief summary of the more notable changes:

Multiversion concurrency control(MVCC)
: This removes our old table−level locking, and replaces it with a locking system that is superior to most commercial database systems. In a traditional system, each row that is modified is locked until committed, preventing reads by other users. MVCC uses the natural multiversion nature of PostgreSQL to allow readers to continue reading consistent data during writer activity. Writers continue to use the compact pg_log transaction system. This is all performed without having to allocate a lock for every row like traditional database systems. So, basically, we no longer are restricted by simple table−level locking; we have quelquechose better than row−level locking.

Hot backups from pg_dump
: pg_dump takes advantage of the new MVCC features to give a consistent database dump/backup while the database stays online and available for queries.

Numeric data type
: We now have a true numeric data type, with user−specified precision.

Temporary tables
: Temporary tables are guaranteed to have unique names within a database session, and are destroyed on session exit.

New SQL features
: We now have CASE, INTERSECT, and EXCEPT statement support. We have new LIMIT/OFFSET, SET TRANSACTION ISOLATION LEVEL, SELECT ... FOR UPDATE, and an improved LOCK TABLE command.

Speedups
: We continue to speed up PostgreSQL, thanks to the variety of talents within our team. We have sped up memory allocation, optimization, table joins, and row transfer routines.

Ports
: We continue to expand our port list, this time including Windows NT/ix86 and NetBSD/arm32.

Interfaces
: Most interfaces have new versions, and existing functionality has been improved.

Documentation
: New and updated material is present throughout the documentation. New FAQs have been contributed for SGI and AIX platforms. The *Tutorial* has introductory information on SQL from Stefan Simkovics. For the *User's Guide*, there are reference pages covering the postmaster and more utility programs, and a new appendix contains details on date/time behavior. The *Administrator's Guide* has a new chapter on troubleshooting from Tom Lane. And the *Programmer's Guide* has a description of query processing, also from Stefan, and details on obtaining the PostgreSQL source tree via anonymous CVS and CVSup.

## E.51.1. Migration vers la version 6.5

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release of PostgreSQL. pg_upgrade can *not* be used to upgrade to this release because the on−disk structure of the tables has changed compared to previous releases.

The new Multiversion Concurrency Control (MVCC) features can give somewhat different behaviors in multiuser environments. *Read and understand the following section to ensure that your existing applications will give you the behavior you need.*

#### E.51.1.1. Multiversion Concurrency Control

Because readers in 6.5 don't lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another. In other words, if a row is returned by `SELECT` it doesn't mean that this row really exists at the time it is returned (i.e. sometime after the statement or transaction began) nor that the row is protected from being deleted or updated by concurrent transactions before the current transaction does a commit or rollback.

To ensure the actual existence of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE` or an appropriate `LOCK TABLE` statement. This should be taken into account when porting applications from previous releases of PostgreSQL and other environments.

Keep the above in mind if you are using `contrib/refint.*` triggers for referential integrity. Additional techniques are required now. One way is to use `LOCK parent_table IN SHARE ROW EXCLUSIVE MODE` command if a transaction is going to update/delete a primary key and use `LOCK parent_table IN SHARE MODE` command if a transaction is going to update/insert a foreign key.

> **Note :** Note that if you run a transaction in SERIALIZABLE mode then you must execute the `LOCK` commands above before exécution of any DML statement (`SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO`) in the transaction.

These inconveniences will disappear in the future when the ability to read dirty (uncommitted) data (regardless of isolation level) and true referential integrity will be implemented.

## E.51.2. Modifications

```
Bug Fixes
---------
Fix text<->float8 and text<->float4 conversion functions(Thomas)
Fix for creating tables with mixed-case constraints(Billy)
Change exp()/pow() behavior to generate error on underflow/overflow(Jan)
Fix bogue in pg_dump -z
Memory overrun cleanups(Tatsuo)
Fix for lo_import crash(Tatsuo)
Adjust handling of data type names to suppress double quotes(Thomas)
Use type coercion for matching columns and DEFAULT(Thomas)
Fix deadlock so it only checks once after one second of sleep(Bruce)
Fixes for aggregates and PL/pgsql(Hiroshi)
Fix for subquery crash(Vadim)
Fix for libpq function PQfnumber and case-insensitive names(Bahman Rafatjoo)
Fix for large object write-in-middle, no extra block, memory consumption(Tatsuo)
Fix for pg_dump -d or -D and  quote special characters in INSERT
Repair serious problems with dynahash(Tom)
Fix INET/CIDR portability problems
Fix problem with selectivity error in ALTER TABLE ADD COLUMN(Bruce)
Fix executor so mergejoin of different column types works(Tom)
Fix for Alpha OR selectivity bogue
Fix OR index selectivity problem(Bruce)
Fix so \d shows proper length for char()/varchar()(Ryan)
Fix tutorial code(Clark)
Improve destroyuser checking(Oliver)
Fix for Kerberos(Rodney McDuff)
Fix for dropping database while dirty buffers(Bruce)
Fix so sequence nextval() can be case-sensitive(Bruce)
Fix !!= operator
Drop buffers before destroying database files(Bruce)
```

```
Fix case where executor evaluates functions twice(Tatsuo)
Allow sequence nextval actions to be case-sensitive(Bruce)
Fix optimizer indexing not working for negative numbers(Bruce)
Fix for memory leak in executor with fjIsNull
Fix for aggregate memory leaks(Erik Riedel)
Allow user name containing a dash to grant privileges
Cleanup of NULL in inet types
Clean up system table bogues(Tom)
Fix problems of PAGER and \? command(Masaaki Sakaida)
Reduce default multisegment file size limit to 1GB(Peter)
Fix for dumping of CREATE OPERATOR(Tom)
Fix for backward scanning of cursors(Hiroshi Inoue)
Fix for COPY FROM STDIN when using \i(Tom)
Fix for subselect is compared inside an expression(Jan)
Fix handling of error reporting while returning rows(Tom)
Fix problems with reference to array types(Tom,Jan)
Prevent UPDATE SET oid(Jan)
Fix pg_dump so -t option can handle case-sensitive tablenames
Fixes for GROUP BY in special cases(Tom, Jan)
Fix for memory leak in failed queries(Tom)
DEFAULT now supports mixed-case identifiers(Tom)
Fix for multisegment uses of DROP/RENAME table, indexes(Ole Gjerde)
Disable use of pg_dump with both -o and -d options(Bruce)
Allow pg_dump to properly dump group privileges(Bruce)
Fix GROUP BY in INSERT INTO table SELECT * FROM table2(Jan)
Fix for computations in views(Jan)
Fix for aggregates on array indexes(Tom)
Fix for DEFAULT handles single quotes in value requiring too many quotes
Fix security problem with non-super users importing/exporting large objects(Tom)
Rollback of transaction that creates table cleaned up properly(Tom)
Fix to allow long table and column names to generate proper serial names(Tom)

Enhancements
------------
Add "vacuumdb" utility
Speed up libpq by allocating memory better(Tom)
EXPLAIN all indexes used(Tom)
Implement CASE, COALESCE, NULLIF  expression(Thomas)
New pg_dump table output format(Constantin)
Add string min()/max() functions(Thomas)
Extend new type coercion techniques to aggregates(Thomas)
New moddatetime contrib(Terry)
Update to pgaccess 0.96(Constantin)
Add routines for single-byte "char" type(Thomas)
Improved substr() function(Thomas)
Improved multibyte handling(Tatsuo)
Multiversion concurrency control/MVCC(Vadim)
New Serialized mode(Vadim)
Fix for tables over 2gigs(Peter)
New SET TRANSACTION ISOLATION LEVEL(Vadim)
New LOCK TABLE IN ... MODE(Vadim)
Update ODBC driver(Byron)
New NUMERIC data type(Jan)
New SELECT FOR UPDATE(Vadim)
Handle "NaN" and "Infinity" for input values(Jan)
Improved date/year handling(Thomas)
Improved handling of backend connections(Magnus)
New options ELOG_TIMESTAMPS and USE_SYSLOG options for log files(Massimo)
New TCL_ARRAYS option(Massimo)
New INTERSECT and EXCEPT(Stefan)
New pg_index.indisprimary for primary key tracking(D'Arcy)
New pg_dump option to allow dropping of tables before creation(Brook)
```

```
Speedup of row output routines(Tom)
New READ COMMITTED isolation level(Vadim)
New TEMP tables/indexes(Bruce)
Prevent sorting if result is already sorted(Jan)
New memory allocation optimization(Jan)
Allow psql to do \p\g(Bruce)
Allow multiple rule actions(Jan)
Added LIMIT/OFFSET functionality(Jan)
Improve optimizer when joining a large number of tables(Bruce)
New intro to SQL from S. Simkovics' Master's Thesis (Stefan, Thomas)
New intro to backend processing from S. Simkovics' Master's Thesis (Stefan)
Improved int8 support(Ryan Bradetich, Thomas, Tom)
New routines to convert between int8 and text/varchar types(Thomas)
New bushy plans, where meta-tables are joined(Bruce)
Enable right-hand queries by default(Bruce)
Allow reliable maximum number of backends to be set at configure time
      (--with-maxbackends and postmaster switch (-N backends))(Tom)
GEQO default now 10 tables because of optimizer speedups(Tom)
Allow NULL=Var for MS-SQL portability(Michael, Bruce)
Modify contrib check_primary_key() so either "automatic" or "dépendent"(Anand)
Allow psql \d on a view show query(Ryan)
Speedup for LIKE(Bruce)
Ecpg fixes/features, see src/interfaces/ecpg/ChangeLog file(Michael)
JDBC fixes/features, see src/interfaces/jdbc/CHANGELOG(Peter)
Make % operator have precedence like /(Bruce)
Add new postgres -O option to allow system table structure changes(Bruce)
Update contrib/pginterface/findoidjoins script(Tom)
Major speedup in vacuum of deleted rows with indexes(Vadim)
Allow non-SQL functions to run different versions based on arguments(Tom)
Add -E option that shows actual queries sent by \dt and friends(Masaaki Sakaida)
Add version number in start-up banners for psql(Masaaki Sakaida)
New contrib/vacuumlo removes large objects not referenced(Peter)
New initialization for table sizes so non-vacuumed tables perform better(Tom)
Improve error messages when a connection is rejected(Tom)
Support for arrays of char() and varchar() fields(Massimo)
Overhaul of hash code to increase reliability and performance(Tom)
Update to PyGreSQL 2.4(D'Arcy)
Changed debogue options so -d4 and -d5 produce different node displays(Jan)
New pg_options: pretty_plan, pretty_parse, pretty_rewritten(Jan)
Better optimization statistics for system table access(Tom)
Better handling of non-default block sizes(Massimo)
Improve GEQO optimizer memory consumption(Tom)
UNION now suppports ORDER BY of columns not in target list(Jan)
Major libpq++ improvements(Vince Vielhaber)
pg_dump now uses -z(ACL's) as default(Bruce)
backend cache, memory speedups(Tom)
have pg_dump do everything in one snapshot transaction(Vadim)
fix for large object memory leakage, fix for pg_dumping(Tom)
INET type now respects netmask for comparisons
Make VACUUM ANALYZE only use a readlock(Vadim)
Allow VIEWs on UNIONS(Jan)
pg_dump now can generate consistent snapshots on active databases(Vadim)

Source Tree Changes
-------------------
Improve port matching(Tom)
Portability fixes for SunOS
Add Windows NT backend port and enable dynamic loading(Magnus and Daniel Horak)
New port to Cobalt Qube(Mips) running Linux(Tatsuo)
Port to NetBSD/m68k(Mr. Mutsuki Nakajima)
Port to NetBSD/sun3(Mr. Mutsuki Nakajima)
Port to NetBSD/macppc(Toshimi Aoki)
```

```
Fix for tcl/tk configuration(Vince)
Removed CURRENT key word for rule queries(Jan)
NT dynamic loading now works(Daniel Horak)
Add ARM32 support(Andrew McMurry)
Better support for HP-UX 11 and UnixWare
Improve file handling to be more uniform, prevent file descriptor leak(Tom)
New install commands for plpgsql(Jan)
```

# E.52. Sortie 6.4.2

**Date de sortie :** 1998-12-20

The 6.4.1 release was improperly packaged. This also has one additional bogue fix.

## E.52.1. Migration vers la version 6.4.2

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 6.4.*.

## E.52.2. Modifications

```
Fix for datetime constant problem on some platforms(Thomas)
```

# E.53. Sortie 6.4.1

**Date de sortie :** 1998-12-18

This is basically a cleanup release for 6.4. We have fixed a variety of problems reported by 6.4 users.

## E.53.1. Migration vers la version 6.4.1

Une sauvegarde/restauration n'est *pas* nécessaires pour ceux utilisant la version 6.4.

## E.53.2. Modifications

```
Add pg_dump -N flag to force double quotes around identifiers.  This is
        the default(Thomas)
Fix for NOT in where clause causing crash(Bruce)
EXPLAIN VERBOSE coredump fix(Vadim)
Fix shared-library problems on Linux
Fix test for table existence to allow mixed-case and whitespace in
        the table name(Thomas)
Fix a couple of pg_dump bogues
Configure matches template/.similar entries better(Tom)
Change builtin function names from SPI_* to spi_*
OR WHERE clause fix(Vadim)
Fixes for mixed-case table names(Billy)
contrib/linux/postgres.init.csh/sh fix(Thomas)
libpq memory overrun fix
SunOS fixes(Tom)
Change exp() behavior to generate error on underflow(Thomas)
pg_dump fixes for memory leak, inheritance constraints, layout change
update pgaccess to 0.93
```

```
Fix prototype for 64-bit platforms
Multibyte fixes(Tatsuo)
New ecpg man page
Fix memory overruns(Tatsuo)
Fix for lo_import() crash(Bruce)
Better search for install program(Tom)
Timezone fixes(Tom)
HP-UX fixes(Tom)
Use implicit type coercion for matching DEFAULT values(Thomas)
Add routines to help with single-byte (internal) character type(Thomas)
Compilation of libpq for Windows fixes(Magnus)
Upgrade to PyGreSQL 2.2(D'Arcy)
```

# E.54. Sortie 6.4

**Date de sortie :** 1998−10−30

There are *many* new features and improvements in this release. Thanks to our developers and maintainers, nearly every aspect of the system has received some attention since the previous release. Here is a brief, incomplete summary:

- Views and rules are now functional thanks to extensive new code in the rewrite rules system from Jan Wieck. He also wrote a chapter on it for the *Programmer's Guide*.
- Jan also contributed a second procedural language, PL/pgSQL, to go with the original PL/pgTCL procedural language he contributed last release.
- We have optional multiple−byte character set support from Tatsuo Ishii to complement our existing locale support.
- Client/server communications has been cleaned up, with better support for asynchronous messages and interrupts thanks to Tom Lane.
- The parser will now perform automatic type coercion to match arguments to available operators and functions, and to match columns and expressions with target columns. This uses a generic mechanism which supports the type extensibility features of PostgreSQL. There is a new chapter in the *User's Guide* which covers this topic.
- Three new data types have been added. Two types, `inet` and `cidr`, support various forms of IP network, subnet, and machine addressing. There is now an 8−byte integer type available on some platforms. See the chapter on data types in the *User's Guide* for details. A fourth type, `serial`, is now supported by the parser as an amalgam of the `int4` type, a sequence, and a unique index.
- Several more SQL92−compatible syntax features have been added, including `INSERT DEFAULT VALUES`
- The automatic configuration and installation system has received some attention, and should be more robust for more platforms than it has ever been.

## E.54.1. Migration vers la version 6.4

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of PostgreSQL.

## E.54.2. Modifications

```
Bug Fixes
---------
Fix for a tiny memory leak in PQsetdb/PQfinish(Bryan)
```

```
Remove char2-16 data types, use char/varchar(Darren)
Pqfn not handles a NOTICE message(Anders)
Reduced busywaiting overhead for spinlocks with many backends (dg)
Stuck spinlock detection (dg)
Fix up "ISO-style" timespan decoding and encoding(Thomas)
Fix problem with table drop after rollback of transaction(Vadim)
Change error message and remove non-functional update message(Vadim)
Fix for COPY array checking
Fix for SELECT 1 UNION SELECT NULL
Fix for buffer leaks in large object calls(Pascal)
Change owner from oid to int4 type(Bruce)
Fix a bogue in the oracle compatibility functions btrim() ltrim() and rtrim()
Fix for shared invalidation cache overflow(Massimo)
Prevent file descriptor leaks in failed COPY's(Bruce)
Fix memory leak in libpgtcl's pg_select(Constantin)
Fix problems with nom_utilisateur/passwords over 8 characters(Tom)
Fix problems with handling of asynchronous NOTIFY in backend(Tom)
Fix of many bad system table entries(Tom)

Enhancements
------------
Upgrade ecpg and ecpglib,see src/interfaces/ecpc/ChangeLog(Michael)
Show the index used in an EXPLAIN(Zeugswetter)
EXPLAIN  invokes  rule system and shows plan(s) for rewritten queries(Jan)
Multibyte awareness of many data types and functions, via configure(Tatsuo)
New configure --with-mb option(Tatsuo)
New initdb --pgencoding option(Tatsuo)
New createdb -E multibyte option(Tatsuo)
Select version(); now returns PostgreSQL version(Jeroen)
libpq now allows asynchronous clients(Tom)
Allow cancel from client of backend query(Tom)
psql now cancels query with Control-C(Tom)
libpq users need not issue dummy queries to get NOTIFY messages(Tom)
NOTIFY now sends sender's PID, so you can tell whether it was your own(Tom)
PGresult struct now includes associated error message, if any(Tom)
Define "tz_hour" and "tz_minute" arguments to date_part()(Thomas)
Add routines to convert between varchar and bpchar(Thomas)
Add routines to allow sizing of varchar and bpchar into target columns(Thomas)
Add bit flags to support timezonehour and minute in data retrieval(Thomas)
Allow more variations on valid floating point numbers (e.g. ".1", "1e6")(Thomas)
Fixes for unary minus parsing with leading spaces(Thomas)
Implement TIMEZONE_HOUR, TIMEZONE_MINUTE per SQL92 specs(Thomas)
Check for and properly ignore FOREIGN KEY column constraints(Thomas)
Define USER as synonym for CURRENT_USER per SQL92 specs(Thomas)
Enable HAVING clause but no fixes elsewhere yet.
Make "char" type a synonym for "char(1)" (actually implemented as
bpchar)(Thomas)
Save string type if specified for DEFAULT clause handling(Thomas)
Coerce operations involving different data types(Thomas)
Allow some index use for columns of different types(Thomas)
Add capabilities for automatic type conversion(Thomas)
Cleanups for large objects, so file is truncated on open(Peter)
Readline cleanups(Tom)
Allow psql  \f \ to make spaces as delimiter(Bruce)
Pass pg_attribute.atttypmod to the frontend for column field lengths(Tom,Bruce)
Msql compatibility library in /contrib(Aldrin)
Remove the requirement that ORDER/GROUP BY clause identifiers be
included in the target list(David)
Convert columns to match columns in UNION clauses(Thomas)
Remove fork()/exec() and only do fork()(Bruce)
Jdbc cleanups(Peter)
Show backend status on ps command line(only works on some platforms)(Bruce)
```

```
Pg_hba.conf now has a sameuser option in the database field
Make lo_unlink take oid param, not int4
New DISABLE_COMPLEX_MACRO for compilers that can't handle our macros(Bruce)
Libpgtcl now handles NOTIFY as a Tcl event, need not send dummy queries(Tom)
libpgtcl cleanups(Tom)
Add -error option to libpgtcl's pg_result command(Tom)
New locale patch, see docs/README/locale(Oleg)
Fix for pg_dump so CONSTRAINT and CHECK syntax is correct(ccb)
New contrib/lo code for large object orphan removal(Peter)
New psql command "SET CLIENT_ENCODING TO 'encoding'" for multibytes
feature, see /doc/README.mb(Tatsuo)
contrib/noupdate code to revoke update permission on a column
libpq can now be compiled on Windows(Magnus)
Add PQsetdbLogin() in libpq
New 8-byte integer type, checked by configure for OS support(Thomas)
Better support for quoted table/column names(Thomas)
Surround table and column names with double-quotes in pg_dump(Thomas)
PQreset() now works with passwords(Tom)
Handle case of GROUP BY target list column number out of range(David)
Allow UNION in subselects
Add auto-size to screen to \d? commands(Bruce)
Use UNION to show all \d? results in one query(Bruce)
Add \d? field search feature(Bruce)
Pg_dump issues fewer \connect requests(Tom)
Make pg_dump -z flag work better, document it in manual page(Tom)
Add HAVING clause with full support for subselects and unions(Stephan)
Full text indexing routines in contrib/fulltextindex(Maarten)
Transaction ids now stored in shared memory(Vadim)
New PGCLIENTENCODING when issuing COPY command(Tatsuo)
Support for SQL92 syntax "SET NAMES"(Tatsuo)
Support for LATIN2-5(Tatsuo)
Add UNICODE regression test case(Tatsuo)
Lock manager cleanup, new locking modes for LLL(Vadim)
Allow index use with OR clauses(Bruce)
Allows "SELECT NULL ORDER BY 1;"
Explain VERBOSE prints the plan, and now pretty-prints the plan to
the postmaster log file(Bruce)
Add indexes display to \d command(Bruce)
Allow GROUP BY on functions(David)
New pg_class.relkind for large objects(Bruce)
New way to send libpq NOTICE messages to a different location(Tom)
New \w write command to psql(Bruce)
New /contrib/findoidjoins scans oid columns to find join relationships(Bruce)
Allow binary-compatible indexes to be considered when checking for valid
Indexes for restriction clauses containing a constant(Thomas)
New ISBN/ISSN code in /contrib/isbn_issn
Allow NOT LIKE, IN, NOT IN, BETWEEN, and NOT BETWEEN constraint(Thomas)
New rewrite system fixes many problems with rules and views(Jan)
        * Rules on relations work
        * Event qualifications on insert/update/delete work
        * New OLD variable to reference CURRENT, CURRENT will be remove in
future
        * Update rules can reference NEW and OLD in rule qualifications/actions
        * Insert/update/delete rules on views work
        * Multiple rule actions are now supported, surrounded by parenthèses
        * Regular users can create views/rules on tables they have RULE permits
        * Rules and views inherit the privileges of the creator
        * No rules at the column level
        * No UPDATE NEW/OLD rules
        * New pg_tables, pg_indexes, pg_rules and pg_views system views
        * Only a single action on SELECT rules
        * Total rewrite overhaul, perhaps for 6.5
```

```
        * handle subselects
        * handle aggregates on views
        * handle insert into select from view works
System indexes are now multikey(Bruce)
Oidint2, oidint4, and oidname types are removed(Bruce)
Use system cache for more system table lookups(Bruce)
New backend programming language PL/pgSQL in backend/pl(Jan)
New SERIAL data type, auto-creates sequence/index(Thomas)
Enable assert checking without a recompile(Massimo)
User lock enhancements(Massimo)
New setval() command to set sequence value(Massimo)
Auto-remove unix socket file on start-up if no postmaster running(Massimo)
Conditional trace package(Massimo)
New UNLISTEN command(Massimo)
psql and libpq now compile under Windows using win32.mak(Magnus)
Lo_read no longer stores trailing NULL(Bruce)
Identifiers are now truncated to 31 characters internally(Bruce)
Createuser options now availble on the command line
Code for 64-bit integer supported added, configure tested, int8 type(Thomas)
Prevent file descriptor leaf from failed COPY(Bruce)
New pg_upgrade command(Bruce)
Updated /contrib directories(Massimo)
New CREATE TABLE DEFAULT VALUES statement available(Thomas)
New INSERT INTO TABLE DEFAULT VALUES statement available(Thomas)
New DECLARE and FETCH feature(Thomas)
libpq's internal structures now not exported(Tom)
Allow up to 8 key indexes(Bruce)
Remove ARCHIVE key word, that is no longer used(Thomas)
pg_dump -n flag to supress quotes around indentifiers
disable system columns for views(Jan)
new INET and CIDR types for network addresses(TomH, Paul)
no more double quotes in psql output
pg_dump now dumps views(Terry)
new SET QUERY_LIMIT(Tatsuo,Jan)

Source Tree Changes
-------------------
/contrib cleanup(Jun)
Inline some small functions called for every row(Bruce)
Alpha/linux fixes
HP-UX cleanups(Tom)
Multibyte regression tests(Soonmyung.)
Remove --disabled options from configure
Define PGDOC to use POSTGRESDIR by default
Make regression optional
Remove extra braces code to pgindent(Bruce)
Add bsdi shared library support(Bruce)
New --without-CXX support configure option(Brook)
New FAQ_CVS
Update backend flowchart in tools/backend(Bruce)
Change atttypmod from int16 to int32(Bruce, Tom)
Getrusage() fix for platforms that do not have it(Tom)
Add PQconnectdb, PGUSER, PGPASSWORD to libpq man page
NS32K platform fixes(Phil Nelson, John Buller)
SCO 7/UnixWare 2.x fixes(Billy,others)
Sparc/Solaris 2.5 fixes(Ryan)
Pgbuiltin.3 is obsolete, move to doc files(Thomas)
Even more documention(Thomas)
Nextstep support(Jacek)
Aix support(David)
pginterface manual page(Bruce)
shared libraries all have version numbers
```

```
merged all OS-specific shared library defines into one file
smarter TCL/TK configuration checking(Billy)
smarter perl configuration(Brook)
configure uses supplied install-sh if no install script found(Tom)
new Makefile.shlib for shared library configuration(Tom)
```

# E.55. Sortie 6.3.2

**Date de sortie :** 1998−04−07

This is a bogue−fix release for 6.3.x. Refer to the release notes for version 6.3 for a more complete summary of new features.

Summary:

- Repairs automatic configuration support for some platforms, including Linux, from breakage inadvertently introduced in version 6.3.1.
- Correctly handles function calls on the left side of BETWEEN and LIKE clauses.

A dump/restore is NOT required for those running 6.3 or 6.3.1. A `make distclean`, `make`, and `make install` is all that is required. This last step should be performed while the postmaster is not running. You should re−link any custom applications that use PostgreSQL libraries.

For upgrades from pre−6.3 installations, refer to the installation and migration instructions for version 6.3.

## E.55.1. Modifications

```
Configure detection improvements for tcl/tk(Brook Milligan, Alvin)
Manual page improvements(Bruce)
BETWEEN and LIKE fix(Thomas)
fix for psql \connect used by pg_dump(Oliver Elphick)
New odbc driver
pgaccess, version 0.86
qsort removed, now uses libc version, cleanups(Jeroen)
fix for buffer over-runs detected(Maurice Gittens)
fix for buffer overrun in libpgtcl(Randy Kunkee)
fix for UNION with DISTINCT or ORDER BY(Bruce)
gettimeofday configure check(Doug Winterburn)
Fix "indexes not used" bogue(Vadim)
docs additions(Thomas)
Fix for backend memory leak(Bruce)
libreadline cleanup(Erwan MAS)
Remove DISTDIR(Bruce)
Makefile dépendency cleanup(Jeroen van Vianen)
ASSERT fixes(Bruce)
```

# E.56. Sortie 6.3.1

**Date de sortie :** 1998−03−23

Summary:

- Additional support for multibyte character sets.
- Repair byte ordering for mixed−endian clients and servers.
- Minor updates to allowed SQL syntax.
- Improvements to the configuration autodetection for installation.

A dump/restore is NOT required for those running 6.3. A `make distclean`, `make`, and `make install` is all that is required. This last step should be performed while the postmaster is not running. You should re−link any custom applications that use PostgreSQL libraries.

For upgrades from pre−6.3 installations, refer to the installation and migration instructions for version 6.3.

## E.56.1. Modifications

```
ecpg cleanup/fixes, now version 1.1(Michael Meskes)
pg_user cleanup(Bruce)
large object fix for pg_dump and tclsh (alvin)
LIKE fix for multiple adjacent underscores
fix for redefining builtin functions(Thomas)
ultrix4 cleanup
upgrade to pg_access 0.83
updated CLUSTER manual page
multibyte character set support, see doc/README.mb(Tatsuo)
configure --with-pgport fix
pg_ident fix
big-endian fix for backend communications(Kataoka)
SUBSTR() and substring() fix(Jan)
several jdbc fixes(Peter)
libpgtcl improvements, see libptcl/README(Randy Kunkee)
Fix for "Datasize = 0" error(Vadim)
Prevent \do from wrapping(Bruce)
Remove duplicate Russian character set entries
Sunos4 cleanup
Allow optional TABLE key word in LOCK and SELECT INTO(Thomas)
CREATE SEQUENCE options to allow a negative integer(Thomas)
Add "PASSWORD" as an allowed column identifier(Thomas)
Add checks for UNION target fields(Bruce)
Fix Alpha port(Dwayne Bailey)
Fix for text arrays containing quotes(Doug Gibson)
Solaris compile fix(Albert Chin-A-Young)
Better identify tcl and tk libs and includes(Bruce)
```

# E.57. Sortie 6.3

**Date de sortie :** 1998−03−01

There are *many* new features and improvements in this release. Here is a brief, incomplete summary:

- Many new SQL features, including full SQL92 subselect capability (everything is here but target−list subselects).
- Support for client−side environment variables to specify time zone and date style.
- Socket interface for client/server connection. This is the default now so you may need to start postmaster with the `-i` flag.
- Better password authorization mechanisms. Default table privileges have changed.
- Old−style *time travel* has been removed. Performance has been improved.

**Note :** Bruce Momjian wrote the following notes to introduce the new release.

There are some general 6.3 issues that I want to mention. These are only the big items that can not be described in one sentence. A review of the detailed changes list is still needed.

First, we now have subselects. Now that we have them, I would like to mention that without subselects, SQL is a very limited language. Subselects are a major feature, and you should review your code for places where subselects provide a better solution for your queries. I think you will find that there are more uses for subselects than you may think. Vadim has put us on the big SQL map with subselects, and fully functional ones too. The only thing you can't do with subselects is to use them in the target list.

Second, 6.3 uses Unix domain sockets rather than TCP/IP by default. To enable connections from other machines, you have to use the new postmaster −i option, and of course edit `pg_hba.conf`. Also, for this reason, the format of `pg_hba.conf` has changed.

Third, `char()` fields will now allow faster access than `varchar()` or `text`. Specifically, the `text` and `varchar()` have a penalty for access to any columns after the first column of this type. `char()` used to also have this access penalty, but it no longer does. This may suggest that you redesign some of your tables, especially if you have short character columns that you have defined as `varchar()` or `text`. This and other changes make 6.3 even faster than earlier releases.

We now have passwords definable indépendent of any Unix file. There are new SQL USER commands. See the *Administrator's Guide* for more information. There is a new table, pg_shadow, which is used to store user information and user passwords, and it by default only SELECT−able by the postgres super−user. pg_user is now a view of pg_shadow, and is SELECT−able by PUBLIC. You should keep using pg_user in your application without changes.

User−created tables now no longer have SELECT privilege to PUBLIC by default. This was done because the ANSI standard requires it. You can of course GRANT any privileges you want after the table is created. System tables continue to be SELECT−able by PUBLIC.

We also have real deadlock detection code. No more sixty−second timeouts. And the new locking code implements a FIFO better, so there should be less resource starvation during heavy use.

Many complaints have been made about inadequate documentation in previous releases. Thomas has put much effort into many new manuals for this release. Check out the doc/ directory.

For performance reasons, time travel is gone, but can be implemented using triggers (see `pgsql/contrib/spi/README`). Please check out the new \d command for types, operators, etc. Also, views have their own privileges now, not based on the underlying tables, so privileges on them have to be set separately. Check `/pgsql/interfaces` for some new ways to talk to PostgreSQL.

This is the first release that really required an explanation for existing users. In many ways, this was necessary because the new release removes many limitations, and the work−arounds people were using are no longer needed.

## E.57.1. Migration vers la version 6.3

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of PostgreSQL.

## E.57.2. Modifications

```
Bug Fixes
---------
Fix binary cursors broken by MOVE implementation(Vadim)
Fix for tcl library crash(Jan)
Fix for array handling, from Gerhard Hintermayer
Fix acl error, and remove duplicate pqtrace(Bruce)
Fix psql \e for empty file(Bruce)
Fix for textcat on varchar() fields(Bruce)
Fix for DBT Sendproc (Zeugswetter Andres)
Fix vacuum analyze syntax problem(Bruce)
Fix for international identifiers(Tatsuo)
Fix aggregates on inherited tables(Bruce)
Fix substr() for out-of-bounds data
Fix for select 1=1 or 2=2, select 1=1 and 2=2, and select sum(2+2)(Bruce)
Fix notty output to show status result.  -q option still turns it off(Bruce)
Fix for count(*), aggs with views and multiple tables and sum(3)(Bruce)
Fix cluster(Bruce)
Fix for PQtrace start/stop several times(Bruce)
Fix a variety of locking problems like newer lock waiters getting
        lock before older waiters, and having readlock people not share
        locks if a writer is waiting for a lock, and waiting writers not
        getting priority over waiting readers(Bruce)
Fix crashes in psql when executing queries from external files(James)
Fix problem with multiple order by columns, with the first one having
        NULL values(Jeroen)
Use correct hash table support functions for float8 and int4(Thomas)
Re-enable JOIN= option in CREATE OPERATOR statement (Thomas)
Change precedence for boolean operators to match expected behavior(Thomas)
Generate elog(ERROR) on over-large integer(Bruce)
Allow multiple-argument functions in constraint clauses(Thomas)
Check boolean input literals for 'true','false','yes','no','1','0'
        and throw elog(ERROR) if unrecognized(Thomas)
Major large objects fix
Fix for GROUP BY showing duplicates(Vadim)
Fix for index scans in MergeJion(Vadim)

Enhancements
------------
Subselects with EXISTS, IN, ALL, ANY key words (Vadim, Bruce, Thomas)
New User Manual(Thomas, others)
Speedup by inlining some frequently-called functions
Real deadlock detection, no more timeouts(Bruce)
Add SQL92 "constants" CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP,
        CURRENT_USER(Thomas)
Modify constraint syntax to be SQL92-compliant(Thomas)
Implement SQL92 PRIMARY KEY and UNIQUE clauses using indexes(Thomas)
Recognize SQL92 syntax for FOREIGN KEY. Throw elog notice(Thomas)
Allow NOT NULL UNIQUE constraint clause (each allowed separately before)(Thomas)
Allow PostgreSQL-style casting ("::") of non-constants(Thomas)
Add support for SQL3 TRUE and FALSE boolean constants(Thomas)
Support SQL92 syntax for IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE(Thomas)
Allow shorter strings for boolean literals (e.g. "t", "tr", "tru")(Thomas)
Allow SQL92 delimited identifiers(Thomas)
Implement SQL92 binary and hexadecimal string decoding (b'10' and x'1F')(Thomas)
Support SQL92 syntax for type coercion of literal strings
        (e.g. "DATETIME 'now'")(Thomas)
Add conversions for int2, int4, and OID types to and from text(Thomas)
Use shared lock when building indexes(Vadim)
```

```
Free memory allocated for an user query inside transaction block after
        this query is done, was turned off in <= 6.2.1(Vadim)
New SQL statement CREATE PROCEDURAL LANGUAGE(Jan)
New PostgreSQL Procedural Language (PL) backend
interface(Jan)
Rename pg_dump -H option to -h(Bruce)
Add Java support for passwords, European dates(Peter)
Use indexes for LIKE and ~, !~ operations(Bruce)
Add hash functions for datetime and timespan(Thomas)
Time Travel removed(Vadim, Bruce)
Add paging for \d and \z, and fix \i(Bruce)
Add Unix domain socket support to backend and to frontend library(Goran)
Implement CREATE DATABASE/WITH LOCATION and initlocation utility(Thomas)
Allow more SQL92 and/or PostgreSQL reserved words as
column identifiers(Thomas)
Augment support for SQL92 SET TIME ZONE...(Thomas)
SET/SHOW/RESET TIME ZONE uses TZ backend environment variable(Thomas)
Implement SET keyword = DEFAULT and SET TIME ZONE DEFAULT(Thomas)
Enable SET TIME ZONE using TZ environment variable(Thomas)
Add PGDATESTYLE environment variable to frontend and backend
initialization(Thomas)
Add PGTZ, PGCOSTHEAP, PGCOSTINDEX, PGRPLANS, PGGEQO
        frontend library initialization environment variables(Thomas)
Regression tests time zone automatically set with "setenv PGTZ PST8PDT"(Thomas)
Add pg_description table for info on tables, columns, operators, types, and
        aggregates(Bruce)
Increase 16 char limit on system table/index names to 32 characters(Bruce)
Rename system indexes(Bruce)
Add 'GERMAN' option to SET DATESTYLE(Thomas)
Define an "ISO-style" timespan output format with "hh:mm:ss" fields(Thomas)
Allow fractional values for delta times (e.g. '2.5 days')(Thomas)
Validate numeric input more carefully for delta times(Thomas)
Implement day of year as possible input to date_part()(Thomas)
Define timespan_finite() and text_timespan() functions(Thomas)
Remove archive stuff(Bruce)
Allow for a pg_password authentication database that is separate from
        the system password file(Todd)
Dump ACLs, GRANT, REVOKE privileges(Matt)
Define text, varchar, and bpchar string length functions(Thomas)
Fix Query handling for inheritance, and cost computations(Bruce)
Implement CREATE TABLE/AS SELECT (alternative to SELECT/INTO)(Thomas)
Allow NOT, IS NULL, IS NOT NULL in constraints(Thomas)
Implement UNIONs for SELECT(Bruce)
Add UNION, GROUP, DISTINCT to INSERT(Bruce)
varchar() stores only necessary bytes on disk(Bruce)
Fix for BLOBs(Peter)
Mega-Patch for JDBC...see README_6.3 for list of changes(Peter)
Remove unused "option" from PQconnectdb()
New LOCK command and lock manual page describing deadlocks(Bruce)
Add new psql \da, \dd, \df, \do, \dS, and \dT commands(Bruce)
Enhance psql \z to show sequences(Bruce)
Show NOT NULL and DEFAULT in psql \d table(Bruce)
New psql .psqlrc file start-up(Andrew)
Modify sample start-up script in contrib/linux to show syslog(Thomas)
New types for IP and MAC addresses in contrib/ip_and_mac(TomH)
Unix system time conversions with date/time types in contrib/unixdate(Thomas)
Update of contrib stuff(Massimo)
Add Unix socket support to DBD::Pg(Goran)
New python interface (PyGreSQL 2.0)(D'Arcy)
New frontend/backend protocol has a version number, network byte order(Phil)
Security features in pg_hba.conf enhanced and documented, many cleanups(Phil)
CHAR() now faster access than VARCHAR() or TEXT
```

```
ecpg embedded SQL preprocessor
Reduce system column overhead(Vadmin)
Remove pg_time table(Vadim)
Add pg_type attribute to identify types that need length (bpchar, varchar)
Add report of offending line when COPY command fails
Allow VIEW privileges to be set separately from the underlying tables.
        For security, use GRANT/REVOKE on views as appropriate(Jan)
Tables now have no default GRANT SELECT TO PUBLIC.  You must
        explicitly grant such privileges.
Clean up tutorial exemples(Darren)

Source Tree Changes
-------------------
Add new html development tools, and flow chart in /tools/backend
Fix for SCO compiles
Stratus computer port Robert Gillies
Added support for shlib for BSD44_derived & i386_solaris
Make configure more automated(Brook)
Add script to check regression test results
Break parser functions into smaller files, group together(Bruce)
Rename heap_create to heap_create_and_catalog, rename heap_creatr
        to heap_create()(Bruce)
Sparc/Linux patch for locking(TomS)
Remove PORTNAME and reorganize port-specific stuff(Marc)
Add optimizer README file(Bruce)
Remove some recursion in optimizer and clean up some code there(Bruce)
Fix for NetBSD locking(Henry)
Fix for libptcl make(Tatsuo)
AIX patch(Darren)
Change IS TRUE, IS FALSE, ... to expressions using "=" rather than
        function calls to istrue() or isfalse() to allow optimization(Thomas)
Various fixes NetBSD/Sparc related(TomH)
Alpha linux locking(Travis,Ryan)
Change elog(WARN) to elog(ERROR)(Bruce)
FAQ for FreeBSD(Marc)
Bring in the PostODBC source tree as part of our standard distribution(Marc)
A minor patch for HP/UX 10 vs 9(Stan)
New pg_attribute.atttypmod for type-specific info like varchar length(Bruce)
UnixWare patches(Billy)
New i386 'lock' for spinlock asm(Billy)
Support for multiplexed backends is removed
Start an OpenBSD port
Start an AUX port
Start a Cygnus port
Add string functions to regression suite(Thomas)
Expand a few function names formerly truncated to 16 characters(Thomas)
Remove un-needed malloc() calls and replace with palloc()(Bruce)
```

# E.58. Sortie 6.2.1

**Date de sortie :** 1997−10−17

6.2.1 is a bogue−fix and usability release on 6.2.

Summary:

- Allow strings to span lines, per SQL92.
- Include exemple trigger function for inserting user names on table updates.

This is a minor bogue−fix release on 6.2. For upgrades from pre−6.2 systems, a full dump/reload is required. Refer to the 6.2 release notes for instructions.

## E.58.1. Migration from version 6.2 to version 6.2.1

This is a minor bogue−fix release. A dump/reload is not required from version 6.2, but is required from any release prior to 6.2.

In upgrading from version 6.2, if you choose to dump/reload you will find that avg(money) is now calculated correctly. All other bogue fixes take effect upon updating the executables.

Another way to avoid dump/reload is to use the following SQL command from `psql` to update the existing system table:

```
update pg_aggregate set aggfinalfn = 'cash_div_flt8'
 where aggname = 'avg' and aggbasetype = 790;
```

This will need to be done to every existing database, including template1.

## E.58.2. Modifications

```
Allow TIME and TYPE column names(Thomas)
Allow larger range of true/false as boolean values(Thomas)
Support output of "now" and "current"(Thomas)
Handle DEFAULT with INSERT of NULL properly(Vadim)
Fix for relation reference counts problem in buffer manager(Vadim)
Allow strings to span lines, like ANSI(Thomas)
Fix for backward cursor with ORDER BY(Vadim)
Fix avg(cash) computation(Thomas)
Fix for specifying a column twice in ORDER/GROUP BY(Vadim)
Documented new libpq function to return affected rows, PQcmdTuples(Bruce)
Trigger function for inserting user names for INSERT/UPDATE(Brook Milligan)
```

# E.59. Sortie 6.2

**Date de sortie :** 1997−10−02

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL.

## E.59.1. Migration from version 6.1 to version 6.2

This migration requires a complete dump of the 6.1 database and a restore of the database in 6.2.

Note that the `pg_dump` and `pg_dumpall` utility from 6.2 should be used to dump the 6.1 database.

## E.59.2. Migration from version 1.*x* to version 6.2

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

## E.59.3. Modifications

```
Bug Fixes
---------
Fix problems with pg_dump for inheritance, sequences, archive tables(Bruce)
Fix compile errors on overflow due to shifts, unsigned, and bad prototypes
         from Solaris(Diab Jerius)
Fix bogues in geometric line arithmetic (bad intersection calculations)(Thomas)
Check for geometric intersections at endpoints to avoid rounding
ugliness(Thomas)
Catch non-functional delete attempts(Vadim)
Change time function names to be more consistent(Michael Reifenberg)
Check for zero divides(Michael Reifenberg)
Fix very old bogue which made rows changed/inserted by a command
        visible to the command itself (so we had multiple update of
        updated rows, etc.)(Vadim)
Fix for SELECT NULL, 'fail' FROM pg_am (Patrick)
SELECT NULL as EMPTY_FIELD now allowed(Patrick)
Remove un-needed signal stuff from contrib/pginterface
Fix OR (where x != 1 or x isNULL didn't return rows with x NULL) (Vadim)
Fix time_cmp function (Vadim)
Fix handling of functions with non-attribute first argument in
        WHERE clauses (Vadim)
Fix GROUP BY when order of entries is different from order
        in target list (Vadim)
Fix pg_dump for aggregates without sfunc1 (Vadim)

Enhancements
------------
Default genetic optimizer GEQO parameter is now 8(Bruce)
Allow use parameters in target list having aggregates in functions(Vadim)
Added JDBC driver as an interface(Adrian & Peter)
pg_password utility
Return number of rows inserted/affected by INSERT/UPDATE/DELETE etc.(Vadim)
Triggers implemented with CREATE TRIGGER (SQL3)(Vadim)
SPI (Server Programming Interface) allows exécution of queries inside
        C-functions (Vadim)
NOT NULL implemented (SQL92)(Robson Paniago de Miranda)
Include reserved words for string handling, outer joins, and unions(Thomas)
Implement extended comments ("/* ... */") using exclusive states(Thomas)
Add "//" single-line comments(Bruce)
Remove some restrictions on characters in operator names(Thomas)
DEFAULT and CONSTRAINT for tables implemented (SQL92)(Vadim & Thomas)
Add text concatenation operator and function (SQL92)(Thomas)
Support WITH TIME ZONE syntax (SQL92)(Thomas)
Support INTERVAL unit TO unit syntax (SQL92)(Thomas)
Define types DOUBLE PRECISION, INTERVAL, CHARACTER,
        and CHARACTER VARYING (SQL92)(Thomas)
Define type FLOAT(p) and rudimentary DECIMAL(p,s), NUMERIC(p,s) (SQL92)(Thomas)
Define EXTRACT(), POSITION(), SUBSTRING(), and TRIM() (SQL92)(Thomas)
Define CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP (SQL92)(Thomas)
Add syntax and warnings for UNION, HAVING, INNER and OUTER JOIN (SQL92)(Thomas)
Add more reserved words, mostly for SQL92 compliance(Thomas)
Allow hh:mm:ss time entry for timespan/reltime types(Thomas)
Add center() routines for lseg, path, polygon(Thomas)
Add distance() routines for circle-polygon, polygon-polygon(Thomas)
Check explicitly for points and polygons contained within polygons
        using an axis-crossing algorithm(Thomas)
Add routine to convert circle-box(Thomas)
Merge conflicting operators for different geometric data types(Thomas)
Replace distance operator "<===>" with "<->"(Thomas)
```

```
Replace "above" operator "!^" with ">^" and "below" operator "!|" with
"<^"(Thomas)
Add routines for text trimming on both ends, substring, and string
position(Thomas)
Added conversion routines circle(box) and poly(circle)(Thomas)
Allow internal sorts to be stored in memory rather than in files(Bruce & Vadim)
Allow functions and operators on internally-identical types to succeed(Bruce)
Speed up backend start-up after profiling analysis(Bruce)
Inline frequently called functions for performance(Bruce)
Reduce open() calls(Bruce)
psql:  Add PAGER for \h and \?,\C fix
Fix for psql pager when no tty(Bruce)
New entab utility(Bruce)
General trigger functions for referential integrity (Vadim)
General trigger functions for time travel (Vadim)
General trigger functions for AUTOINCREMENT/IDENTITY feature (Vadim)
MOVE implementation (Vadim)

Source Tree Changes
-------------------
HP-UX 10 patches (Vladimir Turin)
Added SCO support, (Daniel Harris)
MkLinux patches (Tatsuo Ishii)
Change geometric box terminology from "length" to "width"(Thomas)
Deprecate temporary unstored slope fields in geometric code(Thomas)
Remove restart instructions from INSTALL(Bruce)
Look in /usr/ucb first for install(Bruce)
Fix c++ copy exemple code(Thomas)
Add -o to psql manual page(Bruce)
Prevent relname unallocated string length from being copied into database(Bruce)
Cleanup for NAMEDATALEN use(Bruce)
Fix pg_proc names over 15 chars in output(Bruce)
Add strNcpy() function(Bruce)
remove some (void) casts that are unnecessary(Bruce)
new interfaces directory(Marc)
Replace fopen() calls with calls to fd.c functions(Bruce)
Make functions static where possible(Bruce)
enclose unused functions in #ifdef NOT_USED(Bruce)
Remove call to difftime() in timestamp support to fix SunOS(Bruce & Thomas)
Changes for Digital Unix
Portability fix for pg_dumpall(Bruce)
Rename pg_attribute.attnvals to attdispersion(Bruce)
"intro/unix" manual page now "pgintro"(Bruce)
"built-in" manual page now "pgbuiltin"(Bruce)
"drop" manual page now "drop_table"(Bruce)
Add "create_trigger", "drop_trigger" manual pages(Thomas)
Add constraints regression test(Vadim & Thomas)
Add comments syntax regression test(Thomas)
Add PGINDENT and support program(Bruce)
Massive commit to run PGINDENT on all *.c and *.h files(Bruce)
Files moved to /src/tools directory(Bruce)
SPI and Trigger programming guides (Vadim & D'Arcy)
```

# E.60. Sortie 6.1.1

**Date de sortie :** 1997-07-22

### E.60.1. Migration from version 6.1 to version 6.1.1

This is a minor bogue−fix release. A dump/reload is not required from version 6.1, but is required from any release prior to 6.1. Refer to the release notes for 6.1 for more details.

### E.60.2. Modifications

```
fix for SET with options (Thomas)
allow pg_dump/pg_dumpall to preserve ownership of all tables/objects(Bruce)
new psql \connect option allows changing usernames without changing databases
fix for initdb --debogue option(Yoshihiko Ichikawa))
lextest cleanup(Bruce)
hash fixes(Vadim)
fix date/time month boundary arithmetic(Thomas)
fix timezone daylight handling for some ports(Thomas, Bruce, Tatsuo)
timestamp overhauled to use standard functions(Thomas)
other code cleanup in date/time routines(Thomas)
psql's \d now case-insensitive(Bruce)
psql's antislash commands can now have trailing semicolon(Bruce)
fix memory leak in psql when using \g(Bruce)
major fix for endian handling of communication to server(Thomas, Tatsuo)
Fix for Solaris assembler and include files(Yoshihiko Ichikawa)
allow underscores in usernames(Bruce)
pg_dumpall now returns proper status, portability fix(Bruce)
```

## E.61. Sortie 6.1

**Date de sortie :** 1997−06−08

The regression tests have been adapted and extensively modified for the 6.1 release of PostgreSQL.

Three new data types (`datetime`, `timespan`, and `circle`) have been added to the native set of PostgreSQL types. Points, boxes, paths, and polygons have had their output formats made consistent across the data types. The polygon output in misc.out has only been spot−checked for correctness relative to the original regression output.

PostgreSQL 6.1 introduces a new, alternate optimizer which uses *genetic* algorithms. These algorithms introduce a random behavior in the ordering of query results when the query contains multiple qualifiers or multiple tables (giving the optimizer a choice on order of evaluation). Several regression tests have been modified to explicitly order the results, and hence are insensitive to optimizer choices. A few regression tests are for data types which are inherently unordered (e.g. points and time intervals) and tests involving those types are explicitly bracketed with `set geqo to 'off'` and `reset geqo`.

The interpretation of array specifiers (the curly braces around atomic values) appears to have changed sometime after the original regression tests were generated. The current `./expected/*.out` files reflect this new interpretation, which may not be correct!

The float8 regression test fails on at least some platforms. This is due to différences in implementations of `pow()` and `exp()` and the signaling mechanisms used for overflow and underflow conditions.

The << random >> results in the random test should cause the << random >> test to be << failed >>, since the regression tests are evaluated using a simple diff. However, << random >> does not seem to produce random

results on my test machine (Linux/gcc/i686).

## E.61.1. Migration vers la version 6.1

This migration requires a complete dump of the 6.0 database and a restore of the database in 6.1.

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

## E.61.2. Modifications

```
Bug Fixes
---------
packet length checking in library routines
lock manager priority patch
check for under/over flow of float8(Bruce)
multitable join fix(Vadim)
SIGPIPE crash fix(Darren)
large object fixes(Sven)
allow btree indexes to handle NULLs(Vadim)
timezone fixes(D'Arcy)
select SUM(x) can return NULL on no rows(Thomas)
internal optimizer, executor bogue fixes(Vadim)
fix problem where inner loop in < or <= has no rows(Vadim)
prevent re-commuting join index clauses(Vadim)
fix join clauses for multiple tables(Vadim)
fix hash, hashjoin for arrays(Vadim)
fix btree for abstime type(Vadim)
large object fixes(Raymond)
fix buffer leak in hash indexes (Vadim)
fix rtree for use in inner scan (Vadim)
fix gist for use in inner scan, cleanups (Vadim, Andrea)
avoid unnecessary local buffers allocation (Vadim, Massimo)
fix local buffers leak in transaction aborts (Vadim)
fix file manager memmory leaks, cleanups (Vadim, Massimo)
fix storage manager memmory leaks (Vadim)
fix btree duplicates handling (Vadim)
fix deleted rows reincarnation caused by vacuum (Vadim)
fix SELECT varchar()/char() INTO TABLE made zero-length fields(Bruce)
many psql, pg_dump, and libpq memory leaks fixed using Purify (Igor)

Enhancements
------------
attribute optimization statistics(Bruce)
much faster new btree bulk load code(Paul)
BTREE UNIQUE added to bulk load code(Vadim)
new lock debogue code(Massimo)
massive changes to libpg++(Leo)
new GEQO optimizer speeds table multitable optimization(Martin)
new WARN message for non-unique insert into unique key(Marc)
update x=-3, no spaces, now valid(Bruce)
remove case-sensitive identifier handling(Bruce,Thomas,Dan)
debogue backend now pretty-prints tree(Darren)
new Oracle character functions(Edmund)
new plaintext password functions(Dan)
no such class or insufficient privilege changed to distinct messages(Dan)
new ANSI timestamp function(Dan)
new ANSI Time and Date types (Thomas)
move large chunks of data in backend(Martin)
```

```
multicolumn btree indexes(Vadim)
new SET var TO value command(Martin)
update transaction status on reads(Dan)
new locale settings for character types(Oleg)
new SEQUENCE serial number generator(Vadim)
GROUP BY function now possible(Vadim)
re-organize regression test(Thomas,Marc)
new optimizer operation weights(Vadim)
new psql \z grant/permit option(Marc)
new MONEY data type(D'Arcy,Thomas)
tcp socket communication speed improved(Vadim)
new VACUUM option for attribute statistics, and for certain columns (Vadim)
many geometric type improvements(Thomas,Keith)
additional regression tests(Thomas)
new datestyle variable(Thomas,Vadim,Martin)
more comparison operators for sorting types(Thomas)
new conversion functions(Thomas)
new more compact btree format(Vadim)
allow pg_dumpall to preserve database ownership(Bruce)
new SET GEQO=# and R_PLANS variable(Vadim)
old (!GEQO) optimizer can use right-sided plans (Vadim)
typechecking improvement in SQL parser(Bruce)
new SET, SHOW, RESET commands(Thomas,Vadim)
new \connect database USER option
new destroydb -i option (Igor)
new \dt and \di psql commands (Darren)
SELECT "\n" now escapes newline (A. Duursma)
new geometry conversion functions from old format (Thomas)

Source tree changes
-------------------
new configuration script(Marc)
readline configuration option added(Marc)
OS-specific configuration options removed(Marc)
new OS-specific template files(Marc)
no more need to edit Makefile.global(Marc)
re-arrange include files(Marc)
nextstep patches (Gregor Hoffleit)
removed Windows-specific code(Bruce)
removed postmaster -e option, now only postgres -e option (Bruce)
merge duplicate library code in front/backends(Martin)
now works with eBones, international Kerberos(Jun)
more shared library support
c++ include file cleanup(Bruce)
warn about boguegy flex(Bruce)
DG/UX, Ultrix, IRIX, AIX portability fixes
```

# E.62. Sortie 6.0

**Date de sortie :** 1997-01-29

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL.

## E.62.1. Migration from version 1.09 to version 6.0

This migration requires a complete dump of the 1.09 database and a restore of the database in 6.0.

## E.62.2. Migration from pre−1.09 to version 6.0

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

## E.62.3. Modifications

```
Bug Fixes
---------
ALTER TABLE bogue - running postgress process needs to re-read table définition
Allow vacuum to be run on one table or entire database(Bruce)
Array fixes
Fix array over-runs of memory writes(Kurt)
Fix elusive btree range/non-range bogue(Dan)
Fix for hash indexes on some types like time and date
Fix for pg_log size explosion
Fix permissions on lo_export()(Bruce)
Fix unitialized reads of memory(Kurt)
Fixed ALTER TABLE ... char(3) bogue(Bruce)
Fixed a few small memory leaks
Fixed EXPLAIN handling of options and changed full_path option name
Fixed output of group acl privileges
Memory leaks (hunt and destroy with tools like Purify(Kurt)
Minor improvements to rules system
NOTIFY fixes
New asserts for run-checking
Overhauled parser/analyze code to properly report errors and increase speed
Pg_dump -d now handles NULL's properly(Bruce)
Prevent SELECT NULL from crashing server (Bruce)
Properly report errors when INSERT ... SELECT columns did not match
Properly report errors when insert column names were not correct
psql \g filename now works(Bruce)
psql fixed problem with multiple statements on one line with multiple outputs
Removed duplicate system OID
SELECT * INTO TABLE . GROUP/ORDER BY gives unlink error if table exists(Bruce)
Several fixes for queries that crashed the backend
Starting quote in insert string errors(Bruce)
Submitting an empty query now returns empty status, not just " " query(Bruce)

Enhancements
------------
Add EXPLAIN manual page(Bruce)
Add UNIQUE index capability(Dan)
Add hostname/user level access control rather than just hostname and user
Add synonym of != for <>(Bruce)
Allow "select oid,* from table"
Allow BY,ORDER BY to specify columns by number, or by non-alias
table.column(Bruce)
Allow COPY from the frontend(Bryan)
Allow GROUP BY to use alias column name(Bruce)
Allow actual compression, not just reuse on the same page(Vadim)
Allow installation-configuration option to auto-add all local users(Bryan)
Allow libpq to distinguish between text value '' and NULL(Bruce)
Allow non-postgres users with createdb privs to destroydb's
Allow restriction on who can create C functions(Bryan)
Allow restriction on who can do backend COPY(Bryan)
Can shrink tables, pg_time and pg_log(Vadim & Erich)
Change debogue level 2 to print queries only, changed debogue heading layout(Bruce)
Change default decimal constant representation from float4 to float8(Bruce)
European date format now set when postmaster is started
```

```
Execute lowercase function names if not found with exact case
Fixes for aggregate/GROUP processing, allow 'select sum(func(x),sum(x+y) from z'
Gist now included in the distrubution(Marc)
Idend authentication of local users(Bryan)
Implement BETWEEN qualifier(Bruce)
Implement IN qualifier(Bruce)
libpq has PQgetisNULL()(Bruce)
libpq++ improvements
New options to initdb(Bryan)
Pg_dump allow dump of OID(Bruce)
Pg_dump create indexes after tables are loaded for speed(Bruce)
Pg_dumpall dumps all databases, and the user table
Pginterface additions for NULL values(Bruce)
Prevent postmaster from being run as root
psql \h and \? is now readable(Bruce)
psql allow antislashed, semicolons anywhere on the line(Bruce)
psql changed command prompt for lines in query or in quotes(Bruce)
psql char(3) now displays as (bp)char in \d output(Bruce)
psql return code now more accurate(Bryan?)
psql updated help syntax(Bruce)
Re-visit and fix vacuum(Vadim)
Reduce size of regression diffs, remove timezone name différence(Bruce)
Remove compile-time parameters to enable binary distributions(Bryan)
Reverse meaning of HBA masks(Bryan)
Secure Authentication of local users(Bryan)
Speed up vacuum(Vadim)
Vacuum now had VERBOSE option(Bruce)

Source tree changes
-------------------
All functions now have prototypes that are compared against the calls
Allow asserts to be disabled easly from Makefile.global(Bruce)
Change oid constants used in code to #define names
Decoupled sparc and solaris defines(Kurt)
Gcc -Wall compiles cleanly with warnings only from unfixable constructs
Major include file reorganization/reduction(Marc)
Make now stops on compile failure(Bryan)
Makefile restructuring(Bryan, Marc)
Merge bsdi_2_1 to bsdi(Bruce)
Monitor program removed
Name change from Postgres95 to PostgreSQL
New config.h file(Marc, Bryan)
PG_VERSION now set to 6.0 and used by postmaster
Portability additions, including Ultrix, DG/UX, AIX, and Solaris
Reduced the number of #define's, centeralized #define's
Remove duplicate OIDS in system tables(Dan)
Remove duplicate system catalog info or report mismatches(Dan)
Removed many os-specific #define's
Restructured object file generation/location(Bryan, Marc)
Restructured port-specific file locations(Bryan, Marc)
Unused/uninialized variables corrected
```

# E.63. Sortie 1.09

**Date de sortie :** 1996-11-04

Sorry, we didn't keep track of changes from 1.02 to 1.09. Some of the changes listed in 6.0 were actually included in the 1.02.1 to 1.09 releases.

# E.64. Sortie 1.02

**Date de sortie :** 1996−08−01

## E.64.1. Migration from version 1.02 to version 1.02.1

Here is a new migration file for 1.02.1. It includes the 'copy' change and a script to convert old ASCII files.

> **Note :** The following notes are for the benefit of users who want to migrate databases from Postgres95 1.01 and 1.02 to Postgres95 1.02.1.
>
> If you are starting afresh with Postgres95 1.02.1 and do not need to migrate old databases, you do not need to read any further.

In order to upgrade older Postgres95 version 1.01 or 1.02 databases to version 1.02.1, the following steps are required:

1. Start up a new 1.02.1 postmaster
2. Add the new built−in functions and operators of 1.02.1 to 1.01 or 1.02 databases. This is done by running the new 1.02.1 server against your own 1.01 or 1.02 database and applying the queries attached at the end of the file. This can be done easily through `psql`. If your 1.01 or 1.02 database is named `testdb` and you have cut the commands from the end of this file and saved them in `addfunc.sql`:

   ```
   % psql testdb −f addfunc.sql
   ```

   Those upgrading 1.02 databases will get a warning when executing the last two statements in the file because they are already present in 1.02. This is not a cause for concern.

## E.64.2. Dump/Reload Procedure

If you are trying to reload a pg_dump or text−mode, `copy tablename to stdout` generated with a previous version, you will need to run the attached `sed` script on the ASCII file before loading it into the database. The old format used '.' as end−of−data, while '\.' is now the end−of−data marker. Also, empty strings are now loaded in as '' rather than NULL. See the copy manual page for full details.

```
sed 's/^\.$/\\./g' <in_file >out_file
```

If you are loading an older binary copy or non−stdout copy, there is no end−of−data character, and hence no conversion necessary.

```
-- following lines added by agc to reflect the case-insensitive
-- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure =
texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure =
texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure =
texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure =
texticregexne);
```

## E.64.3. Modifications

```
Source code maintenance and development
 * worldwide team of volunteers
 * the source tree now in CVS at ftp.ki.net

Enhancements
 * psql (and underlying libpq library) now has many more options for
   formatting output, including HTML
 * pg_dump now output the schema and/or the data, with many fixes to
   enhance completeness.
 * psql used in place of monitor in administration shell scripts.
   monitor to be deprecated in next release.
 * date/time functions enhanced
 * NULL insert/update/comparison fixed/enhanced
 * TCL/TK lib and shell fixed to work with both tck7.4/tk4.0 and tcl7.5/tk4.1

Bug Fixes (almost too numerous to mention)
 * indexes
 * storage management
 * check for NULL pointer before dereferencing
 * Makefile fixes

New Ports
 * added SolarisX86 port
 * added BSD/OS 2.1 port
 * added DG/UX port
```

# E.65. Sortie 1.01

**Date de sortie :** 1996−02−23

## E.65.1. Migration from version 1.0 to version 1.01

The following notes are for the benefit of users who want to migrate databases from Postgres95 1.0 to Postgres95 1.01.

If you are starting afresh with Postgres95 1.01 and do not need to migrate old databases, you do not need to read any further.

In order to Postgres95 version 1.01 with databases created with Postgres95 version 1.0, the following steps are required:

1. Set the définition of `NAMEDATALEN` in `src/Makefile.global` to 16 and `OIDNAMELEN` to 20.
2. Decide whether you want to use Host based authentication.

    a. If you do, you must create a file name `pg_hba` in your top−level data directory (typically the value of your `$PGDATA`). `src/libpq/pg_hba` shows an exemple syntax.
    b. If you do not want host−based authentication, you can comment out the line

    ```
    HBA = 1
    ```

    in `src/Makefile.global`

Note that host−based authentication is turned on by default, and if you do not take steps A or B above, the out−of−the−box 1.01 will not allow you to connect to 1.0 databases.

3. Compile and install 1.01, but DO NOT do the `initdb` step.
4. Before doing anything else, terminate your 1.0 postmaster, and backup your existing `$PGDATA` directory.
5. Set your `PGDATA` environment variable to your 1.0 databases, but set up path up so that 1.01 binaries are being used.
6. Modify the file `$PGDATA/PG_VERSION` from 5.0 to 5.1
7. Start up a new 1.01 postmaster
8. Add the new built−in functions and operators of 1.01 to 1.0 databases. This is done by running the new 1.01 server against your own 1.0 database and applying the queries attached and saving in the file 1.0_to_1.01.sql. This can be done easily through `psql`. If your 1.0 database is name `testdb`:

```
% psql testdb -f 1.0_to_1.01.sql
```

and then execute the following commands (cut and paste from here):

```
-- add builtin functions that are new to 1.01

create function int4eqoid (int4, oid) returns bool as 'foo'
language 'internal';
create function oideqint4 (oid, int4) returns bool as 'foo'
language 'internal';
create function char2icregexeq (char2, text) returns bool as 'foo'
language 'internal';
create function char2icregexne (char2, text) returns bool as 'foo'
language 'internal';
create function char4icregexeq (char4, text) returns bool as 'foo'
language 'internal';
create function char4icregexne (char4, text) returns bool as 'foo'
language 'internal';
create function char8icregexeq (char8, text) returns bool as 'foo'
language 'internal';
create function char8icregexne (char8, text) returns bool as 'foo'
language 'internal';
create function char16icregexeq (char16, text) returns bool as 'foo'
language 'internal';
create function char16icregexne (char16, text) returns bool as 'foo'
language 'internal';
create function texticregexeq (text, text) returns bool as 'foo'
language 'internal';
create function texticregexne (text, text) returns bool as 'foo'
language 'internal';

-- add builtin functions that are new to 1.01

create operator = (leftarg = int4, rightarg = oid, procedure = int4eqoid);
create operator = (leftarg = oid, rightarg = int4, procedure = oideqint4);
create operator ~* (leftarg = char2, rightarg = text, procedure =
char2icregexeq);
create operator !~* (leftarg = char2, rightarg = text, procedure =
char2icregexne);
create operator ~* (leftarg = char4, rightarg = text, procedure =
char4icregexeq);
create operator !~* (leftarg = char4, rightarg = text, procedure =
char4icregexne);
create operator ~* (leftarg = char8, rightarg = text, procedure =
char8icregexeq);
create operator !~* (leftarg = char8, rightarg = text, procedure =
```

```
char8icregexne);
create operator ~* (leftarg = char16, rightarg = text, procedure =
char16icregexeq);
create operator !~* (leftarg = char16, rightarg = text, procedure =
char16icregexne);
create operator ~* (leftarg = text, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = text, rightarg = text, procedure =
texticregexne);
```

## E.65.2. Modifications

```
Incompatibilities:
 * 1.01 is backwards compatible with 1.0 database provided the user
   follow the steps outlined in the MIGRATION_from_1.0_to_1.01 file.
   If those steps are not taken, 1.01 is not compatible with 1.0 database.

Enhancements:
 * added PQdisplayTuples() to libpq and changed monitor and psql to use it
 * added NeXT port (requires SysVIPC implementation)
 * added CAST .. AS ... syntax
 * added ASC and DESC key words
 * added 'internal' as a possible language for CREATE FUNCTION
   internal functions are C functions which have been statically linked
   into the postgres backend.
 * a new type "name" has been added for system identifiers (table names,
   attribute names, etc.)  This replaces the old char16 type.   The
   of name is set by the NAMEDATALEN #define in src/Makefile.global
 * a readable reference manual that describes the query language.
 * added host-based access control.  A configuration file ($PGDATA/pg_hba)
   is used to hold the configuration data.  If host-based access control
   is not desired, comment out HBA=1 in src/Makefile.global.
 * changed regex handling to be uniform use of Henry Spencer's regex code
   regardless of platform.  The regex code is included in the distribution
 * added functions and operators for case-insensitive regular expressions.
   The operators are ~* and !~*.
 * pg_dump uses COPY instead of SELECT loop for better performance

Bug fixes:
 * fixed an optimizer bogue that was causing core dumps when
   functions calls were used in comparisons in the WHERE clause
 * changed all uses of getuid to geteuid so that effective uids are used
 * psql now returns non-zero status on errors when using -c
 * applied public patches 1-14
```

# E.66. Sortie 1.0

**Date de sortie :** 1995-09-05

## E.66.1. Modifications

```
Copyright change:
 * The copyright of Postgres 1.0 has been loosened to
be freely modifiable
   and modifiable for any purpose.  Please read the COPYRIGHT file.
   Thanks to Professor Michael Stonebraker for making this possible.

Incompatibilities:
 *  date formats have to be MM-DD-YYYY (or DD-MM-YYYY if you're using
```

```
   EUROPEAN STYLE).  This follows SQL-92 specs.
 *  "delimiters" is now a key word

Enhancements:
 *  sql LIKE syntax has been added
 *  copy command now takes an optional USING DELIMITER specification.
   delimiters can be any single-character string.
 *  IRIX 5.3 port has been added.
   Thanks to Paul Walmsley and others.
 *  updated pg_dump to work with new libpq
 *  \d has been added psql
   Thanks to Keith Parks
 *  regexp performance for architectures that use POSIX regex has been
   improved due to caching of precompiled patterns.
   Thanks to Alistair Crooks
 *  a new version of libpq++
   Thanks to William Wanders

Bug fixes:
 *  arbitrary userids can be specified in the createuser script
 *  \c to connect to other databases in psql now works.
 *  bad pg_proc entry for float4inc() is fixed
 *  users with usecreatedb field set can now create databases without
   having to be usesuper
 *  remove access control entries when the entry no longer has any
   privileges
 *  fixed non-portable datetimes implementation
 *  added kerberos flags to the src/backend/Makefile
 *  libpq now works with kerberos
 *  typographic errors in the user manual have been corrected.
 *  btrees with multiple index never worked, now we tell you they don't
   work when you try to use them
```

# E.67. Postgres95 Release 0.03

**Date de sortie :** 1995-07-21

## E.67.1. Modifications

```
Incompatible changes:
 * BETA-0.3 IS INCOMPATIBLE WITH DATABASES CREATED WITH PREVIOUS VERSIONS
   (due to system catalog changes and indexing structure changes).
 * double-quote (") is deprecated as a quoting character for string literals;
   you need to convert them to single quotes (').
 * name of aggregates (eg. int4sum) are renamed in accordance with the
   SQL standard (eg. sum).
 * CHANGE ACL syntax is replaced by GRANT/REVOKE syntax.
 * float literals (eg. 3.14) are now of type float4 (instead of float8 in
   previous releases); you might have to do typecasting if you dépend on it
   being of type float8.  If you neglect to do the typecasting and you assign
   a float literal to a field of type float8, you may get incorrect values
   stored!
 * LIBPQ has been totally revamped so that frontend applications
   can connect to multiple backends
 * the usesysid field in pg_user has been changed from int2 to int4 to
   allow wider range of Unix user ids.
 * the netbsd/freebsd/bsd o/s ports have been consolidated into a
   single BSD44_derived port.  (thanks to Alistair Crooks)
```

```
SQL standard-compliance (the following details changes that makes postgres95
more compliant to the SQL-92 standard):
 * the following SQL types are now built-in: smallint, int(eger), float, real,
   char(N), varchar(N), date and time.

   The following are aliases to existing postgres types:
               smallint -> int2
               integer, int -> int4
               float, real  -> float4
   char(N) and varchar(N) are implemented as truncated text types. In
   addition, char(N) does blank-padding.
 * single-quote (') is used for quoting string literals; '' (in addition to
   \') is supported as means of inserting a single quote in a string
 * SQL standard aggregate names (MAX, MIN, AVG, SUM, COUNT) are used
   (Also, aggregates can now be overloaded, i.e. you can define your
   own MAX aggregate to take in a user-defined type.)
 * CHANGE ACL removed. GRANT/REVOKE syntax added.
   - Privileges can be given to a group using the "GROUP" key word.
        For exemple:
                GRANT SELECT ON foobar TO GROUP my_group;
        The key word 'PUBLIC' is also supported to mean all users.

        Privileges can only be granted or revoked to one user or group
        at a time.

        "WITH GRANT OPTION" is not supported.  Only class owners can change
        access control
   - The default access control is to to grant users readonly access.
     You must explicitly grant insert/update access to users.  To change
     this, modify the line in
               src/backend/utils/acl.h
     that defines ACL_WORLD_DEFAULT

Bug fixes:
 * the bogue where aggregates of empty tables were not run has been fixed. Now,
   aggregates run on empty tables will return the initial conditions of the
   aggregates. Thus, COUNT of an empty  table will now properly return 0.
   MAX/MIN of an empty table will return a row of value NULL.
 * allow the use of \; inside the monitor
 * the LISTEN/NOTIFY asynchronous notification mechanism now work
 * NOTIFY in rule action bodies now work
 * hash indexes work, and access methods in general should perform better.
   creation of large btree indexes should be much faster.  (thanks to Paul
   Aoki)

Other changes and enhancements:
 * addition of an EXPLAIN statement used for explaining the query exécution
   plan (eg. "EXPLAIN SELECT * FROM EMP" prints out the exécution plan for
   the query).
 * WARN and NOTICE messages no longer have timestamps on them. To turn on
   timestamps of error messages, uncomment the line in
   src/backend/utils/elog.h:
        /* define ELOG_TIMESTAMPS */
 * On an access control violation, the message
        "Either no such class or insufficient privilege"
   will be given.  This is the same message that is returned when
   a class is not found.  This dissuades non-privileged users from
   guessing the existence of privileged classes.
 * some additional system catalog changes have been made that are not
   visible to the user.

libpgtcl changes:
```

```
 * The -oid option has been added to the "pg_result" tcl command.
   pg_result -oid returns oid of the last row inserted.   If the
   last command was not an INSERT, then pg_result -oid returns "".
 * the large object interface is available as pg_lo* tcl commands:
   pg_lo_open, pg_lo_close, pg_lo_creat, etc.

Portability enhancements and New Ports:
 * flex/lex problems have been cleared up.  Now, you should be able to use
   flex instead of lex on any platforms.  We no longer make assumptions of
   what lexer you use based on the platform you use.
 * The Linux-ELF port is now supported.  Various configuration have been
   tested:  The following configuration is known to work:
        kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
   with everything in ELF format,

New utilities:
 * ipcclean added to the distribution
   ipcclean usually does not need to be run, but if your backend crashes
   and leaves shared memory segments hanging around, ipcclean will
   clean them up for you.

New documentation:
 * the user manual has been revised and libpq documentation added.
```

# E.68. Postgres95 Release 0.02

**Date de sortie :** 1995−05−25

## E.68.1. Modifications

```
Incompatible changes:
 * The SQL statement for creating a database is 'CREATE DATABASE' instead
   of 'CREATEDB'. Similarly, dropping a database is 'DROP DATABASE' instead
   of 'DESTROYDB'. However, the names of the executables 'createdb' and
   'destroydb' remain the same.

New tools:
 * pgperl - a Perl (4.036) interface to Postgres95
 * pg_dump - a utility for dumping out a postgres database into a
        script file containing query commands. The script files are in a ASCII
        format and can be used to reconstruct the database, even on other
        machines and other architectures. (Also good for converting
        a Postgres 4.2 database to Postgres95 database.)

The following ports have been incorporated into postgres95-beta-0.02:
 * the NetBSD port by Alistair Crooks
 * the AIX port by Mike Tung
 * the Windows NT port by Jon Forrest (more stuff but not done yet)
 * the Linux ELF port by Brian Gallew

The following bogues have been fixed in postgres95-beta-0.02:
 * new lines not escaped in COPY OUT and problem with COPY OUT when first
   attribute is a '.'
 * cannot type return to use the default user id in createuser
 * SELECT DISTINCT on big tables crashes
 * Linux installation problems
 * monitor doesn't allow use of 'localhost' as PGHOST
 * psql core dumps when doing \c or \l
 * the "pgtclsh" target missing from src/bin/pgtclsh/Makefile
```

```
* libpgtcl has a hard-wired default port number
* SELECT DISTINCT INTO TABLE hangs
* CREATE TYPE doesn't accept 'variable' as the internallength
* wrong result using more than 1 aggregate in a SELECT
```

# E.69. Postgres95 Release 0.01

**Date de sortie :** 1995−05−01

Initial release.

# Annexe F. Dépôt CVS

Le code source de PostgreSQL est stocké et géré en utilisant le système de gestion de codes CVS.

Au moins deux méthodes, CVS anonyme et CVSup, permettent d'extraire de CVS l'arborescence du code source du serveur PostgreSQL vers votre serveur local.

## F.1. Obtenir les sources via CVS anonyme

Si vous voulez mettre régulièrement à jour vos sources, vous pouvez les prendre de notre serveur CVS, puis utiliser CVS pour récupérer les mises à jour de temps en temps.

**CVS anonyme**

1. Vous aurez besoin d'une copie locale de CVS (Concurrent Version Control System), que vous pouvez obtenir depuis http://www.nongnu.org/cvs/ (le site officiel avec la dernière version) ou depuis n'importe quel site d'archive GNU (souvent peu à jour). Nous recommandons la version 1.10 ou la version la plus récente. Beaucoup de systèmes ont une version récente de cvs installé par défaut.
2. Connectez−vous une première fois au serveur CVS :

   ```
   cvs -d :pserver:anoncvs@anoncvs.postgresql.org:/projects/cvsroot login
   ```

   Un mot de passe vous est demandé. Vous pouvez entrer ce que vous voulez, sauf une chaîne vide.

   Vous n'avez besoin de faire ceci qu'une seule fois car le mot de passe est sauvegardé dans le fichier `.cvspass` de votre répertoire personnel.
3. Récupérez les sources de PostgreSQL :

   ```
   cvs -z3 -d :pserver:anoncvs@anoncvs.postgresql.org:/projects/cvsroot co -P pgsql
   ```

   Ceci installe les sources de PostgreSQL dans un sous−répertoire `pgsql` de votre répertoire courant.

   > **Note :** Si vous avez une connexion rapide à Internet, vous n'avez peut−être pas besoin de l'option `-z3`, qui demande à CVS de compresser les données à transférer avec gzip. À la vitesse d'un modem, cela permet un gain de temps substantiel.

   Cette extraction initiale est un peu plus lente que de simplement télécharger un fichier `tar.gz`; attendez vous à ce qu'elle prenne 40 minutes environ si vous avez un modem à 28,8 Kbps. L'avantage de CVS n'apparaîtra que plus tard, lorsque vous voudrez mettre à jour les fichiers.
4. Lorsque vous voulez mettre à jour vos sources CVS avec la dernière version, faites `cd` dans le sous−répertoire `pgsql` et lancez la commande

   ```
   $ cvs -z3 update -d -P
   ```

   Ceci ne récupère que les modifications qui ont eu lieu depuis la dernière fois que vous avez fait une mise à jour. La mise à jour ne dure généralement que quelques minutes, même avec un simple modem.

5. Vous pouvez économiser quelques saisies en vous faisant un fichier `.cvsrc` dans votre répertoire personnel ($HOME) en lui ajoutant :

```
cvs -z3
update -d -P
```

Ceci ajoute l'option `-z3` à toutes les commandes cvs, et les options `-d` et `-P` aux mises à jour par cvs. Vous n'avez plus qu'à lancer

```
$ cvs update
```

pour mettre à jour vos fichiers.

| **Attention** |
| --- |
| Certaines versions anciennes de CVS ont un problème qui fait que tous les fichiers extraits ont des droits en écriture pour tout le monde. Si vous vous apercevez que cela vous arrive, vous pouvez faire quelque chose comme :<br><br>`$ chmod -R go-w pgsql`<br><br>pour remettre les droits d'aplomb. Ce problème a été corrigé dans la version 1.9.28 de CVS. |

CVS peut faire de nombreuses autres choses, comme retrouver des versions antérieures des sources de PostgreSQL plutôt que les dernières versions de développement. Pour plus d'informations, consultez le manuel de CVS ou bien lisez la documentation en ligne à http://www.nongnu.org/cvs/.

## F.2. Organisation de l'arbre CVS

**Auteur :** Écrit par Marc G. Fournier (<scrappy@hub.org>) le 05/11/1998.

La commande `cvs checkout` a un paramètre, `-r`, qui permet d'obtenir une révision particulière d'un module. Ce paramètre permet par exemple de retrouver les fichiers source de la version 6_4 du module 'tc' à tout moment dans le futur.

```
$ cvs checkout -r REL6_4 tc
```

C'est en particulier utile si quelqu'un pense qu'il y a un problème dans cette version, mais que vous ne trouvez pas d'erreur dans la version courante.

**Astuce :** Vous pouvez aussi extraire un module tel qu'il était à une date donnée en utilisant l'option `-D`.

Lorsque vous marquez plus d'un fichier avec une marque particulière, vous pouvez vous représenter la marque comme << une courbe tracée dans la matrice des noms de fichiers et des numéros de révision >>. Supposons que nous ayons 5 fichiers avec les révisions suivantes :

```
fichier1 fichier2 fichier3 fichier4 fichier5

1.1      1.1      1.1      1.1  /---1.1*    <-*-  MARQUE
```

```
1.2*-    1.2      1.2    --1.2*-
1.3  \-- 1.3*-    1.3   /  1.3
1.4           \   1.4  /   1.4
               \--1.5*-    1.5
                  1.6
```

alors la marque `MARQUE` référence fichier1−1.2, fichier2−1.3, etc.

> **Note :** Pour créer une branche de version, à part l'option `-b` ajoutée sur la ligne de commande, c'est la même chose.

Ainsi, pour créer la version 6.4, j'ai fait :

```
$ cd pgsql
$ cvs tag -b REL6_4
```

ce qui créera la marque et la branche pour l'arbre de la version.

Pour ceux qui ont un accès CVS, il est simple de créer des répertoires séparés pour chaque version. D'abord, créez deux répertoires, VERSION et COURANT, afin de ne pas les mélanger. Puis :

```
cd RELEASE
cvs checkout -P -r REL6_4 pgsql
cd ../CURRENT
cvs checkout -P pgsql
```

Cela crée deux arbres de répertoires, `VERSION/pgsql` et `COURANT/pgsql`. À partir de ce moment, CVS gardera trace de quel arbre de référence est dans quel répertoire, et permettra des mises à jour indépendantes de chaque arbre.

Si vous ne travaillez *que* sur l'arbre `CURRENT`, faites tout ce qui est indiqué jusqu'à ce que nous ayons commencé à marquer les branches de versions.

Après avoir fait l'extraction initiale d'une branche

```
$ cvs checkout -r REL6_4
```

tout ce que vous faites dans ce répertoire est limité à cette branche. Si vous appliquez une correction à cette structure de répertoires et faites un

```
cvs commit
```

dans ce répertoire, la correction est appliquée à cette branche et *seulement* à cette branche.

## F.3. Obtenir les sources via CVSup

Une alternative au CVS anonyme pour obtenir l'arbre source de PostgreSQL est CVSup. CVSup a été développé par John Polstra (<jdp@polstra.com>) pour distribuer des arbres de référence CVS et d'autres

arborescences de fichiers pour le projet FreeBSD.

Un des avantages majeurs de CVSup est qu'il peut répliquer fiablement la *totalité* du référentiel CVS sur votre système local, ce qui permet un accès rapide aux opérations CVS comme `log` ou `diff`. Autre avantage, la synchronisation efficace avec le serveur PostgreSQL grâce à un protocole de flux de transfert qui ne transmet que les différences depuis la dernière mise à jour.

## F.3.1. Préparer un système client CVSup

Deux groupes de répertoires sont nécessaires à CVSup : un répertoire contenant le référentiel CVS local (ou simplement un groupe de répertoires si vous ne récupérez qu'un extrait plutôt que le référentiel complet ; voir plus loin) et une zone dans laquelle CVSup enregistre ses propres données. Ces deux groupes de répertoires peuvent coexister dans la même arborescence.

Décidez où vous souhaitez garder votre copie locale du référentiel CVS. Sur un de nos systèmes, nous avons récemment créé un référentiel dans `/home/cvs/`, mais nous avions jusque là utilisé `/opt/postgres/cvs/` comme arbre de développement pour PostgreSQL. Si vous souhaitez placer votre référentiel dans `/home/cvs/`, alors ajoutez

```
setenv CVSROOT /home/cvs
```

dans votre fichier `.cshrc`, ou une ligne similaire dans votre `.bashrc` ou dans votre `.profile`, en fonction de votre interpréteur de commandes (shell).

La zone de référentiel de cvs doit être initialisée. Une fois que `CVSROOT` est initialisée, il suffit de faire :

```
$ cvs init
```

après quoi vous devriez au moins voir un répertoire `CVSROOT` en listant le répertoire `CVSROOT` :

```
$ ls $CVSROOT
CVSROOT/
```

## F.3.2. Utiliser un client CVSup

Vérifiez que cvsup est dans votre chemin. Sur la plupart des systèmes, cela se fait en tapant

```
which cvsup
```

Puis, lancez simplement cvsup en faisant :

```
$ cvsup -L 2 postgres.cvsup
```

où `-L 2` active certains messages de statut pour vous permettre de suivre le progrès pour la mise à jour et *`postgres.cvsup`* est la chemin et le nom que vous avez donné à votre fichier de configuration CVSup.

Voici un fichier de configuration CVSup modifié pour une installation spécifique, et maintient un référentiel CVS local complet.

```
# Ce fichier représente le fichier de distribution CVSup standard
# pour le projet de SGBDRO PostgreSQL
# Modifié par lockart@fourpalms.org 1997-08-28
# - Pointe vers mon arborescence de référence locale
# - Extrait le référentiel CVS complet, pas seulement la
#   dernière version
#
#Valeurs par défaut qui s'appliquent à toutes les collections
*default host=cvsup.postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
# Activez la ligne suivante pour récupérer la dernière version
#*default tag=.
# Activez la ligne suivante récupérer ce qui est précisé en dessus ou
# par défaut à la date précisée en dessous
#*default date=97.08.29.00.00.00

# répertoire de base où CVSup stocke ses fichiers 'marque page'
# créera un sous répertoire sup/
#*default base=/opt/postgres # /usr/local/pgsql
*default base=/home/cvs

# répertoire préfixe où CVSup stocke les distributions.
*default prefix=/home/cvs

# Distribution complète, avec tout ce qui est en dessous
pgsql

# distributions partielles
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

Si vous indiquez `repository` au lieu de `pgsql` dans la configuration précédente, vous obtiendrez une copie complète du dépôt sur cvsup.postgresql.org, incluant son répertoire `CVSROOT`. Si vous faites cela, vous voudrez probablement exclure ces fichiers du répertoire que vous voulez modifier localement, en utilisant un fichier nommé refuse. Par exemple, pour la configuration précédente, vous auriez pu placer ceci dans `/home/cvs/sup/repository/refuse` :

```
CVSROOT/config*
CVSROOT/commitinfo*
CVSROOT/loginfo*
```

Voir les pages man de CVSup pour savoir comment utiliser les fichiers refuse.

Ce qui suit est une suggestion de fichier de configuration CVSup issu du site ftp de PostgreSQL, qui ne récupère que la version courante :

```
# Ce fichier représente la distribution CVSup standard pour le
# projet de SGBDRO PostgreSQL.
#
# Valeurs par défaut qui d'appliquent à toutes les collections.
*default host=cvsup.postgresql.org
```

```
*default compress
*default release=cvs
*default delete use-rel-suffix
*default tag=.

# répertoire de base où CVSup stocke ses fichiers 'marque page'
*default base=/usr/local/pgsql

# répertoire préfixe où CVSup stocke les distributions.
*default prefix=/usr/local/pgsql

# Distribution complète, avec tout ce qui est en dessous
pgsql

# distributions partielles
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

## F.3.3. Installer CVSup

CVSup est disponible sous forme de fichiers source, de binaires pré−compilés ou de RPM Linux. Il est beaucoup plus simple d'utiliser les binaires plutôt que de compiler les sources, principalement parce que cela nécessite le compilateur Modula−3, qui est très puissant mais volumineux.

**Installation de CVSup à partir des fichier binaires**

Vous pouvez utiliser les binaires pré−compilés si vous avez une plate−forme pour laquelle les binaires sont postés sur le site ftp de PostgreSQL, vous pouvez ou si avez FreeBSD, pour lequel CVSup est disponible comme << portage >>.

> **Note :** CVSup a été initialement développé pour distribuer l'arbre des sources de FreeBSD. Il est disponible comme << portage >>, et pour ceux qui ont FreeBSD, si cela n'explique pas suffisamment comment obtenir et installer CVSup, merci d'ajouter une procédure ici.

Au moment de l'écriture de ce chapitre, des binaires sont disponibles pour : Alpha/Tru64, ix86/xBSD, HPPA/HP−UX 10.20, MIPS/IRIX, ix86/linux−libc5, ix86/linux−glibc, Sparc/Solaris et Sparc/SunOS.

1. Récupérez l'archive tar des binaires cvsup (cvsupd n'est pas nécessaire pour être un client) approprié pour votre plate−forme.
    a. Si vous avez FreeBSD, installez le portage CVSup.
    b. Si vous avez une autre plate−forme, vérifiez et télécharger le binaire approprié du site ftp PostgreSQL.
2. Vérifiez dans l'archive tar les contenus et la structure de répertoires. Pour le tar Linux au moins, le binaire statique et la page de manuel sont inclues sans répertoires.
    a. Si le binaire est au plus haut niveau du fichier tar, alors il suffit d'extraire le fichier tar dans le répertoire cible :

```
$ cd /usr/local/bin
$ tar zxvf /usr/local/src/cvsup-16.0-linux-i386.tar.gz
$ mv cvsup.1 ../doc/man/man1/
```

b. S'il y a une structure de répertoires dans le fichier tar, alors extrayez le dans /usr/local/src et déplacez les binaires dans le répertoire approprié, comme indiqué ci−dessus.

3. Assurez−vous que les nouveaux binaires sont dans votre chemin.

```
$ rehash
$ which cvsup
$ set path=(chemin de cvsup $path)
$ which cvsup
/usr/local/bin/cvsup
```

## F.3.4. Installation à partir des sources

Installer CVSup n'est pas entièrement trivial, principalement parce que la plupart des systèmes auront besoin du compilateur Modula−3. Ce compilateur est disponible sous forme de RPM Linux, de paquetage FreeBSD ou de code source.

> **Note :** Une installation de Modula−3 avec les sources prend environ 200 Mo d'espace disque, qui redescendent ensuite à environ 50 Mo lorsque les sources sont supprimées.

**Installation sur Linux**

1. Installer Modula−3.

   a. Récupérer la distribution de Modula−3 à Polytechnique Montréal, qui maintient activement le code initialement développé par le centre de recherches système de DEC. La distribution PM3 sous forme de RPM fait environ 30 Mo compressée. Au moment de l'écriture de ce document, la version 1.1.10−1 s'installe sans problème sur RH−5.2, alors que la version 1.1.11−1 est apparemment prévue pour une autre version (RH−6.0?) et ne fonctionne pas sous RH−5.2.

   > **Astuce :** Cette distribution rpm spécifique a de *nombreux* fichiers RPM, si bien que vous voudrez sans doute les mettre dans un répertoire séparé.

   b. Installer les rpms Modula−3 :

   ```
   # rpm −Uvh pm3*.rpm
   ```

2. Décompresser la distribution CVSup :

```
# cd /usr/local/src
# tar zxf cvsup−16.0.tar.gz
```

3. Compiler la distribution cvsup, en supprimant l'interface graphique pour éviter d'utiliser les bibliothèques X11.

```
# make M3FLAGS="−DNOGUI"
```

   et si vous voulez construire un binaire statique pour l'utiliser sur des systèmes qui n'ont pas Modula−3 installé, essayez :

```
# make M3FLAGS="−DNOGUI −DSTATIC"
```

4. Installer la librairie construite précédemment :

```
# make M3FLAGS="-DNOGUI -DSTATIC" install
```

# Annexe G. Documentation

PostgreSQL fournit quatre formats de documentation de base :

- Texte brut, pour les information de pré−installation ;
- HTML, pour la lecture en ligne et les références ;
- PDF ou Postscript, pour l'impression ;
- les pages man (de manuel), pour la référence rapide.

De plus, un certain nombre de fichiers `README` pourront être trouvés à divers endroits de l'arbre des sources de PostgreSQL. Ils renseigneront l'utilisateur sur différents points d'implémentation.

La documentation HTML et les pages de manuel font parties de la distribution standard et sont installées par défaut. Les documents au format PDF et Postscript sont disponibles indépendemment par le biais du téléchargement.

## G.1. DocBook

Les sources de la documentation sont écrites en *DocBook*, qui est un langage semblable au HTML au premier abord. Ces deux langages sont des applications de *Standard Generalized Markup Language*, SGML, qui est essentiellement un langage décrivant d'autres langages. Dans ce qui suivra, les termes DocBook et SGML seront utilisés, ils ne sont cependant pas techniquement interchangeables.

DocBook permet à l'auteur de spécifier la structure et le contenu d'un document technique sans qu'il ait besoin de se soucier du détail de la présentation. Un style de document définit la manière dont le contenu sera rendu dans un des formats de sortie finaux. DocBook est maintenu par le groupe OASIS . Le site officiel de DocBook présente une bonne documentation d'introduction et de référence ainsi qu'un livre complet de chez O'Reilly disponible en ligne pour votre plaisir. Le projet de documentation FreeBSD utilise également DocBook et fournit également de bonnes informations, incluant un certain nombre de lignes directrices qu'il peut être bon de prendre en considération.

## G.2. Ensemble d'outils

Les outils suivants sont utilisés pour générer la documentation. Certains sont optionnels (comme mentionné).

DTD DocBook
: Il s'agit de la définition de DocBook elle−même. Nous utilisons actuellement la version 4.2. Vous ne pouvez pas utiliser des versions plus récentes ou plus anciennes. Notez qu'il existe également une version XML de DocBook — ne l'utilisez pas !

Les entités de caractère ISO 8879
: Celles−ci sont nécessaires à DocBook mais sont distribués à part car elles sont maintenues par l'ISO.

OpenJade
: C'est le paquetage de base pour le traitement de SGML. Il contient un analyseur SGML, un processeur DSSSL (qui est un programme permettant la conversion de documents SGML en d'autres formats en utilisant des feuilles de styles DSSSL), ainsi qu'un certain nombre d'autres outils. Jade est actuellement maintenu par le groupe OpenJade et non plus par James Clark.

Feuilles de styles DocBook DSSSL
: Celles−ci contiennent les instructions permettant la conversion des sources DocBook en d'autres

formats tels que le HTML.

Les outils DocBook2X
: Ce paquetage est utilisé pour créer les pages de manuel. Un certain nombre d'autres paquetages sont nécessaires pour le faire fonctionner. Pour plus d'informations, vérifiez sur le site web.

JadeTeX
: Si vous le souhaitez, vous pouvez installer JadeTeX qui s'appuie sur TeX en tant qu'outil de formatage pour Jade. JadeTeX est capable de créer des fichiers au formats Postscript ou PDF (pour ce dernier, il implante les signets).

  Cependant, une sortie JadeTeX est de qualité moindre par rapport à ce que vous pouvez obtenir d'une sortie RTF. Les principaux problèmes que l'on peut rencontrer se situent autour des tables et des éléments de placements verticaux et horizontaux. Il est à noter qu'il n'y a aucun recours de correction afin de corriger manuellement ces problèmes.

Nous avons pris le soin de documenter, ci−aprés, plusieurs types de méthodes d'installation pour les différents outils nécessaires au traitement de la documentation. Il peut exister d'autres types de distributions empaquetées de ces outils. Veuillez notifier tout changement de paquetage auprés de la liste de discussion de la documentation, nous tâcherons d'inclure ces informations ici−même.

## G.2.1. Installation RPM Linux

La plupart des fabriquants de distributions mettent à disposition des utilisateurs un ensemble complet de paquetages RPM pour le traitement de DocBook au sein de leur distribution. Lors de l'installation, recherchez une option << SGML >> ou les paquetages suivants : `sgml-common`, `docbook`, `stylesheets`, `openjade`. Vous aurez probablement besoin de `sgml-tools`. Si le fournisseur de la distribution ne permet pas de disposer de ceux−ci, vous devriez être capable d'utiliser des paquetages issus d'une autre distribution compatible.

## G.2.2. Installation pour FreeBSD

Le projet de documentation FreeBSD (FreeBSD Documentation Projetc) est lui−même un utilisateur intensif de DocBook, et c'est sans surprise que l'on retrouve en son sein un ensemble complet de << portages >> des outils de documentation sur FreeBSD. Les portages suivants devront être installés afin de produire la documentation sur FreeBSD.

- `textproc/sp`
- `textproc/openjade`
- `textproc/iso8879`
- `textproc/dsssl-docbook-modular`

Apparamment, il n'existe pas encore de port pour la DTD de DocBook V4.2 SGML. Vous devrez l'installer manuellement.

Vous pourrez également porter un intérêt particulier aux différents éléments de `/usr/ports/print` (`tex` ou `jadetex`).

Il est probable que les portages ne mettent pas à jour le fichier de catalogue général dans `/usr/local/share/sgml/catalog`. Assurez−vous que la ligne suivante y figure bien :

```
CATALOG "/usr/local/share/sgml/docbook/4.2/docbook.cat"
```

Si vous ne voulez pas éditer ce fichier, vous pouvez également modifier la variable d'environnement SGML_CATALOG_FILES en y mettant une liste de fichiers catalogues séparés par des caractères << deux points >>.

Vous pourrez trouver plus d'informations sur les outils dédiés à la documentation de FreeBSD dans les instructions du projet de documentation de FreeBSD.

## G.2.3. Paquetages Debian

Un ensemble complet de paquetages d'outils de documentation sont disponibles pour Debian GNU/Linux. Pour l'installer, utilisez simplement :

```
apt-get install jade
apt-get install docbook
apt-get install docbook-stylesheets
```

## G.2.4. Installation manuelle à partir des sources

L'installation manuelle des outils DocBook est quelque peu complexe. Il est donc préférable que vous utilisiez des paquetages pré−compilés. Nous ne décrivons ici que la mise en œuvre standard utilisant des répertoires d'installation standards et sans fonctionnalités particulières. Pour entrer dans les détails, nous vous recommandons d'étudier la documentation respective de chaque paquetage et de lire les document d'introduction à SGML.

### G.2.4.1. Installer OpenJade

1. L'installation d'OpenJade se fait par l'intermédiaire des outils de construction `./configure;make;make install` classique de GNU. Vous pourrez trouver des informations détaillées dans la distribution des sources d'OpenJade. En quelques mots :

   ```
   ./configure --enable-default-catalog=/usr/local/share/sgml/catalog
   make
   make install
   ```

   Assurez−vous de bien vous souvenir de l'endroit où vous placez le << catalogue par défaut >> ; vous en aurez besoin par la suite. Vous pouvez également vous passer de renseigner cette option. Dans ce cas, vous devrez définir la variable d'environnement SGML_CATALOG_FILES afin qu'elle pointe vers le bon fichier à chaque fois que vous lancerez jade. (Cette méthode est également possible si OpenJade est déjà installé et que vous souhaitez installer le reste de l'environnement de publication localement.)
2. Par ailleurs, vous devez installer les fichiers `dsssl.dtd`, `fot.dtd`, `style-sheet.dtd` et `catalog` du répertoire `dsssl` quelque part, par exemple dans `/usr/local/share/sgml/dsssl`. Il vous sera certainement plus facile de copier le répertoire entier.

   ```
   cp -R dsssl /usr/local/share/sgml
   ```
3. Enfin, créez le fichier `/usr/local/share/sgml/catalog` et ajoutez−y la ligne suivante :

   ```
   CATALOG "dsssl/catalog"
   ```

(Il s'agit d'un chemin relatif référençant le fichier installé dans l'étape 2. Assurez−vous de bien avoir renseigné ce chemin si vous avez utilisé un répertoire d'installation différent.)

### G.2.4.2. Installation du kit DTD de DocBook

1. Récupérez la distribution DocBook V4.2.
2. Créez le répertoire `/usr/local/share/sgml/docbook-4.2` et placez−y vous. (L'emplacement importe peu en fait mais celle−ci a le bénéfice d'être cohérente avec le schéma d'installation que nous vous proposons ici.)

   ```
   $ mkdir /usr/local/share/sgml/docbook-4.2
   $ cd /usr/local/share/sgml/docbook-4.2
   ```
3. Décompressez l'archive.

   ```
   $ unzip -a ...../docbook-4.2.zip
   ```

   (L'archive décompressera ses fichier dans le répertoire courant.)
4. Éditez le fichier `/usr/local/share/sgml/catalog` (ou ce que vous aurez dit à jade lors de l'installation) et placez−y une ligne similaire à celle−ci :

   ```
   CATALOG "docbook-4.2/docbook.cat"
   ```
5. Téléchargez l'archive contenant les entités de caractères ISO 8879, décompressez−la et placez les fichiers dans le même répertoire que celui des fichiers de DocBook.

   ```
   $ cd /usr/local/share/sgml/docbook-4.2
   $ unzip ...../ISOEnts.zip
   ```
6. Lancez la commande suivante dans le répertoire contenant les fichiers DocBook et ISO :

   ```
   perl -pi -e 's/iso-(.*).gml/ISO\1/g' docbook.cat
   ```

   (Cette opération permet de corriger les mélanges entre le fichier de catalogue de DocBook et les noms réels des fichiers contenant les entités de caractères ISO.)

### G.2.4.3. Installation des feuilles de style DSSSL DocBook

Pour installer les feuilles de style, décompressez et déballez la distribution et déplacez−la à un endroit convenable de votre aborescence comme, par exemple, `/usr/local/share/sgml`. (L'archive créera automatiquement un sous−répertoire.)

```
$ gunzip docbook-dsssl-1.xx.tar.gz
$ tar -C /usr/local/share/sgml -xf docbook-dsssl-1.xx.tar
```

L'entrée de catalogue communément admise dans `/usr/local/share/sgml/catalog` peut également être réalisée :

```
CATALOG "docbook-dsssl-1.xx/catalog"
```

Comme les feuilles de styles changent assez souvent et qu'il est parfois avantageux d'essayer des versions alternatives, PostgreSQL n'utilise pas cette entrée dans le catalogue. Regardez dans la Section G.2.5 pour tout renseignement sur la manière de sélectionner une feuille de style.

#### G.2.4.4. Intallation de JadeTeX

Pour installer et utiliser JadeTeX, vous devez disposer d'une installation fonctionnelle de TeX et de LaTeX2e, incluant également les paquetages d'outils de support, la bibliothèque graphics, Babel, les polices AMS et AMS−LaTeX, l'extension PSNFSS et le kit d'accompagnement de << 35 polices >>, le programme dvips permettant de générer du PostScript, le paquetage de macros fancyhdr, hyperref, minitoc, url et enfin ot2enc. Tous ceux−ci peuvent être trouvé sur le site web de CTAN . L'installation de base pour le système TeX va au−delà des buts visés par cette introduction. Des paquetages binaires devraient être disponibles pour tout système pouvant exécuter TeX.

Avant que vous soyez en mesure d'utiliser JadeTeX avec les sources de la documentation de PostgreSQL, vous devrez augmenter la taille des structures de données internes de TeX. Des explications plus détaillées sur JadeTeX pourront être trouvées sur les instructions d'installation de ce produit.

Une fois ceci terminé, vous pouvez installer JadeTeX :

```
$ gunzip jadetex-xxx.tar.gz
$ tar xf jadetex-xxx.tar
$ cd jadetex
$ make install
$ mktexlsr
```

Les deux dernières commandes doivent être exécutées en temps que root.

### G.2.5. Détection par `configure`

Avant de pouvoir générer la documentation, vous devrez lancer le script `configure` comme vous le feriez lors de la génération des programmes PostgreSQL eux−même. Vérifiez la sortie de l'exécution de ce script vers la fin, vous trouverez quelque chose de similaire à ce qui suit :

```
checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V4.2... yes
checking for DocBook stylesheets... /usr/lib/sgml/stylesheets/nwalsh-modular
checking for sgmlspl... sgmlspl
```

Si ni `onsgmls` ni `nsgmls` n'ont été trouvés, vous ne verrez pas les quatre dernières lignes. `onsgmls` fait partie du paquetage Jade. Si << DocBook V4.2 >> n'a pas été trouvé, cela signifie que le kit de DTD DocBook n'a pas été placé à l'endroit où jade peut le trouver ou que vous n'avez pas configuré les fichiers de catalogue correctement. Regardez les indications d'installation ci−dessus. Les feuilles de style DocBook sont recherchées dans un certain nombre d'endroits relativement standardisés, mais si vous les avez placées ailleurs, vous devrez modifier la variable d'environnement `DOCBOOKSTYLE` à cet emplacement et relancer `configure` juste aprés.

## G.3. Génération de la documentation

Une fois que tout est en place, placez−vous dans le répertoire `doc/src/sgml` et lancez une des commandes décrites dans les sections suivantes afin de générer la documentation. (Souvenez−vous bien d'utiliser la version GNU de make.)

### G.3.1. HTML

Pour générer la version HTML de la documentation, effectuez ce qui suit :

```
doc/src/sgml$ gmake html
```

Il s'agit également de la cible par défaut.

Lorsque la documentation HTML est générée, le processus produit également des informations de liaison pour les entrées de l'index. Ainsi, si vous souhaitez disposer d'un index des concepts à la fin de votre documentation, vous devrez produire une première fois la documentation HTML, puis relancer la production de la documentation avec le format que vous souhaitez obtenir.

Pour un côté plus pratique lors de la manipulation de la distribution finale de la documentation, l'ensemble des fichiers y compris la documentation HTML est stockée dans une archive tar qui est décompressée lors de l'installation. Pour créer le paquetage HTML, utilisez les commandes :

```
cd doc/src
gmake postgres.tar.gz
```

Dans la distribution de PostgreSQL, cette archive se trouve dans le répertoire `doc` et peut être installée grace à `gmake install`.

### G.3.2. Pages man (de manuels)

Afin de convertir les pages de références de la documentation DocBook en pages man nous utilisons docbook2man pour les convertir dans un format compatible avec *roff. Les pages man sont également distribuées sous la forme d'une archive tar, à l'instar de la version HTML. Pour créer le paquetage de pages man, utilisez les commandes :

```
cd doc/src
gmake man.tar.gz
```

qui résulteront en un fichier tar généré dans le répertoire `doc/src`.

Afin de générer des pages man de qualité, il sera peut−être nécessaire d'utiliser une version modifiée de l'utilitaire de conversion ou de faire des modification manuelles en post−production. Toutes les pages man doivent être manuellement vérifiées avant toute distribution.

### G.3.3. Sortie pour l'impression via JadeTex

Si vous souhaitez utiliser JadeTex pour produire une documentation à destination de l'impression, vous pouvez utiliser une des commandes suivantes :

- Pour créer une version DVI :

  ```
  doc/src/sgml$ gmake postgres.dvi
  ```
- Pour générer un fichier Postscript à partir du DVI :

  ```
  doc/src/sgml$ gmake postgres.ps
  ```
- Pour créer un PDF :

```
doc/src/sgml$ gmake postgres.pdf
```

(Bien entendu, vous pouvez également créer une version PDF à partir d'un document Postscript, mais si vous le générez directement, votre PDF bénéficiera des liens actifs et de fonctionnalités supérieures.)

## G.3.4. Version imprimable via RTF

Vous pouvez aussi créer une version imprimable de la documentation de PostgreSQL en la convertissant en RTF et en y appliquant des corrections de formatage en utilisant une suite de logiciels de bureau. En fonction des capacité de cette dernière, vous pourrez convertir la documentation aux formats Postscript ou PDF. La procédure ci−dessous décrit une telle procédure en utilisant Applixware.

> **Note :** Il apparaît que la version actuelle de la documentation de PostgreSQL génère quelques bogues au niveau d'OpenJade, dépassant notamment la taille maximale de traitement. Si le processus de génération de la version RTF reste suspendu un long moment et que la sortie garde une taille de 0, c'est que vous devez vous trouver face à ce problème. (Considérez cependant qu'une génération normale doit prendre 5 à 10 minutes, n'abandonnez donc pas trop tôt.)

**Nettoyage du RTF pour Applixware**

OpenJade oublie de spécifier un style par défaut pour le corps du texte. Dans le passé, ce problème non diagnostiqué a amené un processus très long pour la génération du sommaire. Néanmoins, avec un grande aide des gens de Applixware, le symptôme a été diagnostiqué et un contournement est disponible.

1. Produisez la version RTF en saisissant :

   ```
   doc/src/sgml$ gmake postgres.rtf
   ```
2. Réparez le fichier RTF afin de spécifier correctement tous les styles et en particulier le style par défaut (default). Si le document contient des sections `refentry`, vous devrez également supprimer les éléments de formatage qui relient le paragraphe précédent au paragraphe courant. À la place, il faudra relier le paragraphe précédent au paragraphe courant. Un utilitaire, `fixrtf`, est disponible dans `doc/src/sgml` afin de permettre ces corrections :

   ```
   doc/src/sgml$ ./fixrtf --refentry postgres.rtf
   ```

   Le script ajoute `{\s0 Normal;}` en temps que style de rang zéro dans le document. D'après Applixware, le standard RTF prohibe l'utilisation implicite d'un style de rang 0 alors que Microsoft Word est capable de gérer cette éventualité. Afin de réparer les sections `refentry`, le script remplace les balises `\keepn` par `\keep`.
3. Ouvrez un nouveau document dans Applixware Words puis importez le fichier RTF.
4. Générez une nouvelle table des matières en utilisant Applixware.

   a. Sélectionnez les lignes existantes de la table des matières, du premier caractère de la table au dernier caractère de la dernière ligne.
   b. Construisez une nouvelle table des matières en allant dans le menu Tools−>Book Building−>Create Table of Contents. Sélectionnez les trois premiers niveaux de titres pour l'inclusion dans la table des matières. Cela remplacera les lignes existantes importées du RTF en une table des matières issue d'Applixware.

c. Ajustez le formatage de la table des matières en utilisant le menu Format–>Style, en sélectionnant chacun des trois styles de la table des matières et en ajustant les indentations pour `First` et `Left`. En utilisant les valeurs suivantes :

| Style | Indentation de First (pouces) | Indentation de Left (pouces) |
|---|---|---|
| `TOC-Heading 1` | `0.4` | `0.4` |
| `TOC-Heading 2` | `0.8` | `0.8` |
| `TOC-Heading 3` | `1.2` | `1.2` |

5. Travaillez la totalité du document en :

   - Ajustant les sauts de page.
   - Ajustant la largeur des colonnes des tables.

6. Remplacez les numéros de pages justifiés à droite dans les portions d'exemple et de figures de la table des matières avec les bonnes valeurs. Cela ne prend que quelques minutes.
7. Supprimez la section d'index du document si elle est vide.
8. Régénérez et ajustez la table des matières.

   a. Sélectionnez un champ de la table des matières.
   b. Sélectionnez le menu Tools–>Book Building–>Create Table of Contents.
   c. Déliez la table des matières en sélectionnant le menu Tools–>Field Editing–>Unprotect.
   d. Supprimez la première ligne de la table des matières, qui est un des champs de la table elle–même.

9. Sauvegardez le document au format natif de Applixware Words afin de pouvoir faire des modifications facilement dans le futur.
10. << Imprimez >> le document dans un fichier au format Postscript.

## G.3.5. Fichiers texte

Plusieurs fichiers textes sont distribués comme fichiers pour la lecture durant le processus d'installation. Le fichier `INSTALL` correspond au Chapitre 14, avec quelques changements mineurs à mettre sur le compte d'un contexte différent. Pour recréer le fichier, placez–vous dans le répertoire `doc/src/sgml` et entrez la commande suivante **`gmake INSTALL`**. Ceci créera un fichier `INSTALL.html` que vous pourrez sauvegarder au format texte avec Netscape Navigator et remplacer le fichier existant. Netscape semble fournir la meilleure qualité pour la conversion du HTML en texte (comparé à lynx et à w3m).

Le fichier `HISTORY` peut être créé de manière similaire en utilisant la commande **`gmake HISTORY`**. Pour le fichier `src/test/regress/README`, la commande est **`gmake regress_README`**.

## G.3.6. Vérification syntaxique

Fabriquer la documentation peut prendre beaucoup de temps. Il existe cependant une méthode ne prenant que quelques secondes et permettant juste de vérifier que la syntaxe est correcte dans les fichiers de documentation :

```
doc/src/sgml$ gmake check
```

# G.4. Écriture de la documentation

SGML et DocBook ne souffrent pas du foisonnement d'outils d'édition open−source. Le plus commun d'entre eux est l'éditeur Emacs/XEmacs qui dispose d'un mode d'édition approprié. Sur certains systèmes, ces outils sont fournis sous la forme d'une installation complète (un même paquetage ou ensemble).

## G.4.1. Emacs/PSGML

PSGML est le mode d'édition de documents SGML le plus répandu et le plus puissant. S'il est correctement configuré, il vous permettra d'utiliser Emacs pour insérer de balises et en vérifier la consistance. Vous pouvez également l'utiliser pour le HTML. Visitez le site web de PSGML pour le télécharger, avoir des informations sur l'installation et pour disposer d'une documentation détaillée.

Un point important à noter avec PSGML : son auteur suppose que votre répertoire principal pour la DTD SGML est `/usr/local/lib/sgml`. Si, comme c'est le cas dans les exemples de ce chapitre, vous utilisez le répertoire `/usr/local/share/sgml`, vous devrez modifier la variable d'environnement `SGML_CATALOG_FILES` ou vous pouvez personnaliser votre installation de PSGML (son manuel vous renseignant sur la manière de le faire).

Ajoutez les lignes suivantes dans votre fichier d'environnement `~/.emacs` (en ajustant les noms des répertoires pour qu'il convienne à votre système de fichiers) :

```
; ********** pour le mode SGML (psgml)

(setq sgml-omittag t)
(setq sgml-shorttag t)
(setq sgml-minimize-attributes nil)
(setq sgml-always-quote-attributes t)
(setq sgml-indent-step 1)
(setq sgml-indent-data t)
(setq sgml-parent-document nil)
(setq sgml-default-dtd-file "./reference.ced")
(setq sgml-exposed-tags nil)
(setq sgml-catalog-files '("/usr/local/share/sgml/catalog"))
(setq sgml-ecat-files nil)

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

et dans le même fichier ajoutez l'entrée pour le SGML dans la définition (existante) pour `auto-mode-alist` :

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
   ))
```

Actuellement, chaque fichier source SGML contient le bloc suivant à la fin du fichier :

```
<!-- Conservez ce commentaire à la fin du fichier
Local variables:
mode: sgml
sgml-omittag:t
sgml-shorttag:t
sgml-minimize-attributes:nil
sgml-always-quote-attributes:t
```

```
sgml-indent-step:1
sgml-indent-data:t
sgml-parent-document:nil
sgml-default-dtd-file:"./reference.ced"
sgml-exposed-tags:nil
sgml-local-catalogs:("/usr/lib/sgml/catalog")
sgml-local-ecat-files:nil
End:
-->
```

Ce bloc vous permet de définir un certain nombre de paramètres du mode d'édition même si vous n'avez pas configuré votre fichier `~/.emacs`, mais il est un peu malheureux car, si vous avez suivi les instructions d'installation ci–dessus, le chemin de catalogue ne correspond pas à l'endroit où vous l'aurez mis. Si c'est le cas, vous drevrez faire en sorte de ne pas prendre en compte les variables locales :

```
(setq inhibit-local-variables t)
```

La distribution PostgreSQL inclut un fichier DTD pré–traité contenant des définitions appelé `reference.ced`. Vous trouverez certainement plus confortable d'insérer une déclaration `DOCTYPE` lorsque vous éditerez un fichier avec PSGML. Si par exemple vous travaillez sur ce fichier source qui est un chapitre annexe, vous pourrez spécifier que le document est une instance << appendix >> d'un document DocBook en ajoutant en première ligne ceci :

```
<!DOCTYPE appendix PUBLIC "-//OASIS//DTD DocBook V4.2//EN">
```

Ceci signifie que, quelque soit l'outil lisant le SGML, il le fera correctement et que je peux vérifier la validité du document avec `nsgmls -s docguide.sgml`. (Cependant, vous devrez supprimer la ligne avant de fabriquer le document complet à partir des différents fichiers.)

### G.4.2. Autres modes pour Emacs

GNU Emacs dispose d'un mode SGML natif différent qui n'est pas aussi puissant que PSGML, mais qui a le bénéfice d'être plus simple et léger. Il offre la coloration syntaxique (en mode font lock), ce qui peut être relativement utile.

Norm Walsh propose un mode majeur spécifique à DocBook qui dispose également de la coloration syntaxique et d'un certain nombre de fonctions permettant de réduire le temps de saisie.

## G.5. Guide des styles

### G.5.1. Pages de références

Les pages de références doivent obéir à des règles construction standard. Ainsi, un utilisateur pourra trouver l'information qu'il souhaite de manière plus rapide et cela encourage également les rédacteurs à documenter tous les aspects relatifs à une commande. Cette consistance n'est pas uniquement souhaitée pour les pages de références PostgreSQL, mais également pour les pages de références fournies par le système d'exploitation et les autres paquetages. C'est pour cela que les règles suivantes ont été développées. Elles sont pour la plupart consistantes avec d'autres règles similaires établies par les différents systèmes d'exploitation.

Les pages de référence qui décrivent les commandes exécutables doivent contenir les sections suivantes dans l'ordre. Les sections qui ne sont pas applicables peuvent être omises. Des sections de premier niveau ne

pourront être utilisées que dans des circonstances particulières ; dans la plupart des cas, les informations devant y figurer pourront appartenir à la section << Usage >>.

Nom
: Cette section est générée automatiquement. Elle contient le nom de la commande et une courte phrase résumant sa fonctionnalité.

Synopsis
: Cette section contient le schéma syntaxique de la commande. Le synopsis ne doit en général pas lister toutes les options de la commande, cela sera fait juste au dessous. À la place, il est important de lister les composantes majeures de la ligne de commande comme par exemple où les fichiers d'entrée et sortie se trouvent.

Description
: Plusieurs paragraphes décrivant ce que permet de faire la commande.

Options
: Une liste décrivant chacune des options de la ligne de commande. S'il y a beaucoup d'options, il sera possible d'utiliser des sous−sections.

Code de sortie
: Si le programme utilise 0 lors d'un succès et une valeur différente de 0 pour un échec, vous devez le documenter. S'il y a une signification particulière au code de retour différent de zéro, décrivez−les ici.

Usage
: Décrivez ici tout subtilité ou interface de lancement du programme. Si le programme n'est pas interactif, cette section peut être omise. Dans les autres cas, cette section est un fourre−tout pour les fonctionnalités disponibles lors de l'utilisation du programme. Utilisez des sous−sections si cela est approprié.

Environment
: Listez ici toute variable d'environnement qui pourrait être utilisée. Efforcez−vous de ne rien omettre même si des variables vous semblent triviales au premier coup d'œil comme `SHELL` qui pourrait être d'un quelconque intérêt pour l'utilisateur.

Fichiers
: Listez tout fichier que le programme pourrait accéder, même implicitement. Ne listez pas les fichiers d'entrée ou de sortie ayant été spécifiés en ligne de commande, mais plutôt les fichiers de configuration, etc.

Diagnostiques
: Expliquez ici tout sortie inhabituelle que le programme pourrait créer. Retenez−vous cependant de lister tous les messages d'erreur possibles. Cela représente beaucoup de travail et n'est pratiquement jamais utilisé. Mais si, par exemple, les messages d'erreurs ont un format particulier pouvant être traité par l'utilisateur, cette section est le bon endroit pour en décrire la composition.

Notes
: Tout ce qui ne peut être contenu dans les autres sections peut être placé ici. Plus particulièrement, les points concernant les bogues, les largesses d'implémentations, la sécurité et la compatibilité pourront être abordés ici.

Exemples
: Les exemples.

Historique
: S'il y a eu des échéances majeures dans l'histoire du programme vous pouvez les lister ici. Typiquement, cette section peut être omise.

Voir aussi
: Les références croisées, listées dans l'ordre suivant : pages de référence vers d'autres commandes PostgreSQL, pages de références de commmandes SQL de PostgreSQL, citation des manuels PostgreSQL, autres pages de référence (c'est−à−dire système d'exploitation, autres paquetages), autres documentations. Les éléments dans un même groupe sont listés dans l'ordre alphabétique.

Les pages de référence décrivant les commandes SQL doivent contenir les sections suivantes : Nom, Synopsis, Description, Paramètres, Sorties, Notes, Exemples, Compatibilité, Historique, Voir aussi. La section Paramètres est identique à la section Options mais elle offre plus de libertés sur lesquelles des clauses peuvent être listées. La section Sorties n'est seulement nécessaire que si la commande renvoie quelque chose d'autre qu'un complément de commande par défaut. La section Compatibilité doit expliquer en quelle mesure une commande se conforme au(x) standard(s) SQL, ou avec quel autre système de gestion de base de données elle est compatible. La section Voir aussi des commandes SQL doit lister les commandes SQL avant de faire des références croisées vers les programmes.

---

# Annexe H. Projets externes

PostgreSQL est un projet logiciel complexe, et le gérer est difficile. Nous avons trouvé que beaucoup d'ajouts à PostgreSQL pouvaient être développés plus efficacement séparément du projet principal. Les projet séparés peuvent avoir leurs propres équipes de développement, leurs listes de discussion, leurs outils de gestion de bogues et leurs sorties de versions. Bien que cette indépendence rend le développement plus simple, il rend plus difficile le travail des utilisateurs. Ils doivent partir en chasse pour les améliorations de la base de données pour trouver une solution à leurs besoins. Cette section décrit quelques ajouts développés en externe parmi les plus populaires et vous guide pour les trouver.

Beaucoup de projets relatifs à PostgreSQL sont disponibles soit sur GBorg (http://gborg.postgresql.org) soit sur pgFoundry (http://pgfoundry.org). Il existe d'autres projets en relation avec PostgreSQL hébergés ailleurs mais vous devrez les chercher sur Internet.

## H.1. Interfaces développés en externe

PostgreSQL inclut très peu d'interfaces avec la distribution de base. libpq en fait partie parce qu'elle est l'interface C principale et que beaucoup d'autres interfaces sont construites à partir d'elle. ecpg est intégré parce qu'elle est liée à la grammaire côté serveur et donc est très dépendante de la version de la base de données. Toutes les autres interfaces sont des projets indépendents et doivent être installés séparément.

Voici quelques unes des interfaces les plus populaires :

psqlODBC
: Ceci est l'interface la plus commune pour les applications Windows.

pgjdbc
: Une interface JDBC.

Npgsql
: Une interface .Net pour les applications Windows plus récentes.

libpqxx
: Une interface C++ encore plus récente.

libpq++
: Une interface C++ plus ancienne.

pgperl
: Une interface Perl avec une API similaire à libpq.

DBD−Pg
: Une interface Perl qui utilise l'API standard de DBD.

pgtclng
: Une version plus récente de l'interface Tcl.

pgtcl
: La version originale de l'interface Tcl.

PyGreSQL
: Une bibliothèque d'interface Python.

Ils sont tous disponibles sur GBorg (http://gborg.postgresql.org) ou pgFoundry (http://pgfoundry.org).

# H.2. Extensions

PostgreSQL a été conçu depuis début pour être extensible. Pour cette raison, les extensions chargées dans la base de données fonctionnent comme les autres fonctionnalités fournies avec la base de données. Le répertoire `contrib/` donné avec le code source contient un grand nombre d'extensions. Le fichier `README` dans ce répertoire contient un résumé. Ils incluent des outils de conversion, un indexage de texte complet, des outils XML et d'autres types de données et méthodes d'indexage. Les autres extensions sont développées indépendamment comme PostGIS. Même mes solutions de réplication PostgreSQL sont développées en externe. Par exemple, Slony−I est une solution de réplication maître/esclave populaire qui a été développé indépendamment du projet principal.

There are several administration tools available for PostgreSQL. The most popular is pgAdmin, and there are several commercially available ones.

---

# Bibliographie

Références sélectionnées et lectures sur SQL et PostgreSQL.

Quelques livres blancs et rapports techniques réalisés par l'équipe d'origine de développement de POSTGRES sont disponibles sur le site web du département des sciences informatiques de l'université de Californie

## Livres de référence sur SQL


Judith Bowman, Sandra Emerson, et Marcy Darnovsky, *The Practical SQL Handbook: Using Structured Query Language*, Third Edition, Addison−Wesley, ISBN 0−201−44787−8, 1996.

C. J. Date et Hugh Darwen, *A Guide to the SQL Standard: A user's guide to the standard database language SQL*, Fourth Edition, Addison−Wesley, ISBN 0−201−96426−0, 1997.

C. J. Date, *An Introduction to Database Systems*, Volume 1, Sixth Edition, Addison−Wesley, 1994.

Ramez Elmasri et Shamkant Navathe, *Fundamentals of Database Systems*, 3rd Edition, Addison−Wesley, ISBN 0−805−31755−4, August 1999.

Jim Melton et Alan R. Simon, *Understanding the New SQL: A complete guide*, Morgan Kaufmann, ISBN 1−55860−245−3, 1993.

Jeffrey D. Ullman, *Principles of Database and Knowledge: Base Systems*, Volume 1, Computer Science Press, 1988.


## Documentation spécifique sur PostgreSQL


Stefan Simkovics, *Enhancement of the ANSI SQL Implementation of PostgreSQL*, Département des systèmes d'informations, université de technologie de Vienne, 29 novembre 1998.

Discute de l'histoire et de la syntaxe de SQL, et décrit les ajouts des constructions `INTERSECT` et `EXCEPT` dans PostgreSQL. Écrit pour une thèse avec le support du professeur d'université Georg Gottlob et de son assistante Katrin Seyr à l'université de technologie de Vienne.

A. Yu et J. Chen, Le groupe POSTGRES, *The Postgres95 User Manual*, Université de Californie, 5 septembre 1995.

Zelaine Fong, *The design and implementation of the POSTGRES query optimizer*, Université de Californie,

Berkeley, département des sciences informatiques.

## Procédures et articles

Nels Olson, *Partial indexing in POSTGRES: research project*, Université de Californie, UCB Engin T7.49.1993 O676, 1993.

L. Ong et J. Goh, << A Unified Framework for Version Modeling Using Production Rules in a Database System >>, *ERL Technical Memorandum M90/33*, Université de Californie, Avril 1990.

L. Rowe et M. Stonebraker, << The POSTGRES data model >>, Proc. VLDB Conference, Septembre 1987.

P. Seshadri et A. Swami, << Generalized Partial Indexes >>, Proc. Eleventh International Conference on Data Engineering, 6−10 mars 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, 420−7.

M. Stonebraker et L. Rowe, << The design of POSTGRES >>, Proc. ACM−SIGMOD Conference on Management of Data, Mai 1986.

M. Stonebraker, E. Hanson, et C. H. Hong, << The design of the POSTGRES rules system >>, Proc. IEEE Conference on Data Engineering, Février 1987.

M. Stonebraker, << The design of the POSTGRES storage system >>, Proc. VLDB Conference, Septembre 1987.

M. Stonebraker, M. Hearst, et S. Potamianos, << A commentary on the POSTGRES rules system >>, *SIGMOD Record 18(3)*, Septembre 1989.

M. Stonebraker, << The case for partial indexes >>, *SIGMOD Record 18(4)*, Décembre 1989, 4−11.

M. Stonebraker, L. A. Rowe, et M. Hirohama, << The implementation of POSTGRES >>, *Transactions on Knowledge and Data Engineering 2(1)*, IEEE, Mars 1990.

M. Stonebraker, A. Jhingran, J. Goh, et S. Potamianos, << On Rules, Procedures, Caching and Views in Database Systems >>, Proc. ACM−SIGMOD Conference on Management of Data, Juin 1990.

# Index

## Notes

[1] Explications de pourquoi ceci fonctionne : les noms d'utilisateurs de PostgreSQL sont différents des comptes utilisateurs du système d'exploitation. Quand vous vous connectez à une base de données, vous pouvez choisir le nom d'utilisateur PostgreSQL que vous utiliserez ; si vous ne spécifiez rien, cela sera par défaut le même nom que votre compte système courant. Ce qui se passe, c'est qu'il y toujours un compte utilisateur PostgreSQL qui aura le même nom que l'utilisateur du système d'exploitation qui a démarré le serveur, et cet utilisateur aura toujours le droit de créer des bases. Au lieu de vous connecter en tant que cet utilisateur, vous pouvez spécifier partout l'option `-U` pour selectionner un nom d'utilisateur PostgreSQL sous lequel vous connecter.

[2] Alors que `SELECT *` est utile pour des requêtes rapides, il est généralement considéré comme un mauvais style dans un code en production car l'ajout d'une colonne dans la table changerait les résultats.

[3] Dans certains systèmes de bases de données, ceci incluant les anciennes versions de PostgreSQL, l'implémentation de `DISTINCT` ordonne automatiquement les lignes. Du coup, `ORDER BY` est redondant. Mais, ceci n'est pas requis par le standard SQL et PostgreSQL ne vous garantie pas actuellement que `DISTINCT` ordonne les lignes.

[4] En fait, PostgreSQL utilise la *classe d'opérateur B−tree par défaut* pour le type de données de la colonne pour déterminer l'ordre de tri avec `ASC` et `DESC`. De façon conventionnelle, les types de données seront initialisés de façon à ce que les opérateurs < et > correspondent à cet ordre de tri mais un concepteur des types de données définis par l'utilisateur pourrait choisir de faire quelque chose de différent.

[5] 60 si les secondes «leap» sont implémentées par le système d'exploitation

[6] En fait, un système de verrouillage de prédicat empêche les lectures fantômes en restreignant ce qui est écrit alors que MVCC les empêche de restreindre ce qui est lu.

[7] C'est−à−dire que la valeur qui était courante quand l'appel à `ereport` a été atteinte ; les changements d'`errno` dans les routines auxiliaires de rapports ne l'affecteront pas. Cela ne sera pas vrai si vous devez écrire explicitement `strerror(errno)` dans la liste de paramètres de `errmsg` ; en conséquence ne faites pas comme ça.

[8] En réalité, les méthodes d'accès par index n'ont pas besoin d'utiliser ce format de page. Toutes les méthodes d'indexage existantes utilisent ce format de base mais les données conservées dans les métapages des index ne suivent habituellement pas les règles d'emplacement des éléments.