PostgreSQL 7.2 User's Guide

The PostgreSQL Global Development Group

PostgreSQL 7.2 User's Guide

by The PostgreSQL Global Development Group

Legal Notice

Table of Contents

List of Tables

List of Examples

Preface

1. What is PostgreSQL?

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2[1], developed at the University of California at Berkeley Computer Science Department. The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

PostgreSQL is an open-source descendant of this original Berkeley code. It provides SQL92/SQL99 language support and other modern features.

POSTGRES pioneered many of the object-relational concepts now becoming available in some commercial databases. Traditional relational database management systems (RDBMS) support a data model consisting of a collection of named relations, containing attributes of a specific type. In current commercial systems, possible types include floating point numbers, integers, character strings, money, and dates. It is commonly recognized that this model is inadequate for future data-processing applications. The relational model successfully replaced previous models in part because of its "Spartan simplicity". However, this simplicity makes the implementation of certain applications very difficult. PostgreSQL offers substantial additional power by incorporating the following additional concepts in such a way that users can easily extend the system:

- inheritance
- data types
- functions

Other features provide additional power and flexibility:

- constraints
- triggers
- rules
- transactional integrity

These features put PostgreSQL into the category of databases referred to as *object-relational*. Note that this is distinct from those referred to as *object-oriented*, which in general are not as well suited to supporting traditional relational database languages. So, although PostgreSQL has some object-oriented features, it is firmly in the relational database world. In fact, some commercial databases have recently incorporated features pioneered by PostgreSQL.

2. A Short History of PostgreSQL

The object-relational database management system now known as PostgreSQL (and briefly called Postgres95) is derived from the POSTGRES package written at the University of California at Berkeley. With over a decade of development behind it, PostgreSQL is the most advanced open-source database available anywhere, offering multiversion concurrency control, supporting almost all SQL

1. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

constructs (including subselects, transactions, and user-defined types and functions), and having a wide range of language bindings available (including C, C++, Java, Perl, Tcl, and Python).

2.1. The Berkeley POSTGRES Project

Implementation of the POSTGRES DBMS began in 1986. The initial concepts for the system were presented in *The design of POSTGRES* and the definition of the initial data model appeared in *The POSTGRES data model*. The design of the rule system at that time was described in *The design of the POSTGRES rules system*. The rationale and architecture of the storage manager were detailed in *The design of the POSTGRES storage system*.

Postgres has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. Version 1, described in *The implementation of POSTGRES*, was released to a few external users in June 1989. In response to a critique of the first rule system (*A commentary on the POSTGRES rules system*), the rule system was redesigned (*On Rules, Procedures, Caching and Views in Database Systems*) and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rewrite rule system. For the most part, subsequent releases until Postgres95 (see below) focused on portability and reliability.

POSTGRES has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. POSTGRES has also been used as an educational tool at several universities. Finally, Illustra Information Technologies (later merged into Informix[2], which is now owned by IBM[3].) picked up the code and commercialized it. POSTGRES became the primary data manager for the Sequoia 2000[4] scientific computing project in late 1992.

The size of the external user community nearly doubled during 1993. It became increasingly obvious that maintenance of the prototype code and support was taking up large amounts of time that should have been devoted to database research. In an effort to reduce this support burden, the Berkeley POSTGRES project officially ended with Version 4.2.

2.2. Postgres95

In 1994, Andrew Yu and Jolly Chen added a SQL language interpreter to POSTGRES. Postgres95 was subsequently released to the Web to find its own way in the world as an open-source descendant of the original POSTGRES Berkeley code.

Postgres95 code was completely ANSI C and trimmed in size by 25%. Many internal changes improved performance and maintainability. Postgres95 release 1.0.x ran about 30-50% faster on the Wisconsin Benchmark compared to POSTGRES, Version 4.2. Apart from bug fixes, the following were the major enhancements:

- The query language PostQUEL was replaced with SQL (implemented in the server). Subqueries were not supported until PostgreSQL (see below), but they could be imitated in Postgres95 with user-defined SQL functions. Aggregates were re-implemented. Support for the GROUP BY query clause was also added. The `libpq` interface remained available for C programs.
- In addition to the monitor program, a new program (psql) was provided for interactive SQL queries using GNU Readline.

2. http://www.informix.com/
3. http://www.ibm.com/
4. http://meteora.ucsd.edu/s2k/s2k_home.html

- A new front-end library, `libpgtcl`, supported Tcl-based clients. A sample shell, **pgtclsh**, provided new Tcl commands to interface Tcl programs with the Postgres95 backend.
- The large-object interface was overhauled. The Inversion large objects were the only mechanism for storing large objects. (The Inversion file system was removed.)
- The instance-level rule system was removed. Rules were still available as rewrite rules.
- A short tutorial introducing regular SQL features as well as those of Postgres95 was distributed with the source code
- GNU make (instead of BSD make) was used for the build. Also, Postgres95 could be compiled with an unpatched GCC (data alignment of doubles was fixed).

2.3. PostgreSQL

By 1996, it became clear that the name "Postgres95" would not stand the test of time. We chose a new name, PostgreSQL, to reflect the relationship between the original POSTGRES and the more recent versions with SQL capability. At the same time, we set the version numbering to start at 6.0, putting the numbers back into the sequence originally begun by the Berkeley POSTGRES project.

The emphasis during development of Postgres95 was on identifying and understanding existing problems in the backend code. With PostgreSQL, the emphasis has shifted to augmenting features and capabilities, although work continues in all areas.

Major enhancements in PostgreSQL include:

- Table-level locking has been replaced by multiversion concurrency control, which allows readers to continue reading consistent data during writer activity and enables hot backups from pg_dump while the database stays available for queries.
- Important backend features, including subselects, defaults, constraints, and triggers, have been implemented.
- Additional SQL92-compliant language features have been added, including primary keys, quoted identifiers, literal string type coercion, type casting, and binary and hexadecimal integer input.
- Built-in types have been improved, including new wide-range date/time types and additional geometric type support.
- Overall backend code speed has been increased by approximately 20-40%, and backend start-up time has decreased by 80% since version 6.0 was released.

3. Documentation Resources

This manual set is organized into several parts:

Tutorial

An informal introduction for new users

User's Guide

Documents the SQL query language environment, including data types and functions.

Programmer's Guide

Advanced information for application programmers. Topics include type and function extensibility, library interfaces, and application design issues.

Administrator's Guide

Installation and server management information

Reference Manual

Reference pages for SQL command syntax and client and server programs

Developer's Guide

Information for PostgreSQL developers. This is intended for those who are contributing to the PostgreSQL project; application development information appears in the *Programmer's Guide*.

In addition to this manual set, there are other resources to help you with PostgreSQL installation and use:

man pages

The *Reference Manual*'s pages in the traditional Unix man format.

FAQs

Frequently Asked Questions (FAQ) lists document both general issues and some platform-specific issues.

READMEs

README files are available for some contributed packages.

Web Site

The PostgreSQL web site[5] carries details on the latest release, upcoming features, and other information to make your work or play with PostgreSQL more productive.

Mailing Lists

The mailing lists are a good place to have your questions answered, to share experiences with other users, and to contact the developers. Consult the User's Lounge[6] section of the PostgreSQL web site for details.

Yourself!

PostgreSQL is an open-source effort. As such, it depends on the user community for ongoing support. As you begin to use PostgreSQL, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute them.

Even those without a lot of experience can provide corrections and minor changes in the documentation, and that is a good way to start. The <pgsql-docs@postgresql.org> mailing list is the place to get going.

5. http://www.postgresql.org
6. http://www.postgresql.org/users-lounge/

4. Terminology and Notation

The terms “PostgreSQL” and “Postgres” will be used interchangeably to refer to the software that accompanies this documentation.

An *administrator* is generally a person who is in charge of installing and running the server. A *user* could be anyone who is using, or wants to use, any part of the PostgreSQL system. These terms should not be interpreted too narrowly; this documentation set does not have fixed presumptions about system administration procedures.

We use `/usr/local/pgsql/` as the root directory of the installation and `/usr/local/pgsql/data` as the directory with the database files. These directories may vary on your site, details can be derived in the *Administrator’s Guide*.

In a command synopsis, brackets (`[` and `]`) indicate an optional phrase or keyword. Anything in braces (`{` and `}`) and containing vertical bars (`|`) indicates that you must choose one alternative.

Examples will show commands executed from various accounts and programs. Commands executed from a Unix shell may be preceded with a dollar sign (“$”). Commands executed from particular user accounts such as root or postgres are specially flagged and explained. SQL commands may be preceded with “`=>`” or will have no leading prompt, depending on the context.

> **Note:** The notation for flagging commands is not universally consistent throughout the documentation set. Please report problems to the documentation mailing list `<pgsql-docs@postgresql.org>`.

5. Bug Reporting Guidelines

When you find a bug in PostgreSQL we want to hear about it. Your bug reports play an important part in making PostgreSQL more reliable because even the utmost care cannot guarantee that every part of PostgreSQL will work on every platform under every circumstance.

The following suggestions are intended to assist you in forming bug reports that can be handled in an effective fashion. No one is required to follow them but it tends to be to everyone’s advantage.

We cannot promise to fix every bug right away. If the bug is obvious, critical, or affects a lot of users, chances are good that someone will look into it. It could also happen that we tell you to update to a newer version to see if the bug happens there. Or we might decide that the bug cannot be fixed before some major rewrite we might be planning is done. Or perhaps it is simply too hard and there are more important things on the agenda. If you need help immediately, consider obtaining a commercial support contract.

5.1. Identifying Bugs

Before you report a bug, please read and re-read the documentation to verify that you can really do whatever it is you are trying. If it is not clear from the documentation whether you can do something or not, please report that too; it is a bug in the documentation. If it turns out that the program does something different from what the documentation says, that is a bug. That might include, but is not limited to, the following circumstances:

- A program terminates with a fatal signal or an operating system error message that would point to a problem in the program. (A counterexample might be a “disk full” message, since you have to fix that yourself.)

- A program produces the wrong output for any given input.
- A program refuses to accept valid input (as defined in the documentation).
- A program accepts invalid input without a notice or error message. But keep in mind that your idea of invalid input might be our idea of an extension or compatibility with traditional practice.
- PostgreSQL fails to compile, build, or install according to the instructions on supported platforms.

Here "program" refers to any executable, not only the backend server.

Being slow or resource-hogging is not necessarily a bug. Read the documentation or ask on one of the mailing lists for help in tuning your applications. Failing to comply to the SQL standard is not necessarily a bug either, unless compliance for the specific feature is explicitly claimed.

Before you continue, check on the TODO list and in the FAQ to see if your bug is already known. If you cannot decode the information on the TODO list, report your problem. The least we can do is make the TODO list clearer.

5.2. What to report

The most important thing to remember about bug reporting is to state all the facts and only facts. Do not speculate what you think went wrong, what "it seemed to do", or which part of the program has a fault. If you are not familiar with the implementation you would probably guess wrong and not help us a bit. And even if you are, educated explanations are a great supplement to but no substitute for facts. If we are going to fix the bug we still have to see it happen for ourselves first. Reporting the bare facts is relatively straightforward (you can probably copy and paste them from the screen) but all too often important details are left out because someone thought it does not matter or the report would be understood anyway.

The following items should be contained in every bug report:

- The exact sequence of steps *from program start-up* necessary to reproduce the problem. This should be self-contained; it is not enough to send in a bare select statement without the preceding create table and insert statements, if the output should depend on the data in the tables. We do not have the time to reverse-engineer your database schema, and if we are supposed to make up our own data we would probably miss the problem. The best format for a test case for query-language related problems is a file that can be run through the psql frontend that shows the problem. (Be sure to not have anything in your `~/.psqlrc` start-up file.) An easy start at this file is to use pg_dump to dump out the table declarations and data needed to set the scene, then add the problem query. You are encouraged to minimize the size of your example, but this is not absolutely necessary. If the bug is reproducible, we will find it either way.

  If your application uses some other client interface, such as PHP, then please try to isolate the offending queries. We will probably not set up a web server to reproduce your problem. In any case remember to provide the exact input files, do not guess that the problem happens for "large files" or "mid-size databases", etc. since this information is too inexact to be of use.

- The output you got. Please do not say that it "didn't work" or "crashed". If there is an error message, show it, even if you do not understand it. If the program terminates with an operating system error, say which. If nothing at all happens, say so. Even if the result of your test case is a program crash or otherwise obvious it might not happen on our platform. The easiest thing is to copy the output from the terminal, if possible.

> **Note:** In case of fatal errors, the error message reported by the client might not contain all the information available. Please also look at the log output of the database server. If you do not keep your server's log output, this would be a good time to start doing so.

- The output you expected is very important to state. If you just write "This command gives me that output." or "This is not what I expected.", we might run it ourselves, scan the output, and think it looks OK and is exactly what we expected. We should not have to spend the time to decode the exact semantics behind your commands. Especially refrain from merely saying that "This is not what SQL says/Oracle does." Digging out the correct behavior from SQL is not a fun undertaking, nor do we all know how all the other relational databases out there behave. (If your problem is a program crash, you can obviously omit this item.)
- Any command line options and other start-up options, including concerned environment variables or configuration files that you changed from the default. Again, be exact. If you are using a prepackaged distribution that starts the database server at boot time, you should try to find out how that is done.
- Anything you did at all differently from the installation instructions.
- The PostgreSQL version. You can run the command `SELECT version();` to find out the version of the server you are connected to. Most executable programs also support a `--version` option; at least `postmaster --version` and `psql --version` should work. If the function or the options do not exist then your version is more than old enough to warrant an upgrade. You can also look into the `README` file in the source directory or at the name of your distribution file or package name. If you run a prepackaged version, such as RPMs, say so, including any subversion the package may have. If you are talking about a CVS snapshot, mention that, including its date and time.

  If your version is older than 7.2 we will almost certainly tell you to upgrade. There are tons of bug fixes in each new release, that is why we make new releases.

- Platform information. This includes the kernel name and version, C library, processor, memory information. In most cases it is sufficient to report the vendor and version, but do not assume everyone knows what exactly "Debian" contains or that everyone runs on Pentiums. If you have installation problems then information about compilers, make, etc. is also necessary.

Do not be afraid if your bug report becomes rather lengthy. That is a fact of life. It is better to report everything the first time than us having to squeeze the facts out of you. On the other hand, if your input files are huge, it is fair to ask first whether somebody is interested in looking into it.

Do not spend all your time to figure out which changes in the input make the problem go away. This will probably not help solving it. If it turns out that the bug cannot be fixed right away, you will still have time to find and share your work-around. Also, once again, do not waste your time guessing why the bug exists. We will find that out soon enough.

When writing a bug report, please choose non-confusing terminology. The software package in total is called "PostgreSQL", sometimes "Postgres" for short. If you are specifically talking about the backend server, mention that, do not just say "PostgreSQL crashes". A crash of a single backend server process is quite different from crash of the parent "postmaster" process; please don't say "the postmaster crashed" when you mean a single backend went down, nor vice versa. Also, client programs such as the interactive frontend "psql" are completely separate from the backend. Please try to be specific about whether the problem is on the client or server side.

5.3. Where to report bugs

In general, send bug reports to the bug report mailing list at <pgsql-bugs@postgresql.org>. You are requested to use a descriptive subject for your email message, perhaps parts of the error message.

Another method is to fill in the bug report web-form available at the project's web site http://www.postgresql.org/. Entering a bug report this way causes it to be mailed to the <pgsql-bugs@postgresql.org> mailing list.

Do not send bug reports to any of the user mailing lists, such as <pgsql-sql@postgresql.org> or <pgsql-general@postgresql.org>. These mailing lists are for answering user questions and their subscribers normally do not wish to receive bug reports. More importantly, they are unlikely to fix them.

Also, please do *not* send reports to the developers' mailing list <pgsql-hackers@postgresql.org>. This list is for discussing the development of PostgreSQL and it would be nice if we could keep the bug reports separate. We might choose to take up a discussion about your bug report on pgsql-hackers, if the problem needs more review.

If you have a problem with the documentation, the best place to report it is the documentation mailing list <pgsql-docs@postgresql.org>. Please be specific about what part of the documentation you are unhappy with.

If your bug is a portability problem on a non-supported platform, send mail to <pgsql-ports@postgresql.org>, so we (and you) can work on porting PostgreSQL to your platform.

> **Note:** Due to the unfortunate amount of spam going around, all of the above email addresses are closed mailing lists. That is, you need to be subscribed to a list to be allowed to post on it. (You need not be subscribed to use the bug report web-form, however.) If you would like to send mail but do not want to receive list traffic, you can subscribe and set your subscription option to nomail. For more information send mail to <majordomo@postgresql.org> with the single word help in the body of the message.

6. Y2K Statement

> **Author:** Written by Thomas Lockhart (<lockhart@fourpalms.org>) on 1998-10-22. Updated 2000-03-31.

The PostgreSQL Global Development Group provides the PostgreSQL software code tree as a public service, without warranty and without liability for its behavior or performance. However, at the time of writing:

- The author of this statement, a volunteer on the PostgreSQL support team since November, 1996, is not aware of any problems in the PostgreSQL code base related to time transitions around Jan 1, 2000 (Y2K).
- The author of this statement is not aware of any reports of Y2K problems uncovered in regression testing or in other field use of recent or current versions of PostgreSQL. We might have expected to hear about problems if they existed, given the installed base and the active participation of users on the support mailing lists.

- To the best of the author's knowledge, the assumptions PostgreSQL makes about dates specified with a two-digit year are documented in the current *User's Guide* in the chapter on data types. For two-digit years, the significant transition year is 1970, not 2000; e.g. `70-01-01` is interpreted as 1970-01-01, whereas `69-01-01` is interpreted as 2069-01-01.
- Any Y2K problems in the underlying OS related to obtaining the "current time" may propagate into apparent Y2K problems in PostgreSQL.

Refer to The GNU Project[8] and The Perl Institute[9] for further discussion of Y2K issues, particularly as it relates to open source, no fee software.

8. http://www.gnu.org/software/year2000.html
9. http://language.perl.com/news/y2k.html

Chapter 1. SQL Syntax

This chapter describes the syntax of SQL.

1.1. Lexical Structure

SQL input consists of a sequence of *commands*. A command is composed of a sequence of *tokens*, terminated by a semicolon (";"). The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a *key word*, an *identifier*, a *quoted identifier*, a *literal* (or constant), or a special character symbol. Tokens are normally separated by whitespace (space, tab, newline), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

Additionally, *comments* can occur in SQL input. They are not tokens, they are effectively equivalent to whitespace.

For example, the following is (syntactically) valid SQL input:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usefully be split across lines).

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a "SELECT", an "UPDATE", and an "INSERT" command. But for instance the **UPDATE** command always requires a SET token to appear in a certain position, and this particular variation of **INSERT** also requires a VALUES in order to be complete. The precise syntax rules for each command are described in the *Reference Manual*.

1.1.1. Identifiers and Key Words

Tokens such as SELECT, UPDATE, or VALUES in the example above are examples of *key words*, that is, words that have a fixed meaning in the SQL language. The tokens MY_TABLE and A are examples of *identifiers*. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called "names". Key words and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a key word without knowing the language. A complete list of key words can be found in Appendix B.

SQL identifiers and key words must begin with a letter (`a-z`, but also letters with diacritical marks and non-Latin letters) or an underscore (_). Subsequent characters in an identifier or key word can be letters, digits (`0-9`), or underscores, although the SQL standard will not define a key word that contains digits or starts or ends with an underscore.

The system uses no more than `NAMEDATALEN`-1 characters of an identifier; longer names can be written in commands, but they will be truncated. By default, `NAMEDATALEN` is 32 so the maxi-

mum identifier length is 31 (but at the time the system is built, `NAMEDATALEN` can be changed in `src/include/postgres_ext.h`).

Identifier and key word names are case insensitive. Therefore

```
UPDATE MY_TABLE SET A = 5;
```

can equivalently be written as

```
uPDaTE my_TabLE SeT a = 5;
```

A convention often used is to write key words in upper case and names in lower case, e.g.,

```
UPDATE my_table SET a = 5;
```

There is a second kind of identifier: the *delimited identifier* or *quoted identifier*. It is formed by enclosing an arbitrary sequence of characters in double-quotes (`"`). A delimited identifier is always an identifier, never a key word. So `"select"` could be used to refer to a column or table named "select", whereas an unquoted `select` would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

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

Quoted identifiers can contain any character other than a double quote itself. This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers `FOO`, `foo` and `"foo"` are considered the same by PostgreSQL, but `"Foo"` and `"FOO"` are different from these three and each other. [1]

1.1.2. Constants

There are four kinds of *implicitly-typed constants* in PostgreSQL: strings, bit strings, integers, and floating-point numbers. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. The implicit constants are described below; explicit constants are discussed afterwards.

1.1.2.1. String Constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes ("'"), e.g., `'This is a string'`. SQL allows single quotes to be embedded in strings by typing two adjacent single quotes (e.g., `'Dianne''s horse'`). In PostgreSQL single quotes may alternatively be escaped with a backslash ("\", e.g., `'Dianne\'s horse'`).

C-style backslash escapes are also available: `\b` is a backspace, `\f` is a form feed, `\n` is a newline, `\r` is a carriage return, `\t` is a tab, and `\`*`xxx`*, where *`xxx`* is an octal number, is the character with

1. The folding of unquoted names to lower case in PostgreSQL is incompatible with the SQL standard, which says that unquoted names should be folded to upper case. Thus, `foo` should be equivalent to `"FOO"` not `"foo"` according to the standard. If you want to write portable applications you are advised to always quote a particular name or never quote it.

the corresponding ASCII code. Any other character following a backslash is taken literally. Thus, to include a backslash in a string constant, type two backslashes.

The character with the code zero cannot be in a string constant.

Two string constants that are only separated by whitespace *with at least one newline* are concatenated and effectively treated as if the string had been written in one constant. For example:

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

is equivalent to

```
SELECT 'foobar';
```

but

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

is not valid syntax, and PostgreSQL is consistent with SQL9x in this regard.

1.1.2.2. Bit-String Constants

Bit-string constants look like string constants with a `B` (upper or lower case) immediately before the opening quote (no intervening whitespace), e.g., `B'1001'`. The only characters allowed within bit-string constants are `0` and `1`. Bit-string constants can be continued across lines in the same way as regular string constants.

1.1.2.3. Integer Constants

Integer constants in SQL are sequences of decimal digits (0 though 9) with no decimal point and no exponent. The range of legal values depends on which integer data type is used, but the plain `integer` type accepts values ranging from -2147483648 to +2147483647. (The optional plus or minus sign is actually a separate unary operator and not part of the integer constant.)

1.1.2.4. Floating-Point Constants

Floating-point constants are accepted in these general forms:

digits.[*digits*][e[+-]*digits*]
[*digits*].*digits*[e[+-]*digits*]
*digits*e[+-]*digits*

where *digits* is one or more decimal digits. At least one digit must be before or after the decimal point. At least one digit must follow the exponent delimiter (e) if that field is present. Thus, a floating-point constant is distinguished from an integer constant by the presence of either the decimal point or the exponent clause (or both). There must not be a space or other characters embedded in the constant.

These are some examples of valid floating-point constants:

3.5
4.
.001
5e2
1.925e-3

Floating-point constants are of type DOUBLE PRECISION. REAL can be specified explicitly by using SQL string notation or PostgreSQL type notation:

```
REAL '1.23'  -- string style
'1.23'::REAL -- PostgreSQL (historical) style
```

1.1.2.5. Constants of Other Types

A constant of an *arbitrary* type can be entered using any one of the following notations:

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

The string's text is passed to the input conversion routine for the type called *type*. The result is a constant of the indicated type. The explicit type cast may be omitted if there is no ambiguity as to the type the constant must be (for example, when it is passed as an argument to a non-overloaded function), in which case it is automatically coerced.

It is also possible to specify a type coercion using a function-like syntax:

```
typename ( 'string' )
```

but not all type names may be used in this way; see Section 1.3.6 for details.

The ::, CAST(), and function-call syntaxes can also be used to specify runtime type conversions of arbitrary expressions, as discussed in Section 1.3.6. But the form *type* '*string*' can only be used to specify the type of a literal constant. Another restriction on *type* '*string*' is that it does not work for array types; use :: or CAST() to specify the type of an array constant.

1.1.2.6. Array constants

The general format of an array constant is the following:

```
'{ val1 delim val2 delim ... }'
```

where *delim* is the delimiter character for the type, as recorded in its pg_type entry. (For all built-in types, this is the comma character ",".) Each *val* is either a constant of the array element type, or a subarray. An example of an array constant is

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

This constant is a two-dimensional, 3-by-3 array consisting of three subarrays of integers.

Individual array elements can be placed between double-quote marks (") to avoid ambiguity problems with respect to whitespace. Without quote marks, the array-value parser will skip leading whitespace.

(Array constants are actually only a special case of the generic type constants discussed in the previous section. The constant is initially treated as a string and passed to the array input conversion routine. An explicit type specification might be necessary.)

1.1.3. Operators

An operator is a sequence of up to NAMEDATALEN-1 (31 by default) characters from the following list:

+ - * / < > = ~ ! @ # % ^ & | ‘ ? $

There are a few restrictions on operator names, however:

- $ (dollar) cannot be a single-character operator, although it can be part of a multiple-character operator name.
- -- and /* cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multiple-character operator name cannot end in + or -, unless the name also contains at least one of these characters:

  ~ ! @ # % ^ & | ‘ ? $

  For example, @- is an allowed operator name, but *- is not. This restriction allows PostgreSQL to parse SQL-compliant queries without requiring spaces between tokens.

When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a left unary operator named @, you cannot write X*@Y; you must write X* @Y to ensure that PostgreSQL reads it as two operator names not one.

1.1.4. Special Characters

Some characters that are not alphanumeric have a special meaning that is different from being an operator. Details on the usage can be found at the location where the respective syntax element is described. This section only exists to advise the existence and summarize the purposes of these characters.

- A dollar sign ($) followed by digits is used to represent the positional parameters in the body of a function definition. In other contexts the dollar sign may be part of an operator name.
- Parentheses (()) have their usual meaning to group expressions and enforce precedence. In some cases parentheses are required as part of the fixed syntax of a particular SQL command.
- Brackets ([]) are used to select the elements of an array. See Chapter 6 for more information on arrays.
- Commas (,) are used in some syntactical constructs to separate the elements of a list.
- The semicolon (;) terminates an SQL command. It cannot appear anywhere within a command, except within a string constant or quoted identifier.
- The colon (:) is used to select “slices” from arrays. (See Chapter 6.) In certain SQL dialects (such as Embedded SQL), the colon is used to prefix variable names.
- The asterisk (*) has a special meaning when used in the **SELECT** command or with the COUNT aggregate function.
- The period (.) is used in floating-point constants, and to separate table and column names.

1.1.5. Comments

A comment is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```
-- This is a standard SQL92 comment
```

Alternatively, C-style block comments can be used:

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

where the comment begins with `/*` and extends to the matching occurrence of `*/`. These block comments nest, as specified in SQL99 but unlike C, so that one can comment out larger blocks of code that may contain existing block comments.

A comment is removed from the input stream before further syntax analysis and is effectively replaced by whitespace.

1.2. Columns

A *column* is either a user-defined column of a given table or one of the following system-defined columns:

`oid`

The object identifier (object ID) of a row. This is a serial number that is automatically added by PostgreSQL to all table rows (unless the table was created WITHOUT OIDS, in which case this column is not present).

`tableoid`

The OID of the table containing this row. This attribute is particularly handy for queries that select from inheritance hierarchies, since without it, it's difficult to tell which individual table a row came from. The `tableoid` can be joined against the `oid` column of `pg_class` to obtain the table name.

`xmin`

The identity (transaction ID) of the inserting transaction for this tuple. (Note: A tuple is an individual state of a row; each update of a row creates a new tuple for the same logical row.)

`cmin`

The command identifier (starting at zero) within the inserting transaction.

`xmax`

The identity (transaction ID) of the deleting transaction, or zero for an undeleted tuple. It is possible for this field to be nonzero in a visible tuple: that usually indicates that the deleting transaction hasn't committed yet, or that an attempted deletion was rolled back.

`cmax`

The command identifier within the deleting transaction, or zero.

`ctid`

The tuple ID of the tuple within its table. This is a pair (block number, tuple index within block) that identifies the physical location of the tuple. Note that although the `ctid` can be used to locate the tuple very quickly, a row's `ctid` will change each time it is updated or moved by **VACUUM FULL**. Therefore `ctid` is useless as a long-term row identifier. The OID, or even better a user-defined serial number, should be used to identify logical rows.

OIDs are 32-bit quantities and are assigned from a single cluster-wide counter. In a large or long-lived database, it is possible for the counter to wrap around. Hence, it is bad practice to assume that OIDs are unique, unless you take steps to ensure that they are unique. Recommended practice when using OIDs for row identification is to create a unique constraint on the OID column of each table for which the OID will be used. Never assume that OIDs are unique across tables; use the combination of `tableoid` and row OID if you need a database-wide identifier. (Future releases of PostgreSQL are likely to use a separate OID counter for each table, so that `tableoid` *must* be included to arrive at a globally unique identifier.)

Transaction identifiers are 32-bit quantities. In a long-lived database it is possible for transaction IDs to wrap around. This is not a fatal problem given appropriate maintenance procedures; see the *Administrator's Guide* for details. However, it is unwise to depend on uniqueness of transaction IDs over the long term (more than one billion transactions).

Command identifiers are also 32-bit quantities. This creates a hard limit of 2^{32} (4 billion) SQL commands within a single transaction. In practice this limit is not a problem --- note that the limit is on number of SQL queries, not number of tuples processed.

1.3. Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the **SELECT** command, as new column values in **INSERT** or **UPDATE**, or in search conditions in a number of commands. The result of a value expression is sometimes called a *scalar*, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called *scalar expressions* (or even simply *expressions*). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

A value expression is one of the following:

- A constant or literal value; see Section 1.1.2.
- A column reference.
- A positional parameter reference, in the body of a function declaration.
- An operator invocation.
- A function call.
- An aggregate expression.
- A type cast.
- A scalar subquery.
- `( expression )`

  Parentheses are used to group subexpressions and override precedence.

In addition to this list, there are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in the appropriate location in Chapter 4. An example is the `IS NULL` clause.

We have already discussed constants in Section 1.1.2. The following sections discuss the remaining options.

1.3.1. Column References

A column can be referenced in the form:

correlation.columnname '['subscript']'

correlation is either the name of a table, an alias for a table defined by means of a FROM clause, or the key words `NEW` or `OLD`. (NEW and OLD can only appear in the action portion of a rule, while other correlation names can be used in any SQL statement.) The correlation name and separating dot may be omitted if the column name is unique across all the tables being used in the current query. If *column* is of an array type, then the optional *subscript* selects a specific element or elements in the array. If no subscript is provided, then the whole array is selected. (See Chapter 6 for more about arrays.)

1.3.2. Positional Parameters

A positional parameter reference is used to indicate a parameter in an SQL function. Typically this is used in SQL function definition statements. The form of a parameter is:

$*number*

For example, consider the definition of a function, `dept`, as

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

Here the `$1` will be replaced by the first function argument when the function is invoked.

1.3.3. Operator Invocations

There are three possible syntaxes for an operator invocation:

expression operator expression (binary infix operator)
operator expression (unary prefix operator)
expression operator (unary postfix operator)

where the *operator* token follows the syntax rules of Section 1.1.3 or is one of the tokens AND, OR, and NOT. Which particular operators exist and whether they are unary or binary depends on what operators have been defined by the system or the user. Chapter 4 describes the built-in operators.

1.3.4. Function Calls

The syntax for a function call is the name of a function (which is subject to the syntax rules for identifiers of Section 1.1.1), followed by its argument list enclosed in parentheses:

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

For example, the following computes the square root of 2:

```
sqrt(2)
```

The list of built-in functions is in Chapter 4. Other functions may be added by the user.

1.3.5. Aggregate Expressions

An *aggregate expression* represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

aggregate_name (*expression*)
aggregate_name (ALL *expression*)
aggregate_name (DISTINCT *expression*)
aggregate_name (*)

where *aggregate_name* is a previously defined aggregate, and *expression* is any value expression that does not itself contain an aggregate expression.

The first form of aggregate expression invokes the aggregate across all input rows for which the given expression yields a non-NULL value. (Actually, it is up to the aggregate function whether to ignore NULLs or not --- but all the standard ones do.) The second form is the same as the first, since ALL is the default. The third form invokes the aggregate for all distinct non-NULL values of the expression found in the input rows. The last form invokes the aggregate once for each input row regardless of NULL or non-NULL values; since no particular input value is specified, it is generally only useful for the count() aggregate function.

For example, count(*) yields the total number of input rows; count(f1) yields the number of input rows in which f1 is non-NULL; count(distinct f1) yields the number of distinct non-NULL values of f1.

The predefined aggregate functions are described in Section 4.14. Other aggregate functions may be added by the user.

1.3.6. Type Casts

A type cast specifies a conversion from one data type to another. PostgreSQL accepts two equivalent syntaxes for type casts:

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

The CAST syntax conforms to SQL92; the syntax with :: is historical PostgreSQL usage.

When a cast is applied to a value expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion function is available. Notice that this is subtly different from the use of casts with constants, as shown in Section 1.1.2.5. A cast applied to an unadorned string literal represents the initial assignment of a type to a literal constant value, and so it will succeed for any type (if the contents of the string literal are acceptable input syntax for the data type).

An explicit type cast may be omitted if there is no ambiguity as to the type that a value expression must produce (for example, when it is assigned to a table column); the system will automatically apply a type cast in such cases.

It is also possible to specify a type cast using a function-like syntax:

```
typename ( expression )
```

However, this only works for types whose names are also valid as function names. For example, `double precision` can't be used this way, but the equivalent `float8` can. Also, the names `interval`, `time`, and `timestamp` can only be used in this fashion if they are double-quoted, because of parser conflicts. Therefore, the use of the function-like cast syntax leads to inconsistencies and should probably be avoided in new applications.

1.3.7. Scalar Subqueries

A scalar subquery is an ordinary **SELECT** in parentheses that returns exactly one row with one column. The **SELECT** query is executed and the single returned value is used in the surrounding value expression. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. (But if, during a particular execution, the subquery returns no rows, there is no error; the scalar result is taken to be NULL.) The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery. See also Section 4.15.

For example, the following finds the largest city population in each state:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
FROM states;
```

1.4. Lexical Precedence

The precedence and associativity of the operators is hard-wired into the parser. Most operators have the same precedence and are left-associative. This may lead to non-intuitive behavior; for example the Boolean operators < and > have a different precedence than the Boolean operators <= and >=. Also, you will sometimes need to add parentheses when using combinations of binary and unary operators. For instance

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

will be parsed as

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

because the parser has no idea -- until it is too late -- that ! is defined as a postfix operator, not an infix one. To get the desired behavior in this case, you must write

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

This is the price one pays for extensibility.

Table 1-1. Operator Precedence (decreasing)

Operator/Element	Associativity	Description
::	left	PostgreSQL-style typecast
[ ]	left	array element selection
.	left	table/column name separator
-	right	unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
IS		test for TRUE, FALSE, UNKNOWN, NULL
ISNULL		test for NULL
NOTNULL		test for NOT NULL
(any other)	left	all other native and user-defined operators
IN		set membership
BETWEEN		containment
OVERLAPS		time interval overlap
LIKE ILIKE		string pattern matching
< >		less than, greater than
=	right	equality, assignment
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

Note that the operator precedence rules also apply to user-defined operators that have the same names as the built-in operators mentioned above. For example, if you define a “+” operator for some custom data type it will have the same precedence as the built-in “+” operator, no matter what yours does.

Chapter 2. Queries

2.1. Overview

A *query* is the process of retrieving or the command to retrieve data from a database. In SQL the **SELECT** command is used to specify queries. The general syntax of the **SELECT** command is

```
SELECT select_list FROM table_expression [sort_specification]
```

The following sections describe the details of the select list, the table expression, and the sort specification. The simplest kind of query has the form

```
SELECT * FROM table1;
```

Assuming that there is a table called table1, this command would retrieve all rows and all columns from table1. (The method of retrieval depends on the client application. For example, the psql program will display an ASCII-art table on the screen, client libraries will offer functions to retrieve individual rows and columns.) The select list specification `*` means all columns that the table expression happens to provide. A select list can also select a subset of the available columns or even make calculations on the columns before retrieving them; see Section 2.3. For example, if table1 has columns named a, b, and c (and perhaps others) you can make the following query:

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

(assuming that b and c are of a numeric data type).

`FROM table1` is a particularly simple kind of table expression. In general, table expressions can be complex constructs of base tables, joins, and subqueries. But you can also omit the table expression entirely and use the SELECT command as a calculator:

```
SELECT 3 * 4;
```

This is more useful if the expressions in the select list return varying results. For example, you could call a function this way.

```
SELECT random();
```

2.2. Table Expressions

A *table expression* specifies a table. The table expression contains a FROM clause that is optionally followed by WHERE, GROUP BY, and HAVING clauses. Trivial table expressions simply refer to a table on disk, a so-called base table, but more complex expressions can be used to modify or combine base tables in various ways.

The optional WHERE, GROUP BY, and HAVING clauses in the table expression specify a pipeline of successive transformations performed on the table derived in the FROM clause. The derived table that is produced by all these transformations provides the input rows used to compute output rows as specified by the select list of column value expressions.

2.2.1. FROM clause

The FROM clause derives a table from one or more other tables given in a comma-separated table reference list.

```
FROM table_reference [, table_reference [, ...]]
```

A table reference may be a table name or a derived table such as a subquery, a table join, or complex combinations of these. If more than one table reference is listed in the FROM clause they are cross-joined (see below) to form the derived table that may then be subject to transformations by the WHERE, GROUP BY, and HAVING clauses and is finally the result of the overall table expression.

When a table reference names a table that is the supertable of a table inheritance hierarchy, the table reference produces rows of not only that table but all of its subtable successors, unless the keyword ONLY precedes the table name. However, the reference produces only the columns that appear in the named table --- any columns added in subtables are ignored.

2.2.1.1. Joined Tables

A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type. INNER, OUTER, and CROSS JOIN are supported.

Join Types

CROSS JOIN

```
T1 CROSS JOIN T2
```

For each combination of rows from *T1* and *T2*, the derived table will contain a row consisting of all columns in *T1* followed by all columns in *T2*. If the tables have N and M rows respectively, the joined table will have N * M rows. A cross join is equivalent to an `INNER JOIN ON TRUE`.

> **Tip:** `FROM T1 CROSS JOIN T2` is equivalent to `FROM T1, T2`.

Qualified joins

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

The words INNER and OUTER are optional for all joins. INNER is the default; LEFT, RIGHT, and FULL imply an OUTER JOIN.

The *join condition* is specified in the ON or USING clause, or implicitly by the word NATURAL. The join condition determines which rows from the two source tables are considered to “match”, as explained in detail below.

The ON clause is the most general kind of join condition: it takes a Boolean value expression of the same kind as is used in a WHERE clause. A pair of rows from T1 and T2 match if the ON expression evaluates to TRUE for them.

USING is a shorthand notation: it takes a comma-separated list of column names, which the joined tables must have in common, and forms a join condition specifying equality of each of these pairs of columns. Furthermore, the output of a JOIN USING has one column for each of the equated pairs of input columns, followed by all of the other columns from each table. Thus, `USING (a, b, c)` is equivalent to `ON (t1.a = t2.a AND t1.b = t2.b AND`

`t1.c = t2.c)` with the exception that if ON is used there will be two columns a, b, and c in the result, whereas with USING there will be only one of each.

Finally, NATURAL is a shorthand form of USING: it forms a USING list consisting of exactly those column names that appear in both input tables. As with USING, these columns appear only once in the output table.

The possible types of qualified JOIN are:

INNER JOIN

For each row R1 of T1, the joined table has a row for each row in T2 that satisfies the join condition with R1.

LEFT OUTER JOIN

First, an INNER JOIN is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is returned with NULL values in columns of T2. Thus, the joined table unconditionally has at least one row for each row in T1.

RIGHT OUTER JOIN

First, an INNER JOIN is performed. Then, for each row in T2 that does not satisfy the join condition with any row in T1, a joined row is returned with NULL values in columns of T1. This is the converse of a left join: the result table will unconditionally have a row for each row in T2.

FULL OUTER JOIN

First, an INNER JOIN is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is returned with null values in columns of T2. Also, for each row of T2 that does not satisfy the join condition with any row in T1, a joined row with null values in the columns of T1 is returned.

Joins of all types can be chained together or nested: either or both of *`T1`* and *`T2`* may be joined tables. Parentheses may be used around JOIN clauses to control the join order. In the absence of parentheses, JOIN clauses nest left-to-right.

2.2.1.2. Subqueries

Subqueries specifying a derived table must be enclosed in parentheses and *must* be named using an AS clause. (See Section 2.2.1.3.)

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

This example is equivalent to `FROM table1 AS alias_name`. More interesting cases, which can't be reduced to a plain join, arise when the subquery involves grouping or aggregation.

2.2.1.3. Table and Column Aliases

A temporary name can be given to tables and complex table references to be used for references to the derived table in further processing. This is called a *table alias*.

```
FROM table_reference AS alias
```

Here, *`alias`* can be any regular identifier. The alias becomes the new name of the table reference for the current query -- it is no longer possible to refer to the table by the original name. Thus

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

is not valid SQL syntax. What will actually happen (this is a PostgreSQL extension to the standard) is that an implicit table reference is added to the FROM clause, so the query is processed as if it were written as

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

Table aliases are mainly for notational convenience, but it is necessary to use them when joining a table to itself, e.g.,

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

Additionally, an alias is required if the table reference is a subquery.

Parentheses are used to resolve ambiguities. The following statement will assign the alias `b` to the result of the join, unlike the previous example:

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

```
FROM table_reference alias
```

This form is equivalent to the previously treated one; the AS key word is noise.

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

In this form, in addition to renaming the table as described above, the columns of the table are also given temporary names for use by the surrounding query. If fewer column aliases are specified than the actual table has columns, the remaining columns are not renamed. This syntax is especially useful for self-joins or subqueries.

When an alias is applied to the output of a JOIN clause, using any of these forms, the alias hides the original names within the JOIN. For example,

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

is valid SQL, but

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

is not valid: the table alias A is not visible outside the alias C.

2.2.1.4. Examples

```
FROM T1 INNER JOIN T2 USING (C)
FROM T1 LEFT OUTER JOIN T2 USING (C)
FROM (T1 RIGHT OUTER JOIN T2 ON (T1.C1=T2.C1)) AS DT1
FROM (T1 FULL OUTER JOIN T2 USING (C)) AS DT1 (DT1C1, DT1C2)

FROM T1 NATURAL INNER JOIN T2
FROM T1 NATURAL LEFT OUTER JOIN T2
FROM T1 NATURAL RIGHT OUTER JOIN T2
FROM T1 NATURAL FULL OUTER JOIN T2
```

```
FROM (SELECT * FROM T1) DT1 CROSS JOIN T2, T3
FROM (SELECT * FROM T1) DT1, T2, T3
```

Above are some examples of joined tables and complex derived tables. Notice how the AS clause renames or names a derived table and how the optional comma-separated list of column names that follows renames the columns. The last two FROM clauses produce the same derived table from T1, T2, and T3. The AS keyword was omitted in naming the subquery as DT1. The keywords OUTER and INNER are noise that can be omitted also.

2.2.2. WHERE clause

The syntax of the WHERE clause is

```
WHERE search_condition
```

where *search_condition* is any value expression as defined in Section 1.3 that returns a value of type boolean.

After the processing of the FROM clause is done, each row of the derived table is checked against the search condition. If the result of the condition is true, the row is kept in the output table, otherwise (that is, if the result is false or NULL) it is discarded. The search condition typically references at least some column in the table generated in the FROM clause; this is not required, but otherwise the WHERE clause will be fairly useless.

> **Note:** Before the implementation of the JOIN syntax, it was necessary to put the join condition of an inner join in the WHERE clause. For example, these table expressions are equivalent:
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```
>
> and
>
> ```
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
> ```
>
> or perhaps even
>
> ```
> FROM a NATURAL JOIN b WHERE b.val > 5
> ```
>
> Which one of these you use is mainly a matter of style. The JOIN syntax in the FROM clause is probably not as portable to other products. For outer joins there is no choice in any case: they must be done in the FROM clause. A ON/USING clause of an outer join is *not* equivalent to a WHERE condition, because it determines the addition of rows (for unmatched input rows) as well as the removal of rows from the final result.

```
FROM FDT WHERE
    C1 > 5

FROM FDT WHERE
    C1 IN (1, 2, 3)
FROM FDT WHERE
    C1 IN (SELECT C1 FROM T2)
FROM FDT WHERE
    C1 IN (SELECT C3 FROM T2 WHERE C2 = FDT.C1 + 10)

FROM FDT WHERE
```

```
    C1 BETWEEN (SELECT C3 FROM T2 WHERE C2 = FDT.C1 + 10) AND 100

FROM FDT WHERE
    EXISTS (SELECT C1 FROM T2 WHERE C2 > FDT.C1)
```

In the examples above, `FDT` is the table derived in the FROM clause. Rows that do not meet the search condition of the where clause are eliminated from `FDT`. Notice the use of scalar subqueries as value expressions. Just like any other query, the subqueries can employ complex table expressions. Notice how `FDT` is referenced in the subqueries. Qualifying `C1` as `FDT.C1` is only necessary if `C1` is also the name of a column in the derived input table of the subquery. Qualifying the column name adds clarity even when it is not needed. This shows how the column naming scope of an outer query extends into its inner queries.

2.2.3. GROUP BY and HAVING clauses

After passing the WHERE filter, the derived input table may be subject to grouping, using the GROUP BY clause, and elimination of group rows using the HAVING clause.

```
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference [, grouping_column_reference]...
```

The GROUP BY clause is used to group together rows in a table that share the same values in all the columns listed. The order in which the columns are listed does not matter (as opposed to an ORDER BY clause). The purpose is to reduce each group of rows sharing common values into one group row that is representative of all rows in the group. This is done to eliminate redundancy in the output and/or obtain aggregates that apply to these groups.

Once a table is grouped, columns that are not used in the grouping cannot be referenced except in aggregate expressions, since a specific value in those columns is ambiguous - which row in the group should it come from? The grouped-by columns can be referenced in select list column expressions since they have a known constant value per group. Aggregate functions on the ungrouped columns provide values that span the rows of a group, not of the whole table. For instance, a `sum(sales)` on a table grouped by product code gives the total sales for each product, not the total sales on all products. Aggregates computed on the ungrouped columns are representative of the group, whereas individual values of an ungrouped column are not.

Example:

```
SELECT pid, p.name, (sum(s.units) * p.price) AS sales
  FROM products p LEFT JOIN sales s USING ( pid )
  GROUP BY pid, p.name, p.price;
```

In this example, the columns `pid`, `p.name`, and `p.price` must be in the GROUP BY clause since they are referenced in the query select list. The column s.units does not have to be in the GROUP BY list since it is only used in an aggregate expression (`sum()`), which represents the group of sales of a product. For each product, a summary row is returned about all sales of the product.

In strict SQL, GROUP BY can only group by columns of the source table but PostgreSQL extends this to also allow GROUP BY to group by select columns in the query select list. Grouping by value expressions instead of simple column names is also allowed.

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

If a table has been grouped using a GROUP BY clause, but then only certain groups are of interest, the HAVING clause can be used, much like a WHERE clause, to eliminate groups from a grouped table. PostgreSQL allows a HAVING clause to be used without a GROUP BY, in which case it acts like another WHERE clause, but the point in using HAVING that way is not clear. A good rule of thumb is that a HAVING condition should refer to the results of aggregate functions. A restriction that does not involve an aggregate is more efficiently expressed in the WHERE clause.

Example:

```
SELECT pid    AS "Products",
       p.name AS "Over 5000",
       (sum(s.units) * (p.price - p.cost)) AS "Past Month Profit"
  FROM products p LEFT JOIN sales s USING ( pid )
  WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
  GROUP BY pid, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

In the example above, the WHERE clause is selecting rows by a column that is not grouped, while the HAVING clause restricts the output to groups with total gross sales over 5000.

2.3. Select Lists

As shown in the previous section, the table expression in the **SELECT** command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the *select list*. The select list determines which *columns* of the intermediate table are actually output. The simplest kind of select list is * which emits all columns that the table expression produces. Otherwise, a select list is a comma-separated list of value expressions (as defined in Section 1.3). For instance, it could be a list of column names:

```
SELECT a, b, c FROM ...
```

The columns names a, b, and c are either the actual names of the columns of tables referenced in the FROM clause, or the aliases given to them as explained in Section 2.2.1.3. The name space available in the select list is the same as in the WHERE clause (unless grouping is used, in which case it is the same as in the HAVING clause). If more than one table has a column of the same name, the table name must also be given, as in

```
SELECT tbl1.a, tbl2.b, tbl1.c FROM ...
```

(see also Section 2.2.2).

If an arbitrary value expression is used in the select list, it conceptually adds a new virtual column to the returned table. The value expression is evaluated once for each retrieved row, with the row's values substituted for any column references. But the expressions in the select list do not have to reference any columns in the table expression of the FROM clause; they could be constant arithmetic expressions as well, for instance.

2.3.1. Column Labels

The entries in the select list can be assigned names for further processing. The "further processing" in this case is an optional sort specification and the client application (e.g., column headers for display). For example:

```
SELECT a AS value, b + c AS sum FROM ...
```

If no output column name is specified via AS, the system assigns a default name. For simple column references, this is the name of the referenced column. For function calls, this is the name of the function. For complex expressions, the system will generate a generic name.

> **Note:** The naming of output columns here is different from that done in the FROM clause (see Section 2.2.1.3). This pipeline will in fact allow you to rename the same column twice, but the name chosen in the select list is the one that will be passed on.

2.3.2. DISTINCT

After the select list has been processed, the result table may optionally be subject to the elimination of duplicates. The DISTINCT key word is written directly after the SELECT to enable this:

```
SELECT DISTINCT select_list ...
```

(Instead of DISTINCT the word ALL can be used to select the default behavior of retaining all rows.)

Obviously, two rows are considered distinct if they differ in at least one column value. NULLs are considered equal in this comparison.

Alternatively, an arbitrary expression can determine what rows are to be considered distinct:

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

Here *expression* is an arbitrary value expression that is evaluated for all rows. A set of rows for which all the expressions are equal are considered duplicates, and only the first row of the set is kept in the output. Note that the "first row" of a set is unpredictable unless the query is sorted on enough columns to guarantee a unique ordering of the rows arriving at the DISTINCT filter. (DISTINCT ON processing occurs after ORDER BY sorting.)

The DISTINCT ON clause is not part of the SQL standard and is sometimes considered bad style because of the potentially indeterminate nature of its results. With judicious use of GROUP BY and subselects in FROM the construct can be avoided, but it is very often the most convenient alternative.

2.4. Combining Queries

The results of two queries can be combined using the set operations union, intersection, and difference. The syntax is

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

query1 and *query2* are queries that can use any of the features discussed up to this point. Set operations can also be nested and chained, for example

```
query1 UNION query2 UNION query3
```

which really says

```
(query1 UNION query2) UNION query3
```

UNION effectively appends the result of *query2* to the result of *query1* (although there is no guarantee that this is the order in which the rows are actually returned). Furthermore, it eliminates all duplicate rows, in the sense of DISTINCT, unless ALL is specified.

INTERSECT returns all rows that are both in the result of *query1* and in the result of *query2*. Duplicate rows are eliminated unless ALL is specified.

EXCEPT returns all rows that are in the result of *query1* but not in the result of *query2*. Again, duplicates are eliminated unless ALL is specified.

In order to calculate the union, intersection, or difference of two queries, the two queries must be "union compatible", which means that they both return the same number of columns, and that the corresponding columns have compatible data types, as described in Section 5.6.

2.5. Sorting Rows

After a query has produced an output table (after the select list has been processed) it can optionally be sorted. If sorting is not chosen, the rows will be returned in random order. The actual order in that case will depend on the scan and join plan types and the order on disk, but it must not be relied on. A particular output ordering can only be guaranteed if the sort step is explicitly chosen.

The ORDER BY clause specifies the sort order:

```
SELECT select_list
    FROM table_expression
    ORDER BY column1 [ASC | DESC] [, column2 [ASC | DESC] ...]
```

column1, etc., refer to select list columns. These can be either the output name of a column (see Section 2.3.1) or the number of a column. Some examples:

```
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;
```

As an extension to the SQL standard, PostgreSQL also allows ordering by arbitrary expressions:

```
SELECT a, b FROM table1 ORDER BY a + b;
```

References to column names in the FROM clause that are renamed in the select list are also allowed:

```
SELECT a AS b FROM table1 ORDER BY a;
```

But these extensions do not work in queries involving UNION, INTERSECT, or EXCEPT, and are not portable to other DBMS.

Each column specification may be followed by an optional ASC or DESC to set the sort direction. ASC is default. Ascending order puts smaller values first, where "smaller" is defined in terms of the $<$ operator. Similarly, descending order is determined with the $>$ operator.

If more than one sort column is specified, the later entries are used to sort rows that are equal under the order imposed by the earlier sort specifications.

2.6. LIMIT and OFFSET

```
SELECT select_list
    FROM table_expression
    [LIMIT { number | ALL }] [OFFSET number]
```

LIMIT allows you to retrieve just a portion of the rows that are generated by the rest of the query. If a limit count is given, no more than that many rows will be returned. LIMIT ALL is the same as omitting a LIMIT clause.

OFFSET says to skip that many rows before beginning to return rows to the client. OFFSET 0 is the same as omitting an OFFSET clause. If both OFFSET and LIMIT appear, then OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

When using LIMIT, it is a good idea to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows---you may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? The ordering is unknown, unless you specified ORDER BY.

The query optimizer takes LIMIT into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you give for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

Chapter 3. Data Types

PostgreSQL has a rich set of native data types available to users. Users may add new types to PostgreSQL using the **CREATE TYPE** command.

Table 3-1 shows all general-purpose data types included in the standard distribution. Most of the alternative names listed in the "Aliases" column are the names used internally by PostgreSQL for historical reasons. In addition, some internally used or deprecated types are available, but they are not listed here.

Table 3-1. Data Types

Type Name	Aliases	Description
`bigint`	`int8`	signed eight-byte integer
`bigserial`	`serial8`	autoincrementing eight-byte integer
`bit`		fixed-length bit string
`bit varying(`*n*`)`	`varbit(`*n*`)`	variable-length bit string
`boolean`	`bool`	logical Boolean (true/false)
`box`		rectangular box in 2D plane
`bytea`		binary data
`character(`*n*`)`	`char(`*n*`)`	fixed-length character string
`character varying(`*n*`)`	`varchar(`*n*`)`	variable-length character string
`cidr`		IP network address
`circle`		circle in 2D plane
`date`		calendar date (year, month, day)
`double precision`	`float8`	double precision floating-point number
`inet`		IP host address
`integer`	`int`, `int4`	signed four-byte integer
`interval(`*p*`)`		general-use time span
`line`		infinite line in 2D plane
`lseg`		line segment in 2D plane
`macaddr`		MAC address
`money`		US-style currency
`numeric [ (`*p*`, `*s*`) ]`	`decimal [ (`*p*`, `*s*`) ]`	exact numeric with selectable precision
`oid`		object identifier
`path`		open and closed geometric path in 2D plane
`point`		geometric point in 2D plane
`polygon`		closed geometric path in 2D plane
`real`	`float4`	single precision floating-point number

Type Name	Aliases	Description
smallint	int2	signed two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(*p*)] [without time zone]		time of day
time [(*p*)] with time zone	timetz	time of day, including time zone
timestamp [(*p*)] without time zone	timestamp	date and time
timestamp [(*p*)] [with time zone]	timestamptz	date and time, including time zone

Compatibility: The following types (or spellings thereof) are specified by SQL: bit, bit varying, boolean, char, character, character varying, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time, timestamp (both with or without time zone).

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL, such as open and closed paths, or have several possibilities for formats, such as the date and time types. Most of the input and output functions corresponding to the base types (e.g., integers and floating-point numbers) do some error-checking. Some of the input and output functions are not invertible. That is, the result of an output function may lose precision when compared to the original input.

Some of the operators and functions (e.g., addition and multiplication) do not perform run-time error-checking in the interests of improving execution speed. On some systems, for example, the numeric operators for some data types may silently underflow or overflow.

3.1. Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers and fixed-precision decimals.

Table 3-2. Numeric Types

Type name	Storage size	Description	Range
smallint	2 bytes	Fixed-precision	-32768 to +32767
integer	4 bytes	Usual choice for fixed-precision	-2147483648 to +2147483647
bigint	8 bytes	Very large range fixed-precision	-9223372036854775808 to 9223372036854775807

Type name	Storage size	Description	Range
`decimal`	variable	user-specified precision, exact	no limit
`numeric`	variable	user-specified precision, exact	no limit
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	autoincrementing integer	1 to 9223372036854775807

The syntax of constants for the numeric types is described in Section 1.1.2. The numeric types have a full set of corresponding arithmetic operators and functions. Refer to Chapter 4 for more information. The following sections describe the types in detail.

3.1.1. The Integer Types

The types `smallint`, `integer`, `bigint` store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error.

The type `integer` is the usual choice, as it offers the best balance between range, storage size, and performance. The `smallint` type is generally only used if disk space is at a premium. The `bigint` type should only be used if the `integer` range is not sufficient, because the latter is definitely faster.

The `bigint` type may not function correctly on all platforms, since it relies on compiler support for eight-byte integers. On a machine without such support, `bigint` acts the same as `integer` (but still takes up eight bytes of storage). However, we are not aware of any reasonable platform where this is actually the case.

SQL only specifies the integer types `integer` (or `int`) and `smallint`. The type `bigint`, and the type names `int2`, `int4`, and `int8` are extensions, which are shared with various other RDBMS products.

> **Note:** If you have a column of type `smallint` or `bigint` with an index, you may encounter problems getting the system to use that index. For instance, a clause of the form
>
> ```
> ... WHERE smallint_column = 42
> ```
>
> will not use an index, because the system assigns type `integer` to the constant 42, and PostgreSQL currently cannot use an index when two different data types are involved. A workaround is to single-quote the constant, thus:
>
> ```
> ... WHERE smallint_column = '42'
> ```
>
> This will cause the system to delay type resolution and will assign the right type to the constant.

3.1.2. Arbitrary Precision Numbers

The type `numeric` can store numbers of practically unlimited size and precision, while being able to store all numbers and carry out all calculations exactly. It is especially recommended for storing monetary amounts and other quantities where exactness is required. However, the `numeric` type is very slow compared to the floating-point types described in the next section.

In what follows we use these terms: The *scale* of a `numeric` is the count of decimal digits in the fractional part, to the right of the decimal point. The *precision* of a `numeric` is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the precision and the scale of the numeric type can be configured. To declare a column of type `numeric` use the syntax

```
NUMERIC(precision, scale)
```

The precision must be positive, the scale zero or positive. Alternatively,

```
NUMERIC(precision)
```

selects a scale of 0. Specifying

```
NUMERIC
```

without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas `numeric` columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer accuracy. We find this a bit useless. If you're concerned about portability, always specify the precision and scale explicitly.)

If the precision or scale of a value is greater than the declared precision or scale of a column, the system will attempt to round the value. If the value cannot be rounded so as to satisfy the declared limits, an error is raised.

The types `decimal` and `numeric` are equivalent. Both types are part of the SQL standard.

3.1.3. Floating-Point Types

The data types `real` and `double precision` are inexact, variable-precision numeric types. In practice, these types are usually implementations of IEEE 754 binary floating point (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and printing back out a value may show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed further here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the `numeric` type instead.

- If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
- Comparing two floating-point values for equality may or may not work as expected.

Normally, the `real` type has a range of at least -1E+37 to +1E+37 with a precision of at least 6 decimal digits. The `double precision` type normally has a range of around -1E+308 to +1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding may take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

3.1.4. The Serial Types

The `serial` data types are not truly types, but are a notational convenience for setting up unique identifier columns in tables. In the current implementation, specifying

```
CREATE TABLE tablename (
    colname SERIAL
);
```

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
    colname integer DEFAULT nextval('tablename_colname_seq') UNIQUE NOT NULL
);
```

Thus, we have created an integer column and arranged for its default values to be assigned from a sequence generator. UNIQUE and NOT NULL constraints are applied to ensure that explicitly-inserted values will never be duplicates, either.

The type names `serial` and `serial4` are equivalent: both create `integer` columns. The type names `bigserial` and `serial8` work just the same way, except that they create a `bigint` column. `bigserial` should be used if you anticipate use of more than 2^{31} identifiers over the lifetime of the table.

Implicit sequences supporting the `serial` types are not automatically dropped when a table containing a serial type is dropped. So, the following commands executed in order will likely fail:

```
CREATE TABLE tablename (colname SERIAL);
DROP TABLE tablename;
CREATE TABLE tablename (colname SERIAL);
```

The sequence will remain in the database until explicitly dropped using **DROP SEQUENCE**. (This annoyance will probably be changed in some future release.)

3.2. Monetary Type

> **Deprecated:** The `money` type is deprecated. Use `numeric` or `decimal` instead, in combination with the `to_char` function. The money type may become a locale-aware layer over the `numeric` type in a future release.

The `money` type stores U.S.-style currency with fixed decimal point representation. If PostgreSQL is compiled with locale support then the `money` type uses locale-specific output formatting.

Input is accepted in a variety of formats, including integer and floating-point literals, as well as "typical" currency formatting, such as `'$1,000.00'`. Output is in the latter form.

Table 3-3. Monetary Types

Type Name	Storage	Description	Range
money	4 bytes	Fixed-precision	-21474836.48 to +21474836.47

3.3. Character Types

Table 3-4. Character Types

Type name	Description
`character(`*n*`)`, `char(`*n*`)`	Fixed-length blank padded
`character varying(`*n*`)`, `varchar(`*n*`)`	Variable-length with limit
`text`	Variable unlimited length

SQL defines two primary character types: `character(`*n*`)` and `character varying(`*n*`)`, where *n* is a positive integer. Both of these types can store strings up to *n* characters in length. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) If the string to be stored is shorter than the declared length, values of type `character` will be space-padded; values of type `character varying` will simply store the shorter string.

> **Note:** Prior to PostgreSQL 7.2, strings that were too long were silently truncated, no error was raised.

The notations `char(`*n*`)` and `varchar(`*n*`)` are aliases for `character(`*n*`)` and `character varying(`*n*`)`, respectively. `character` without length specifier is equivalent to `character(1)`; if `character varying` is used without length specifier, the type accepts strings of any size. The latter is a PostgreSQL extension.

In addition, PostgreSQL supports the more general `text` type, which stores strings of any length. Unlike `character varying`, `text` does not require an explicit declared upper limit on the size of the string. Although the type `text` is not in the SQL standard, many other RDBMS packages have it as well.

The storage requirement for data of these types is 4 bytes plus the actual string, and in case of `character` plus the padding. Long strings will be compressed by the system automatically, so the physical requirement on disk may be less. In any case, the longest possible character string that can be stored is about 1 GB. (The maximum value that will be allowed for *n* in the data type declaration is less than that. It wouldn't be very useful to change this because with multibyte character encodings the number of characters and bytes can be quite different anyway. If you desire to store long strings with no spe-

cific upper limit, use `text` or `character varying` without a length specifier, rather than making up an arbitrary length limit.)

> **Tip:** There are no performance differences between these three types, apart from the increased storage size when using the blank-padded type.

Refer to Section 1.1.2.1 for information about the syntax of string literals, and to Chapter 4 for information about available operators and functions.

Example 3-1. Using the character types

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- ❶
  a   | char_length
------+-------------
 ok   |           4

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
SELECT b, char_length(b) FROM test2;
   b   | char_length
-------+-------------
 ok    |           2
 good  |           5
```

❶ The `char_length` function is discussed in Section 4.4.

There are two other fixed-length character types in PostgreSQL. The `name` type exists *only* for storage of internal catalog names and is not intended for use by the general user. Its length is currently defined as 32 bytes (31 usable characters plus terminator) but should be referenced using the macro `NAMEDATALEN`. The length is set at compile time (and is therefore adjustable for special uses); the default maximum length may change in a future release. The type `"char"` (note the quotes) is different from `char(1)` in that it only uses one byte of storage. It is internally used in the system catalogs as a poor-man's enumeration type.

Table 3-5. Specialty Character Type

Type Name	Storage	Description
"char"	1 byte	Single character internal type
name	32 bytes	Thirty-one character internal type

3.4. Binary Strings

The `bytea` data type allows storage of binary strings.

Table 3-6. Binary String Types

Type Name	Storage	Description
bytea	4 bytes plus the actual binary string	Variable (not specifically limited) length binary string

A binary string is a sequence of octets that does not have either a character set or collation associated with it. Bytea specifically allows storing octets of zero value and other "non-printable" octets.

Octets of certain values *must* be escaped (but all octet values *may* be escaped) when used as part of a string literal in an SQL statement. In general, to escape an octet, it is converted into the three-digit octal number equivalent of its decimal octet value, and preceded by two backslashes. Some octet values have alternate escape sequences, as shown in Table 3-7.

Table 3-7. SQL Literal Escaped Octets

Decimal Octet Value	Description	Input Escaped Representation	Example	Printed Result
0	zero octet	'\\000'	select '\\000'::bytea;	\000
39	single quote	'\\" or '\\047'	select '\"::bytea;	'
92	backslash	'\\\\' or '\\134'	select '\\\\'::bytea;	\\

Note that the result in each of the examples above was exactly one octet in length, even though the output representation of the zero octet and backslash are more than one character. Bytea output octets are also escaped. In general, each "non-printable" octet decimal value is converted into its equivalent three digit octal value, and preceded by one backslash. Most "printable" octets are represented by their standard representation in the client character set. The octet with decimal value 92 (backslash) has a special alternate output representation. Details are in Table 3-8.

Table 3-8. SQL Output Escaped Octets

Decimal Octet Value	Description	Output Escaped Representation	Example	Printed Result
92	backslash	\\	select '\\134'::bytea;	\\
0 to 31 and 127 to 255	"non-printable" octets	\### (octal value)	select '\\001'::bytea;	\001
32 to 126	"printable" octets	ASCII representation	select '\\176'::bytea;	~

SQL string literals (input strings) must be preceded with two backslashes due to the fact that they must pass through two parsers in the PostgreSQL backend. The first backslash is interpreted as an escape character by the string-literal parser, and therefore is consumed, leaving the octets that follow. The remaining backslash is recognized by the bytea input function as the prefix of a three digit octal

value. For example, a string literal passed to the backend as `'\\001'` becomes `'\001'` after passing through the string-literal parser. The `'\001'` is then sent to the `bytea` input function, where it is converted to a single octet with a decimal value of 1.

For a similar reason, a backslash must be input as `'\\\\'` (or `'\\134'`). The first and third backslashes are interpreted as escape characters by the string-literal parser, and therefore are consumed, leaving two backslashes in the string passed to the `bytea` input function, which interprets them as representing a single backslash. For example, a string literal passed to the backend as `'\\\\'` becomes `'\\'` after passing through the string-literal parser. The `'\\'` is then sent to the `bytea` input function, where it is converted to a single octet with a decimal value of 92.

A single quote is a bit different in that it must be input as `'\"` (or `'\\134'`), *not* as `'\\"`. This is because, while the literal parser interprets the single quote as a special character, and will consume the single backslash, the `bytea` input function does *not* recognize a single quote as a special octet. Therefore a string literal passed to the backend as `'\"` becomes `"'` after passing through the string-literal parser. The `"'` is then sent to the `bytea` input function, where it is retains its single octet decimal value of 39.

Depending on the front end to PostgreSQL you use, you may have additional work to do in terms of escaping and unescaping `bytea` strings. For example, you may also have to escape line feeds and carriage returns if your interface automatically translates these. Or you may have to double up on backslashes if the parser for your language or choice also treats them as an escape character.

`Bytea` provides most of the functionality of the binary string type per SQL99 section 4.3. A comparison of SQL99 Binary Strings and PostgreSQL `bytea` is presented in Table 3-9.

Table 3-9. Comparison of SQL99 Binary String and PostgreSQL `BYTEA` types

SQL99	BYTEA
Name of data type `BINARY LARGE OBJECT` or `BLOB`	Name of data type `BYTEA`
Sequence of octets that does not have either a character set or collation associated with it.	same
Described by a binary data type descriptor containing the name of the data type and the maximum length in octets	Described by a binary data type descriptor containing the name of the data type with no specific maximum length
All binary strings are mutually comparable in accordance with the rules of comparison predicates.	same
Binary string values can only be compared for equality.	Binary string values can be compared for equality, greater than, greater than or equal, less than, less than or equal
Operators operating on and returning binary strings include concatenation, substring, overlay, and trim	Operators operating on and returning binary strings include concatenation, substring, and trim. The `leading` and `trailing` arguments for trim are not yet implemented.
Other operators involving binary strings include length, position, and the like predicate	same
A binary string literal is comprised of an even number of hexadecimal digits, in single quotes, preceded by “X”, e.g. `X'1a43fe'`	A binary string literal is comprised of octets escaped according to the rules shown in Table 3-7

3.5. Date/Time Types

PostgreSQL supports the full set of SQL date and time types.

Table 3-10. Date/Time Types

Type	Description	Storage	Earliest	Latest	Resolution
`timestamp [ (p) ] without time zone`	both date and time	8 bytes	4713 BC	AD 1465001	1 microsecond / 14 digits
`timestamp [ (p) ] [ with time zone ]`	both date and time	8 bytes	4713 BC	AD 1465001	1 microsecond / 14 digits
`interval [ (p) ]`	for time intervals	12 bytes	-178000000 years	178000000 years	1 microsecond
`date`	dates only	4 bytes	4713 BC	32767 AD	1 day
`time [ (p) ] [ without time zone ]`	times of day only	8 bytes	00:00:00.00	23:59:59.99	1 microsecond
`time [ (p) ] with time zone`	times of day only	12 bytes	00:00:00.00+12	23:59:59.99-12	1 microsecond

`time`, `timestamp`, and `interval` accept an optional precision value *p* which specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The effective limit of precision is determined by the underlying double precision floating-point number used to store values (in seconds for `interval` and in seconds since 2000-01-01 for `timestamp`). The useful range of *p* is from 0 to about 6 for `timestamp`, but may be more for `interval`. The system will accept *p* ranging from 0 to 13.

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900's, but continue to be prone to arbitrary changes. PostgreSQL uses your operating system's underlying features to provide output time-zone support, and these systems usually contain information for only the time period 1902 through 2038 (corresponding to the full range of conventional Unix system time). `timestamp with time zone` and `time with time zone` will use time zone information only within that year range, and assume that times outside that range are in UTC.

To ensure an upgrade path from versions of PostgreSQL earlier than 7.0, we recognize `datetime` (equivalent to `timestamp`) and `timespan` (equivalent to `interval`). These types are now restricted to having an implicit translation to `timestamp` and `interval`, and support for these will be removed in the next release of PostgreSQL (likely named 7.3).

The types `abstime` and `reltime` are lower precision types which are used internally. You are discouraged from using any of these types in new applications and are encouraged to move any old ones over when appropriate. Any or all of these internal types might disappear in a future release.

3.5.1. Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional PostgreSQL, and others. For some formats, ordering of month and day in date input can be ambiguous and there is support for specifying the expected ordering of these fields. The command `SET DateStyle TO 'US'` or `SET DateStyle TO 'NonEuropean'` specifies the variant "month before day", the command `SET DateStyle TO 'European'` sets the variant "day before month". The `ISO` style is the default but this default can be changed at compile time or at run time.

PostgreSQL is more flexible in handling date/time than the SQL standard requires. See Appendix A for the exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones.

Remember that any date or time literal input needs to be enclosed in single quotes, like text strings. Refer to Section 1.1.2.5 for more information. SQL9x requires the following syntax

type [(*p*)] '*value*'

where *p* in the optional precision specification is an integer corresponding to the number of fractional digits in the seconds field. Precision can be specified for `time`, `timestamp`, and `interval` types.

3.5.1.1. `date`

The following are some possible inputs for the `date` type.

Table 3-11. Date Input

Example	Description
January 8, 1999	Unambiguous
1999-01-08	ISO-8601 format, preferred
1/8/1999	U.S.; read as August 1 in European mode
8/1/1999	European; read as August 1 in U.S. mode
1/18/1999	U.S.; read as January 18 in any mode
19990108	ISO-8601 year, month, day
990108	ISO-8601 year, month, day
1999.008	Year and day of year
99008	Year and day of year
J2451187	Julian day
January 8, 99 BC	Year 99 before the Common Era

3.5.1.2. `time [ ( p ) ] [ without time zone ]`

Per SQL99, this type can be specified as `time` or as `time without time zone`. The optional precision *p* should be between 0 and 13, and defaults to the precision of the input time literal.

The following are valid `time` inputs.

Table 3-12. Time Input

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	Same as 04:05; AM does not affect value
04:05 PM	Same as 16:05; input hour must be <= 12
allballs	Same as 00:00:00

3.5.1.3. `time [ ( precision ) ] with time zone`

This type is defined by SQL92, but the definition exhibits properties which lead to questionable usefulness. In most cases, a combination of `date`, `time`, `timestamp without time zone` and `timestamp with time zone` should provide a complete range of date/time functionality required by any application.

The optional precision *p* should be between 0 and 13, and defaults to the precision of the input time literal.

`time with time zone` accepts all input also legal for the `time` type, appended with a legal time zone, as follows:

Table 3-13. Time With Time Zone Input

Example	Description
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601

Refer to Table 3-14 for more examples of time zones.

3.5.1.4. `timestamp [ (precision) ] without time zone`

Valid input for the `timestamp [ (p) ] without time zone` type consists of a concatenation of a date and a time, followed by an optional `AD` or `BC`, followed by an optional time zone. (See below.) Thus

```
1999-01-08 04:05:06
```

is a valid `timestamp without time zone` value that is ISO-compliant. In addition, the widespread format

```
January 8 04:05:06 1999 PST
```

is supported.

The optional precision *p* should be between 0 and 13, and defaults to the precision of the input `timestamp` literal.

For `timestamp without time zone`, any explicit time zone specified in the input is silently swallowed. That is, the resulting date/time value is derived from the explicit date/time fields in the input value, and is not adjusted for time zone.

3.5.1.5. `timestamp [ (`*`precision`*`) ] with time zone`

Valid input for the `timestamp` type consists of a concatenation of a date and a time, followed by an optional `AD` or `BC`, followed by an optional time zone. (See below.) Thus

```
1999-01-08 04:05:06 -8:00
```

is a valid `timestamp` value that is ISO-compliant. In addition, the wide-spread format

```
January 8 04:05:06 1999 PST
```

is supported.

The optional precision *p* should be between 0 and 13, and defaults to the precision of the input `timestamp` literal.

Table 3-14. Time Zone Input

Time Zone	Description
PST	Pacific Standard Time
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST

3.5.1.6. `interval [ (` *`precision`* `) ]`

`interval` values can be written with the following syntax:

```
  Quantity Unit [Quantity Unit...] [Direction]
@ Quantity Unit [Quantity Unit...] [Direction]
```

where: `Quantity` is a number (possibly signed), `Unit` is `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, or abbreviations or plurals of these units; `Direction` can be `ago` or empty. The at sign (`@`) is optional noise. The amounts of different units are implicitly added up with appropriate sign accounting.

Quantities of days, hours, minutes, and seconds can be specified without explicit unit markings. For example, `'1 12:59:10'` is read the same as `'1 day 12 hours 59 min 10 sec'`.

The optional precision *p* should be between 0 and 13, and defaults to the precision of the input literal.

3.5.1.7. Special values

The following SQL-compatible functions can be used as date or time input for the corresponding data type: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`. The latter two accept an optional

precision specification.

PostgreSQL also supports several special constants for convenience.

Table 3-15. Special Date/Time Constants

Constant	Description
epoch	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	Later than other valid times
-infinity	Earlier than other valid times
invalid	Illegal entry
now	Current transaction time
today	Midnight today
tomorrow	Midnight tomorrow
yesterday	Midnight yesterday
zulu, allballs, z	00:00:00.00 GMT

`'now'` is evaluated when the value is first interpreted.

> **Note:** As of PostgreSQL version 7.2, `'current'` is no longer supported as a date/time constant. Previously, `'current'` was stored as a special value, and evaluated to `'now'` only when used in an expression or type conversion.

3.5.2. Date/Time Output

Output formats can be set to one of the four styles ISO 8601, SQL (Ingres), traditional PostgreSQL, and German, using the **SET DateStyle**. The default is the ISO format.

Table 3-16. Date/Time Output Styles

Style Specification	Description	Example
'ISO'	ISO-8601 standard	1997-12-17 07:37:16-08
'SQL'	Traditional style	12/17/1997 07:37:16.00 PST
'PostgreSQL'	Original style	Wed Dec 17 07:37:16 1997 PST
'German'	Regional style	17.12.1997 07:37:16.00 PST

The output of the `date` and `time` styles is of course only the date or time part in accordance with the above examples.

The SQL style has European and non-European (U.S.) variants, which determines whether month follows day or vice versa. (See also Section 3.5.1 for how this setting affects interpretation of input values.)

Table 3-17. Date-Order Conventions

Style Specification	Description	Example
European	*day*/*month*/*year*	17/12/1997 15:37:16.00 MET
US	*month*/*day*/*year*	12/17/1997 07:37:16.00 PST

`interval` output looks like the input format, except that units like `week` or `century` are converted to years and days. In ISO mode the output looks like

```
[ Quantity Units [ ... ] ] [ Days ] Hours:Minutes [ ago ]
```

There are several ways to affect the appearance of date/time types:

- The `PGDATESTYLE` environment variable used by the backend directly on postmaster start-up.
- The `PGDATESTYLE` environment variable used by the frontend libpq on session start-up.
- **SET DATESTYLE** SQL command.

3.5.3. Time Zones

PostgreSQL endeavors to be compatible with SQL92 definitions for typical usage. However, the SQL92 standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the `date` type does not have an associated time zone, the `time` type can. Time zones in the real world can have no meaning unless associated with a date as well as a time since the offset may vary through the year with daylight-saving time boundaries.
- The default time zone is specified as a constant integer offset from GMT/UTC. It is not possible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.

To address these difficulties, we recommend using date/time types that contain both date and time when using time zones. We recommend *not* using the SQL92 type `time with time zone` (though it is supported by PostgreSQL for legacy applications and for compatibility with other RDBMS implementations). PostgreSQL assumes your local time zone for any type containing only date or time. Further, time zone support is derived from the underlying operating system time-zone capabilities, and hence can handle daylight-saving time and other expected behavior.

PostgreSQL obtains time-zone support from the underlying operating system for dates between 1902 and 2038 (near the typical date limits for Unix-style systems). Outside of this range, all dates are assumed to be specified and used in Universal Coordinated Time (UTC).

All dates and times are stored internally in UTC, traditionally known as Greenwich Mean Time (GMT). Times are converted to local time on the database server before being sent to the client frontend, hence by default are in the server time zone.

There are several ways to affect the time-zone behavior:

- The `TZ` environment variable is used by the backend directly on postmaster start-up as the default time zone.
- The `PGTZ` environment variable, if set at the client, is used by libpq to send a **SET TIME ZONE** command to the backend upon connection.

- The SQL command **SET TIME ZONE** sets the time zone for the session.
- The SQL92 qualifier on

  timestamp AT TIME ZONE *'zone'*

  where *zone* can be specified as a text time zone (e.g. 'PST') or as an interval (e.g. INTERVAL '-08:00').

Note: If an invalid time zone is specified, the time zone becomes GMT (on most systems anyway).

Note: If the runtime option AUSTRALIAN_TIMEZONES is set then CST and EST refer to Australian time zones, not American ones.

3.5.4. Internals

PostgreSQL uses Julian dates for all date/time calculations. They have the nice property of correctly predicting/calculating any date more recent than 4713BC to far into the future, using the assumption that the length of the year is 365.2425 days.

Date conventions before the 19th century make for interesting reading, but are not consistent enough to warrant coding into a date/time handler.

3.6. Boolean Type

PostgreSQL provides the SQL99 type boolean. boolean can have one of only two states: “true” or “false”. A third state, “unknown”, is represented by the SQL NULL state.

Valid literal values for the “true” state are:

```
TRUE
't'
'true'
'y'
'yes'
'1'
```

For the “false” state, the following values can be used:

```
FALSE
'f'
'false'
'n'
'no'
'0'
```

Using the key words TRUE and FALSE is preferred (and SQL-compliant).

Example 3-2. Using the `boolean` type

```
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
```

Example 3-2 shows that `boolean` values are output using the letters `t` and `f`.

> **Tip:** Values of the `boolean` type cannot be cast directly to other types (e.g., `CAST (boolval AS integer)` does not work). This can be accomplished using the `CASE` expression: `CASE WHEN boolval THEN 'value if true' ELSE 'value if false' END`. See also Section 4.12.

`boolean` uses 1 byte of storage.

3.7. Geometric Types

Geometric types represent two-dimensional spatial objects. The most fundamental type, the point, forms the basis for all of the other types.

Table 3-18. Geometric Types

Geometric Type	Storage	Representation	Description
`point`	16 bytes	(x,y)	Point in space
`line`	32 bytes	((x1,y1),(x2,y2))	Infinite line
`lseg`	32 bytes	((x1,y1),(x2,y2))	Finite line segment
`box`	32 bytes	((x1,y1),(x2,y2))	Rectangular box
`path`	4+32n bytes	((x1,y1),...)	Closed path (similar to polygon)
`path`	4+32n bytes	[(x1,y1),...]	Open path
`polygon`	4+32n bytes	((x1,y1),...)	Polygon (similar to closed path)
`circle`	24 bytes	<(x,y),r>	Circle (center and radius)

A rich set of functions and operators is available to perform various geometric operations such as scaling, translation, rotation, and determining intersections.

3.7.1. Point

Points are the fundamental two-dimensional building block for geometric types.

`point` is specified using the following syntax:

```
( x , y )
  x , y
```

where the arguments are

x

The x-axis coordinate as a floating-point number

y

The y-axis coordinate as a floating-point number

3.7.2. Line Segment

Line segments (`lseg`) are represented by pairs of points.

`lseg` is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where the arguments are

(*x1*,*y1*)
(*x2*,*y2*)

The end points of the line segment

3.7.3. Box

Boxes are represented by pairs of points that are opposite corners of the box.

`box` is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where the arguments are

(*x1*,*y1*)
(*x2*,*y2*)

Opposite corners of the box

Boxes are output using the first syntax. The corners are reordered on input to store the upper right corner, then the lower left corner. Other corners of the box can be entered, but the lower left and upper right corners are determined from the input and stored.

3.7.4. Path

Paths are represented by connected sets of points. Paths can be *open*, where the first and last points in the set are not connected, and *closed*, where the first and last point are connected. Functions `popen(p)` and `pclose(p)` are supplied to force a path to be open or closed, and functions `isopen(p)` and `isclosed(p)` are supplied to test for either type in a query.

`path` is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the arguments are

(*x*,*y*)

End points of the line segments comprising the path. A leading square bracket ("[") indicates an open path, while a leading parenthesis ("(") indicates a closed path.

Paths are output using the first syntax.

3.7.5. Polygon

Polygons are represented by sets of points. Polygons should probably be considered equivalent to closed paths, but are stored differently and have their own set of support routines.

`polygon` is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the arguments are

(*x*,*y*)

End points of the line segments comprising the boundary of the polygon

Polygons are output using the first syntax.

3.7.6. Circle

Circles are represented by a center point and a radius.

`circle` is specified using the following syntax:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

where the arguments are

(*x*,*y*)

Center of the circle

r

Radius of the circle

Circles are output using the first syntax.

3.8. Network Address Data Types

PostgreSQL offers data types to store IP and MAC addresses. It is preferable to use these types over plain text types, because these types offer input error checking and several specialized operators and functions.

Table 3-19. Network Address Data Types

Name	Storage	Description	Range
`cidr`	12 bytes	IP networks	valid IPv4 networks
`inet`	12 bytes	IP hosts and networks	valid IPv4 hosts or networks
`macaddr`	6 bytes	MAC addresses	customary formats

IP v6 is not supported, yet.

3.8.1. `inet`

The `inet` type holds an IP host address, and optionally the identity of the subnet it is in, all in one

field. The subnet identity is represented by the number of bits in the network part of the address (the "netmask"). If the netmask is 32, then the value does not indicate a subnet, only a single host. Note that if you want to accept networks only, you should use the `cidr` type rather than `inet`.

The input format for this type is *x.x.x.x/y* where *x.x.x.x* is an IP address and *y* is the number of bits in the netmask. If the */y* part is left off, then the netmask is 32, and the value represents just a single host. On display, the */y* portion is suppressed if the netmask is 32.

3.8.2. `cidr`

The `cidr` type holds an IP network specification. Input and output formats follow Classless Internet Domain Routing conventions. The format for specifying classless networks is *x.x.x.x/y* where *x.x.x.x* is the network and *y* is the number of bits in the netmask. If *y* is omitted, it is calculated using assumptions from the older classful numbering system, except that it will be at least large enough to include all of the octets written in the input.

Here are some examples:

Table 3-20. `cidr` Type Input Examples

`CIDR` Input	`CIDR` Displayed	`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

3.8.3. `inet` vs `cidr`

The essential difference between `inet` and `cidr` data types is that `inet` accepts values with nonzero bits to the right of the netmask, whereas `cidr` does not.

> **Tip:** If you do not like the output format for `inet` or `cidr` values, try the `host`(), `text`(), and `abbrev`() functions.

3.8.4. `macaddr`

The `macaddr` type stores MAC addresses, i.e., Ethernet card hardware addresses (although MAC ad-

dresses are used for other purposes as well). Input is accepted in various customary formats, including

```
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08-00-2b-01-02-03'
'08:00:2b:01:02:03'
```

which would all specify the same address. Upper and lower case is accepted for the digits a through f. Output is always in the last of the shown forms.

The directory contrib/mac in the PostgreSQL source distribution contains tools that can be used to map MAC addresses to hardware manufacturer names.

3.9. Bit String Types

Bit strings are strings of 1's and 0's. They can be used to store or visualize bit masks. There are two SQL bit types: BIT(*x*) and BIT VARYING(*x*); where *x* is a positive integer.

BIT type data must match the length *x* exactly; it is an error to attempt to store shorter or longer bit strings. BIT VARYING is of variable length up to the maximum length *x*; longer strings will be rejected. BIT without length is equivalent to BIT(1), BIT VARYING without length specification means unlimited length.

> **Note:** Prior to PostgreSQL 7.2, BIT type data was zero-padded on the right. This was changed to comply with the SQL standard. To implement zero-padded bit strings, a combination of the concatenation operator and the substring function can be used.

Refer to Section 1.1.2.2 for information about the syntax of bit string constants. Bit-logical operators and string manipulation functions are available; see Chapter 4.

Example 3-3. Using the bit string types

```
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 does not match type bit(3)
SELECT SUBSTRING(b FROM 1 FOR 2) FROM test;
```

Chapter 4. Functions and Operators

PostgreSQL provides a large number of functions and operators for the built-in data types. Users can also define their own functions and operators, as described in the *Programmer's Guide*. The psql commands **\df** and **\do** can be used to show the list of all actually available functions and operators, respectively.

If you are concerned about portability then take note that most of the functions and operators described in this chapter, with the exception of the most trivial arithmetic and comparison operators and some explicitly marked functions, are not specified by the SQL standard. Some of this extended functionality is present in other RDBMS products, and in many cases this functionality is compatible and consistent between various products.

4.1. Logical Operators

The usual logical operators are available:

AND
OR
NOT

SQL uses a three-valued Boolean logic where NULL represents "unknown". Observe the following truth tables:

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

4.2. Comparison Operators

Table 4-1. Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to

Operator	Description
>=	greater than or equal to
=	equal
<> or !=	not equal

> **Note:** The != operator is converted to <> in the parser stage. It is not possible to implement != and <> operators that do different things.

Comparison operators are available for all data types where this makes sense. All comparison operators are binary operators that return values of type `boolean`; expressions like `1 < 2 < 3` are not valid (because there is no < operator to compare a Boolean value with `3`).

In addition to the comparison operators, the special BETWEEN construct is available.

a `BETWEEN` *x* `AND` *y*

is equivalent to

a `>=` *x* `AND` *a* `<=` *y*

Similarly,

a `NOT BETWEEN` *x* `AND` *y*

is equivalent to

a `<` *x* `OR` *a* `>` *y*

There is no difference between the two respective forms apart from the CPU cycles required to rewrite the first one into the second one internally.

To check whether a value is or is not NULL, use the constructs

expression `IS NULL`
expression `IS NOT NULL`

or the equivalent, but less standard, constructs

expression `ISNULL`
expression `NOTNULL`

Do *not* write *expression* `= NULL` because NULL is not "equal to" NULL. (NULL represents an unknown value, and it is not known whether two unknown values are equal.)

Some applications may (incorrectly) require that *expression* `= NULL` returns true if *expression* evaluates to the NULL value. To support these applications, the run-time option `transform_null_equals` can be turned on (e.g., `SET transform_null_equals TO ON;`). PostgreSQL will then convert `x = NULL` clauses to `x IS NULL`. This was the default behavior in releases 6.5 through 7.1.

Boolean values can also be tested using the constructs

expression `IS TRUE`
expression `IS NOT TRUE`

```
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

These are similar to `IS NULL` in that they will always return TRUE or FALSE, never NULL, even when the operand is NULL. A NULL input is treated as the logical value UNKNOWN.

4.3. Mathematical Functions and Operators

Mathematical operators are provided for many PostgreSQL types. For types without common mathematical conventions for all possible permutations (e.g. date/time types) we describe the actual behavior in subsequent sections.

Table 4-2. Mathematical Operators

Name	Description	Example	Result
+	Addition	2 + 3	5
-	Subtraction	2 - 3	-1
*	Multiplication	2 * 3	6
/	Division (integer division truncates results)	4 / 2	2
%	Modulo (remainder)	5 % 4	1
^	Exponentiation	2.0 ^ 3.0	8
|/	Square root	|/ 25.0	5
||/	Cube root	||/ 27.0	3
!	Factorial	5 !	120
!!	Factorial (prefix operator)	!! 5	120
@	Absolute value	@ -5.0	5
&	Binary AND	91 & 15	11
|	Binary OR	32 | 3	35
#	Binary XOR	17 # 5	20
~	Binary NOT	~1	-2
<<	Binary shift left	1 << 4	16
>>	Binary shift right	8 >> 2	2

The “binary” operators are also available for the bit string types `BIT` and `BIT VARYING`.

Table 4-3. Bit String Binary Operators

Example	Result
B’10001’ & B’01101’	00001
B’10001’ | B’01101’	11101
B’10001’ # B’01101’	11110

Example	Result
~ B'10001'	01110
B'10001' << 3	01000
B'10001' >> 2	00100

Bit string arguments to `&`, `|`, and # must be of equal length. When bit shifting, the original length of the string is preserved, as shown here.

Table 4-4. Mathematical Functions

Function	Return Type	Description	Example	Result
`abs(`*x*`)`	(same as x)	absolute value	`abs(-17.4)`	17.4
`cbrt(dp)`	`dp`	cube root	`cbrt(27.0)`	3
`ceil(numeric)`	`numeric`	smallest integer not less than argument	`ceil(-42.8)`	-42
`degrees(dp)`	`dp`	radians to degrees	`degrees(0.5)`	28.6478897565412
`exp(dp)`	`dp`	exponential	`exp(1.0)`	2.71828182845905
`floor(numeric)`	`numeric`	largest integer not greater than argument	`floor(-42.8)`	-43
`ln(dp)`	`dp`	natural logarithm	`ln(2.0)`	0.693147180559945
`log(dp)`	`dp`	base 10 logarithm	`log(100.0)`	2
`log(`*b* `numeric, `*x* `numeric)`	`numeric`	logarithm to base *b*	`log(2.0, 64.0)`	6.0000000000
`mod(`*y*`, `*x*`)`	(same as argument types)	remainder of *y*/*x*	`mod(9,4)`	1
`pi()`	`dp`	"Pi" constant	`pi()`	3.14159265358979
`pow(`*e* `dp, `*n* `dp)`	`dp`	raise a number to exponent *e*	`pow(9.0, 3.0)`	729
`radians(dp)`	`dp`	degrees to radians	`radians(45.0)`	0.785398163397448
`random()`	`dp`	value between 0.0 to 1.0	`random()`	
`round(dp)`	`dp`	round to nearest integer	`round(42.4)`	42
`round(`*v* `numeric, `*s* `integer)`	`numeric`	round to *s* decimal places	`round(42.4382, 2)`	42.44
`sign(numeric)`	`numeric`	sign of the argument (-1, 0, +1)	`sign(-8.4)`	-1
`sqrt(dp)`	`dp`	square root	`sqrt(2.0)`	1.4142135623731

Function	Return Type	Description	Example	Result
trunc(dp)	dp	truncate toward zero	trunc(42.8)	42
trunc(numeric, *s* integer)	numeric	truncate to *s* decimal places	trunc(42.4382, 2)	42.43

In the table above, dp indicates double precision. The functions exp, ln, log, pow, round (1 argument), sqrt, and trunc (1 argument) are also available for the type numeric in place of double precision. Functions returning a numeric result take numeric input arguments, unless otherwise specified. Many of these functions are implemented on top of the host system's C library; accuracy and behavior in boundary cases could therefore vary depending on the host system.

Table 4-5. Trigonometric Functions

Function	Description
acos(*x*)	inverse cosine
asin(*x*)	inverse sine
atan(*x*)	inverse tangent
atan2(*x*, *y*)	inverse tangent of *y*/*x*
cos(*x*)	cosine
cot(*x*)	cotangent
sin(*x*)	sine
tan(*x*)	tangent

All trigonometric functions have arguments and return values of type double precision.

4.4. String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of all the types CHARACTER, CHARACTER VARYING, and TEXT. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of the automatic padding when using the CHARACTER type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first. Some functions also exist natively for bit-string types.

SQL defines some string functions with a special syntax where certain keywords rather than commas are used to separate the arguments. Details are in Table 4-6. These functions are also implemented using the regular syntax for function invocation. (See Table 4-7.)

Table 4-6. SQL String Functions and Operators

Function	Return Type	Description	Example	Result
string || *string*	text	string concatenation	'Postgre' || 'SQL'	PostgreSQL
bit_length(*string*)	integer	number of bits in string	bit_length('jose')	32

Function	Return Type	Description	Example	Result
char_length(*string*) or character_length(*string*)	integer	number of characters in string	char_length('jose')	4
lower(*string*)	text	Convert string to lower case.	lower('TOM')	tom
octet_length(*string*)	integer	number of bytes in string	octet_length('jose')	4
position(*substring* in *string*)	integer	location of specified substring	position('om' in 'Thomas')	3
substring(*string* [from integer] [for integer])	text	extract substring	substring('Thomas' from 2 for 3)	hom
trim([leading | trailing | both] [*characters*] from *string*)	text	Removes the longest string containing only the *characters* (a space by default) from the beginning/end/both ends of the *string*.	trim(both 'x' from 'xTomxx')	Tom
upper(*string*)	text	Convert string to upper case.	upper('tom')	TOM

Additional string manipulation functions are available and are listed below. Some of them are used internally to implement the SQL-standard string functions listed above.

Table 4-7. Other String Functions

Function	Return Type	Description	Example	Result
ascii(text)	integer	Returns the ASCII code of the first character of the argument.	ascii('x')	120
btrim(*string* text, *trim* text)	text	Remove (trim) the longest string consisting only of characters in *trim* from the start and end of *string*.	btrim('xyxtrimyyx', 'xy')	trim
chr(integer)	text	Returns the character with the given ASCII code.	chr(65)	A

Function	Return Type	Description	Example	Result
convert(*string* text, [*src_encoding* name,] *dest_encoding* name)	text	Converts string using *dest_encoding*. The original encoding is specified by *src_encoding*. If *src_encoding* is omitted, database encoding is assumed.	convert('text_in_unicode', 'UNICODE', 'LATIN1')	text_in_unicode represented in ISO 8859-1
initcap(text)	text	Converts first letter of each word (whitespace separated) to upper case.	initcap('hi thomas')	Hi Thomas
length(*string*)	integer	length of string	length('jose')	4
lpad(*string* text, *length* integer [, *fill* text])	text	Fills up the *string* to length *length* by prepending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated (on the right).	lpad('hi', 5, 'xy')	xyxhi
ltrim(*string* text, *trim* text)	text	Removes the longest string containing only characters from *trim* from the start of the string.	ltrim('zzzytrim', 'xyz')	trim
pg_client_encoding()	name	Returns current client encoding name.	pg_client_encoding()	SQL_ASCII
repeat(text, integer)	text	Repeat text a number of times.	repeat('Pg', 4)	PgPgPgPg

Function	Return Type	Description	Example	Result
rpad(*string* text, *length* integer [, *fill* text])	text	Fills up the *string* to length *length* by appending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated.	rpad('hi', 5, 'xy')	hixyx
rtrim(*string* text, *trim* text)	text	Removes the longest string containing only characters from *trim* from the end of the string.	rtrim('trimxxxx', 'x')	trim
strpos(*string*, *substring*)	text	Locates specified substring. (same as position(*substring* in *string*), but note the reversed argument order)	strpos('high', 'ig')	2
substr(*string*, *from* [, *count*])	text	Extracts specified substring. (same as substring(*string* from *from* for *count*))	substr('alphabet', 3, 2)	ph
to_ascii(text [, *encoding*])	text	Converts text from multibyte encoding to ASCII.	to_ascii('Karel')	Karel
translate(*string* text, *from* text, *to* text)	text	Any character in *string* that matches a character in the *from* set is replaced by the corresponding character in the *to* set.	translate('12345', '14', 'ax')	a23x5
encode(*data* bytea, *type* text)	text	Encodes binary data to ASCII-only representation. Supported types are: 'base64', 'hex', 'escape'.	encode('123\\000\001', 'base64')	MTIzAAE=

Function	Return Type	Description	Example	Result
decode(*string* text, *type* text)	bytea	Decodes binary data from *string* previously encoded with encode(). Parameter type is same as in encode().	decode('MTIzAAE=', 'base64')	123\000\001

The to_ascii function supports conversion from LATIN1, LATIN2, WIN1250 (CP1250) only.

4.5. Binary String Functions and Operators

This section describes functions and operators for examining and manipulating binary string values. Strings in this context include values of the type BYTEA.

SQL defines some string functions with a special syntax where certain keywords rather than commas are used to separate the arguments. Details are in Table 4-8. Some functions are also implemented using the regular syntax for function invocation. (See Table 4-9.)

Table 4-8. SQL Binary String Functions and Operators

Function	Return Type	Description	Example	Result
string || *string*	bytea	string concatenation	'\\\\Postgre'::bytea || '\\047SQL\\000'::bytea	\\Postgre'SQL\000
octet_length(*string*)	integer	number of bytes in binary string	octet_length('jo\\000se'::bytea)	5
position(*substring* in *string*)	integer	location of specified substring	position('\\000m'::bytea in 'Th\\000omas'::bytea)	3
substring(*string* [from integer] [for integer])	bytea	extract substring	substring('Th\\000omas'::bytea from 2 for 3)	h\000o
trim([both] *characters* from *string*)	bytea	Removes the longest string containing only the *characters* from the beginning/end/both ends of the *string*.	trim('\\000'::bytea from '\\000Tom\\000'::bytea)	Tom

Additional binary string manipulation functions are available and are listed below. Some of them are used internally to implement the SQL-standard string functions listed above.

Table 4-9. Other Binary String Functions

Function	Return Type	Description	Example	Result
btrim(*string* bytea, *trim* bytea)	bytea	Remove (trim) the longest string consisting only of characters in *trim* from the start and end of *string*.	btrim('\\000trim\\000'::bytea,'\\000'::bytea)	trim
length(*string*)	integer	length of binary string	length('jo\\005e'::bytea)	5
encode(*string* bytea, *type* text)	text	Encodes binary string to ASCII-only representation. Supported types are: 'base64', 'hex', 'escape'.	encode('123\\000\\001456'::bytea, 'escape')	123\000\001456
decode(*string* text, *type* text)	bytea	Decodes binary string from *string* previously encoded with encode(). Parameter type is same as in encode().	decode('123\\000\\001456', 'escape')	123\000\001456

4.6. Pattern Matching

There are two separate approaches to pattern matching provided by PostgreSQL: the SQL LIKE operator and POSIX-style regular expressions.

> **Tip:** If you have pattern matching needs that go beyond this, or want to make pattern-driven substitutions or translations, consider writing a user-defined function in Perl or Tcl.

4.6.1. Pattern Matching with LIKE

```
string LIKE pattern [ ESCAPE escape-character ]
string NOT LIKE pattern [ ESCAPE escape-character ]
```

Every *pattern* defines a set of strings. The LIKE expression returns true if the *string* is contained in the set of strings represented by *pattern*. (As expected, the NOT LIKE expression returns false if LIKE returns true, and vice versa. An equivalent expression is NOT (*string* LIKE *pattern*).)

If *pattern* does not contain percent signs or underscore, then the pattern only represents the string itself; in that case LIKE acts like the equals operator. An underscore (_) in *pattern* stands for (matches) any single character; a percent sign (%) matches any string of zero or more characters.

Some examples:

```
'abc' LIKE 'abc'    true
```

```
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

LIKE pattern matches always cover the entire string. To match a pattern anywhere within a string, the pattern must therefore start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in *pattern* must be preceded by the escape character. The default escape character is the backslash but a different one may be selected by using the ESCAPE clause. To match the escape character itself, write two escape characters.

Note that the backslash already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in the query. Thus, writing a pattern that actually matches a literal backslash means writing four backslashes in the query. You can avoid this by selecting a different escape character with ESCAPE; then backslash is not special to LIKE anymore. (But it is still special to the string literal parser, so you still need two of them.)

It's also possible to select no escape character by writing ESCAPE ''. In this case there is no way to turn off the special meaning of underscore and percent signs in the pattern.

The keyword ILIKE can be used instead of LIKE to make the match case insensitive according to the active locale. This is not in the SQL standard but is a PostgreSQL extension.

The operator ~~ is equivalent to LIKE, and ~~* corresponds to ILIKE. There are also !~~ and !~~* operators that represent NOT LIKE and NOT ILIKE. All of these operators are PostgreSQL-specific.

4.6.2. POSIX Regular Expressions

Table 4-10. Regular Expression Match Operators

Operator	Description	Example
~	Matches regular expression, case sensitive	'thomas' ~ '.*thomas.*'
~*	Matches regular expression, case insensitive	'thomas' ~* '.*Thomas.*'
!~	Does not match regular expression, case sensitive	'thomas' !~ '.*Thomas.*'
!~*	Does not match regular expression, case insensitive	'thomas' !~* '.*vadim.*'

POSIX regular expressions provide a more powerful means for pattern matching than the LIKE function. Many Unix tools such as **egrep**, **sed**, or **awk** use a pattern matching language that is similar to the one described here.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a *regular set*). A string is said to match a regular expression if it is a member of the regular set described by the regular expression. As with LIKE, pattern characters match string characters exactly unless they are special characters in the regular expression language --- but regular expressions use different special characters than LIKE does. Unlike LIKE patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.

Regular expressions ("RE"s), as defined in POSIX 1003.2, come in two forms: modern REs (roughly those of **egrep**; 1003.2 calls these "extended" REs) and obsolete REs (roughly those of **ed**; 1003.2 "basic" REs). PostgreSQL implements the modern form.

A (modern) RE is one or more non-empty *branches*, separated by `|`. It matches anything that matches one of the branches.

A branch is one or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by a single `*`, `+`, `?`, or *bound*. An atom followed by `*` matches a sequence of 0 or more matches of the atom. An atom followed by `+` matches a sequence of 1 or more matches of the atom. An atom followed by `?` matches a sequence of 0 or 1 matches of the atom.

A *bound* is `{` followed by an unsigned decimal integer, possibly followed by `,` possibly followed by another unsigned decimal integer, always followed by `}`. The integers must lie between 0 and `RE_DUP_MAX` (255) inclusive, and if there are two of them, the first may not exceed the second. An atom followed by a bound containing one integer *`i`* and no comma matches a sequence of exactly *`i`* matches of the atom. An atom followed by a bound containing one integer *`i`* and a comma matches a sequence of *`i`* or more matches of the atom. An atom followed by a bound containing two integers *`i`* and *`j`* matches a sequence of *`i`* through *`j`* (inclusive) matches of the atom.

> **Note:** A repetition operator (`?`, `*`, `+`, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow `^` or `|`.

An *atom* is a regular expression enclosed in `()` (matching a match for the regular expression), an empty set of `()` (matching the null string), a *bracket expression* (see below), `.` (matching any single character), `^` (matching the null string at the beginning of the input string), `$` (matching the null string at the end of the input string), a `\` followed by one of the characters `^.[$()|*+?{\` (matching that character taken as an ordinary character), a `\` followed by any other character (matching that character taken as an ordinary character, as if the `\` had not been present), or a single character with no other significance (matching that character). A `{` followed by a character other than a digit is an ordinary character, not the beginning of a bound. It is illegal to end an RE with `\`.

Note that the backslash (`\`) already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in the query.

A *bracket expression* is a list of characters enclosed in `[]`. It normally matches any single character from the list (but see below). If the list begins with `^`, it matches any single character (but see below) not from the rest of the list. If two characters in the list are separated by `-`, this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g. `[0-9]` in ASCII matches any decimal digit. It is illegal for two ranges to share an endpoint, e.g. `a-c-e`. Ranges are very collating-sequence-dependent, and portable programs should avoid relying on them.

To include a literal `]` in the list, make it the first character (following a possible `^`). To include a literal `-`, make it the first or last character, or the second endpoint of a range. To use a literal `-` as the first endpoint of a range, enclose it in `[.` and `.]` to make it a collating element (see below). With the exception of these and some combinations using `[` (see next paragraphs), all other special characters, including `\`, lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multiple-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in `[.` and `.]` stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression's list. A bracket expression containing a multiple-character collating element

can thus match more than one character, e.g. if the collating sequence includes a `ch` collating element, then the RE `[[.ch.]]*c` matches the first five characters of `chchcc`.

Within a bracket expression, a collating element enclosed in `[=` and `=]` is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were `[.` and `.]`.) For example, if `o` and `^` are the members of an equivalence class, then `[[=o=]]`, `[[=^=]]`, and `[o^]` are all synonymous. An equivalence class may not be an endpoint of a range.

Within a bracket expression, the name of a character class enclosed in `[:` and `:]` stands for the list of all characters belonging to that class. Standard character class names are: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`. These stand for the character classes defined in ctype. A locale may provide others. A character class may not be used as an endpoint of a range.

There are two special cases of bracket expressions: the bracket expressions `[[:<:]]` and `[[:>:]]` match the null string at the beginning and end of a word respectively. A word is defined as a sequence of word characters which is neither preceded nor followed by word characters. A word character is an alnum character (as defined by ctype) or an underscore. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint that the whole match be as long as possible, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions.

Match lengths are measured in characters, not collating elements. A null string is considered longer than no match at all. For example, `bb*` matches the three middle characters of `abbbc`, `(wee|week)(knights|nights)` matches all ten characters of `weeknights`, when `(.*).*` is matched against `abc` the parenthesized subexpression matches all three characters, and when `(a*)*` is matched against `bc` both the whole RE and the parenthesized subexpression match the null string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g. `x` becomes `[xX]`. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that (e.g.) `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. Bounded repetitions are implemented by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like, say, `((((a{1,100}){1,100}){1,100}){1,100}){1,100}` will (eventually) run almost any existing machine out of swap space. [1]

4.7. Data Type Formatting Functions

Author: Written by Karel Zak (<zakkr@zf.jcu.cz>) on 2000-01-24

1. This was written in 1994, mind you. The numbers have probably changed, but the problem persists.

The PostgreSQL formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a template that defines the output or input format.

Table 4-11. Formatting Functions

Function	Returns	Description	Example
`to_char(timestamp, text)`	`text`	convert time stamp to string	`to_char(timestamp 'now','HH12:MI:SS')`
`to_char(interval, text)`	`text`	convert interval to string	`to_char(interval '15h 2m 12s','HH24:MI:SS')`
`to_char(int, text)`	`text`	convert int4/int8 to string	`to_char(125, '999')`
`to_char(double precision, text)`	`text`	convert real/double precision to string	`to_char(125.8, '999D9')`
`to_char(numeric, text)`	`text`	convert numeric to string	`to_char(numeric '-125.8', '999D99S')`
`to_date(text, text)`	`date`	convert string to date	`to_date('05 Dec 2000', 'DD Mon YYYY')`
`to_timestamp(text, text)`	`timestamp`	convert string to time stamp	`to_timestamp('05 Dec 2000', 'DD Mon YYYY')`
`to_number(text, text)`	`numeric`	convert string to numeric	`to_number('12,454.8-', '99G999D9S')`

In an output template string, there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string, template patterns identify the parts of the input data string to be looked at and the values to be found there.

Table 4-12. Template patterns for date/time conversions

Pattern	Description
`HH`	hour of day (01-12)
`HH12`	hour of day (01-12)
`HH24`	hour of day (00-23)
`MI`	minute (00-59)
`SS`	second (00-59)
`MS`	millisecond (000-999)
`US`	microsecond (000000-999999)

Pattern	Description
SSSS	seconds past midnight (0-86399)
AM or A.M. or PM or P.M.	meridian indicator (upper case)
am or a.m. or pm or p.m.	meridian indicator (lower case)
Y,YYY	year (4 and more digits) with comma
YYYY	year (4 and more digits)
YYY	last 3 digits of year
YY	last 2 digits of year
Y	last digit of year
BC or B.C. or AD or A.D.	era indicator (upper case)
bc or b.c. or ad or a.d.	era indicator (lower case)
MONTH	full upper case month name (blank-padded to 9 chars)
Month	full mixed case month name (blank-padded to 9 chars)
month	full lower case month name (blank-padded to 9 chars)
MON	abbreviated upper case month name (3 chars)
Mon	abbreviated mixed case month name (3 chars)
mon	abbreviated lower case month name (3 chars)
MM	month number (01-12)
DAY	full upper case day name (blank-padded to 9 chars)
Day	full mixed case day name (blank-padded to 9 chars)
day	full lower case day name (blank-padded to 9 chars)
DY	abbreviated upper case day name (3 chars)
Dy	abbreviated mixed case day name (3 chars)
dy	abbreviated lower case day name (3 chars)
DDD	day of year (001-366)
DD	day of month (01-31)
D	day of week (1-7; SUN=1)
W	week of month (1-5) where first week start on the first day of the month
WW	week number of year (1-53) where first week start on the first day of the year
IW	ISO week number of year (The first Thursday of the new year is in week 1.)
CC	century (2 digits)
J	Julian Day (days since January 1, 4712 BC)
Q	quarter
RM	month in Roman Numerals (I-XII; I=January) - upper case

Pattern	Description
rm	month in Roman Numerals (I-XII; I=January) - lower case
TZ	timezone name - upper case
tz	timezone name - lower case

Certain modifiers may be applied to any template pattern to alter its behavior. For example, "FMMonth" is the "Month" pattern with the "FM" prefix.

Table 4-13. Template pattern modifiers for date/time conversions

Modifier	Description	Example
FM prefix	fill mode (suppress padding blanks and zeroes)	FMMonth
TH suffix	add upper-case ordinal number suffix	DDTH
th suffix	add lower-case ordinal number suffix	DDth
FX prefix	Fixed format global option (see below)	FX Month DD Day
SP suffix	spell mode (not yet implemented)	DDSP

Usage notes:

- FM suppresses leading zeroes or trailing blanks that would otherwise be added to make the output of a pattern be fixed-width.
- to_timestamp and to_date skip multiple blank spaces in the input string if the FX option is not used. FX must be specified as the first item in the template; for example to_timestamp('2000 JUN','YYYY MON') is right, but to_timestamp('2000 JUN','FXYYYY MON') returns an error, because to_timestamp expects one blank space only.
- If a backslash ("\") is desired in a string constant, a double backslash ("\\") must be entered; for example '\\HH\\MI\\SS'. This is true for any string constant in PostgreSQL.
- Ordinary text is allowed in to_char templates and will be output literally. You can put a substring in double quotes to force it to be interpreted as literal text even if it contains pattern keywords. For example, in '"Hello Year: "YYYY', the YYYY will be replaced by year data, but the single Y will not be.
- If you want to have a double quote in the output you must precede it with a backslash, for example '\\"YYYY Month\\"'.
- YYYY conversion from string to timestamp or date is restricted if you use a year with more than 4 digits. You must use some non-digit character or template after YYYY, otherwise the year is always interpreted as 4 digits. For example (with year 20000): to_date('200001131', 'YYYYMMDD') will be interpreted as a 4-digit year; better is to use a non-digit separator after the year, like to_date('20000-1131', 'YYYY-MMDD') or to_date('20000Nov31', 'YYYYMonDD').
- Millisecond MS and microsecond US values in a conversion from string to time stamp are used as part of the seconds after the decimal point. For example to_timestamp('12:3', 'SS:MS') is not 3 milliseconds, but 300, because the conversion counts it as 12 + 0.3. This means for the format

`SS:MS`, the input values `12:3`, `12:30`, and `12:300` specify the same number of milliseconds. To get three milliseconds, one must use `12:003`, which the conversion counts as 12 + 0.003 = 12.003 seconds.

Here is a more complex example: `to_timestamp('15:12:02.020.001230','HH:MI:SS.MS.US')` is 15 hours, 12 minutes, and 2 seconds + 20 milliseconds + 1230 microseconds = 2.021230 seconds.

Table 4-14. Template patterns for numeric conversions

Pattern	Description
`9`	value with the specified number of digits
`0`	value with leading zeros
`.` (period)	decimal point
`,` (comma)	group (thousand) separator
`PR`	negative value in angle brackets
`S`	negative value with minus sign (uses locale)
`L`	currency symbol (uses locale)
`D`	decimal point (uses locale)
`G`	group separator (uses locale)
`MI`	minus sign in specified position (if number < 0)
`PL`	plus sign in specified position (if number > 0)
`SG`	plus/minus sign in specified position
`RN`	roman numeral (input between 1 and 3999)
`TH` or `th`	convert to ordinal number
`V`	shift *n* digits (see notes)
`EEEE`	scientific notation (not implemented yet)

Usage notes:

- A sign formatted using `SG`, `PL`, or `MI` is not an anchor in the number; for example, `to_char(-12, 'S9999')` produces `' -12'`, but `to_char(-12, 'MI9999')` produces `'- 12'`. The Oracle implementation does not allow the use of `MI` ahead of `9`, but rather requires that `9` precede `MI`.
- `9` specifies a value with the same number of digits as there are `9`s. If a digit is not available use blank space.
- `TH` does not convert values less than zero and does not convert decimal numbers.
- `PL`, `SG`, and `TH` are PostgreSQL extensions.
- `V` effectively multiplies the input values by `10^n`, where *n* is the number of digits following `V`. `to_char` does not support the use of `V` combined with a decimal point. (E.g., `99.9V99` is not allowed.)

Table 4-15. `to_char` Examples

Input	Output
`to_char(now(),'Day, DD HH12:MI:SS')`	`'Tuesday , 06 05:39:18'`
`to_char(now(),'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,'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,'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'`

4.8. Date/Time Functions and Operators

Table 4-17 shows the available functions for date/time value processing. Table 4-16 illustrates the behaviors of the basic arithmetic operators (+, *, etc.). For formatting functions, refer to Section 4.7. You should be familiar with the background information on date/time data types (see Section 3.5).

The date/time operators described below behave similarly for types involving time zones as well as those without.

Table 4-16. Date/Time Operators

Name	Example	Result
+	timestamp ’2001-09-28 01:00’ + interval ’23 hours’	timestamp ’2001-09-29 00:00’
+	date ’2001-09-28’ + interval ’1 hour’	timestamp ’2001-09-28 01:00’
+	time ’01:00’ + interval ’3 hours’	time ’04:00’
-	timestamp ’2001-09-28 23:00’ - interval ’23 hours’	timestamp ’2001-09-28’
-	date ’2001-09-28’ - interval ’1 hour’	timestamp ’2001-09-27 23:00’
-	time ’05:00’ - interval ’2 hours’	time ’03:00’
-	interval ’2 hours’ - time ’05:00’	time ’03:00:00’
*	interval ’1 hour’ * int ’3’	interval ’03:00’
/	interval ’1 hour’ / int ’3’	interval ’00:20’

The date/time functions are summarized below, with additional details in subsequent sections.

Table 4-17. Date/Time Functions

Name	Return Type	Description	Example	Result
age(timestamp)	interval	Subtract from today	age(timestamp ’1957-06-13’)	43 years 8 mons 3 days
age(timestamp, timestamp)	interval	Subtract arguments	age(’2001-04-10’, timestamp ’1957-06-13’)	43 years 9 mons 27 days
current_date	date	Today’s date; see below		
current_time	time	Time of day; see below		
current_timestamp	timestamp	Date and time; see below		
date_part(text, timestamp)	double precision	Get subfield (equivalent to extract); see also below	date_part(’hour’, timestamp ’2001-02-16 20:38:40’)	20
date_part(text, interval)	double precision	Get subfield (equivalent to extract); see also below	date_part(’month’, interval ’2 years 3 months’)	3

Name	Return Type	Description	Example	Result
date_trunc(text, timestamp)	timestamp	Truncate to specified precision; see also below	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00+00
extract(*field* from timestamp)	double precision	Get subfield; see also below	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(*field* from interval)	double precision	Get subfield; see also below	extract(month from interval '2 years 3 months')	3
isfinite(timestamp)	boolean	Test for finite time stamp (neither invalid nor infinity)	isfinite(timestamp '2001-02-16 21:28:30')	true
isfinite(interval)	boolean	Test for finite interval	isfinite(interval '4 hours')	true
now()	timestamp	Current date and time (equivalent to current_timestamp); see below		
timeofday()	text	Current date and time; see below	timeofday()	Wed Feb 21 17:01:13.000126 2001 EST
timestamp(date)	timestamp	date to timestamp	timestamp(date '2000-12-25')	2000-12-25 00:00:00
timestamp(date, time)	timestamp	date and time to timestamp	timestamp(date '1998-02-24',time '23:07')	1998-02-24 23:07:00

4.8.1. EXTRACT, date_part

EXTRACT (*field* FROM *source*)

The extract function retrieves sub-fields from date/time values, such as year or hour. *source* is a value expression that evaluates to type timestamp or interval. (Expressions of type date or time will be cast to timestamp and can therefore be used as well.) *field* is an identifier or string that selects what field to extract from the source value. The extract function returns values of type double precision. The following are valid values:

century

The year field divided by 100

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

Note that the result for the century field is simply the year field divided by 100, and not the conventional definition which puts most years in the 1900's in the twentieth century.

`day`

The day (of the month) field (1 - 31)

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16
```

`decade`

The year field divided by 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 200
```

`dow`

The day of the week (0 - 6; Sunday is 0) (for `timestamp` values only)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5
```

`doy`

The day of the year (1 - 365/366) (for `timestamp` values only)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47
```

`epoch`

For `date` and `timestamp` values, the number of seconds since 1970-01-01 00:00:00-00 (Result may be negative.); for `interval` values, the total number of seconds in the interval

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 982352320

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800
```

`hour`

The hour field (0 - 23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

`microseconds`

The seconds field, including fractional parts, multiplied by 1 000 000. Note that this includes full seconds.

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Result: 28500000
```

`millennium`

The year field divided by 1000

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2
```

Note that the result for the millennium field is simply the year field divided by 1000, and not the conventional definition which puts years in the 1900's in the second millennium.

`milliseconds`

The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Result: 28500
```

`minute`

The minutes field (0 - 59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

`month`

For `timestamp` values, the number of the month within the year (1 - 12) ; for `interval` values the number of months, modulo 12 (0 - 11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Result: 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Result: 1
```

`quarter`

The quarter of the year (1 - 4) that the day is in (for `timestamp` values only)

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
```

`second`

The seconds field, including fractional parts (0 - 59[2])

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.5
```

60 if leap seconds are implemented by the operating system

`timezone_hour`

The hour component of the time zone offset.

`timezone_minute`

The minute component of the time zone offset.

`week`

From a `timestamp` value, calculate the number of the week of the year that the day is in. By definition (ISO 8601), the first week of a year contains January 4 of that year. (The ISO week starts on Monday.) In other words, the first Thursday of a year is in week 1 of that year.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
```

`year`

The year field

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2001
```

The `extract` function is primarily intended for computational processing. For formatting date/time values for display, see Section 4.7.

The `date_part` function is modeled on the traditional Ingres equivalent to the SQL-function `extract`:

```
date_part('field', source)
```

Note that here the *`field`* value needs to be a string. The valid field values for `date_part` are the same as for `extract`.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4
```

4.8.2. `date_trunc`

The function `date_trunc` is conceptually similar to the `trunc` function for numbers.

```
date_trunc('field', source)
```

`source` is a value expression of type `timestamp` (values of type `date` and `time` are cast automatically). *`field`* selects to which precision to truncate the time stamp value. The return value is of type `timestamp` with all fields that are less than the selected one set to zero (or one, for day and month).

Valid values for *`field`* are:

microseconds
milliseconds
second

minute
hour
day
month
year
decade
century
millennium

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00+00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00+00
```

4.8.3. Current Date/Time

The following functions are available to obtain the current date and/or time:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME ( precision )
CURRENT_TIMESTAMP ( precision )
```

`CURRENT_TIME` and `CURRENT_TIMESTAMP` can optionally be given a precision parameter, which causes the result to be rounded to that many fractional digits. Without a precision parameter, the result is given to full available precision.

> **Note:** Prior to PostgreSQL 7.2, the precision parameters were unimplemented, and the result was always given in integer seconds.

> **Note:** The SQL99 standard requires these functions to be written without any parentheses, unless a precision parameter is given. As of PostgreSQL 7.2, an empty pair of parentheses can be written, but this is deprecated and may be removed in a future release.

```
SELECT CURRENT_TIME;
14:39:53.662522-05

SELECT CURRENT_DATE;
2001-12-23

SELECT CURRENT_TIMESTAMP;
2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
```

```
2001-12-23 14:39:53.66-05
```

The function `now()` is the traditional PostgreSQL equivalent to `CURRENT_TIMESTAMP`.

There is also `timeofday()`, which for historical reasons returns a text string rather than a `timestamp` value:

```
SELECT timeofday();
 Sat Feb 17 19:07:32.000126 2001 EST
```

It is quite important to realize that `CURRENT_TIMESTAMP` and related functions all return the time as of the start of the current transaction; their values do not increment while a transaction is running. But `timeofday()` returns the actual current time.

All the date/time data types also accept the special literal value `now` to specify the current date and time. Thus, the following three all return the same result:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';
```

> **Note:** You do not want to use the third form when specifying a DEFAULT value while creating a table. The system will convert `now` to a `timestamp` as soon as the constant is parsed, so that when the default value is needed, the time of the table creation would be used! The first two forms will not be evaluated until the default value is used, because they are function calls. Thus they will give the desired behavior of defaulting to the time of row insertion.

4.9. Geometric Functions and Operators

The geometric types `point`, `box`, `lseg`, `line`, `path`, `polygon`, and `circle` have a large set of native support functions and operators.

Table 4-18. Geometric Operators

Operator	Description	Usage
+	Translation	`box '((0,0),(1,1))' + point '(2.0,0)'`
-	Translation	`box '((0,0),(1,1))' - point '(2.0,0)'`
*	Scaling/rotation	`box '((0,0),(1,1))' * point '(2.0,0)'`
/	Scaling/rotation	`box '((0,0),(2,2))' / point '(2.0,0)'`
#	Intersection	`'((1,-1),(-1,1))' # '((1,1),(-1,-1))'`
#	Number of points in polygon	`# '((1,0),(0,1),(-1,0))'`

Operator	Description	Usage
##	Point of closest proximity	`point '(0,0)' ## lseg '((2,0),(0,2))'`
&&	Overlaps?	`box '((0,0),(1,1))' && box '((0,0),(2,2))'`
&<	Overlaps to left?	`box '((0,0),(1,1))' &< box '((0,0),(2,2))'`
&>	Overlaps to right?	`box '((0,0),(3,3))' &> box '((0,0),(2,2))'`
<->	Distance between	`circle '((0,0),1)' <-> circle '((5,0),1)'`
<<	Left of?	`circle '((0,0),1)' << circle '((5,0),1)'`
<^	Is below?	`circle '((0,0),1)' <^ circle '((0,5),1)'`
>>	Is right of?	`circle '((5,0),1)' >> circle '((0,0),1)'`
>^	Is above?	`circle '((0,5),1)' >^ circle '((0,0),1)'`
?#	Intersects or overlaps	`lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'`
?-	Is horizontal?	`point '(1,0)' ?- point '(0,0)'`
?-|	Is perpendicular?	`lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'`
@-@	Length or circumference	`@-@ path '((0,0),(1,0))'`
?|	Is vertical?	`point '(0,1)' ?| point '(0,0)'`
?||	Is parallel?	`lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'`
@	Contained or on	`point '(1,1)' @ circle '((0,0),2)'`
@@	Center of	`@@ circle '((0,0),10)'`
~=	Same as	`polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'`

Table 4-19. Geometric Functions

Function	Returns	Description	Example
`area`(object)	`double precision`	area of item	`area(box '((0,0),(1,1))')`

Function	Returns	Description	Example
box(box, box)	box	intersection box	box(box '((0,0),(1,1))',box '((0.5,0.5),(2,2))')
center(object)	point	center of item	center(box '((0,0),(1,2))')
diameter(circle)	double precision	diameter of circle	diameter(circle '((0,0),2.0)')
height(box)	double precision	vertical size of box	height(box '((0,0),(1,1))')
isclosed(path)	boolean	a closed path?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	an open path?	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	double precision	length of item	length(path '((-1,0),(1,0))')
pclose(path)	path	convert path to closed	popen(path '[(0,0),(1,1),(2,0)]')
npoint(path)	integer	number of points	npoints(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	convert path to open path	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	radius of circle	radius(circle '((0,0),2.0)')
width(box)	double precision	horizontal size	width(box '((0,0),(1,1))')

Table 4-20. Geometric Type Conversion Functions

Function	Returns	Description	Example
box(circle)	box	circle to box	box(circle '((0,0),2.0)')
box(point, point)	box	points to box	box(point '(0,0)', point '(1,1)')
box(polygon)	box	polygon to box	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	to circle	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	point to circle	circle(point '(0,0)', 2.0)

Function	Returns	Description	Example
`lseg(box)`	`lseg`	box diagonal to lseg	`lseg(box '((-1,0),(1,0))')`
`lseg(point, point)`	`lseg`	points to lseg	`lseg(point '(-1,0)', point '(1,0)')`
`path(polygon)`	`point`	polygon to path	`path(polygon '((0,0),(1,1),(2,0))')`
`point(circle)`	`point`	center	`point(circle '((0,0),2.0)')`
`point(lseg, lseg)`	`point`	intersection	`point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')`
`point(polygon)`	`point`	center	`point(polygon '((0,0),(1,1),(2,0))')`
`polygon(box)`	`polygon`	12 point polygon	`polygon(box '((0,0),(1,1))')`
`polygon(circle)`	`polygon`	12-point polygon	`polygon(circle '((0,0),2.0)')`
`polygon(`*`npts`*`, circle)`	`polygon`	*`npts`* polygon	`polygon(12, circle '((0,0),2.0)')`
`polygon(path)`	`polygon`	path to polygon	`polygon(path '((0,0),(1,1),(2,0))')`

4.10. Network Address Type Functions

Table 4-21. `cidr` and `inet` Operators

Operator	Description	Usage
<	Less than	`inet '192.168.1.5' < inet '192.168.1.6'`
<=	Less than or equal	`inet '192.168.1.5' <= inet '192.168.1.5'`
=	Equals	`inet '192.168.1.5' = inet '192.168.1.5'`
>=	Greater or equal	`inet '192.168.1.5' >= inet '192.168.1.5'`
>	Greater	`inet '192.168.1.5' > inet '192.168.1.4'`
<>	Not equal	`inet '192.168.1.5' <> inet '192.168.1.4'`

Operator	Description	Usage
<<	is contained within	inet '192.168.1.5' << inet '192.168.1/24'
<<=	is contained within or equals	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	contains	inet'192.168.1/24' >> inet '192.168.1.5'
>>=	contains or equals	inet '192.168.1/24' >>= inet '192.168.1/24'

All of the operators for inet can be applied to cidr values as well. The operators <<, <<=, >>, >>= test for subnet inclusion: they consider only the network parts of the two addresses, ignoring any host part, and determine whether one network part is identical to or a subnet of the other.

Table 4-22. cidr and inet Functions

Function	Returns	Description	Example	Result
broadcast(inet)	inet	broadcast address for network	broadcast('192.168.1.5/24')	192.168.1.255/24
host(inet)	text	extract IP address as text	host('192.168.1.5/24')	192.168.1.5
masklen(inet)	integer	extract netmask length	masklen('192.168.1.5/24')	24
set_masklen(inet, integer)	inet	set netmask length for inet value	set_masklen('192.168.1.5/24', 16)	192.168.1.5/16
netmask(inet)	inet	construct netmask for network	netmask('192.168.1.5/24')	255.255.255.0
network(inet)	cidr	extract network part of address	network('192.168.1.5/24')	192.168.1.0/24
text(inet)	text	extract IP address and masklen as text	text(inet '192.168.1.5')	192.168.1.5/32
abbrev(inet)	text	extract abbreviated display as text	abbrev(cidr '10.1.0.0/16')	10.1/16

All of the functions for inet can be applied to cidr values as well. The host(), text(), and abbrev() functions are primarily intended to offer alternative display formats. You can cast a text field to inet using normal casting syntax: inet(expression) or colname::inet.

Table 4-23. macaddr Functions

Function	Returns	Description	Example	Result
trunc(macaddr)	macaddr	set last 3 bytes to zero	trunc(macaddr '12:34:56:78:90:ab')	12:34:56:00:00:00

The function trunc(macaddr) returns a MAC address with the last 3 bytes set to 0. This can be used to associate the remaining prefix with a manufacturer. The directory contrib/mac in the source distribution contains some utilities to create and maintain such an association table.

The macaddr type also supports the standard relational operators (>, <=, etc.) for lexicographical ordering.

4.11. Sequence-Manipulation Functions

Table 4-24. Sequence Functions

Function	Returns	Description
nextval(text)	bigint	Advance sequence and return new value
currval(text)	bigint	Return value most recently obtained with nextval
setval(text,bigint)	bigint	Set sequence's current value
setval(text,bigint,boolean)	bigint	Set sequence's current value and is_called flag

This section describes PostgreSQL's functions for operating on *sequence objects*. Sequence objects (also called sequence generators or just sequences) are special single-row tables created with **CREATE SEQUENCE**. A sequence object is usually used to generate unique identifiers for rows of a table. The sequence functions provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

For largely historical reasons, the sequence to be operated on by a sequence-function call is specified by a text-string argument. To achieve some compatibility with the handling of ordinary SQL names, the sequence functions convert their argument to lower case unless the string is double-quoted. Thus

```
nextval('foo')      operates on sequence foo
nextval('FOO')      operates on sequence foo
nextval('"Foo"')    operates on sequence Foo
```

Of course, the text argument can be the result of an expression, not only a simple literal, which is occasionally useful.

The available sequence functions are:

nextval

Advance the sequence object to its next value and return that value. This is done atomically: even if multiple server processes execute nextval concurrently, each will safely receive a distinct sequence value.

currval

Return the value most recently obtained by nextval for this sequence in the current server process. (An error is reported if nextval has never been called for this sequence in this process.) Notice that because this is returning a process-local value, it gives a predictable answer even if other server processes are executing nextval meanwhile.

setval

Reset the sequence object's counter value. The two-parameter form sets the sequence's last_value field to the specified value and sets its is_called field to true, meaning that the next nextval will advance the sequence before returning a value. In the three-parameter form, is_called may be set either true or false. If it's set to false, the next nextval will

return exactly the specified value, and sequence advancement commences with the following `nextval`. For example,

```
SELECT setval('foo', 42);           Next nextval() will return 43
SELECT setval('foo', 42, true);     Same as above
SELECT setval('foo', 42, false);    Next nextval() will return 42
```

The result returned by `setval` is just the value of its second argument.

Important: To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a `nextval` operation is never rolled back; that is, once a value has been fetched it is considered used, even if the transaction that did the `nextval` later aborts. This means that aborted transactions may leave unused "holes" in the sequence of assigned values. `setval` operations are never rolled back, either.

If a sequence object has been created with default parameters, `nextval()` calls on it will return successive values beginning with one. Other behaviors can be obtained by using special parameters in the **CREATE SEQUENCE** command; see its command reference page for more information.

4.12. Conditional Expressions

This section describes the SQL-compliant conditional expressions available in PostgreSQL.

Tip: If your needs go beyond the capabilities of these conditional expressions you might want to consider writing a stored procedure in a more expressive programming language.

CASE

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

The SQL CASE expression is a generic conditional expression, similar to if/else statements in other languages. CASE clauses can be used wherever an expression is valid. *`condition`* is an expression that returns a `boolean` result. If the result is true then the value of the CASE expression is *`result`*. If the result is false any subsequent WHEN clauses are searched in the same manner. If no WHEN *`condition`* is true then the value of the case expression is the *`result`* in the ELSE clause. If the ELSE clause is omitted and no condition matches, the result is NULL.

An example:

```
=> SELECT * FROM test;
 a
---
 1
 2
 3

=> SELECT a,
```

```
          CASE WHEN a=1 THEN 'one'
               WHEN a=2 THEN 'two'
               ELSE 'other'
          END
    FROM test;
 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

The data types of all the *result* expressions must be coercible to a single output type. See Section 5.6 for more detail.

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

This "simple" CASE expression is a specialized variant of the general form above. The *expression* is computed and compared to all the *value*s in the WHEN clauses until one is found that is equal. If no match is found, the *result* in the ELSE clause (or NULL) is returned. This is similar to the `switch` statement in C.

The example above can be written using the simple CASE syntax:

```
=> SELECT a,
          CASE a WHEN 1 THEN 'one'
                 WHEN 2 THEN 'two'
                 ELSE 'other'
          END
    FROM test;
 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

COALESCE

```
COALESCE(value[, ...])
```

The `COALESCE` function returns the first of its arguments that is not NULL. This is often useful to substitute a default value for NULL values when data is retrieved for display, for example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

NULLIF

```
NULLIF(value1, value2)
```

The NULLIF function returns NULL if and only if *value1* and *value2* are equal. Otherwise it returns *value1*. This can be used to perform the inverse operation of the COALESCE example given above:

```
SELECT NULLIF(value, '(none)') ...
```

> **Tip:** COALESCE and NULLIF are just shorthand for CASE expressions. They are actually converted into CASE expressions at a very early stage of processing, and subsequent processing thinks it is dealing with CASE. Thus an incorrect COALESCE or NULLIF usage may draw an error message that refers to CASE.

4.13. Miscellaneous Functions

Table 4-25. Session Information Functions

Name	Return Type	Description
current_user	name	user name of current execution context
session_user	name	session user name
user	name	equivalent to current_user

The session_user is the user that initiated a database connection; it is fixed for the duration of that connection. The current_user is the user identifier that is applicable for permission checking. Currently it is always equal to the session user, but in the future there might be "setuid" functions and other facilities to allow the current user to change temporarily. In Unix parlance, the session user is the "real user" and the current user is the "effective user".

Note that these functions have special syntactic status in SQL: they must be called without trailing parentheses.

> **Deprecated:** The function getpgusername() is an obsolete equivalent of current_user.

Table 4-26. System Information Functions

Name	Return Type	Description
version	text	PostgreSQL version information

version() returns a string describing the PostgreSQL server's version.

Table 4-27. Access Privilege Inquiry Functions

Name	Return Type	Description
has_table_privilege(*user*, *table*, *access*)	boolean	does user have access to table
has_table_privilege(*table*, *access*)	boolean	does current user have access to table

has_table_privilege determines whether a user can access a table in a particular way. The user can be specified by name or by ID (pg_user.usesysid), or if the argument is omitted current_user is assumed. The table can be specified by name or by OID. (Thus, there are actually six variants of has_table_privilege, which can be distinguished by the number and types of their arguments.) The desired access type is specified by a text string, which must evaluate to one of the values SELECT, INSERT, UPDATE, DELETE, RULE, REFERENCES, or TRIGGER. (Case of the string is not significant, however.)

Table 4-28. Catalog Information Functions

Name	Return Type	Description
pg_get_viewdef(*viewname*)	text	Get CREATE VIEW command for view
pg_get_ruledef(*rulename*)	text	Get CREATE RULE command for rule
pg_get_indexdef(*indexOID*)	text	Get CREATE INDEX command for index
pg_get_userbyid(*userid*)	name	Get user name given ID

These functions extract information from the system catalogs. pg_get_viewdef(), pg_get_ruledef(), and pg_get_indexdef() respectively reconstruct the creating command for a view, rule, or index. (Note that this is a decompiled reconstruction, not the verbatim text of the command.) pg_get_userbyid() extracts a user's name given a usesysid value.

Table 4-29. Comment Information Functions

Name	Return Type	Description
obj_description(*objectOID*, *tablename*)	text	Get comment for a database object
obj_description(*objectOID*)	text	Get comment for a database object (*deprecated*)
col_description(*tableOID*, *columnnumber*)	text	Get comment for a table column

These functions extract comments previously stored with the **COMMENT** command. NULL is returned if no comment can be found matching the specified parameters.

The two-parameter form of obj_description() returns the comment for a database object specified by its OID and the name of the containing system catalog. For example, obj_description(123456,'pg_class') would retrieve the comment for a table with OID 123456. The one-parameter form of obj_description() requires only the object OID. It is now deprecated since there is no guarantee that OIDs are unique across different system catalogs; therefore, the wrong comment could be returned.

col_description() returns the comment for a table column, which is specified by the OID of its

table and its column number. `obj_description()` cannot be used for table columns since columns do not have OIDs of their own.

4.14. Aggregate Functions

Author: Written by Isaac Wilcox <isaac@azartmedia.com> on 2000-06-16

Aggregate functions compute a single result value from a set of input values. The special syntax considerations for aggregate functions are explained in Section 1.3.5. Consult the *PostgreSQL Tutorial* for additional introductory information.

Table 4-30. Aggregate Functions

Function	Description	Notes
AVG(*expression*)	the average (arithmetic mean) of all input values	Finding the average value is available on the following data types: `smallint`, `integer`, `bigint`, `real`, `double precision`, `numeric`, `interval`. The result is of type `numeric` for any integer type input, `double precision` for floating-point input, otherwise the same as the input data type.
`count`(*)	number of input values	The return value is of type `bigint`.
`count`(*expression*)	Counts the input values for which the value of *expression* is not NULL.	The return value is of type `bigint`.
`max`(*expression*)	the maximum value of *expression* across all input values	Available for all numeric, string, and date/time types. The result has the same type as the input expression.
`min`(*expression*)	the minimum value of *expression* across all input values	Available for all numeric, string, and date/time types. The result has the same type as the input expression.
`stddev`(*expression*)	the sample standard deviation of the input values	Finding the standard deviation is available on the following data types: `smallint`, `integer`, `bigint`, `real`, `double precision`, `numeric`. The result is of type `double precision` for floating-point input, otherwise `numeric`.

Function	Description	Notes
sum(*expression*)	sum of *expression* across all input values	Summation is available on the following data types: smallint, integer, bigint, real, double precision, numeric, interval. The result is of type bigint for smallint or integer input, numeric for bigint input, double precision for floating-point input, otherwise the same as the input data type.
variance(*expression*)	the sample variance of the input values	The variance is the square of the standard deviation. The supported data types and result types are the same as for standard deviation.

It should be noted that except for COUNT, these functions return NULL when no rows are selected. In particular, SUM of no rows returns NULL, not zero as one might expect. COALESCE may be used to substitute zero for NULL when necessary.

4.15. Subquery Expressions

This section describes the SQL-compliant subquery expressions available in PostgreSQL. All of the expression forms documented in this section return Boolean (true/false) results.

EXISTS

```
EXISTS ( subquery )
```

The argument of EXISTS is an arbitrary SELECT statement, or *subquery*. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is TRUE; if the subquery returns no rows, the result of EXISTS is FALSE.

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed far enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has any side-effects (such as calling sequence functions); whether the side-effects occur or not may be difficult to predict.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally uninteresting. A common coding convention is to write all EXISTS tests in the form EXISTS(SELECT 1 WHERE ...). There are exceptions to this rule however, such as subqueries that use INTERSECT.

This simple example is like an inner join on col2, but it produces at most one output row for each tab1 row, even if there are multiple matching tab2 rows:

```
SELECT col1 FROM tab1
    WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

IN (scalar form)

```
expression IN (value[, ...])
```

The right-hand side of this form of IN is a parenthesized list of scalar expressions. The result is TRUE if the left-hand expression's result is equal to any of the right-hand expressions. This is a shorthand notation for

```
expression = value1
OR
expression = value2
OR
...
```

Note that if the left-hand expression yields NULL, or if there are no equal right-hand values and at least one right-hand expression yields NULL, the result of the IN construct will be NULL, not FALSE. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

> **Note:** This form of IN is not truly a subquery expression, but it seems best to document it in the same place as subquery IN.

IN (subquery form)

```
expression IN (subquery)
```

The right-hand side of this form of IN is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of IN is TRUE if any equal subquery row is found. The result is FALSE if no equal row is found (including the special case where the subquery returns no rows).

Note that if the left-hand expression yields NULL, or if there are no equal right-hand values and at least one right-hand row yields NULL, the result of the IN construct will be NULL, not FALSE. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
(expression, expression[, ...]) IN (subquery)
```

The right-hand side of this form of IN is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of IN is TRUE if any equal subquery row is found. The result is FALSE if no equal row is found (including the special case where the subquery returns no rows).

As usual, NULLs in the expressions or subquery rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (NULL). If all the row results are either unequal or NULL, with at least one NULL, then the result of IN is NULL.

NOT IN (scalar form)

```
expression NOT IN (value[, ...])
```

The right-hand side of this form of NOT IN is a parenthesized list of scalar expressions. The result is TRUE if the left-hand expression's result is unequal to all of the right-hand expressions. This is a shorthand notation for

```
expression <> value1
AND
expression <> value2
AND
...
```

Note that if the left-hand expression yields NULL, or if there are no equal right-hand values and at least one right-hand expression yields NULL, the result of the NOT IN construct will be NULL, not TRUE as one might naively expect. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

> **Tip:** `x NOT IN y` is equivalent to `NOT (x IN y)` in all cases. However, NULLs are much more likely to trip up the novice when working with NOT IN than when working with IN. It's best to express your condition positively if possible.

NOT IN (subquery form)

```
expression NOT IN (subquery)
```

The right-hand side of this form of NOT IN is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of NOT IN is TRUE if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is FALSE if any equal row is found.

Note that if the left-hand expression yields NULL, or if there are no equal right-hand values and at least one right-hand row yields NULL, the result of the NOT IN construct will be NULL, not TRUE. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
(expression, expression[, ...]) NOT IN (subquery)
```

The right-hand side of this form of NOT IN is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of NOT IN is TRUE if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is FALSE if any equal row is found.

As usual, NULLs in the expressions or subquery rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (NULL). If all the row results are either unequal or NULL, with at least one NULL, then the result of NOT IN is NULL.

ANY

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

The right-hand side of this form of ANY is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using

the given *operator*, which must yield a Boolean result. The result of ANY is TRUE if any true result is obtained. The result is FALSE if no true result is found (including the special case where the subquery returns no rows).

SOME is a synonym for ANY. IN is equivalent to = ANY.

Note that if there are no successes and at least one right-hand row yields NULL for the operator's result, the result of the ANY construct will be NULL, not FALSE. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
(expression, expression[, ...]) operator ANY (subquery)
(expression, expression[, ...]) operator SOME (subquery)
```

The right-hand side of this form of ANY is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. Presently, only = and <> operators are allowed in row-wise ANY queries. The result of ANY is TRUE if any equal or unequal row is found, respectively. The result is FALSE if no such row is found (including the special case where the subquery returns no rows).

As usual, NULLs in the expressions or subquery rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (NULL). If there is at least one NULL row result, then the result of ANY cannot be FALSE; it will be TRUE or NULL.

ALL

```
expression operator ALL (subquery)
```

The right-hand side of this form of ALL is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given *operator*, which must yield a Boolean result. The result of ALL is TRUE if all rows yield TRUE (including the special case where the subquery returns no rows). The result is FALSE if any false result is found.

NOT IN is equivalent to <> ALL.

Note that if there are no failures but at least one right-hand row yields NULL for the operator's result, the result of the ALL construct will be NULL, not TRUE. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
(expression, expression[, ...]) operator ALL (subquery)
```

The right-hand side of this form of ALL is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. Presently, only = and <> operators are allowed in row-wise ALL queries. The result of ALL is TRUE if all subquery rows are equal or unequal, respectively (including the special case where the subquery returns no rows). The result is FALSE if any row is found to be unequal or equal, respectively.

As usual, NULLs in the expressions or subquery rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise

the result of that row comparison is unknown (NULL). If there is at least one NULL row result, then the result of ALL cannot be TRUE; it will be FALSE or NULL.

Row-wise comparison

```
(expression, expression[, ...]) operator (subquery)
(expression, expression[, ...]) operator (expression, expression[, ...])
```

The left-hand side is a list of scalar expressions. The right-hand side can be either a list of scalar expressions of the same length, or a parenthesized subquery, which must return exactly as many columns as there are expressions on the left-hand side. Furthermore, the subquery cannot return more than one row. (If it returns zero rows, the result is taken to be NULL.) The left-hand side is evaluated and compared row-wise to the single subquery result row, or to the right-hand expression list. Presently, only = and <> operators are allowed in row-wise comparisons. The result is TRUE if the two rows are equal or unequal, respectively.

As usual, NULLs in the expressions or subquery rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of the row comparison is unknown (NULL).

Chapter 5. Type Conversion

5.1. Introduction

SQL queries can, intentionally or not, require mixing of different data types in the same expression. PostgreSQL has extensive facilities for evaluating mixed-type expressions.

In many cases a user will not need to understand the details of the type conversion mechanism. However, the implicit conversions done by PostgreSQL can affect the results of a query. When necessary, these results can be tailored by a user or programmer using *explicit* type coercion.

This chapter introduces the PostgreSQL type conversion mechanisms and conventions. Refer to the relevant sections in Chapter 3 and Chapter 4 for more information on specific data types and allowed functions and operators.

The *Programmer's Guide* has more details on the exact algorithms used for implicit type conversion and coercion.

5.2. Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. PostgreSQL has an extensible type system that is much more general and flexible than other RDBMS implementations. Hence, most type conversion behavior in PostgreSQL should be governed by general rules rather than by *ad hoc* heuristics, to allow mixed-type expressions to be meaningful even with user-defined types.

The PostgreSQL scanner/parser decodes lexical elements into only five fundamental categories: integers, floating-point numbers, strings, names, and key words. Most extended types are first tokenized into strings. The SQL language definition allows specifying type names with strings, and this mechanism can be used in PostgreSQL to start the parser down the correct path. For example, the query

```
tgl=> SELECT text 'Origin' AS "Label", point '(0,0)' AS "Value";
 Label  | Value
--------+-------
 Origin | (0,0)
(1 row)
```

has two literal constants, of type `text` and `point`. If a type is not specified for a string literal, then the placeholder type *unknown* is assigned initially, to be resolved in later stages as described below.

There are four fundamental SQL constructs requiring distinct type conversion rules in the PostgreSQL parser:

Operators

PostgreSQL allows expressions with prefix and postfix unary (one-argument) operators, as well as binary (two-argument) operators.

Function calls

Much of the PostgreSQL type system is built around a rich set of functions. Function calls have one or more arguments which, for any specific query, must be matched to the functions available in the system catalog. Since PostgreSQL permits function overloading, the function name alone does not uniquely identify the function to be called; the parser must select the right function based on the data types of the supplied arguments.

Query targets

SQL **INSERT** and **UPDATE** statements place the results of expressions into a table. The expressions in the query must be matched up with, and perhaps converted to, the types of the target columns.

`UNION` and `CASE` constructs

Since all select results from a unionized `SELECT` statement must appear in a single set of columns, the types of the results of each `SELECT` clause must be matched up and converted to a uniform set. Similarly, the result expressions of a `CASE` construct must be coerced to a common type so that the `CASE` expression as a whole has a known output type.

Many of the general type conversion rules use simple conventions built on the PostgreSQL function and operator system tables. There are some heuristics included in the conversion rules to better support conventions for the SQL standard native types such as `smallint`, `integer`, and `real`.

The PostgreSQL parser uses the convention that all type conversion functions take a single argument of the source type and are named with the same name as the target type. Any function meeting these criteria is considered to be a valid conversion function, and may be used by the parser as such. This simple assumption gives the parser the power to explore type conversion possibilities without hardcoding, allowing extended user-defined types to use these same features transparently.

An additional heuristic is provided in the parser to allow better guesses at proper behavior for SQL standard types. There are several basic *type categories* defined: `boolean`, `numeric`, `string`, `bitstring`, `datetime`, `timespan`, `geometric`, `network`, and user-defined. Each category, with the exception of user-defined, has a *preferred type* which is preferentially selected when there is ambiguity. In the user-defined category, each type is its own preferred type. Ambiguous expressions (those with multiple candidate parsing solutions) can often be resolved when there are multiple possible built-in types, but they will raise an error when there are multiple choices for user-defined types.

All type conversion rules are designed with several principles in mind:

- Implicit conversions should never have surprising or unpredictable outcomes.
- User-defined types, of which the parser has no *a priori* knowledge, should be “higher” in the type hierarchy. In mixed-type expressions, native types shall always be converted to a user-defined type (of course, only if conversion is necessary).
- User-defined types are not related. Currently, PostgreSQL does not have information available to it on relationships between types, other than hardcoded heuristics for built-in types and implicit relationships based on available functions in the catalog.
- There should be no extra overhead from the parser or executor if a query does not need implicit type conversion. That is, if a query is well formulated and the types already match up, then the query should proceed without spending extra time in the parser and without introducing unnecessary implicit conversion functions into the query.

  Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines an explicit function with the correct argument types, the parser should use this new function and will no longer do the implicit conversion using the old function.

5.3. Operators

The operand types of an operator invocation are resolved following the procedure below. Note that this procedure is indirectly affected by the precedence of the involved operators. See Section 1.4 for more information.

Operand Type Resolution

1. Check for an exact match in the `pg_operator` system catalog.

    a. If one argument of a binary operator is `unknown` type, then assume it is the same type as the other argument for this check. Other cases involving `unknown` will never find a match at this step.

2. Look for the best match.

    a. Make a list of all operators of the same name for which the input types match or can be coerced to match. (`unknown` literals are assumed to be coercible to anything for this purpose.) If there is only one, use it; else continue to the next step.

    b. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have any exact matches. If only one candidate remains, use it; else continue to the next step.

    c. Run through all candidates and keep those with the most exact or binary-compatible matches on input types. Keep all candidates if none have any exact or binary-compatible matches. If only one candidate remains, use it; else continue to the next step.

    d. Run through all candidates and keep those that accept preferred types at the most positions where type coercion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

    e. If any input arguments are “unknown”, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the "string" category if any candidate accepts that category (this bias towards string is appropriate since an unknown-type literal does look like a string). Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Also note whether any of the candidates accept a preferred data type within the selected category. Now discard operator candidates that do not accept the selected type category; furthermore, if any candidate accepts a preferred type at a given argument position, discard candidates that accept non-preferred types for that argument.

    f. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

Examples

Example 5-1. Exponentiation Operator Type Resolution

There is only one exponentiation operator defined in the catalog, and it takes arguments of type `double precision`. The scanner assigns an initial type of `integer` to both arguments of this query expression:

```
tgl=> SELECT 2 ^ 3 AS "Exp";
 Exp
-----
```

```
   8
(1 row)
```

So the parser does a type conversion on both operands and the query is equivalent to

```
tgl=> SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS "Exp";
 Exp
-----
   8
(1 row)
```

or

```
tgl=> SELECT 2.0 ^ 3.0 AS "Exp";
 Exp
-----
   8
(1 row)
```

> **Note:** This last form has the least overhead, since no functions are called to do implicit type conversion. This is not an issue for small queries, but may have an impact on the performance of queries involving large tables.

Example 5-2. String Concatenation Operator Type Resolution

A string-like syntax is used for working with string types as well as for working with complex extended types. Strings with unspecified type are matched with likely operator candidates.

An example with one unspecified argument:

```
tgl=> SELECT text 'abc' || 'def' AS "Text and Unknown";
 Text and Unknown
------------------
 abcdef
(1 row)
```

In this case the parser looks to see if there is an operator taking `text` for both arguments. Since there is, it assumes that the second argument should be interpreted as of type `text`.

Concatenation on unspecified types:

```
tgl=> SELECT 'abc' || 'def' AS "Unspecified";
 Unspecified
-------------
 abcdef
(1 row)
```

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that there are candidates accepting both string-category and bit-string-category inputs. Since string category is preferred when available, that category is selected, and then the "preferred type" for strings, `text`, is used as the specific type to resolve the unknown literals to.

Example 5-3. Absolute-Value and Factorial Operator Type Resolution

The PostgreSQL operator catalog has several entries for the prefix operator `@`, all of which implement absolute-value operations for various numeric data types. One of these entries is for type `float8`, which is the preferred type in the numeric category. Therefore, PostgreSQL will use that entry when faced with a non-numeric input:

```
tgl=> select @ text '-4.5' as "abs";
 abs
-----
 4.5
(1 row)
```

Here the system has performed an implicit text-to-float8 conversion before applying the chosen operator. We can verify that float8 and not some other type was used:

```
tgl=> select @ text '-4.5e500' as "abs";
ERROR:  Input '-4.5e500' is out of range for float8
```

On the other hand, the postfix operator `!` (factorial) is defined only for integer data types, not for float8. So, if we try a similar case with `!`, we get:

```
tgl=> select text '44' ! as "factorial";
ERROR:  Unable to identify a postfix operator '!' for type 'text'
        You may need to add parentheses or an explicit cast
```

This happens because the system can't decide which of the several possible `!` operators should be preferred. We can help it out with an explicit cast:

```
tgl=> select cast(text '44' as int8) ! as "factorial";
      factorial
---------------------
 2673996885588443136
(1 row)
```

5.4. Functions

The argument types of function calls are resolved according to the following steps.

Function Argument Type Resolution

1. Check for an exact match in the `pg_proc` system catalog. (Cases involving `unknown` will never find a match at this step.)
2. If no exact match appears in the catalog, see whether the function call appears to be a trivial type coercion request. This happens if the function call has just one argument and the function name is the same as the (internal) name of some data type. Furthermore, the function argument must be either an unknown-type literal or a type that is binary-compatible with the named data type. When these conditions are met, the function argument is coerced to the named data type without any explicit function call.
3. Look for the best match.

    a. Make a list of all functions of the same name with the same number of arguments for which the input types match or can be coerced to match. (`unknown` literals are assumed to be coercible to anything for this purpose.) If there is only one, use it; else continue to the next step.

b. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have any exact matches. If only one candidate remains, use it; else continue to the next step.

c. Run through all candidates and keep those with the most exact or binary-compatible matches on input types. Keep all candidates if none have any exact or binary-compatible matches. If only one candidate remains, use it; else continue to the next step.

d. Run through all candidates and keep those that accept preferred types at the most positions where type coercion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

e. If any input arguments are `unknown`, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the `string` category if any candidate accepts that category (this bias towards string is appropriate since an unknown-type literal does look like a string). Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Also note whether any of the candidates accept a preferred data type within the selected category. Now discard candidates that do not accept the selected type category; furthermore, if any candidate accepts a preferred type at a given argument position, discard candidates that accept non-preferred types for that argument.

f. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

Examples

Example 5-4. Factorial Function Argument Type Resolution

There is only one `int4fac` function defined in the `pg_proc` catalog. So the following query automatically converts the `int2` argument to `int4`:

```
tgl=> SELECT int4fac(int2 '4');
 int4fac
---------
      24
(1 row)
```

and is actually transformed by the parser to

```
tgl=> SELECT int4fac(int4(int2 '4'));
 int4fac
---------
      24
(1 row)
```

Example 5-5. Substring Function Type Resolution

There are two `substr` functions declared in `pg_proc`. However, only one takes two arguments, of types `text` and `int4`.

If called with a string constant of unspecified type, the type is matched up directly with the only candidate function type:

```
tgl=> SELECT substr('1234', 3);
 substr
```

```
--------
     34
(1 row)
```

If the string is declared to be of type `varchar`, as might be the case if it comes from a table, then the parser will try to coerce it to become `text`:

```
tgl=> SELECT substr(varchar ’1234’, 3);
 substr
--------
     34
(1 row)
```

which is transformed by the parser to become

```
tgl=> SELECT substr(text(varchar ’1234’), 3);
 substr
--------
     34
(1 row)
```

> **Note:** Actually, the parser is aware that `text` and `varchar` are *binary-compatible*, meaning that one can be passed to a function that accepts the other without doing any physical conversion. Therefore, no explicit type conversion call is really inserted in this case.

And, if the function is called with an `int4`, the parser will try to convert that to `text`:

```
tgl=> SELECT substr(1234, 3);
 substr
--------
     34
(1 row)
```

which actually executes as

```
tgl=> SELECT substr(text(1234), 3);
 substr
--------
     34
(1 row)
```

This succeeds because there is a conversion function text(int4) in the system catalog.

5.5. Query Targets

Values to be inserted into a table are coerced to the destination column’s data type according to the following steps.

Query Target Type Resolution

1. Check for an exact match with the target.
2. Otherwise, try to coerce the expression to the target type. This will succeed if the two types are known binary-compatible, or if there is a conversion function. If the expression is an unknown-type literal, the contents of the literal string will be fed to the input conversion routine for the target type.

3. If the target is a fixed-length type (e.g. `char` or `varchar` declared with a length) then try to find a sizing function for the target type. A sizing function is a function of the same name as the type, taking two arguments of which the first is that type and the second is an integer, and returning the same type. If one is found, it is applied, passing the column's declared length as the second parameter.

Example 5-6. `character` Storage Type Conversion

For a target column declared as `character(20)` the following query ensures that the target is sized correctly:

```
tgl=> CREATE TABLE vv (v character(20));
CREATE
tgl=> INSERT INTO vv SELECT 'abc' || 'def';
INSERT 392905 1
tgl=> SELECT v, length(v) FROM vv;
          v           | length
----------------------+--------
 abcdef               |     20
(1 row)
```

What has really happened here is that the two unknown literals are resolved to `text` by default, allowing the `||` operator to be resolved as `text` concatenation. Then the `text` result of the operator is coerced to `bpchar` ("blank-padded char", the internal name of the character data type) to match the target column type. (Since the parser knows that `text` and `bpchar` are binary-compatible, this coercion is implicit and does not insert any real function call.) Finally, the sizing function `bpchar(bpchar, integer)` is found in the system catalogs and applied to the operator's result and the stored column length. This type-specific function performs the required length check and addition of padding spaces.

5.6. `UNION` and `CASE` Constructs

SQL `UNION` constructs must match up possibly dissimilar types to become a single result set. The resolution algorithm is applied separately to each output column of a union query. The `INTERSECT` and `EXCEPT` constructs resolve dissimilar types in the same way as `UNION`. A `CASE` construct also uses the identical algorithm to match up its component expressions and select a result data type.

`UNION` and `CASE` Type Resolution

1. If all inputs are of type `unknown`, resolve as type `text` (the preferred type for string category). Otherwise, ignore the `unknown` inputs while choosing the type.
2. If the non-unknown inputs are not all of the same type category, fail.
3. If one or more non-unknown inputs are of a preferred type in that category, resolve as that type.
4. Otherwise, resolve as the type of the first non-unknown input.
5. Coerce all inputs to the selected type.

Examples

Example 5-7. Underspecified Types in a Union

```
tgl=> SELECT text 'a' AS "Text" UNION SELECT 'b';
 Text
------
 a
 b
(2 rows)
```

Here, the unknown-type literal `'b'` will be resolved as type text.

Example 5-8. Type Conversion in a Simple Union

```
tgl=> SELECT 1.2 AS "Double" UNION SELECT 1;
 Double
--------
      1
    1.2
(2 rows)
```

The literal `1.2` is of type `double precision`, the preferred type in the numeric category, so that type is used.

Example 5-9. Type Conversion in a Transposed Union

Here the output type of the union is forced to match the type of the first clause in the union:

```
tgl=> SELECT 1 AS "All integers"
tgl-> UNION SELECT CAST('2.2' AS REAL);
 All integers
--------------
            1
            2
(2 rows)
```

Since `REAL` is not a preferred type, the parser sees no reason to select it over `INTEGER` (which is what the 1 is), and instead falls back on the use-the-first-alternative rule. This example demonstrates that the preferred-type mechanism doesn't encode as much information as we'd like. Future versions of PostgreSQL may support a more general notion of type preferences.

Chapter 6. Arrays

PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays. Arrays of any built-in type or user-defined type can be created. To illustrate their use, we create this table:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

As shown, an array data type is named by appending square brackets (`[]`) to the data type name of the array elements. The above query will create a table named `sal_emp` with a `text` string (`name`), a one-dimensional array of type `integer` (`pay_by_quarter`), which represents the employee's salary by quarter, and a two-dimensional array of `text` (`schedule`), which represents the employee's weekly schedule.

Now we do some **INSERT**s. Observe that to write an array value, we enclose the element values within curly braces and separate them by commas. If you know C, this is not unlike the syntax for initializing structures.

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

Now, we can run some queries on `sal_emp`. First, we show how to access a single element of an array at a time. This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

The array subscript numbers are written within square brackets. PostgreSQL uses the "one-based" numbering convention for arrays, that is, an array of *n* elements starts with `array[1]` and ends with `array[`*n*`]`.

This query retrieves the third quarter pay of all employees:

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

We can also access arbitrary rectangular slices of an array, or subarrays. An array slice is denoted by writing *`lower subscript : upper subscript`* for one or more array dimensions. This query retrieves the first item on Bill's schedule for the first two days of the week:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

      schedule
--------------------
 {{"meeting"},{""}}
(1 row)
```

We could also have written

```
SELECT schedule[1:2][1] FROM sal_emp WHERE name = 'Bill';
```

with the same result. An array subscripting operation is taken to represent an array slice if any of the subscripts are written in the form *`lower : upper`*. A lower bound of 1 is assumed for any subscript where only one value is specified.

An array value can be replaced completely:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

or updated at a single element:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

or updated in a slice:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

An array can be enlarged by assigning to an element adjacent to those already present, or by assigning to a slice that is adjacent to or overlaps the data already present. For example, if an array value currently has 4 elements, it will have five elements after an update that assigns to array[5]. Currently, enlargement in this fashion is only allowed for one-dimensional arrays, not multidimensional arrays.

The syntax for **CREATE TABLE** allows fixed-length arrays to be defined:

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

However, the current implementation does not enforce the array size limits --- the behavior is the same as for arrays of unspecified length.

Actually, the current implementation does not enforce the declared number of dimensions either. Arrays of a particular element type are all considered to be of the same type, regardless of size or number of dimensions.

The current dimensions of any array value can be retrieved with the `array_dims` function:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';
```

```
 array_dims
------------
 [1:2][1:1]
(1 row)
```

`array_dims` produces a `text` result, which is convenient for people to read but perhaps not so convenient for programs.

To search for a value in an array, you must check each value of the array. This can be done by hand (if you know the size of the array):

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

However, this quickly becomes tedious for large arrays, and is not helpful if the size of the array is unknown. Although it is not part of the primary PostgreSQL distribution, there is an extension available that defines new functions and operators for iterating over array values. Using this, the above query could be:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1:4] *= 10000;
```

To search the entire array (not just specified columns), you could use:

```
SELECT * FROM sal_emp WHERE pay_by_quarter *= 10000;
```

In addition, you could find rows where the array had all values equal to 10 000 with:

```
SELECT * FROM sal_emp WHERE pay_by_quarter **= 10000;
```

To install this optional module, look in the `contrib/array` directory of the PostgreSQL source distribution.

> **Tip:** Arrays are not sets; using arrays in the manner described in the previous paragraph is often a sign of database misdesign. The array field should generally be split off into a separate table. Tables can obviously be searched easily.

> **Note:** A limitation of the present array implementation is that individual elements of an array cannot be SQL NULLs. The entire array can be set to NULL, but you can't have an array with some elements NULL and some not. Fixing this is on the to-do list.

Quoting array elements. As shown above, when writing an array literal value you may write double quotes around any individual array element. You *must* do so if the element value would otherwise confuse the array-value parser. For example, elements containing curly braces, commas, double quotes, backslashes, or white space must be double-quoted. To put a double quote or backslash in an array element value, precede it with a backslash.

> **Tip:** Remember that what you write in an SQL query will first be interpreted as a string literal, and then as an array. This doubles the number of backslashes you need. For example, to insert a `text` array value containing a backslash and a double quote, you'd need to write
>
> ```
> INSERT ... VALUES ('{"\\\\","\\""}');
> ```

The string-literal processor removes one level of backslashes, so that what arrives at the array-value parser looks like `{"\\",\"\"}`. In turn, the strings fed to the `text` data type's input routine become `\` and `"` respectively. (If we were working with a data type whose input routine also treated backslashes specially, `bytea` for example, we might need as many as eight backslashes in the query to get one backslash into the stored array element.)

Chapter 7. Indexes

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

7.1. Introduction

The classical example for the need of an index is if there is a table similar to this:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

and the application requires a lot of queries of the form

```
SELECT content FROM test1 WHERE id = constant;
```

Ordinarily, the system would have to scan the entire `test1` table row by row to find all matching entries. If there are a lot of rows in `test1` and only a few rows (possibly zero or one) returned by the query, then this is clearly an inefficient method. If the system were instructed to maintain an index on the `id` column, then it could use a more efficient method for locating matching rows. For instance, it might only have to walk a few levels deep into a search tree.

A similar approach is used in most books of non-fiction: Terms and concepts that are frequently looked up by readers are collected in an alphabetic index at the end of the book. The interested reader can scan the index relatively quickly and flip to the appropriate page, and would not have to read the entire book to find the interesting location. As it is the task of the author to anticipate the items that the readers are most likely to look up, it is the task of the database programmer to foresee which indexes would be of advantage.

The following command would be used to create the index on the `id` column, as discussed:

```
CREATE INDEX test1_id_index ON test1 (id);
```

The name `test1_id_index` can be chosen freely, but you should pick something that enables you to remember later what the index was for.

To remove an index, use the **DROP INDEX** command. Indexes can be added to and removed from tables at any time.

Once the index is created, no further intervention is required: the system will use the index when it thinks it would be more efficient than a sequential table scan. But you may have to run the **ANALYZE** command regularly to update statistics to allow the query planner to make educated decisions. Also read Chapter 11 for information about how to find out whether an index is used and when and why the planner may choose to *not* use an index.

Indexes can benefit **UPDATE**s and **DELETE**s with search conditions. Indexes can also be used in join queries. Thus, an index defined on a column that is part of a join condition can significantly speed up queries with joins.

When an index is created, the system has to keep it synchronized with the table. This adds overhead to data manipulation operations. Therefore indexes that are non-essential or do not get used at all should be removed. Note that a query or data manipulation command can use at most one index per table.

7.2. Index Types

PostgreSQL provides several index types: B-tree, R-tree, GiST, and Hash. Each index type is more appropriate for a particular query type because of the algorithm it uses. By default, the **CREATE INDEX** command will create a B-tree index, which fits the most common situations. In particular, the PostgreSQL query optimizer will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators: <, <=, =, >=, >

R-tree indexes are especially suited for spatial data. To create an R-tree index, use a command of the form

```
CREATE INDEX name ON table USING RTREE (column);
```

The PostgreSQL query optimizer will consider using an R-tree index whenever an indexed column is involved in a comparison using one of these operators: <<, &<, &>, >>, @, ~=, && (Refer to Section 4.9 about the meaning of these operators.)

The query optimizer will consider using a hash index whenever an indexed column is involved in a comparison using the = operator. The following command is used to create a hash index:

```
CREATE INDEX name ON table USING HASH (column);
```

> **Note:** Because of the limited utility of hash indexes, a B-tree index should generally be preferred over a hash index. We do not have sufficient evidence that hash indexes are actually faster than B-trees even for = comparisons. Moreover, hash indexes require coarser locks; see Section 9.7.

The B-tree index is an implementation of Lehman-Yao high-concurrency B-trees. The R-tree index method implements standard R-trees using Guttman's quadratic split algorithm. The hash index is an implementation of Litwin's linear hashing. We mention the algorithms used solely to indicate that all of these access methods are fully dynamic and do not have to be optimized periodically (as is the case with, for example, static hash access methods).

7.3. Multicolumn Indexes

An index can be defined on more than one column. For example, if you have a table of this form:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

(Say, you keep your `/dev` directory in a database...) and you frequently make queries like

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

then it may be appropriate to define an index on the columns `major` and `minor` together, e.g.,

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

Currently, only the B-tree and GiST implementations support multicolumn indexes. Up to 16 columns may be specified. (This limit can be altered when building PostgreSQL; see the file `pg_config.h`.)

The query optimizer can use a multicolumn index for queries that involve the first *n* consecutive columns in the index (when used with appropriate operators), up to the total number of columns specified in the index definition. For example, an index on `(a, b, c)` can be used in queries involving all of `a`, `b`, and `c`, or in queries involving both `a` and `b`, or in queries involving only `a`, but not in other combinations. (In a query involving `a` and `c` the optimizer might choose to use the index for `a` only and treat `c` like an ordinary unindexed column.)

Multicolumn indexes can only be used if the clauses involving the indexed columns are joined with `AND`. For instance,

```
SELECT name FROM test2 WHERE major = constant OR minor = constant;
```

cannot make use of the index `test2_mm_idx` defined above to look up both columns. (It can be used to look up only the `major` column, however.)

Multicolumn indexes should be used sparingly. Most of the time, an index on a single column is sufficient and saves space and time. Indexes with more than three columns are almost certainly inappropriate.

7.4. Unique Indexes

Indexes may also be used to enforce uniqueness of a column's value, or the uniqueness of the combined values of more than one column.

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

Currently, only B-tree indexes can be declared unique.

When an index is declared unique, multiple table rows with equal indexed values will not be allowed. NULL values are not considered equal.

PostgreSQL automatically creates unique indexes when a table is declared with a unique constraint or a primary key, on the columns that make up the primary key or unique columns (a multicolumn index, if appropriate), to enforce that constraint. A unique index can be added to a table at any later time, to add a unique constraint.

> **Note:** The preferred way to add a unique constraint to a table is `ALTER TABLE ... ADD CONSTRAINT`. The use of indexes to enforce unique constraints could be considered an implementation detail that should not be accessed directly.

7.5. Functional Indexes

For a *functional index*, an index is defined on the result of a function applied to one or more columns of a single table. Functional indexes can be used to obtain fast access to data based on the result of function calls.

For example, a common way to do case-insensitive comparisons is to use the `lower` function:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

This query can use an index, if one has been defined on the result of the `lower(column)` operation:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

The function in the index definition can take more than one argument, but they must be table columns, not constants. Functional indexes are always single-column (namely, the function result) even if the function uses more than one input field; there cannot be multicolumn indexes that contain function calls.

> **Tip:** The restrictions mentioned in the previous paragraph can easily be worked around by defining a custom function to use in the index definition that computes any desired result internally.

7.6. Operator Classes

An index definition may specify an *operator class* for each column of an index.

```
CREATE INDEX name ON table (column opclass [, ...]);
```

The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. There are also some operator classes with special purposes:

- The operator classes `box_ops` and `bigbox_ops` both support R-tree indexes on the `box` data type. The difference between them is that `bigbox_ops` scales box coordinates down, to avoid floating-point exceptions from doing multiplication, addition, and subtraction on very large floating-point coordinates. If the field on which your rectangles lie is about 20 000 units square or larger, you should use `bigbox_ops`.

The following query shows all defined operator classes:

```
SELECT am.amname AS acc_method,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    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 acc_method, ops_name, ops_comp
```

7.7. Keys

Author: Written by Herouth Maoz (<herouth@oumail.openu.ac.il>). This originally appeared on the User's Mailing List on 1998-03-02 in response to the question: "What is the difference between PRIMARY KEY and UNIQUE constraints?".

Subject: Re: [QUESTIONS] PRIMARY KEY | UNIQUE

What's the difference between:

PRIMARY KEY(fields,...) and
UNIQUE (fields,...)

- Is this an alias?
- If PRIMARY KEY is already unique, then why is there another kind of key named UNIQUE?

A primary key is the field(s) used to identify a specific row. For example, Social Security numbers identifying a person.

A simply UNIQUE combination of fields has nothing to do with identifying the row. It's simply an integrity constraint. For example, I have collections of links. Each collection is identified by a unique number, which is the primary key. This key is used in relations.

However, my application requires that each collection will also have a unique name. Why? So that a human being who wants to modify a collection will be able to identify it. It's much harder to know, if you have two collections named "Life Science", the one tagged 24433 is the one you need, and the one tagged 29882 is not.

So, the user selects the collection by its name. We therefore make sure, within the database, that names are unique. However, no other table in the database relates to the collections table by the collection Name. That would be very inefficient.

Moreover, despite being unique, the collection name does not actually define the collection! For example, if somebody decided to change the name of the collection from "Life Science" to "Biology", it will still be the same collection, only with a different name. As long as the name is unique, that's OK.

So:

- Primary key:
  - Is used for identifying the row and relating to it.
  - Is impossible (or hard) to update.
  - Should not allow NULLs.
- Unique field(s):
  - Are used as an alternative access to the row.
  - Are updatable, so long as they are kept unique.
  - NULLs are acceptable.

As for why no non-unique keys are defined explicitly in standard SQL syntax? Well, you must understand that indexes are implementation-dependent. SQL does not define the implementation, merely the relations between data in the database. PostgreSQL does allow non-unique indexes, but indexes used to enforce SQL keys are always unique.

Thus, you may query a table by any combination of its columns, despite the fact that you don't have an index on these columns. The indexes are merely an implementation aid that each RDBMS offers you, in order to cause commonly used queries to be done more efficiently. Some RDBMS may give you additional measures, such as keeping a key stored in main memory. They will have a special command, for example

```
CREATE MEMSTORE ON table COLUMNS cols
```

(This is not an existing command, just an example.)

In fact, when you create a primary key or a unique combination of fields, nowhere in the SQL specification does it say that an index is created, nor that the retrieval of data by the key is going to be more efficient than a sequential scan!

So, if you want to use a combination of fields that is not unique as a secondary key, you really don't have to specify anything - just start retrieving by that combination! However, if you want to make the retrieval efficient, you'll have to resort to the means your RDBMS provider gives you - be it an index, my imaginary `MEMSTORE` command, or an intelligent RDBMS that creates indexes without your knowledge based on the fact that you have sent it many queries based on a specific combination of keys... (It learns from experience).

7.8. Partial Indexes

A *partial index* is an index built over a subset of a table; the subset is defined by a conditional expression (called the *predicate* of the partial index). The index contains entries for only those table rows that satisfy the predicate.

A major motivation for partial indexes is to avoid indexing common values. Since a query searching for a common value (one that accounts for more than a few percent of all the table rows) will not use the index anyway, there is no point in keeping those rows in the index at all. This reduces the size of the index, which will speed up queries that do use the index. It will also speed up many table update operations because the index does not need to be updated in all cases. Example 7-1 shows a possible application of this idea.

Example 7-1. Setting up a Partial Index to Exclude Common Values

Suppose you are storing web server access logs in a database. Most accesses originate from the IP range of your organization but some are from elsewhere (say, employees on dial-up connections). If your searches by IP are primarily for outside accesses, you probably do not need to index the IP range that corresponds to your organization's subnet.

Assume a table like this:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

To create a partial index that suits our example, use a command such as this:

```
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');
```

A typical query that can use this index would be:

```
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

A query that cannot use this index is:

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

Observe that this kind of partial index requires that the common values be predetermined. If the distribution of values is inherent (due to the nature of the application) and static (not changing over time), this is not difficult, but if the common values are merely due to the coincidental data load this can require a lot of maintenance work.

Another possibility is to exclude values from the index that the typical query workload is not interested in; this is shown in Example 7-2. This results in the same advantages as listed above, but it prevents the "uninteresting" values from being accessed via that index at all, even if an index scan might be profitable in that case. Obviously, setting up partial indexes for this kind of scenario will require a lot of care and experimentation.

Example 7-2. Setting up a Partial Index to Exclude Uninteresting Values

If you have a table that contains both billed and unbilled orders, where the unbilled orders take up a small fraction of the total table and yet those are the most-accessed rows, you can improve performance by creating an index on just the unbilled rows. The command to create the index would look like this:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

A possible query to use this index would be

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

However, the index can also be used in queries that do not involve `order_nr` at all, e.g.,

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

This is not as efficient as a partial index on the `amount` column would be, since the system has to scan the entire index. Yet, if there are relatively few unbilled orders, using this partial index just to find the unbilled orders could be a win.

Note that this query cannot use this index:

```
SELECT * FROM orders WHERE order_nr = 3501;
```

The order 3501 may be among the billed or among the unbilled orders.

Example 7-2 also illustrates that the indexed column and the column used in the predicate do not need to match. PostgreSQL supports partial indexes with arbitrary predicates, so long as only columns of the table being indexed are involved. However, keep in mind that the predicate must match the conditions used in the queries that are supposed to benefit from the index. To be precise, a partial index can be used in a query only if the system can recognize that the query's WHERE condition mathematically *implies* the index's predicate. PostgreSQL does not have a sophisticated theorem prover that can recognize mathematically equivalent predicates that are written in different forms. (Not only is such a general theorem prover extremely difficult to create, it would probably be too slow to be of any real use.) The system can recognize simple inequality implications, for example "x < 1" implies "x < 2"; otherwise the predicate condition must exactly match the query's WHERE condition or the index will not be recognized to be usable.

A third possible use for partial indexes does not require the index to be used in queries at all. The idea here is to create a unique index over a subset of a table, as in Example 7-3. This enforces uniqueness among the rows that satisfy the index predicate, without constraining those that do not.

Example 7-3. Setting up a Partial Unique Index

Suppose that we have a table describing test outcomes. We wish to ensure that there is only one "successful" entry for a given subject and target combination, but there might be any number of "unsuccessful" entries. Here is one way to do it:

```
CREATE TABLE tests (subject text,
                    target text,
                    success bool,
                    ...);
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

This is a particularly efficient way of doing it when there are few successful trials and many unsuccessful ones.

Finally, a partial index can also be used to override the system's query plan choices. It may occur that data sets with peculiar distributions will cause the system to use an index when it really should not. In that case the index can be set up so that it is not available for the offending query. Normally, PostgreSQL makes reasonable choices about index usage (e.g., it avoids them when retrieving common values, so the earlier example really only saves index size, it is not required to avoid index usage), and grossly incorrect plan choices are cause for a bug report.

Keep in mind that setting up a partial index indicates that you know at least as much as the query planner knows, in particular you know when an index might be profitable. Forming this knowledge requires experience and understanding of how indexes in PostgreSQL work. In most cases, the advantage of a partial index over a regular index will not be much.

More information about partial indexes can be found in *The case for partial indexes*, *Partial indexing in POSTGRES: research project*, and *Generalized Partial Indexes*.

7.9. Examining Index Usage

Although indexes in PostgreSQL do not need maintenance and tuning, it is still important to check which indexes are actually used by the real-life query workload. Examining index usage is done with the **EXPLAIN** command; its application for this purpose is illustrated in Section 11.1.

It is difficult to formulate a general procedure for determining which indexes to set up. There are a number of typical cases that have been shown in the examples throughout the previous sections. A good deal of experimentation will be necessary in most cases. The rest of this section gives some tips for that.

- Always run **ANALYZE** first. This command collects statistics about the distribution of the values in the table. This information is required to guess the number of rows returned by a query, which is needed by the planner to assign realistic costs to each possible query plan. In absence of any real statistics, some default values are assumed, which are almost certain to be inaccurate. Examining an application's index usage without having run **ANALYZE** is therefore a lost cause.
- Use real data for experimentation. Using test data for setting up indexes will tell you what indexes you need for the test data, but that is all.

It is especially fatal to use proportionally reduced data sets. While selecting 1000 out of 100000 rows could be a candidate for an index, selecting 1 out of 100 rows will hardly be, because the 100 rows will probably fit within a single disk page, and there is no plan that can beat sequentially fetching 1 disk page.

Also be careful when making up test data, which is often unavoidable when the application is not in production use yet. Values that are very similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.

- When indexes are not used, it can be useful for testing to force their use. There are run-time parameters that can turn off various plan types (described in the *Administrator's Guide*). For instance, turning off sequential scans (`enable_seqscan`) and nested-loop joins (`enable_nestloop`), which are the most basic plans, will force the system to use a different plan. If the system still chooses a sequential scan or nested-loop join then there is probably a more fundamental problem for why the index is not used, for example, the query condition does not match the index. (What kind of query can use what kind of index is explained in the previous sections.)
- If forcing index usage does use the index, then there are two possibilities: Either the system is right and using the index is indeed not appropriate, or the cost estimates of the query plans are not reflecting reality. So you should time your query with and without indexes. The **EXPLAIN ANALYZE** command can be useful here.
- If it turns out that the cost estimates are wrong, there are, again, two possibilities. The total cost is computed from the per-row costs of each plan node times the selectivity estimate of the plan node. The costs of the plan nodes can be tuned with run-time parameters (described in the *Administrator's Guide*). An inaccurate selectivity estimate is due to insufficient statistics. It may be possible to help this by tuning the statistics-gathering parameters (see **ALTER TABLE** reference).

  If you do not succeed in adjusting the costs to be more appropriate, then you may have to resort to forcing index usage explicitly. You may also want to contact the PostgreSQL developers to examine the issue.

Chapter 8. Inheritance

Let’s create two tables. The capitals table contains state capitals which are also cities. Naturally, the capitals table should inherit from cities.

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- (in ft)
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

In this case, a row of capitals *inherits* all attributes (name, population, and altitude) from its parent, cities. The type of the attribute name is `text`, a native PostgreSQL type for variable length ASCII strings. The type of the attribute population is `float`, a native PostgreSQL type for double precision floating-point numbers. State capitals have an extra attribute, state, that shows their state. In PostgreSQL, a table can inherit from zero or more other tables, and a query can reference either all rows of a table or all rows of a table plus all of its descendants.

> **Note:** The inheritance hierarchy is actually a directed acyclic graph.

For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500ft:

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;
```

which returns:

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

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude of 500ft or higher:

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;
```

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

Here the “ONLY” before cities indicates that the query should be run over only cities and not tables below cities in the inheritance hierarchy. Many of the commands that we have already discussed -- **SELECT**, **UPDATE** and **DELETE** -- support this “ONLY” notation.

In some cases you may wish to know which table a particular tuple originated from. There is a system column called `TABLEOID` in each table which can tell you the originating table:

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

which returns:

```
+---------+----------+----------+
|tableoid |name      | altitude |
+---------+----------+----------+
|37292    |Las Vegas | 2174     |
+---------+----------+----------+
|37280    |Mariposa  | 1953     |
+---------+----------+----------+
|37280    |Madison   | 845      |
+---------+----------+----------+
```

If you do a join with pg_class you can see the actual table name:

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 and c.tableoid = p.oid;
```

which returns:

```
+---------+----------+----------+
|relname  |name      | altitude |
+---------+----------+----------+
|capitals |Las Vegas | 2174     |
+---------+----------+----------+
|cities   |Mariposa  | 1953     |
+---------+----------+----------+
|cities   |Madison   | 845      |
+---------+----------+----------+
```

Deprecated: In previous versions of PostgreSQL, the default was not to get access to child tables. This was found to be error prone and is also in violation of SQL99. Under the old syntax, to get the sub-tables you append `*` to the table name. For example

```
SELECT * from cities*;
```

You can still explicitly specify scanning child tables by appending `*`, as well as explicitly specify not scanning child tables by writing “ONLY”. But beginning in version 7.1, the default behavior for an undecorated table name is to scan its child tables too, whereas before the default was not to do so. To get the old default behavior, set the configuration option `SQL_Inheritance` to off, e.g.,

```
SET SQL_Inheritance TO OFF;
```

or add a line in your `postgresql.conf` file.

A limitation of the inheritance feature is that indexes (including unique constraints) and foreign key constraints only apply to single tables, not to their inheritance children. Thus, in the above example, specifying that another table’s column `REFERENCES cities(name)` would allow the other table to contain city names but not capital names. This deficiency will probably be fixed in some future release.

Chapter 9. Multiversion Concurrency Control

Multiversion Concurrency Control (MVCC) is an advanced technique for improving database performance in a multiuser environment. Vadim Mikheev (<vadim@krs.ru>) provided the implementation for PostgreSQL.

9.1. Introduction

Unlike most other database systems which use locks for concurrency control, PostgreSQL maintains data consistency by using a multiversion model. This means that while querying a database each transaction sees a snapshot of data (a *database version*) as it was some time ago, regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data that could be caused by (other) concurrent transaction updates on the same data rows, providing *transaction isolation* for each database session.

The main difference between multiversion and lock models is that in MVCC locks acquired for querying (reading) data don't conflict with locks acquired for writing data and so reading never blocks writing and writing never blocks reading.

9.2. Transaction Isolation

The ANSI/ISO SQL standard defines four levels of transaction isolation in terms of three phenomena that must be prevented between concurrent transactions. These undesirable phenomena are:

dirty reads

A transaction reads data written by concurrent uncommitted transaction.

non-repeatable reads

A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

phantom read

A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

The four transaction isolation levels and the corresponding behaviors are described in Table 9-1.

Table 9-1. SQL Transaction Isolation Levels

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

PostgreSQL offers the read committed and serializable isolation levels.

9.3. Read Committed Isolation Level

Read Committed is the default isolation level in PostgreSQL. When a transaction runs on this isolation level, a **SELECT** query sees only data committed before the query began and never sees either uncommitted data or changes committed during query execution by concurrent transactions. (However, the **SELECT** does see the effects of previous updates executed within this same transaction, even though they are not yet committed.) Notice that two successive **SELECT**s can see different data, even though they are within a single transaction, when other transactions commit changes during execution of the first **SELECT**.

If a target row found by a query while executing an **UPDATE** statement (or **DELETE** or **SELECT FOR UPDATE**) has already been updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback. In the case of rollback, the waiting transaction can proceed to change the row. In the case of commit (and if the row still exists; i.e. was not deleted by the other transaction), the query will be re-executed for this row to check that the new row version still satisfies the query search condition. If the new row version satisfies the query search condition then the row will be updated (or deleted or marked for update). Note that the starting point for the update will be the new row version; moreover, after the update the doubly-updated row is visible to subsequent **SELECT**s in the current transaction. Thus, the current transaction is able to see the effects of the other transaction for this specific row.

The partial transaction isolation provided by Read Committed level is adequate for many applications, and this level is fast and simple to use. However, for applications that do complex queries and updates, it may be necessary to guarantee a more rigorously consistent view of the database than the Read Committed level provides.

9.4. Serializable Isolation Level

Serializable provides the highest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. However, applications using this level must be prepared to retry transactions due to serialization failures.

When a transaction is on the serializable level, a **SELECT** query sees only data committed before the transaction began and never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. (However, the **SELECT** does see the effects of previous updates executed within this same transaction, even though they are not yet committed.) This is different from Read Committed in that the **SELECT** sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction.

If a target row found by a query while executing an **UPDATE** statement (or **DELETE** or **SELECT FOR UPDATE**) has already been updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback. In the case of rollback, the waiting transaction can proceed to change the row. In the case of a concurrent transaction commit, a serializable transaction will be rolled back with the message

```
ERROR:  Can't serialize access due to concurrent update
```

because a serializable transaction cannot modify rows changed by other transactions after the serializable transaction began.

When the application receives this error message, it should abort the current transaction and then retry the whole transaction from the beginning. The second time through, the transaction sees the previously-committed change as part of its initial view of the database, so there is no logical conflict in using the new version of the row as the starting point for the new transaction's update. Note that only updating transactions may need to be retried --- read-only transactions never have serialization conflicts.

The Serializable transaction level provides a rigorous guarantee that each transaction sees a wholly consistent view of the database. However, the application has to be prepared to retry transactions when concurrent updates make it impossible to sustain the illusion of serial execution, and the cost of redoing complex transactions may be significant. So this level is recommended only when update queries contain logic sufficiently complex that they may give wrong answers in the Read Committed level.

9.5. Data consistency checks at the application level

Because readers in PostgreSQL don't lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another concurrent transaction. In other words, if a row is returned by **SELECT** it doesn't mean that the row still exists at the time it is returned (i.e., sometime after the current transaction began); the row might have been modified or deleted by an already-committed transaction that committed after this one started. Even if the row is still valid "now", it could be changed or deleted before the current transaction does a commit or rollback.

Another way to think about it is that each transaction sees a snapshot of the database contents, and concurrently executing transactions may very well see different snapshots. So the whole concept of "now" is somewhat suspect anyway. This is not normally a big problem if the client applications are isolated from each other, but if the clients can communicate via channels outside the database then serious confusion may ensue.

To ensure the current existence of a row and protect it against concurrent updates one must use **SELECT FOR UPDATE** or an appropriate **LOCK TABLE** statement. (**SELECT FOR UPDATE** locks just the returned rows against concurrent updates, while **LOCK TABLE** protects the whole table.) This should be taken into account when porting applications to PostgreSQL from other environments.

Note: Before version 6.5 PostgreSQL used read-locks and so the above consideration is also the case when upgrading to 6.5 (or higher) from previous PostgreSQL versions.

9.6. Locking and Tables

PostgreSQL provides various lock modes to control concurrent access to data in tables. Some of these lock modes are acquired by PostgreSQL automatically before statement execution, while others are provided to be used by applications. All lock modes acquired in a transaction are held for the duration of the transaction.

9.6.1. Table-level locks

AccessShareLock

A read-lock mode acquired automatically on tables being queried.

Conflicts with AccessExclusiveLock only.

RowShareLock

Acquired by **SELECT FOR UPDATE** and **LOCK TABLE** for `IN ROW SHARE MODE` statements.

Conflicts with ExclusiveLock and AccessExclusiveLock modes.

RowExclusiveLock

Acquired by **UPDATE**, **DELETE**, **INSERT** and **LOCK TABLE** for `IN ROW EXCLUSIVE MODE` statements.

Conflicts with ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ShareUpdateExclusiveLock

Acquired by **VACUUM** (without `FULL`) and **LOCK TABLE** table for `IN SHARE UPDATE EXCLUSIVE MODE` statements.

Conflicts with ShareUpdateExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ShareLock

Acquired by **CREATE INDEX** and **LOCK TABLE** table for `IN SHARE MODE` statements.

Conflicts with RowExclusiveLock, ShareUpdateExclusiveLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ShareRowExclusiveLock

Acquired by **LOCK TABLE** for `IN SHARE ROW EXCLUSIVE MODE` statements.

Conflicts with RowExclusiveLock, ShareUpdateExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ExclusiveLock

Acquired by **LOCK TABLE** table for `IN EXCLUSIVE MODE` statements.

Conflicts with RowShareLock, RowExclusiveLock, ShareUpdateExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

AccessExclusiveLock

Acquired by **ALTER TABLE**, **DROP TABLE**, **VACUUM FULL** and **LOCK TABLE** statements.

Conflicts with all modes (AccessShareLock, RowShareLock, RowExclusiveLock, ShareUpdateExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock).

Note: Only AccessExclusiveLock blocks **SELECT** (without `FOR UPDATE`) statement.

9.6.2. Row-level locks

Row-level locks are acquired when rows are being updated (or deleted or marked for update). Row-level locks don't affect data querying. They block writers to *the same row* only.

PostgreSQL doesn't remember any information about modified rows in memory and so has no limit to the number of rows locked at one time. However, locking a row may cause a disk write; thus, for example, **SELECT FOR UPDATE** will modify selected rows to mark them and so will result in disk writes.

In addition to table and row locks, short-term share/exclusive locks are used to control read/write access to table pages in the shared buffer pool. These locks are released immediately after a tuple is fetched or updated. Application writers normally need not be concerned with page-level locks, but we mention them for completeness.

9.7. Locking and Indexes

Though PostgreSQL provides nonblocking read/write access to table data, nonblocking read/write access is not currently offered for every index access method implemented in PostgreSQL.

The various index types are handled as follows:

GiST and R-Tree indexes

> Share/exclusive index-level locks are used for read/write access. Locks are released after statement is done.

Hash indexes

> Share/exclusive page-level locks are used for read/write access. Locks are released after page is processed.
>
> Page-level locks provide better concurrency than index-level ones but are subject to deadlocks.

B-tree indexes

> Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index tuple is fetched/inserted.
>
> B-tree indexes provide the highest concurrency without deadlock conditions.

In short, B-tree indexes are the recommended index type for concurrent applications.

Chapter 10. Managing a Database

Although the *site administrator* is responsible for overall management of the PostgreSQL installation, some databases within the installation may be managed by another person, designated the *database administrator*. This assignment of responsibilities occurs when a database is created. A user may be assigned explicit privileges to create databases and/or to create new users. A user assigned both privileges can perform most administrative tasks within PostgreSQL, but will not by default have the same operating system privileges as the site administrator.

The *Administrator's Guide* covers these topics in more detail.

10.1. Database Creation

Databases are created by the **CREATE DATABASE** command issued from within PostgreSQL. createdb is a shell script provided to give the same functionality from the Unix command line.

The PostgreSQL backend must be running for either method to succeed, and the user issuing the command must be the PostgreSQL *superuser* or have been assigned database creation privileges by the superuser.

To create a new database named `mydb` from the command line, type

```
% createdb mydb
```

and to do the same from within psql type

```
=> CREATE DATABASE mydb;
```

If you do not have the privileges required to create a database, you will see the following:

```
ERROR:  CREATE DATABASE: Permission denied.
```

You automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 31 characters in length. PostgreSQL allows you to create any number of databases at a given site.

The *Administrator's Guide* discusses database creation in more detail, including advanced options of the **CREATE DATABASE** command.

10.2. Accessing a Database

Once you have constructed a database, you can access it by:

- Running the PostgreSQL interactive terminal program, called psql, which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing graphical frontend tool like PgAccess or ApplixWare (via ODBC) to create and manipulate a database. These possibilities are not covered in this tutorial.
- Writing a custom application, using one of the several available language bindings. These possibilities are discussed further in *The PostgreSQL Programmer's Guide*.

You probably want to start up psql, to try out the examples in this manual. It can be activated for the mydb database by typing the command:

```
% psql mydb
```

You will be greeted with the following message:

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

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

mydb=>
```

This prompt indicates that **psql** is listening to you and that you can type SQL queries into a work space maintained by the terminal monitor. The psql program itself responds to special commands that begin with the backslash character, `\`. For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

```
mydb=> \h
```

Once you have finished entering your queries into the work space, you can pass the contents of the work space to the PostgreSQL server by typing:

```
mydb=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the `\g` is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, say `myFile`, instead of entering them interactively, type:

```
mydb=> \i myFile
```

To get out of psql and return to Unix, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type **\?** at the **psql** prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by `--`. Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by `/* ... */`.

10.3. Destroying a Database

If you are the owner of the database mydb, you can destroy it using the SQL command

```
=> DROP DATABASE mydb;
```

or the Unix shell script

```
% dropdb mydb
```

This action physically removes all of the Unix files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

Chapter 11. Performance Tips

Query performance can be affected by many things. Some of these can be manipulated by the user, while others are fundamental to the underlying design of the system. This chapter provides some hints about understanding and tuning PostgreSQL performance.

11.1. Using EXPLAIN

PostgreSQL devises a *query plan* for each query it is given. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance. You can use the **EXPLAIN** command to see what query plan the system creates for any query. Plan-reading is an art that deserves an extensive tutorial, which this is not; but here is some basic information.

The numbers that are currently quoted by **EXPLAIN** are:

- Estimated start-up cost (time expended before output scan can start, e.g., time to do the sorting in a SORT node).
- Estimated total cost (if all tuples are retrieved, which they may not be --- a query with a LIMIT will stop short of paying the total cost, for example).
- Estimated number of rows output by this plan node (again, without regard for any LIMIT).
- Estimated average width (in bytes) of rows output by this plan node.

The costs are measured in units of disk page fetches. (CPU effort estimates are converted into disk-page units using some fairly arbitrary fudge-factors. If you want to experiment with these factors, see the list of run-time configuration parameters in the *Administrator's Guide*.)

It's important to note that the cost of an upper-level node includes the cost of all its child nodes. It's also important to realize that the cost only reflects things that the planner/optimizer cares about. In particular, the cost does not consider the time spent transmitting result tuples to the frontend --- which could be a pretty dominant factor in the true elapsed time, but the planner ignores it because it cannot change it by altering the plan. (Every correct plan will output the same tuple set, we trust.)

Rows output is a little tricky because it is *not* the number of rows processed/scanned by the query --- it is usually less, reflecting the estimated selectivity of any WHERE-clause constraints that are being applied at this node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

Here are some examples (using the regress test database after a vacuum analyze, and 7.2 development sources):

```
regression=# EXPLAIN SELECT * FROM tenk1;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

This is about as straightforward as it gets. If you do

```
SELECT * FROM pg_class WHERE relname = 'tenk1';
```

you will find out that `tenk1` has 233 disk pages and 10000 tuples. So the cost is estimated at 233 page reads, defined as 1.0 apiece, plus 10000 * `cpu_tuple_cost` which is currently 0.01 (try **show cpu_tuple_cost**).

Now let's modify the query to add a qualification clause:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..358.00 rows=1007 width=148)
```

The estimate of output rows has gone down because of the WHERE clause. However, the scan will still have to visit all 10000 rows, so the cost hasn't decreased; in fact it has gone up a bit to reflect the extra CPU time spent checking the WHERE condition.

The actual number of rows this query would select is 1000, but the estimate is only approximate. If you try to duplicate this experiment, you will probably get a slightly different estimate; moreover, it will change after each **ANALYZE** command, because the statistics produced by **ANALYZE** are taken from a randomized sample of the table.

Modify the query to restrict the qualification even more:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50;
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..181.09 rows=49 width=148)
```

and you will see that if we make the WHERE condition selective enough, the planner will eventually decide that an index scan is cheaper than a sequential scan. This plan will only have to visit 50 tuples because of the index, so it wins despite the fact that each individual fetch is more expensive than reading a whole disk page sequentially.

Add another condition to the qualification:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50 AND
regression-# stringu1 = 'xxx';
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..181.22 rows=1 width=148)
```

The added clause `stringu1 = 'xxx'` reduces the output-rows estimate, but not the cost because we still have to visit the same set of tuples.

Let's try joining two tables, using the fields we have been discussing:

```
regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50
regression-# AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..330.41 rows=49 width=296)
  ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..181.09 rows=49 width=148)
  ->  Index Scan using tenk2_unique2 on tenk2 t2
              (cost=0.00..3.01 rows=1 width=148)
```

In this nested-loop join, the outer scan is the same index scan we had in the example before last, and so its cost and row count are the same because we are applying the `unique1 < 50` WHERE clause at that node. The `t1.unique2 = t2.unique2` clause is not relevant yet, so it doesn't affect row count of the outer scan. For the inner scan, the unique2 value of the current outer-scan tuple is plugged into the inner index scan to produce an index qualification like `t2.unique2 = constant`. So we get the same inner-scan plan and costs that we'd get from, say, `explain select * from tenk2 where unique2 = 42`. The costs of the loop node are then set on the basis of the cost of the outer scan, plus one repetition of the inner scan for each outer tuple (49 * 3.01, here), plus a little CPU time for join processing.

In this example the loop's output row count is the same as the product of the two scans' row counts, but that's not true in general, because in general you can have WHERE clauses that mention both relations and so can only be applied at the join point, not to either input scan. For example, if we added `WHERE ... AND t1.hundred < t2.hundred`, that would decrease the output row count of the join node, but not change either input scan.

One way to look at variant plans is to force the planner to disregard whatever strategy it thought was the winner, using the enable/disable flags for each plan type. (This is a crude tool, but useful. See also Section 11.3.)

```
regression=# set enable_nestloop = off;
SET VARIABLE
regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50
regression-# AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Hash Join  (cost=181.22..564.83 rows=49 width=296)
  ->  Seq Scan on tenk2 t2
              (cost=0.00..333.00 rows=10000 width=148)
  ->  Hash  (cost=181.09..181.09 rows=49 width=148)
        ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..181.09 rows=49 width=148)
```

This plan proposes to extract the 50 interesting rows of `tenk1` using ye same olde index scan, stash them into an in-memory hash table, and then do a sequential scan of `tenk2`, probing into the hash table for possible matches of `t1.unique2 = t2.unique2` at each `tenk2` tuple. The cost to read `tenk1` and set up the hash table is entirely start-up cost for the hash join, since we won't get any tuples out until we can start reading `tenk2`. The total time estimate for the join also includes a hefty charge for CPU time to probe the hash table 10000 times. Note, however, that we are NOT charging 10000 times 181.09; the hash table setup is only done once in this plan type.

It is possible to check on the accuracy of the planner's estimated costs by using EXPLAIN ANALYZE. This command actually executes the query, and then displays the true runtime accumulated within each plan node along with the same estimated costs that a plain EXPLAIN shows. For example, we might get a result like this:

```
regression=# EXPLAIN ANALYZE
regression-# SELECT * FROM tenk1 t1, tenk2 t2
regression-# WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..330.41 rows=49 width=296) (actual time=1.31..28.90 rows=50 loop
  ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..181.09 rows=49 width=148) (actual time=0.69..8.84 rows=50 loo
```

```
  ->  Index Scan using tenk2_unique2 on tenk2 t2
              (cost=0.00..3.01 rows=1 width=148) (actual time=0.28..0.31 rows=1 loops=5
Total runtime: 30.67 msec
```

Note that the “actual time” values are in milliseconds of real time, whereas the “cost” estimates are expressed in arbitrary units of disk fetches; so they are unlikely to match up. The thing to pay attention to is the ratios.

In some query plans, it is possible for a subplan node to be executed more than once. For example, the inner index scan is executed once per outer tuple in the above nested-loop plan. In such cases, the “loops” value reports the total number of executions of the node, and the actual time and rows values shown are averages per-execution. This is done to make the numbers comparable with the way that the cost estimates are shown. Multiply by the “loops” value to get the total time actually spent in the node.

The “total runtime” shown by EXPLAIN ANALYZE includes executor startup and shutdown time, as well as time spent processing the result tuples. It does not include parsing, rewriting, or planning time. For a SELECT query, the total runtime will normally be just a little larger than the total time reported for the top-level plan node. For INSERT, UPDATE, and DELETE queries, the total runtime may be considerably larger, because it includes the time spent processing the output tuples. In these queries, the time for the top plan node essentially is the time spent computing the new tuples and/or locating the old ones, but it doesn’t include the time spent making the changes.

It is worth noting that EXPLAIN results should not be extrapolated to situations other than the one you are actually testing; for example, results on a toy-sized table can’t be assumed to apply to large tables. The planner’s cost estimates are not linear and so it may well choose a different plan for a larger or smaller table. An extreme example is that on a table that only occupies one disk page, you’ll nearly always get a sequential scan plan whether indexes are available or not. The planner realizes that it’s going to take one disk page read to process the table in any case, so there’s no value in expending additional page reads to look at an index.

11.2. Statistics used by the Planner

As we saw in the previous section, the query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans. This section provides a quick look at the statistics that the system uses for these estimates.

One component of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in `pg_class`’s `reltuples` and `relpages` columns. We can look at it with queries similar to this one:

```
regression=# select relname, relkind, reltuples, relpages from pg_class
regression-# 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)
```

Here we can see that `tenk1` contains 10000 rows, as do its indexes, but the indexes are (unsurprisingly) much smaller than the table.

For efficiency reasons, `reltuples` and `relpages` are not updated on-the-fly, and so they usually contain only approximate values (which is good enough for the planner's purposes). They are initialized with dummy values (presently 1000 and 10 respectively) when a table is created. They are updated by certain commands, presently **VACUUM**, **ANALYZE**, and **CREATE INDEX**. A standalone **ANALYZE**, that is one not part of **VACUUM**, generates an approximate `reltuples` value since it does not read every row of the table.

Most queries retrieve only a fraction of the rows in a table, due to having WHERE clauses that restrict the rows to be examined. The planner thus needs to make an estimate of the *selectivity* of WHERE clauses, that is, the fraction of rows that match each clause of the WHERE condition. The information used for this task is stored in the `pg_statistic` system catalog. Entries in `pg_statistic` are updated by **ANALYZE** and **VACUUM ANALYZE** commands, and are always approximate even when freshly updated.

Rather than look at `pg_statistic` directly, it's better to look at its view `pg_stats` when examining the statistics manually. `pg_stats` is designed to be more easily readable. Furthermore, `pg_stats` is readable by all, whereas `pg_statistic` is only readable by the superuser. (This prevents unprivileged users from learning something about the contents of other people's tables from the statistics. The `pg_stats` view is restricted to show only rows about tables that the current user can read.) For example, we might do:

```
regression=# select attname, n_distinct, most_common_vals from pg_stats where table-
name = 'road';
 attname | n_distinct |
---------+------------+-----------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
---------------------
 name    |  -0.467008 | {"I- 580                        Ramp","I- 880
road                        ","I- 580                          ","I- 680
80                          Ramp","14th                        St  ","5th
sion                        Blvd","I- 880                          "}
 thepath |         20 | {"[(-122.089,37.71),(-122.0886,37.711)]"}
(2 rows)
regression=#
```

As of PostgreSQL 7.2 the following columns exist in `pg_stats`:

Table 11-1. `pg_stats` Columns

Name	Type	Description
`tablename`	`name`	Name of table containing column
`attname`	`name`	Column described by this row
`null_frac`	`real`	Fraction of column's entries that are NULL
`avg_width`	`integer`	Average width in bytes of column's entries

Name	Type	Description
`n_distinct`	`real`	If greater than zero, the estimated number of distinct values in the column. If less than zero, the negative of the number of distinct values divided by the number of rows. (The negated form is used when ANALYZE believes that the number of distinct values is likely to increase as the table grows; the positive form is used when the column seems to have a fixed number of possible values.) For example, -1 indicates a unique column in which the number of distinct values is the same as the number of rows.
`most_common_vals`	`text[]`	A list of the most common values in the column. (Omitted if no values seem to be more common than any others.)
`most_common_freqs`	`real[]`	A list of the frequencies of the most common values, ie, number of occurrences of each divided by total number of rows.
histogram_bounds	`text[]`	A list of values that divide the column's values into groups of approximately equal population. The `most_common_vals`, if present, are omitted from the histogram calculation. (Omitted if column data type does not have a < operator, or if the `most_common_vals` list accounts for the entire population.)
correlation	`real`	Statistical correlation between physical row ordering and logical ordering of the column values. This ranges from -1 to +1. When the value is near -1 or +1, an index scan on the column will be estimated to be cheaper than when it is near zero, due to reduction of random access to the disk. (Omitted if column data type does not have a < operator.)

The maximum number of entries in the `most_common_vals` and `histogram_bounds` arrays can be set on a column-by-column basis using the **ALTER TABLE SET STATISTICS** command. The default limit is presently 10 entries. Raising the limit may allow more accurate planner estimates to be made, particularly for columns with irregular data distributions, at the price of consuming more space in `pg_statistic` and slightly more time to compute the estimates. Conversely, a lower limit may be appropriate for columns with simple data distributions.

11.3. Controlling the Planner with Explicit JOINs

Beginning with PostgreSQL 7.1 it is possible to control the query planner to some extent by using explicit JOIN syntax. To see why this matters, we first need some background.

In a simple join query, such as

```
SELECT * FROM a,b,c WHERE a.id = b.id AND b.ref = c.id;
```

the planner is free to join the given tables in any order. For example, it could generate a query plan that joins A to B, using the WHERE clause a.id = b.id, and then joins C to this joined table, using the other WHERE clause. Or it could join B to C and then join A to that result. Or it could join A to C and then join them with B --- but that would be inefficient, since the full Cartesian product of A and C would have to be formed, there being no applicable WHERE clause to allow optimization of the join. (All joins in the PostgreSQL executor happen between two input tables, so it's necessary to build up the result in one or another of these fashions.) The important point is that these different join possibilities give semantically equivalent results but may have hugely different execution costs. Therefore, the planner will explore all of them to try to find the most efficient query plan.

When a query only involves two or three tables, there aren't many join orders to worry about. But the number of possible join orders grows exponentially as the number of tables expands. Beyond ten or so input tables it's no longer practical to do an exhaustive search of all the possibilities, and even for six or seven tables planning may take an annoyingly long time. When there are too many input tables, the PostgreSQL planner will switch from exhaustive search to a *genetic* probabilistic search through a limited number of possibilities. (The switch-over threshold is set by the `GEQO_THRESHOLD` run-time parameter described in the *Administrator's Guide*.) The genetic search takes less time, but it won't necessarily find the best possible plan.

When the query involves outer joins, the planner has much less freedom than it does for plain (inner) joins. For example, consider

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Although this query's restrictions are superficially similar to the previous example, the semantics are different because a row must be emitted for each row of A that has no matching row in the join of B and C. Therefore the planner has no choice of join order here: it must join B to C and then join A to that result. Accordingly, this query takes less time to plan than the previous query.

In PostgreSQL 7.1, the planner treats all explicit JOIN syntaxes as constraining the join order, even though it is not logically necessary to make such a constraint for inner joins. Therefore, although all of these queries give the same result:

```
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);
```

the second and third take less time to plan than the first. This effect is not worth worrying about for only three tables, but it can be a lifesaver with many tables.

You do not need to constrain the join order completely in order to cut search time, because it's OK to use JOIN operators in a plain FROM list. For example,

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

forces the planner to join A to B before joining them to other tables, but doesn't constrain its choices otherwise. In this example, the number of possible join orders is reduced by a factor of 5.

If you have a mix of outer and inner joins in a complex query, you might not want to constrain the planner's search for a good ordering of inner joins inside an outer join. You can't do that directly in the JOIN syntax, but you can get around the syntactic limitation by using subselects. For example,

```
SELECT * FROM d LEFT JOIN
        (SELECT * FROM a, b, c WHERE ...) AS ss
        ON (...);
```

Here, joining D must be the last step in the query plan, but the planner is free to consider various join orders for A,B,C.

Constraining the planner's search in this way is a useful technique both for reducing planning time and for directing the planner to a good query plan. If the planner chooses a bad join order by default, you can force it to choose a better order via JOIN syntax --- assuming that you know of a better order, that is. Experimentation is recommended.

11.4. Populating a Database

One may need to do a large number of table insertions when first populating a database. Here are some tips and techniques for making that as efficient as possible.

11.4.1. Disable Autocommit

Turn off autocommit and just do one commit at the end. (In plain SQL, this means issuing **BEGIN** at the start and **COMMIT** at the end. Some client libraries may do this behind your back, in which case you need to make sure the library does it when you want it done.) If you allow each insertion to be committed separately, PostgreSQL is doing a lot of work for each record added.

11.4.2. Use COPY FROM

Use **COPY FROM STDIN** to load all the records in one command, instead of using a series of **INSERT** commands. This reduces parsing, planning, etc. overhead a great deal. If you do this then it is not necessary to turn off autocommit, since it is only one command anyway.

11.4.3. Remove Indexes

If you are loading a freshly created table, the fastest way is to create the table, bulk-load with **COPY**, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each record is loaded.

If you are augmenting an existing table, you can **DROP INDEX**, load the table, then recreate the index. Of course, the database performance for other users may be adversely affected during the time that the index is missing. One should also think twice before dropping unique indexes, since the error checking afforded by the unique constraint will be lost while the index is missing.

11.4.4. ANALYZE Afterwards

It's a good idea to run **ANALYZE** or **VACUUM ANALYZE** anytime you've added or updated a lot of data, including just after initially populating a table. This ensures that the planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner may make poor choices of query plans, leading to bad performance on queries that use your table.

Appendix A. Date/Time Support

PostgreSQL uses an internal heuristic parser for all date/time support. Dates and times are input as strings, and are broken up into distinct fields with a preliminary determination of what kind of information may be in the field. Each field is interpreted and either assigned a numeric value, ignored, or rejected. The parser contains internal lookup tables for all textual fields, including months, days of the week, and time zones.

This appendix includes information on the content of these lookup tables and describes the steps used by the parser to decode dates and times.

A.1. Date/Time Keywords

Table A-1. Month Abbreviations

Month	Abbreviations
April	Apr
August	Aug
December	Dec
February	Feb
January	Jan
July	Jul
June	Jun
March	Mar
November	Nov
October	Oct
September	Sep, Sept

Note: The month `May` has no explicit abbreviation, for obvious reasons.

Table A-2. Day of the Week Abbreviations

Day	Abbreviation
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

Table A-3. PostgreSQL Field Modifiers

Identifier	Description
ABSTIME	Keyword ignored
AM	Time is before 12:00
AT	Keyword ignored
JULIAN, JD, J	Next field is Julian Day
ON	Keyword ignored
PM	Time is on or after after 12:00
T	Next field is time

The keyword `ABSTIME` is ignored for historical reasons; in very old releases of PostgreSQL invalid `ABSTIME` fields were emitted as "Invalid Abstime". This is no longer the case however and this keyword will likely be dropped in a future release.

A.2. Time Zones

PostgreSQL contains internal tabular information for time zone decoding, since there is no *nix standard system interface to provide access to general, cross-timezone information. The underlying OS *is* used to provide time zone information for *output*, however.

The following table of time zones recognized by PostgreSQL is organized by time zone offset from UTC, rather than alphabetically; this is intended to facilitate matching local usage with recognized abbreviations for cases where these might differ.

Table A-4. PostgreSQL Recognized Time Zones

Time Zone	Offset from UTC	Description
NZDT	+13:00	New Zealand Daylight 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 Savings Time
SADT	+10:30	South Australian Daylight Time
AEST	+10:00	Australia Eastern Standard Time
EAST	+10:00	East Australian Standard Time
GST	+10:00	Guam Standard Time, USSR Zone 9
LIGT	+10:00	Melbourne, Australia
SAST	+09:30	South Australia Standard Time

Time Zone	Offset from UTC	Description
CAST	+09:30	Central Australia Standard Time
AWSST	+09:00	Australia Western Summer Standard Time
JST	+09:00	Japan Standard Time,USSR Zone 8
KST	+09:00	Korea Standard Time
MHT	+09:00	Kwajalein Time
WDT	+09:00	West Australian Daylight 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 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
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	Afganistan Time
EAST	+04:00	Antananarivo Savings Time
MUT	+04:00	Mauritius Island Time
RET	+04:00	Reunion Island Time
SCT	+04:00	Mahe Island Time
IT	+03:30	Iran Time
EAT	+03:00	Antananarivo, Comoro Time
BT	+03:00	Baghdad Time
EETDST	+03:00	Eastern Europe Daylight Savings Time
HMT	+03:00	Hellas Mediterranean Time (?)
BDST	+02:00	British Double Standard Time
CEST	+02:00	Central European Savings Time
CETDST	+02:00	Central European Daylight Savings Time
EET	+02:00	Eastern Europe, USSR Zone 1
FWT	+02:00	French Winter Time
IST	+02:00	Israel Standard Time
MEST	+02:00	Middle Europe Summer Time

Time Zone	Offset from UTC	Description
METDST	+02:00	Middle Europe Daylight 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 Europe Time
MEWT	+01:00	Middle Europe Winter Time
MEZ	+01:00	Middle Europe Zone
NOR	+01:00	Norway Standard Time
SET	+01:00	Seychelles Time
SWT	+01:00	Swedish Winter Time
WETDST	+01:00	Western Europe Daylight Savings Time
GMT	+00:00	Greenwich Mean Time
UT	+00:00	Universal Time
UTC	+00:00	Universal Time, Coordinated
Z	+00:00	Same as UTC
ZULU	+00:00	Same as UTC
WET	+00:00	Western Europe
WAT	-01:00	West Africa Time
NDT	-02:30	Newfoundland Daylight Time
ADT	-03:00	Atlantic Daylight Time
AWT	-03:00	(unknown)
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
ACT	-05:00	Atlantic/Porto Acre Standard Time
EDT	-04:00	Eastern Daylight Time
CDT	-05:00	Central Daylight Time
EST	-05:00	Eastern Standard Time
CST	-06:00	Central Standard Time
MDT	-06:00	Mountain Daylight Time
MST	-07:00	Mountain Standard Time
PDT	-07:00	Pacific Daylight Time
AKDT	-08:00	Alaska Daylight Time
PST	-08:00	Pacific Standard Time
YDT	-08:00	Yukon Daylight Time

Time Zone	Offset from UTC	Description
AKST	-09:00	Alaska Standard Time
HDT	-09:00	Hawaii/Alaska Daylight Time
YST	-09:00	Yukon Standard 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

A.2.1. Australian Time Zones

Australian time zones and their naming variants account for fully one quarter of all time zones in the PostgreSQL time zone lookup table. There are two naming conflicts with time zones commonly used in the United States, `CST` and `EST`.

If the runtime option `AUSTRALIAN_TIMEZONES` is set then `CST`, `EST`, and `SAT` will be interpreted as Australian timezone names. Without this option, `CST` and `EST` are taken as American timezone names, while `SAT` is interpreted as a noise word indicating `Saturday`.

Table A-5. PostgreSQL Australian Time Zones

Time Zone	Offset from UTC	Description
ACST	+09:30	Central Australia Standard Time
CST	+10:30	Australian Central Standard Time
EST	+10:00	Australian Eastern Standard Time
SAT	+09:30	South Australian Standard Time

A.2.2. Date/Time Input Interpretation

The date/time types are all decoded using a common set of routines.

Date/Time Input Interpretation

1. Break the input string into tokens and categorize each token as a string, time, time zone, or number.

    a. If the numeric token contains a colon (":"), this is a time string. Include all subsequent digits and colons.

    b. If the numeric token contains a dash ("-"), slash ("/"), or two or more dots ("."), this is a date string which may have a text month.

c. If the token is numeric only, then it is either a single field or an ISO-8601 concatenated date (e.g. `19990113` for January 13, 1999) or time (e.g. 141516 for 14:15:16).

d. If the token starts with a plus ("+") or minus ("-"), then it is either a time zone or a special field.

2. If the token is a text string, match up with possible strings.

   a. Do a binary-search table lookup for the token as either a special string (e.g. `today`), day (e.g. `Thursday`), month (e.g. `January`), or noise word (e.g. `at`, `on`).

      Set field values and bit mask for fields. For example, set year, month, day for `today`, and additionally hour, minute, second for `now`.

   b. If not found, do a similar binary-search table lookup to match the token with a time zone.

   c. If not found, throw an error.

3. The token is a number or number field.

   a. If there are more than 4 digits, and if no other date fields have been previously read, then interpret as a "concatenated date" (e.g. `19990118`). 8 and 6 digits are interpreted as year, month, and day, while 7 and 5 digits are interpreted as year, day of year, respectively.

   b. If the token is three digits and a year has already been decoded, then interpret as day of year.

   c. If four or six digits and a year has already been read, then interpret as a time.

   d. If four or more digits, then interpret as a year.

   e. If in European date mode, and if the day field has not yet been read, and if the value is less than or equal to 31, then interpret as a day.

   f. If the month field has not yet been read, and if the value is less than or equal to 12, then interpret as a month.

   g. If the day field has not yet been read, and if the value is less than or equal to 31, then interpret as a day.

   h. If two digits or four or more digits, then interpret as a year.

   i. Otherwise, throw an error.

4. If BC has been specified, negate the year and add one for internal storage (there is no year zero in the Gregorian calendar, so numerically `1BC` becomes year zero).

5. If BC was not specified, and if the year field was two digits in length, then adjust the year to 4 digits. If the field was less than 70, then add 2000; otherwise, add 1900.

Tip: Gregorian years 1-99AD may be entered by using 4 digits with leading zeros (e.g. 0099 is 99AD). Previous versions of PostgreSQL accepted years with three digits and with single digits, but as of version 7.0 the rules have been tightened up to reduce the possibility of ambiguity.

A.3. History of Units

> **Note:** Contributed by José Soares (<jose@sferacarta.com>)

The Julian Day was invented by the French scholar Joseph Justus Scaliger (1540-1609) and probably takes its name from the Scaliger's father, the Italian scholar Julius Caesar Scaliger (1484-1558). Astronomers have used the Julian period to assign a unique number to every day since 1 January 4713 BC. This is the so-called Julian Day (JD). JD 0 designates the 24 hours from noon UTC on 1 January 4713 BC to noon UTC on 2 January 4713 BC.

"Julian Day" is different from "Julian Date". The Julian calendar was introduced by Julius Caesar in 45 BC. It was in common use until the 1582, when countries started changing to the Gregorian calendar. In the Julian calendar, the tropical year is approximated as 365 1/4 days = 365.25 days. This gives an error of about 1 day in 128 years. The accumulating calendar error prompted Pope Gregory XIII to reform the calendar in accordance with instructions from the Council of Trent.

In the Gregorian calendar, the tropical year is approximated as 365 + 97 / 400 days = 365.2425 days. Thus it takes approximately 3300 years for the tropical year to shift one day with respect to the Gregorian calendar.

The approximation 365+97/400 is achieved by having 97 leap years every 400 years, using the following rules:

Every year divisible by 4 is a leap year.
However, every year divisible by 100 is not a leap year.
However, every year divisible by 400 is a leap year after all.

So, 1700, 1800, 1900, 2100, and 2200 are not leap years. But 1600, 2000, and 2400 are leap years. By contrast, in the older Julian calendar only years divisible by 4 are leap years.

The papal bull of February 1582 decreed that 10 days should be dropped from October 1582 so that 15 October should follow immediately after 4 October. This was observed in Italy, Poland, Portugal, and Spain. Other Catholic countries followed shortly after, but Protestant countries were reluctant to change, and the Greek orthodox countries didn't change until the start of this century. The reform was observed by Great Britain and Dominions (including what is now the USA) in 1752. Thus 2 Sep 1752 was followed by 14 Sep 1752. This is why Unix systems have cal produce the following:

```
% cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

> **Note:** SQL92 states that "Within the definition of a 'datetime literal', the 'datetime value's are constrained by the natural rules for dates and times according to the Gregorian calendar". Dates between 1752-09-03 and 1752-09-13, although eliminated in some countries by Papal fiat, conform to "natural rules" and are hence valid dates.

Different calendars have been developed in various parts of the world, many predating the Gregorian system. For example, the beginnings of the Chinese calendar can be traced back to the 14th century

BC. Legend has it that the Emperor Huangdi invented the calendar in 2637 BC. The People's Republic of China uses the Gregorian calendar for civil purposes. Chinese calendar is used for determining festivals.

Appendix B. SQL Key Words

Table B-1 lists all tokens that are key words in the SQL standard and in PostgreSQL 7.2. Background information can be found in Section 1.1.1.

SQL distinguishes between *reserved* and *non-reserved* key words. According to the standard, reserved key words are the only real key words; they are never allowed as identifiers. Non-reserved key words only have a special meaning in particular contexts and can be used as identifiers in other contexts. Most non-reserved key words are actually the names of built-in tables and functions specified by SQL. The concept of non-reserved key words essentially only exists to declare that some predefined meaning is attached to a word in some contexts.

In the PostgreSQL parser life is a bit more complicated. There are several different classes of tokens ranging from those that can never be used as an identifier to those that have absolutely no special status in the parser as compared to an ordinary identifier. (The latter is usually the case for functions specified by SQL.) Even reserved key words are not completely reserved in PostgreSQL, but can be used as column labels (for example, `SELECT 55 AS CHECK`, even though CHECK is a reserved key word).

In Table B-1 in the column for PostgreSQL we classify as "non-reserved" those key words that are explicitly known to the parser but are allowed in most or all contexts where an identifier is expected. Some key words that are otherwise non-reserved cannot be used as function or data type names and are marked accordingly. (Most of these words represent built-in functions or data types with special syntax. The function or type is still available but it cannot be redefined by the user.) Labeled "reserved" are those tokens that are only allowed as "AS" column label names (and perhaps in very few other contexts). Some reserved key words are allowable as names for functions; this is also shown in the table.

As a general rule, if you get spurious parser errors for commands that contain any of the listed key words as an identifier you should try to quote the identifier to see if the problem goes away.

It is important to understand before studying Table B-1 that the fact that a key word is not reserved in PostgreSQL does not mean that the feature related to the word is not implemented. Conversely, the presence of a key word does not indicate the existence of a feature.

Table B-1. SQL Key Words

Key Word	PostgreSQL	SQL 99	SQL 92
ABORT	non-reserved		
ABS		non-reserved	
ABSOLUTE	non-reserved	reserved	reserved
ACCESS	non-reserved		
ACTION	non-reserved	reserved	reserved
ADA		non-reserved	non-reserved
ADD	non-reserved	reserved	reserved
ADMIN		reserved	
AFTER	non-reserved	reserved	
AGGREGATE	non-reserved	reserved	
ALIAS		reserved	
ALL	reserved	reserved	reserved
ALLOCATE		reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
ALTER	non-reserved	reserved	reserved
ANALYSE	reserved		
ANALYZE	reserved		
AND	reserved	reserved	reserved
ANY	reserved	reserved	reserved
ARE		reserved	reserved
ARRAY		reserved	
AS	reserved	reserved	reserved
ASC	reserved	reserved	reserved
ASENSITIVE		non-reserved	
ASSERTION		reserved	reserved
ASSIGNMENT		non-reserved	
ASYMMETRIC		non-reserved	
AT	non-reserved	reserved	reserved
ATOMIC		non-reserved	
AUTHORIZATION	non-reserved	reserved	reserved
AVG		non-reserved	reserved
BACKWARD	non-reserved		
BEFORE	non-reserved	reserved	
BEGIN	non-reserved	reserved	reserved
BETWEEN	reserved (can be function)	non-reserved	reserved
BINARY	reserved (can be function)	reserved	
BIT	non-reserved (cannot be function or type)	reserved	reserved
BITVAR		non-reserved	
BIT_LENGTH		non-reserved	reserved
BLOB		reserved	
BOOLEAN		reserved	
BOTH	reserved	reserved	reserved
BREADTH		reserved	
BY	non-reserved	reserved	reserved
C		non-reserved	non-reserved
CACHE	non-reserved		
CALL		reserved	
CALLED		non-reserved	
CARDINALITY		non-reserved	
CASCADE	non-reserved	reserved	reserved
CASCADED		reserved	reserved
CASE	reserved	reserved	reserved
CAST	reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
CATALOG		reserved	reserved
CATALOG_NAME		non-reserved	non-reserved
CHAIN	non-reserved	non-reserved	
CHAR	non-reserved (cannot be function or type)	reserved	reserved
CHARACTER	non-reserved (cannot be function or type)	reserved	reserved
CHARACTERISTICS	non-reserved		
CHARACTER_LENGTH		non-reserved	reserved
CHARACTER_SET_CATALOG		non-reserved	non-reserved
CHARACTER_SET_NAME		non-reserved	non-reserved
CHARACTER_SET_SCHEMA		non-reserved	non-reserved
CHAR_LENGTH		non-reserved	reserved
CHECK	reserved	reserved	reserved
CHECKED		non-reserved	
CHECKPOINT	non-reserved		
CLASS		reserved	
CLASS_ORIGIN		non-reserved	non-reserved
CLOB		reserved	
CLOSE	non-reserved	reserved	reserved
CLUSTER	non-reserved		
COALESCE	non-reserved (cannot be function or type)	non-reserved	reserved
COBOL		non-reserved	non-reserved
COLLATE	reserved	reserved	reserved
COLLATION		reserved	reserved
COLLATION_CATALOG		non-reserved	non-reserved
COLLATION_NAME		non-reserved	non-reserved
COLLATION_SCHEMA		non-reserved	non-reserved
COLUMN	reserved	reserved	reserved
COLUMN_NAME		non-reserved	non-reserved
COMMAND_FUNCTION		non-reserved	non-reserved
COMMAND_FUNCTION_CODE		non-reserved	
COMMENT	non-reserved		
COMMIT	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
COMMITTED	non-reserved	non-reserved	non-reserved
COMPLETION		reserved	
CONDITION_NUMBER		non-reserved	non-reserved
CONNECT		reserved	reserved
CONNECTION		reserved	reserved
CONNECTION_NAME		non-reserved	non-reserved
CONSTRAINT	reserved	reserved	reserved
CONSTRAINTS	non-reserved	reserved	reserved
CONSTRAINT_CATALOG		non-reserved	non-reserved
CONSTRAINT_NAME		non-reserved	non-reserved
CONSTRAINT_SCHEMA		non-reserved	non-reserved
CONSTRUCTOR		reserved	
CONTAINS		non-reserved	
CONTINUE		reserved	reserved
CONVERT		non-reserved	reserved
COPY	non-reserved		
CORRESPONDING		reserved	reserved
COUNT		non-reserved	reserved
CREATE	non-reserved	reserved	reserved
CREATEDB	non-reserved		
CREATEUSER	non-reserved		
CROSS	reserved (can be function)	reserved	reserved
CUBE		reserved	
CURRENT		reserved	reserved
CURRENT_DATE	reserved	reserved	reserved
CURRENT_PATH		reserved	
CURRENT_ROLE		reserved	
CURRENT_TIME	reserved	reserved	reserved
CURRENT_TIMESTAMP	reserved	reserved	reserved
CURRENT_USER	reserved	reserved	reserved
CURSOR	non-reserved	reserved	reserved
CURSOR_NAME		non-reserved	non-reserved
CYCLE	non-reserved	reserved	
DATA		reserved	non-reserved
DATABASE	non-reserved		
DATE		reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
DATETIME_INTERVAL_CODE		non-reserved	non-reserved
DATETIME_INTERVAL_PRECISION		non-reserved	non-reserved
DAY	non-reserved	reserved	reserved
DEALLOCATE		reserved	reserved
DEC	non-reserved (cannot be function or type)	reserved	reserved
DECIMAL	non-reserved (cannot be function or type)	reserved	reserved
DECLARE	non-reserved	reserved	reserved
DEFAULT	reserved	reserved	reserved
DEFERRABLE	reserved	reserved	reserved
DEFERRED	non-reserved	reserved	reserved
DEFINED		non-reserved	
DEFINER		non-reserved	
DELETE	non-reserved	reserved	reserved
DELIMITERS	non-reserved		
DEPTH		reserved	
DEREF		reserved	
DESC	reserved	reserved	reserved
DESCRIBE		reserved	reserved
DESCRIPTOR		reserved	reserved
DESTROY		reserved	
DESTRUCTOR		reserved	
DETERMINISTIC		reserved	
DIAGNOSTICS		reserved	reserved
DICTIONARY		reserved	
DISCONNECT		reserved	reserved
DISPATCH		non-reserved	
DISTINCT	reserved	reserved	reserved
DO	reserved		
DOMAIN		reserved	reserved
DOUBLE	non-reserved	reserved	reserved
DROP	non-reserved	reserved	reserved
DYNAMIC		reserved	
DYNAMIC_FUNCTION		non-reserved	non-reserved
DYNAMIC_FUNCTION_CODE		non-reserved	
EACH	non-reserved	reserved	
ELSE	reserved	reserved	reserved
ENCODING	non-reserved		

Key Word	PostgreSQL	SQL 99	SQL 92
ENCRYPTED	non-reserved		
END	reserved	reserved	reserved
END-EXEC		reserved	reserved
EQUALS		reserved	
ESCAPE	non-reserved	reserved	reserved
EVERY		reserved	
EXCEPT	reserved	reserved	reserved
EXCEPTION		reserved	reserved
EXCLUSIVE	non-reserved		
EXEC		reserved	reserved
EXECUTE	non-reserved	reserved	reserved
EXISTING		non-reserved	
EXISTS	non-reserved (cannot be function or type)	non-reserved	reserved
EXPLAIN	non-reserved		
EXTERNAL		reserved	reserved
EXTRACT	non-reserved (cannot be function or type)	non-reserved	reserved
FALSE	reserved	reserved	reserved
FETCH	non-reserved	reserved	reserved
FINAL		non-reserved	
FIRST		reserved	reserved
FLOAT	non-reserved (cannot be function or type)	reserved	reserved
FOR	reserved	reserved	reserved
FORCE	non-reserved		
FOREIGN	reserved	reserved	reserved
FORTRAN		non-reserved	non-reserved
FORWARD	non-reserved		
FOUND		reserved	reserved
FREE		reserved	
FREEZE	reserved (can be function)		
FROM	reserved	reserved	reserved
FULL	reserved (can be function)	reserved	reserved
FUNCTION	non-reserved	reserved	
G		non-reserved	
GENERAL		reserved	
GENERATED		non-reserved	
GET		reserved	reserved
GLOBAL	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
GO		reserved	reserved
GOTO		reserved	reserved
GRANT	non-reserved	reserved	reserved
GRANTED		non-reserved	
GROUP	reserved	reserved	reserved
GROUPING		reserved	
HANDLER	non-reserved		
HAVING	reserved	reserved	reserved
HIERARCHY		non-reserved	
HOLD		non-reserved	
HOST		reserved	
HOUR	non-reserved	reserved	reserved
IDENTITY		reserved	reserved
IGNORE		reserved	
ILIKE	reserved (can be function)		
IMMEDIATE	non-reserved	reserved	reserved
IMPLEMENTATION		non-reserved	
IN	reserved (can be function)	reserved	reserved
INCREMENT	non-reserved		
INDEX	non-reserved		
INDICATOR		reserved	reserved
INFIX		non-reserved	
INHERITS	non-reserved		
INITIALIZE		reserved	
INITIALLY	reserved	reserved	reserved
INNER	reserved (can be function)	reserved	reserved
INOUT	non-reserved	reserved	
INPUT		reserved	reserved
INSENSITIVE	non-reserved	non-reserved	reserved
INSERT	non-reserved	reserved	reserved
INSTANCE		non-reserved	
INSTANTIABLE		non-reserved	
INSTEAD	non-reserved		
INT		reserved	reserved
INTEGER		reserved	reserved
INTERSECT	reserved	reserved	reserved
INTERVAL	non-reserved (cannot be function or type)	reserved	reserved
INTO	reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
INVOKER		non-reserved	
IS	reserved (can be function)	reserved	reserved
ISNULL	reserved (can be function)		
ISOLATION	non-reserved	reserved	reserved
ITERATE		reserved	
JOIN	reserved (can be function)	reserved	reserved
K		non-reserved	
KEY	non-reserved	reserved	reserved
KEY_MEMBER		non-reserved	
KEY_TYPE		non-reserved	
LANCOMPILER	non-reserved		
LANGUAGE	non-reserved	reserved	reserved
LARGE		reserved	
LAST		reserved	reserved
LATERAL		reserved	
LEADING	reserved	reserved	reserved
LEFT	reserved (can be function)	reserved	reserved
LENGTH		non-reserved	non-reserved
LESS		reserved	
LEVEL	non-reserved	reserved	reserved
LIKE	reserved (can be function)	reserved	reserved
LIMIT	reserved	reserved	
LISTEN	non-reserved		
LOAD	non-reserved		
LOCAL	non-reserved	reserved	reserved
LOCALTIME		reserved	
LOCALTIMESTAMP		reserved	
LOCATION	non-reserved		
LOCATOR		reserved	
LOCK	non-reserved		
LOWER		non-reserved	reserved
M		non-reserved	
MAP		reserved	
MATCH	non-reserved	reserved	reserved
MAX		non-reserved	reserved
MAXVALUE	non-reserved		
MESSAGE_LENGTH		non-reserved	non-reserved

Key Word	PostgreSQL	SQL 99	SQL 92
MESSAGE_OCTET_LENGTH		non-reserved	non-reserved
MESSAGE_TEXT		non-reserved	non-reserved
METHOD		non-reserved	
MIN		non-reserved	reserved
MINUTE	non-reserved	reserved	reserved
MINVALUE	non-reserved		
MOD		non-reserved	
MODE	non-reserved		
MODIFIES		reserved	
MODIFY		reserved	
MODULE		reserved	reserved
MONTH	non-reserved	reserved	reserved
MORE		non-reserved	non-reserved
MOVE	non-reserved		
MUMPS		non-reserved	non-reserved
NAME		non-reserved	non-reserved
NAMES	non-reserved	reserved	reserved
NATIONAL	non-reserved	reserved	reserved
NATURAL	reserved (can be function)	reserved	reserved
NCHAR	non-reserved (cannot be function or type)	reserved	reserved
NCLOB		reserved	
NEW	reserved	reserved	
NEXT	non-reserved	reserved	reserved
NO	non-reserved	reserved	reserved
NOCREATEDB	non-reserved		
NOCREATEUSER	non-reserved		
NONE	non-reserved (cannot be function or type)	reserved	
NOT	reserved	reserved	reserved
NOTHING	non-reserved		
NOTIFY	non-reserved		
NOTNULL	reserved (can be function)		
NULL	reserved	reserved	reserved
NULLABLE		non-reserved	non-reserved
NULLIF	non-reserved (cannot be function or type)	non-reserved	reserved
NUMBER		non-reserved	non-reserved
NUMERIC	non-reserved (cannot be function or type)	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
OBJECT		reserved	
OCTET_LENGTH		non-reserved	reserved
OF	non-reserved	reserved	reserved
OFF	reserved	reserved	
OFFSET	reserved		
OIDS	non-reserved		
OLD	reserved	reserved	
ON	reserved	reserved	reserved
ONLY	reserved	reserved	reserved
OPEN		reserved	reserved
OPERATION		reserved	
OPERATOR	non-reserved		
OPTION	non-reserved	reserved	reserved
OPTIONS		non-reserved	
OR	reserved	reserved	reserved
ORDER	reserved	reserved	reserved
ORDINALITY		reserved	
OUT	non-reserved	reserved	
OUTER	reserved (can be function)	reserved	reserved
OUTPUT		reserved	reserved
OVERLAPS	reserved (can be function)	non-reserved	reserved
OVERLAY		non-reserved	
OVERRIDING		non-reserved	
OWNER	non-reserved		
PAD		reserved	reserved
PARAMETER		reserved	
PARAMETERS		reserved	
PARAMETER_MODE		non-reserved	
PARAMETER_NAME		non-reserved	
PARAMETER_ORDINAL_POSITION		non-reserved	
PARAMETER_SPECIFIC_CATALOG		non-reserved	
PARAMETER_SPECIFIC_NAME		non-reserved	
PARAMETER_SPECIFIC_SCHEMA		non-reserved	
PARTIAL	non-reserved	reserved	reserved
PASCAL		non-reserved	non-reserved
PASSWORD	non-reserved		
PATH	non-reserved	reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
PENDANT	non-reserved		
PLI		non-reserved	non-reserved
POSITION	non-reserved (cannot be function or type)	non-reserved	reserved
POSTFIX		reserved	
PRECISION	non-reserved	reserved	reserved
PREFIX		reserved	
PREORDER		reserved	
PREPARE		reserved	reserved
PRESERVE		reserved	reserved
PRIMARY	reserved	reserved	reserved
PRIOR	non-reserved	reserved	reserved
PRIVILEGES	non-reserved	reserved	reserved
PROCEDURAL	non-reserved		
PROCEDURE	non-reserved	reserved	reserved
PUBLIC	reserved (can be function)	reserved	reserved
READ	non-reserved	reserved	reserved
READS		reserved	
REAL		reserved	reserved
RECURSIVE		reserved	
REF		reserved	
REFERENCES	reserved	reserved	reserved
REFERENCING		reserved	
REINDEX	non-reserved		
RELATIVE	non-reserved	reserved	reserved
RENAME	non-reserved		
REPEATABLE		non-reserved	non-reserved
REPLACE	non-reserved		
RESET	non-reserved		
RESTRICT	non-reserved	reserved	reserved
RESULT		reserved	
RETURN		reserved	
RETURNED_LENGTH		non-reserved	non-reserved
RETURNED_OCTET_LENGTH		non-reserved	non-reserved
RETURNED_SQLSTATE		non-reserved	non-reserved
RETURNS	non-reserved	reserved	
REVOKE	non-reserved	reserved	reserved
RIGHT	reserved (can be function)	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
ROLE		reserved	
ROLLBACK	non-reserved	reserved	reserved
ROLLUP		reserved	
ROUTINE		reserved	
ROUTINE_CATALOG		non-reserved	
ROUTINE_NAME		non-reserved	
ROUTINE_SCHEMA		non-reserved	
ROW	non-reserved	reserved	
ROWS		reserved	reserved
ROW_COUNT		non-reserved	non-reserved
RULE	non-reserved		
SAVEPOINT		reserved	
SCALE		non-reserved	non-reserved
SCHEMA	non-reserved	reserved	reserved
SCHEMA_NAME		non-reserved	non-reserved
SCOPE		reserved	
SCROLL	non-reserved	reserved	reserved
SEARCH		reserved	
SECOND	non-reserved	reserved	reserved
SECTION		reserved	reserved
SECURITY		non-reserved	
SELECT	reserved	reserved	reserved
SELF		non-reserved	
SENSITIVE		non-reserved	
SEQUENCE	non-reserved	reserved	
SERIALIZABLE	non-reserved	non-reserved	non-reserved
SERVER_NAME		non-reserved	non-reserved
SESSION	non-reserved	reserved	reserved
SESSION_USER	reserved	reserved	reserved
SET	non-reserved	reserved	reserved
SETOF	non-reserved (cannot be function or type)		
SETS		reserved	
SHARE	non-reserved		
SHOW	non-reserved		
SIMILAR		non-reserved	
SIMPLE		non-reserved	
SIZE		reserved	reserved
SMALLINT		reserved	reserved
SOME	reserved	reserved	reserved
SOURCE		non-reserved	
SPACE		reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
SPECIFIC		reserved	
SPECIFICTYPE		reserved	
SPECIFIC_NAME		non-reserved	
SQL		reserved	reserved
SQLCODE			reserved
SQLERROR			reserved
SQLEXCEPTION		reserved	
SQLSTATE		reserved	reserved
SQLWARNING		reserved	
START	non-reserved	reserved	
STATE		reserved	
STATEMENT	non-reserved	reserved	
STATIC		reserved	
STATISTICS	non-reserved		
STDIN	non-reserved		
STDOUT	non-reserved		
STRUCTURE		reserved	
STYLE		non-reserved	
SUBCLASS_ORIGIN		non-reserved	non-reserved
SUBLIST		non-reserved	
SUBSTRING	non-reserved (cannot be function or type)	non-reserved	reserved
SUM		non-reserved	reserved
SYMMETRIC		non-reserved	
SYSID	non-reserved		
SYSTEM		non-reserved	
SYSTEM_USER		reserved	reserved
TABLE	reserved	reserved	reserved
TABLE_NAME		non-reserved	non-reserved
TEMP	non-reserved		
TEMPLATE	non-reserved		
TEMPORARY	non-reserved	reserved	reserved
TERMINATE		reserved	
THAN		reserved	
THEN	reserved	reserved	reserved
TIME	non-reserved (cannot be function or type)	reserved	reserved
TIMESTAMP	non-reserved (cannot be function or type)	reserved	reserved
TIMEZONE_HOUR		reserved	reserved
TIMEZONE_MINUTE		reserved	reserved
TO	reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
TOAST	non-reserved		
TRAILING	reserved	reserved	reserved
TRANSACTION	non-reserved	reserved	reserved
TRANSACTIONS_COMMITTED		non-reserved	
TRANSACTIONS_ROLLED_BACK		non-reserved	
TRANSACTION_ACTIVE		non-reserved	
TRANSFORM		non-reserved	
TRANSFORMS		non-reserved	
TRANSLATE		non-reserved	reserved
TRANSLATION		reserved	reserved
TREAT		reserved	
TRIGGER	non-reserved	reserved	
TRIGGER_CATALOG		non-reserved	
TRIGGER_NAME		non-reserved	
TRIGGER_SCHEMA		non-reserved	
TRIM	non-reserved (cannot be function or type)	non-reserved	reserved
TRUE	reserved	reserved	reserved
TRUNCATE	non-reserved		
TRUSTED	non-reserved		
TYPE	non-reserved	non-reserved	non-reserved
UNCOMMITTED		non-reserved	non-reserved
UNDER		reserved	
UNENCRYPTED	non-reserved		
UNION	reserved	reserved	reserved
UNIQUE	reserved	reserved	reserved
UNKNOWN	non-reserved	reserved	reserved
UNLISTEN	non-reserved		
UNNAMED		non-reserved	non-reserved
UNNEST		reserved	
UNTIL	non-reserved		
UPDATE	non-reserved	reserved	reserved
UPPER		non-reserved	reserved
USAGE		reserved	reserved
USER	reserved	reserved	reserved
USER_DEFINED_TYPE_CATALOG		non-reserved	
USER_DEFINED_TYPE_NAME		non-reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
USER_DEFINED_TYPE_SCHEMA		non-reserved	
USING	reserved	reserved	reserved
VACUUM	non-reserved		
VALID	non-reserved		
VALUE		reserved	reserved
VALUES	non-reserved	reserved	reserved
VARCHAR	non-reserved (cannot be function or type)	reserved	reserved
VARIABLE		reserved	
VARYING	non-reserved	reserved	reserved
VERBOSE	reserved (can be function)		
VERSION	non-reserved		
VIEW	non-reserved	reserved	reserved
WHEN	reserved	reserved	reserved
WHENEVER		reserved	reserved
WHERE	reserved	reserved	reserved
WITH	non-reserved	reserved	reserved
WITHOUT	non-reserved	reserved	
WORK	non-reserved	reserved	reserved
WRITE		reserved	reserved
YEAR	non-reserved	reserved	reserved
ZONE	non-reserved	reserved	reserved

Bibliography

Selected references and readings for SQL and PostgreSQL.

Some white papers and technical reports from the original POSTGRES development team are available at the University of California, Berkeley, Computer Science Department web site[1]

SQL Reference Books

Judith Bowman, Sandra Emerson, and Marcy Darnovsky, *The Practical SQL Handbook: Using Structured Query Language*, Third Edition, Addison-Wesley, ISBN 0-201-44787-8, 1996.

C. J. Date and 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 and Shamkant Navathe, *Fundamentals of Database Systems*, 3rd Edition, Addison-Wesley, ISBN 0-805-31755-4, August 1999.

Jim Melton and 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.

PostgreSQL-Specific Documentation

Stefan Simkovics, *Enhancement of the ANSI SQL Implementation of PostgreSQL*, Department of Information Systems, Vienna University of Technology, November 29, 1998.

Discusses SQL history and syntax, and describes the addition of `INTERSECT` and `EXCEPT` constructs into PostgreSQL. Prepared as a Master's Thesis with the support of O. Univ. Prof. Dr. Georg Gottlob and Univ. Ass. Mag. Katrin Seyr at Vienna University of Technology.

A. Yu and J. Chen, The POSTGRES Group, *The Postgres95 User Manual*, University of California, Sept. 5, 1995.

Zelaine Fong, *The design and implementation of the POSTGRES query optimizer*[2], University of California, Berkeley, Computer Science Department.

1. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/
2. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/UCB-MS-zfong.pdf

Proceedings and Articles

Nels Olson, *Partial indexing in POSTGRES: research project*, University of California, UCB Engin T7.49.1993 O676, 1993.

L. Ong and J. Goh, "A Unified Framework for Version Modeling Using Production Rules in a Database System", *ERL Technical Memorandum M90/33*, University of California, April, 1990.

L. Rowe and M. Stonebraker, "The POSTGRES data model[3]", Proc. VLDB Conference, Sept. 1987.

P. Seshadri and A. Swami, "Generalized Partial Indexes[4] ", Proc. Eleventh International Conference on Data Engineering, 6-10 March 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, p. 420-7.

M. Stonebraker and L. Rowe, "The design of POSTGRES[5]", Proc. ACM-SIGMOD Conference on Management of Data, May 1986.

M. Stonebraker, E. Hanson, and C. H. Hong, "The design of the POSTGRES rules system", Proc. IEEE Conference on Data Engineering, Feb. 1987.

M. Stonebraker, "The design of the POSTGRES storage system[6]", Proc. VLDB Conference, Sept. 1987.

M. Stonebraker, M. Hearst, and S. Potamianos, "A commentary on the POSTGRES rules system[7]", *SIGMOD Record 18(3)*, Sept. 1989.

M. Stonebraker, "The case for partial indexes[8]", *SIGMOD Record 18(4)*, Dec. 1989, p. 4-11.

M. Stonebraker, L. A. Rowe, and M. Hirohama, "The implementation of POSTGRES[9]", *Transactions on Knowledge and Data Engineering 2(1)*, IEEE, March 1990.

M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, "On Rules, Procedures, Caching and Views in Database Systems[10]", Proc. ACM-SIGMOD Conference on Management of Data, June 1990.

3. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-13.pdf
4. http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z
5. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M85-95.pdf
6. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-06.pdf
7. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-82.pdf
8. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf
9. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-34.pdf
10. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-36.pdf

Index

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W