Oracle*i*

SQL Reference

Release 3 (8.1.7)

September 2000

Part No. A85397-01

ORACLE®

SQL Reference, Release 3 (8.1.7)

Part No. A85397-01

Primary Author: Diana Lorentz

Contributors: Dave Alpern, Vikas Arora, Lance Ashdown, Hermann Baer, Vladimir Barriere, Lucy Burgess, Souripriya Das, Carolyn Gray, John Haydu, Thuvan Hoang, Wei Hu, Namit Jain, Hakan Jakobsson, Bob Jenkins, Mark Johnson, Jonathan Klein, Susan Kotsovolos, Vishu Krishnamurthy, Muralidhar Krishnaprasad, Paul Lane, Geoff Lee, Nina Lewis, Bryn Llewellyn, Phil Locke, David McElhoes, Jack Melnick, Ari Mozes, Subramanian Muralidhar, Ravi Murthy, Sujatha Muthulingam, Bruce Olsen, Alla S Pfauntsch, Tom Portfolio, Kevin Quinn, Ananth Raghavan, Den Raphaely, John Russell, Anant Singh, Rajesh Sivaramasubramaniom, Roger Snowden, Jags Srinivisan, Sankar Subramanian, Murali Thiyagarajah, Michael Tobie, AhnTuan Tran, Randy Urbano, Andy Witkowski, Daniel Wong, Aravind Yalamanchi, Qin Yu, Fred Zemke, Mohamed Ziauddin

Contents

4 Functions

7 SQL Statements: ALTER CLUSTER to ALTER SYSTEM

8 SQL Statements: ALTER TABLE to *constraint_clause*

9 SQL Statements: CREATE CLUSTER to CREATE SEQUENCE

10 SQL Statements: CREATE SYNONYM to DROP ROLLBACK SEGMENT

11 SQL Statements: DROP SEQUENCE to UPDATE

Send Us Your Comments

SQL Reference, Release 3 (8.1.7)

Part No. A85397-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- E-mail - infodev@us.oracle.com
- FAX - (650) 506-7228. Attn: Information Development
- Postal service:
  Oracle Corporation
  Server Technologies Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065
  U.S.A.

If you would like a reply, please give your name, address, and telephone number below.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This reference contains a complete description of the Structured Query Language (SQL) used to manage information in an Oracle database. Oracle SQL is a superset of the American National Standards Institute (ANSI) and the International Standards Organization (ISO) SQL92 standard at entry level conformance.

See Also:

- *PL/SQL User's Guide and Reference* for information on PL/SQL, Oracle's procedural language extension to SQL
- *Pro*C/C++ Precompiler Programmer's Guide*, *SQL*Module for Ada Programmer's Guide*, and the *Pro*COBOL Precompiler Programmer's Guide* for detailed descriptions of Oracle embedded SQL

Features and Functionality

Oracle8i SQL Reference contains information about the features and functionality of the Oracle8*i* and the Oracle8*i* Enterprise Edition products. Oracle8*i* and Oracle8*i* Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional.

See Also: *Getting to Know Oracle8i* for information about the differences between Oracle8*i* and the Oracle8*i* Enterprise Edition and the available features and options. That book also describes all the features that are new in Oracle8*i*.

Audience

This reference is intended for all users of Oracle SQL.

What's New in Oracle8*i*?

Each release of Oracle8*i* contains new features, many of which are documented throughout this reference.

See Also: *Getting to Know Oracle8i* for a description of all features new to this release

---

Release 3 (8.1.7)

The following SQL functions are new to this release:

- BITAND on page 4-20
- NVL2 on page 4-72

Release 2 (8.1.6)

The following SQL functions are new to this release:

- CORR on page 4-25
- COVAR_POP
- COVAR_SAMP
- CUME_DIST
- DENSE_RANK
- FIRST_VALUE
- LAG
- LAST_VALUE
- LEAD
- NTILE
- NUMTOYMINTERVAL
- NUMTODSINTERVAL
- PERCENT_RANK

- RATIO_TO_REPORT
- REGR_ (linear regression) functions
- STDDEV_POP
- STDDEV_SAMP
- VAR_POP
- VAR_SAMP

In addition, the following features have been enhanced:

- The aggregate functions have expanded functionality. See "Aggregate Functions" on page 4-6.
- When specifying LOB storage parameters, you can now specify caching of LOBs for read-only purposes. See CREATE TABLE on page 10-7.
- The section on Expressions now contains a new expression. See "CASE Expressions" on page 5-14.
- Subqueries can now be unnested. See "Unnesting of Nested Subqueries" on page 5-28.

Release 8.1.5

The following top-level SQL statements are new to Release 8.1.5:

- ALTER DIMENSION on page 7-34
- ALTER JAVA on page 7-58
- ALTER OUTLINE on page 7-83
- ASSOCIATE STATISTICS on page 8-110
- CALL on page 8-128
- CREATE CONTEXT on page 9-13
- CREATE DIMENSION on page 9-34
- CREATE INDEXTYPE on page 9-76
- CREATE JAVA on page 9-79
- CREATE OPERATOR on page 9-115
- CREATE OUTLINE on page 9-119
- CREATE TEMPORARY TABLESPACE on page 10-63

- DISASSOCIATE STATISTICS on page 10-123
- DROP CONTEXT on page 10-128
- DROP DIMENSION on page 10-131
- DROP INDEXTYPE on page 10-138
- DROP JAVA on page 10-140
- DROP OPERATOR on page 10-147
- DROP OUTLINE on page 10-149

Organization

This reference is divided into the following parts:

Chapter 1, "Introduction"

This chapter defines SQL and describes its history as well as the advantages of using it to access relational databases.

Chapter 2, "Basic Elements of Oracle SQL"

This chapter describes the basic building blocks of an Oracle database and of Oracle SQL.

Chapter 3, "Operators"

This chapter describes how to use SQL operators to combine data into expressions and conditions.

Chapter 4, "Functions"

This chapter describes how to use SQL functions to combine data into expressions and conditions.

Chapter 5, "Expressions, Conditions, and Queries"

This chapter describes SQL expressions and conditions and discusses the various ways of extracting information from your database through queries.

Chapter 6, "About SQL Statements"

This chapter lists the various types of SQL statements, and provides a table to help you find the appropriate SQL statement for your database task.

Chapter 7, "SQL Statements: ALTER CLUSTER to ALTER SYSTEM"
Chapter 8, "SQL Statements: ALTER TABLE to constraint_clause"
Chapter 9, "SQL Statements: CREATE CLUSTER to CREATE SEQUENCE"
Chapter 10, "SQL Statements: CREATE SYNONYM to DROP ROLLBACK SEGMENT"
Chapter 11, "SQL Statements: DROP SEQUENCE to UPDATE"
These chapters list and describe all Oracle SQL statements in alphabetical order.

Appendix A, "Syntax Diagrams"
This appendix describes how to read the syntax diagrams in this reference.

Appendix B, "Oracle and Standard SQL"
This appendix describes Oracle compliance with ANSI and ISO standards.

Appendix C, "Oracle Reserved Words"
This appendix lists words that are reserved for internal use by Oracle.

Structural Changes in the Reference in Release 8.1.7

The chapter containing all SQL statements (formerly Chapter 7) has been divided into four chapters for printing purposes.

The following top-level SQL statements have been revised in Release 8.1.7:

- The two SQL statements `GRANT` *`object_privileges`* and `GRANT` *`system_privileges_and_roles`* have been combined into one `GRANT` statement. See GRANT on page 11-31.
- The two SQL statements `REVOKE` *`schema_object_privileges`* and `REVOKE` *`system_privileges_and_roles`* have been combined into one `REVOKE` statement. See REVOKE on page 11-73.
- The two SQL statements `AUDIT` *`sql_statements`* and `AUDIT` *`schema_objects`* have been combined into one `AUDIT` statement. See AUDIT on page 8-114.
- The two SQL statements `NOAUDIT` *`sql_statements`* and `NOAUDIT` *`schema_objects`* have been combined into one `NOAUDIT` statement. See NOAUDIT on page 11-66.

Structural Changes in the Reference in Release 8.1.5

Users familiar with the Release 8.0 documentation will find that the following sections have been moved or renamed:

- The section "Format Models" now appears in Chapter 2 on page 2-41.
- Chapter 3 has been divided into several smaller chapters:
  - Chapter 3, "Operators"
  - Chapter 4, "Functions"
  - Chapter 5, "Expressions, Conditions, and Queries". The last section, "Queries and Subqueries" on page 5-21, provides background for the syntactic and semantic information in SELECT and subquery on page 11-88.
- A new chapter, Chapter 6, "About SQL Statements", has been added to help you find the correct SQL statement for a particular task.
- The *archive_log_clause* is no longer a separate section, but has been incorporated into ALTER SYSTEM on page 7-127.
- The *deallocate_unused_clause* is no longer a separate section, but has been incorporated into ALTER TABLE on page 8-2, ALTER CLUSTER on page 7-3, and ALTER INDEX on page 7-40.
- The *disable_clause* is no longer a separate section, but has been incorporated into CREATE TABLE on page 10-7 and ALTER TABLE on page 8-2.
- The *drop_clause* is no longer a separate section. It has become the *drop_constraint_clause* of the ALTER TABLE statement (to distinguish it from the new *drop_column_clause* of that statement). See ALTER TABLE on page 8-2.
- The *enable_clause* is no longer a separate section, but has been incorporated into CREATE TABLE on page 10-7 and ALTER TABLE on page 8-2.
- The *parallel_clause* is no longer a separate section. The clause has been simplified, and has been incorporated into the various statements where it is relevant.
- The *recover_clause* is no longer a separate section. Recovery functionality has been enhanced, and because it is always implemented through the ALTER DATABASE statement, it has been incorporated into that section. See ALTER DATABASE on page 7-9.

- The sections on **snapshots** and **snapshot logs** have been moved and renamed. Snapshot functionality has been greatly enhanced, and these objects are now called **materialized views**. See CREATE MATERIALIZED VIEW on page 9-88, ALTER MATERIALIZED VIEW on page 7-61, DROP MATERIALIZED VIEW on page 10-143, "CREATE MATERIALIZED VIEW LOG" on page 9-107, ALTER MATERIALIZED VIEW LOG on page 7-76, and DROP MATERIALIZED VIEW LOG on page 10-145.
- The section on **subqueries** has now been combined with the SELECT statement. See SELECT and subquery on page 11-88.

Conventions Used in this Reference

This section explains the conventions used in this book including:

- Text
- Syntax Diagrams and Notation
- Code Examples
- Example Data

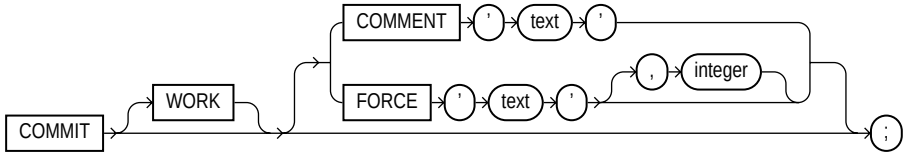
Text

The text in this reference adheres to the following conventions:

UPPERCASE	Uppercase text calls attention to SQL keywords, filenames, and initialization parameters.
italics	Italicized text calls attention to parameters of SQL statements.
boldface	Boldface text calls attention to definitions of terms.

Syntax Diagrams and Notation

Syntax Diagrams This reference uses syntax diagrams to show SQL statements in Chapter 7 through Chapter 11, and to show other elements of the SQL language in Chapter 2, “Basic Elements of Oracle SQL”; Chapter 3, “Operators”; Chapter 4, “Functions”; and Chapter 5, “Expressions, Conditions, and Queries”. These syntax diagrams use lines and arrows to show syntactic structure, as shown here:



If you are not familiar with this type of syntax diagram, refer to Appendix A, “Syntax Diagrams”, for a description of how to read them. This section describes the components of syntax diagrams and gives examples of how to write SQL statements. Syntax diagrams are made up of these items:

Keywords Keywords have special meanings in the SQL language. In the syntax diagrams, keywords appear in UPPERCASE. You must use keywords in your SQL statements exactly as they appear in the syntax diagram, except that they can be either uppercase or lowercase. For example, you must use the CREATE keyword to begin your CREATE TABLE statements just as it appears in the CREATE TABLE syntax diagram.

Parameters Parameters act as placeholders in syntax diagrams. They appear in lowercase. Parameters are usually names of database objects, Oracle datatype names, or expressions. When you see a parameter in a syntax diagram, substitute an object or expression of the appropriate type in your SQL statement. For example, to write a CREATE TABLE statement, use the name of the table you want to create, such as emp, in place of the *table* parameter in the syntax diagram. (Note that parameter names appear in italics in the text.)

Code Examples

This reference contains many examples of SQL statements. These examples show you how to use elements of SQL. The following example shows a CREATE TABLE statement:

```
CREATE TABLE accounts
    ( accno     NUMBER,
      owner     VARCHAR2(10),
      balance   NUMBER(7,2) );
```

Code examples appear in a different font than the text.

Examples follow these conventions:

- Keywords, such as CREATE and NUMBER, appear in uppercase.

- Names of database objects and their parts, such as `accounts` and `accno`, appear in lowercase.
- PL/SQL blocks appear in italics. Keywords and parameters in these blocks may not be documented in this reference unless they are also SQL keywords and parameters. For more information see *PL/SQL User's Guide and Reference.*

Many examples assume the existence of objects that are not created in the example itself. The examples will not work as expected unless you first create those underlying objects.

SQL is not case sensitive (except for quoted identifiers), so you need not follow these conventions when writing your own SQL statements. However, your statements may be easier for you to read if you do.

Some Oracle tools require you to terminate SQL statements with a special character. For example, the code examples in this reference were issued through SQL*Plus, and therefore are terminated with a semicolon (;). If you issue these example statements to Oracle, you must terminate them with the special character expected by the Oracle tool you are using.

Example Data

Many examples in this reference operate on sample tables. The definitions of some of these tables appear in a SQL script available on your distribution medium. On most operating systems the name of this script is `UTLSAMPL.SQL`, although its exact name and location depend on your operating system. This script creates sample users and creates these sample tables in the schema of the user `scott` (password `tiger`):

```
CREATE TABLE dept
    (deptno    NUMBER(2)      CONSTRAINT pk_dept PRIMARY KEY,
     dname     VARCHAR2(14),
     loc       VARCHAR2(13) );
CREATE TABLE emp
    (empno     NUMBER(4)      CONSTRAINT pk_emp PRIMARY KEY,
     ename     VARCHAR2(10),
     job       VARCHAR2(9),
     mgr       NUMBER(4),
     hiredate  DATE,
     sal       NUMBER(7,2),
     comm      NUMBER(7,2),
     deptno    NUMBER(2)    CONSTRAINT fk_deptno REFERENCES dept );
CREATE TABLE bonus
    (ename     VARCHAR2(10),
```

```
     job       VARCHAR2(9),
     sal       NUMBER,
     comm      NUMBER );
CREATE TABLE salgrade
    (grade     NUMBER,
     losal     NUMBER,
     hisal     NUMBER );
```

The script also fills the sample tables with this data:

```
SELECT * FROM dept;

DEPTNO  DNAME      LOC
------- ---------- ---------
10      ACCOUNTING NEW YORK
20      RESEARCH   DALLAS
30      SALES      CHICAGO
40      OPERATIONS BOSTON

SELECT * FROM emp;

EMPNO  ENAME   JOB          MGR  HIREDATE    SAL   COMM  DEPTNO
-----  ------- --------- ------ --------- ------ ------ -------
 7369  SMITH   CLERK       7902 17-DEC-80    800             20
 7499  ALLEN   SALESMAN    7698 20-FEB-81   1600    300      30
 7521  WARD    SALESMAN    7698 22-FEB-81   1250    500      30
 7566  JONES   MANAGER     7839 02-APR-81   2975             20
 7654  MARTIN  SALESMAN    7698 28-SEP-81   1250   1400      30
 7698  BLAKE   MANAGER     7839 01-MAY-81   2850             30
 7782  CLARK   MANAGER     7839 09-JUN-81   2450             10
 7788  SCOTT   ANALYST     7566 19-APR-87   3000             20
 7839  KING    PRESIDENT         17-NOV-81   5000             10
 7844  TURNER  SALESMAN    7698 08-SEP-81   1500             30
 7876  ADAMS   CLERK       7788 23-MAY-87   1100             20
 7900  JAMES   CLERK       7698 03-DEC-81    950             30
 7902  FORD    ANALYST     7566 03-DEC-81   3000             20
 7934  MILLER  CLERK       7782 23-JAN-82   1300             10

SELECT * FROM salgrade;

GRADE LOSAL HISAL
----- ----- -----
1     700   1200
2     1201  1400
3     1401  2000
```

```
4     2001  3000
5     3001  9999
```

The bonus table does not contain any data.

To perform all the operations of the script, run it when you are logged into Oracle as the user SYSTEM.

1

Introduction

Structured Query Language (SQL) is the set of statements with which all programs and users access data in an Oracle database. Application programs and Oracle tools often allow users access to the database without using SQL directly, but these applications in turn must use SQL when executing the user's request. This chapter provides background information on SQL as used by most relational database systems.

This chapter contains these topics:

History of SQL

Dr. E. F. Codd published the paper, "A Relational Model of Data for Large Shared Data Banks", in June 1970 in the Association of Computer Machinery (ACM) journal, *Communications of the ACM*. Codd's model is now accepted as the definitive model for relational database management systems (RDBMS). The language, Structured English Query Language ("SEQUEL") was developed by IBM Corporation, Inc., to use Codd's model. SEQUEL later became SQL (still pronounced "sequel"). In 1979, Relational Software, Inc. (now Oracle Corporation) introduced the first commercially available implementation of SQL. Today, SQL is accepted as the standard RDBMS language.

SQL Standards

Oracle Corporation strives to comply with industry-accepted standards and participates actively in SQL standards committees. Industry-accepted committees are the American National Standards Institute (ANSI) and the International Standards Organization (ISO), which is affiliated with the International Electrotechnical Commission (IEC). Both ANSI and the ISO / IEC have accepted SQL as the standard language for relational databases. When a new SQL standard is simultaneously published by these organizations, the names of the standards conform to conventions used by the organization, but the standards are technically identical.

The latest SQL standard was adopted in July 1999 and is often called SQL-99. The formal names of this standard are:

- ANSI X3.135-1999, "Database Language SQL", Parts 1 ("Framework"), 2 ("Foundation"), and 5 ("Bindings")
- ISO / IEC 9075:1999, "Database Language SQL", Parts 1 ("Framework"), 2 ("Foundation"), and 5 ("Bindings")

SQL-99 replaced the previous version of the standard, commonly known as SQL-92. SQL-99 is an upward compatible extension of SQL-92, except for a few minor incompatibilities noted in Annex E of Part 2, "Foundation," of SQL-99.

SQL-92 defined four levels of compliance: Entry, Transitional, Intermediate, and Full. A conforming SQL implementation must support at least Entry SQL. Oracle8*i* fully supports Entry SQL as outlined in Federal Information Processing Standard (FIPS) PUB 127-2, and has many features that conform to Transitional, Intermediate, or Full SQL.

The minimal conformance level for SQL-99 is known as Core. Core SQL-99 is a superset of SQL-92 Entry Level specification. Oracle8*i* also is broadly compatible

with the SQL-99 Core specification. However, some SQL-99 Core features are not currently implemented in Oracle8*i* or differ from the Oracle8*i* implementation. Oracle Corporation is committed to fully supporting SQL-99 Core functionality in a future release, while providing upward compatibility for existing applications.

> **See Also:** Appendix B, "Oracle and Standard SQL" for more information about Oracle and standard SQL

How SQL Works

The strengths of SQL provide benefits for all types of users, including application programmers, database administrators, managers, and end users. Technically speaking, SQL is a data sublanguage. The purpose of SQL is to provide an interface to a relational database such as Oracle, and all SQL statements are instructions to the database. In this SQL differs from general-purpose programming languages like C and BASIC. Among the features of SQL are the following:

- It processes sets of data as groups rather than as individual units.
- It provides automatic navigation to the data.
- It uses statements that are complex and powerful individually, and that therefore stand alone. Flow-control statements were not part of SQL originally, but they are found in the recently accepted optional part of SQL, ISO / IEC 9075-5: 1996. Flow-control statements are commonly known as "persistent stored modules" (PSM), and Oracle's PL / SQL extension to SQL is similar to PSM.

Essentially, SQL lets you work with data at the logical level. You need to be concerned with the implementation details only when you want to manipulate the data. For example, to retrieve a set of rows from a table, you define a condition used to filter the rows. All rows satisfying the condition are retrieved in a single step and can be passed as a unit to the user, to another SQL statement, or to an application. You need not deal with the rows one by one, nor do you have to worry about how they are physically stored or retrieved. All SQL statements use the **optimizer**, a part of Oracle that determines the most efficient means of accessing the specified data. Oracle also provides techniques that you can use to make the optimizer perform its job better.

SQL provides statements for a variety of tasks, including:

- Querying data
- Inserting, updating, and deleting rows in a table
- Creating, replacing, altering, and dropping objects

- Controlling access to the database and its objects
- Guaranteeing database consistency and integrity

SQL unifies all of the above tasks in one consistent language.

Common Language for All Relational Databases

All major relational database management systems support SQL, so you can transfer all skills you have gained with SQL from one database to another. In addition, all programs written in SQL are portable. They can often be moved from one database to another with very little modification.

Embedded SQL

Embedded SQL refers to the use of standard SQL statements embedded within a procedural programming language. The embedded SQL statements are documented in the Oracle precompiler books.

Embedded SQL is a collection of these statements:

- All SQL commands, such as `SELECT` and `INSERT`, available with SQL with interactive tools
- Dynamic SQL execution commands, such as `PREPARE` and `OPEN`, which integrate the standard SQL statements with a procedural programming language

Embedded SQL also includes extensions to some standard SQL statements. Embedded SQL is supported by the Oracle precompilers. The Oracle precompilers interpret embedded SQL statements and translate them into statements that can be understood by procedural language compilers.

Each of these Oracle precompilers translates embedded SQL programs into a different procedural language:

- Pro*C/C++ precompiler
- Pro*COBOL precompiler
- SQL*Module for ADA

> **See Also:** ,*SQL*Module for Ada Programmer's Guide*, *Pro*C/C++ Precompiler Programmer's Guide*, and *Pro*COBOL Precompiler Programmer's Guide* for a definition of the Oracle precompilers and the embedded SQL statements

Lexical Conventions

The following lexical conventions for issuing SQL statements apply specifically to Oracle's implementation of SQL, but are generally acceptable in other SQL implementations.

When you issue a SQL statement, you can include one or more tabs, carriage returns, spaces, or comments anywhere a space occurs within the definition of the statement. Thus, Oracle evaluates the following two statements in the same manner:

```
SELECT ENAME,SAL*12,MONTHS_BETWEEN(HIREDATE,SYSDATE) FROM EMP;

SELECT ENAME,
     SAL * 12,
          MONTHS_BETWEEN( HIREDATE, SYSDATE )
FROM EMP;
```

Case is insignificant in reserved words, keywords, identifiers and parameters. However, case is significant in text literals and quoted names.

See Also: "Text" on page 2-33 for a syntax description

Tools Support

Most (but not all) Oracle tools support all features of Oracle SQL. This reference describes the complete functionality of SQL. If the Oracle tool that you are using does not support this complete functionality, you can find a discussion of the restrictions in the manual describing the tool, such as *SQL*Plus User's Guide and Reference*.

If you are using Trusted Oracle, see your Trusted Oracle documentation for information about SQL statements specific to that environment.

2

Basic Elements of Oracle SQL

This chapter contains reference information on the basic elements of Oracle SQL. These elements are simplest building blocks of SQL statements. Therefore, before using the statements described in Chapter 7 through Chapter 11, you should familiarize yourself with the concepts covered in this chapter, as well as in Chapter 3, "Operators", Chapter 4, "Functions", Chapter 5, "Expressions, Conditions, and Queries", and Chapter 6, "About SQL Statements".

This chapter contains these sections:

- Datatypes
- Literals
- Format Models
- Nulls
- Pseudocolumns
- Comments
- Database Objects
- Schema Object Names and Qualifiers
- Syntax for Schema Objects and Parts in SQL Statements

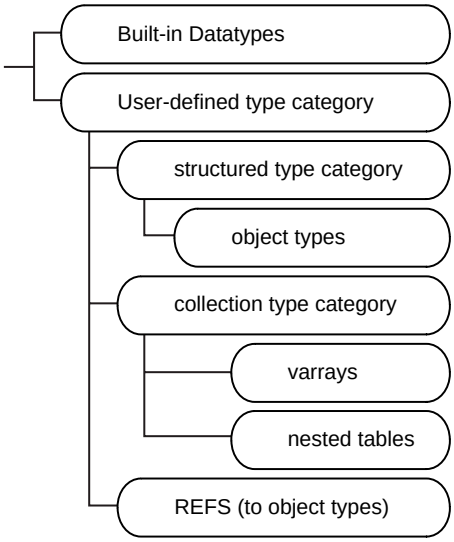
Datatypes

Each value manipulated by Oracle has a **datatype**. A value's datatype associates a fixed set of properties with the value. These properties cause Oracle to treat values of one datatype differently from values of another. For example, you can add values of NUMBER datatype, but not values of RAW datatype.

When you create a table or cluster, you must specify a datatype for each of its columns. When you create a procedure or stored function, you must specify a datatype for each of its arguments. These datatypes define the domain of values that each column can contain or each argument can have. For example, DATE columns cannot accept the value February 29 (except for a leap year) or the values 2 or 'SHOE'. Each value subsequently placed in a column assumes the column's datatype. For example, if you insert '01-JAN-98' into a DATE column, Oracle treats the '01-JAN-98' character string as a DATE value after verifying that it translates to a valid date.

Oracle provides a number of built-in datatypes as well as several categories for user-defined types, as shown in Figure 2–1.

Figure 2–1 Oracle Type Categories

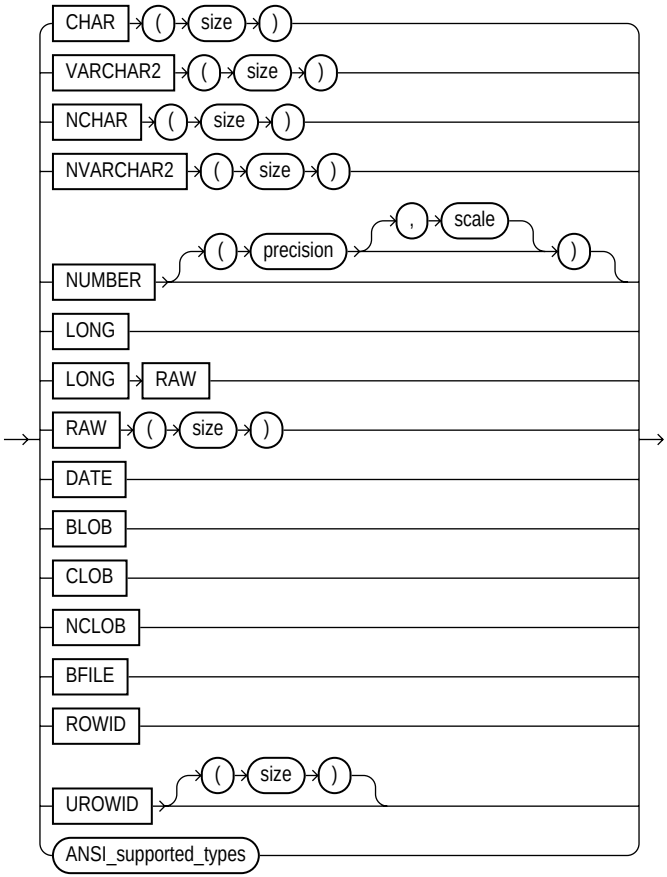


The syntax of the Oracle built-in datatypes appears in the next diagram. Table 2–1 summarizes Oracle built-in datatypes. The rest of this section describes these datatypes as well as the various kinds of user-defined types.

Note: The Oracle precompilers recognize other datatypes in embedded SQL programs. These datatypes are called **external datatypes** and are associated with host variables. Do not confuse built-in and user-defined datatypes with external datatypes. For information on external datatypes, including how Oracle converts between them and built-in or user-defined datatypes, see *Pro*COBOL Precompiler Programmer's Guide*, *Pro*C/C++ Precompiler Programmer's Guide*, and *SQL*Module for Ada Programmer's Guide*.

---

built-in datatypes:



The ANSI-supported datatypes appear in the figure that follows. Table 2–2 shows the mapping of ANSI-supported datatypes to Oracle build-in datatypes.

ANSI-supported datatypes:

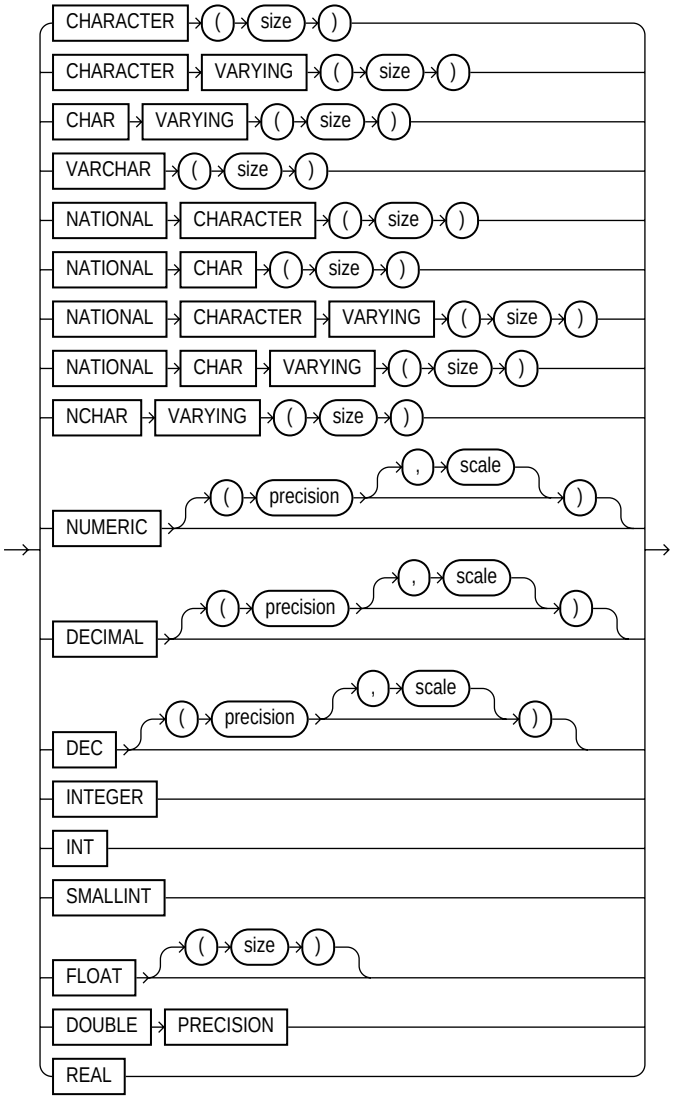


Table 2–1 Built-In Datatype Summary

Code[a]	Built-In Datatype	Description
1	VARCHAR2(*size*)	Variable-length character string having maximum length *size* bytes. Maximum *size* is 4000, and minimum is 1. You must specify *size* for VARCHAR2.
1	NVARCHAR2(*size*)	Variable-length character string having maximum length *size* characters or bytes, depending on the choice of national character set. Maximum *size* is determined by the number of bytes required to store each character, with an upper limit of 4000 bytes. You must specify *size* for NVARCHAR2.
2	NUMBER(*p*,*s*)	Number having precision *p* and scale *s*. The precision *p* can range from 1 to 38. The scale *s* can range from -84 to 127.
8	LONG	Character data of variable length up to 2 gigabytes, or 2^{31} -1 bytes.
12	DATE	Valid date range from January 1, 4712 BC to December 31, 9999 AD.
23	RAW(*size*)	Raw binary data of length *size* bytes. Maximum *size* is 2000 bytes. You must specify *size* for a RAW value.
24	LONG RAW	Raw binary data of variable length up to 2 gigabytes.
69	ROWID	Hexadecimal string representing the unique address of a row in its table. This datatype is primarily for values returned by the ROWID pseudocolumn.
208	UROWID [(*size*)]	Hexadecimal string representing the logical address of a row of an index-organized table. The optional *size* is the size of a column of type UROWID. The maximum size and default is 4000 bytes.
96	CHAR(*size*)	Fixed-length character data of length *size* bytes. Maximum *size* is 2000 bytes. Default and minimum *size* is 1 byte.

[a] The codes listed for the datatypes are used internally by Oracle. The datatype code of a column or object attribute is returned by the DUMP function.

Table 2–1 (Cont.) Built-In Datatype Summary

Code[a]	Built-In Datatype	Description
96	NCHAR(*size*)	Fixed-length character data of length *size* characters or bytes, depending on the choice of national character set. Maximum *size* is determined by the number of bytes required to store each character, with an upper limit of 2000 bytes. Default and minimum *size* is 1 character or 1 byte, depending on the character set.
112	CLOB	A character large object containing single-byte characters. Both fixed-width and variable-width character sets are supported, both using the CHAR database character set. Maximum size is 4 gigabytes.
112	NCLOB	A character large object containing multibyte characters. Both fixed-width and variable-width character sets are supported, both using the NCHAR database character set. Maximum size is 4 gigabytes. Stores national character set data.
113	BLOB	A binary large object. Maximum size is 4 gigabytes.
114	BFILE	Contains a locator to a large binary file stored outside the database. Enables byte stream I/O access to external LOBs residing on the database server. Maximum size is 4 gigabytes.

[a] The codes listed for the datatypes are used internally by Oracle. The datatype code of a column or object attribute is returned by the DUMP function.

Character Datatypes

Character datatypes store character (alphanumeric) data, which are words and free-form text, in the database character set or national character set. They are less restrictive than other datatypes and consequently have fewer properties. For example, character columns can store all alphanumeric values, but NUMBER columns can store only numeric values.

Character data is stored in strings with byte values corresponding to one of the character sets, such as 7-bit ASCII or EBCDIC, specified when the database was created. Oracle supports both single-byte and multibyte character sets.

These datatypes are used for character data:

- CHAR Datatype
- NCHAR Datatype

- NVARCHAR2 Datatype
- VARCHAR2 Datatype

CHAR Datatype

The CHAR datatype specifies a fixed-length character string. When you create a table with a CHAR column, you supply the column length in bytes. Oracle subsequently ensures that all values stored in that column have this length. If you insert a value that is shorter than the column length, Oracle blank-pads the value to column length. If you try to insert a value that is too long for the column, Oracle returns an error.

The default length for a CHAR column is 1 character and the maximum allowed is 2000 characters. A zero-length string can be inserted into a CHAR column, but the column is blank-padded to 1 character when used in comparisons.

> **See Also:** "Datatype Comparison Rules" on page 2-26 for information on comparison semantics

---

Note: To ensure proper data conversion between databases with different character sets, you must ensure that CHAR data consists of well-formed strings. See *Oracle8i National Language Support Guide* for more information on character set support.

NCHAR Datatype

The NCHAR datatype specifies a fixed-length national character set character string. When you create a table with an NCHAR column, you define the column length either in characters or in bytes. You define the national character set when you create your database.

If the national character set of the database is fixed width, such as JA16EUCFIXED, then you declare the NCHAR column size as the number of characters desired for the string length. If the national character set is variable width, such as JA16SJIS, you declare the column size in bytes. The following statement creates a table with one NCHAR column that can store strings up to 30 characters in length using JA16EUCFIXED as the national character set:

```
CREATE TABLE tab1 (col1 NCHAR(30));
```

The column's maximum length is determined by the national character set definition. Width specifications of character datatype NCHAR refer to the number of

characters if the national character set is fixed width and refer to the number of bytes if the national character set is variable width. The maximum column size allowed is 2000 bytes. For fixed-width, multibyte character sets, the maximum length of a column allowed is the number of characters that fit into no more than 2000 bytes.

If you insert a value that is shorter than the column length, Oracle blank-pads the value to column length. You cannot insert a CHAR value into an NCHAR column, nor can you insert an NCHAR value into a CHAR column.

The following example compares the `col1` column of `tab1` with national character set string 'NCHAR literal':

```
SELECT * FROM tab1 WHERE col1 = N'NCHAR literal';
```

NVARCHAR2 Datatype

The NVARCHAR2 datatype specifies a variable-length national character set character string. When you create a table with an NVARCHAR2 column, you supply the maximum number of characters or bytes it can hold. Oracle subsequently stores each value in the column exactly as you specify it, provided the value does not exceed the column's maximum length.

The column's maximum length is determined by the national character set definition. Width specifications of character datatype NVARCHAR2 refer to the number of characters if the national character set is fixed width and refer to the number of bytes if the national character set is variable width. The maximum column size allowed is 4000 bytes. For fixed-width, multibyte character sets, the maximum length of a column allowed is the number of characters that fit into no more than 4000 bytes.

The following statement creates a table with one NVARCHAR2 column of 2000 characters in length (stored as 4000 bytes, because each character takes two bytes) using JA16EUCFIXED as the national character set:

```
CREATE TABLE tab1 (col1 NVARCHAR2(2000));
```

VARCHAR2 Datatype

The VARCHAR2 datatype specifies a variable-length character string. When you create a VARCHAR2 column, you supply the maximum number of bytes of data that it can hold. Oracle subsequently stores each value in the column exactly as you specify it, provided the value does not exceed the column's maximum length. If you try to insert a value that exceeds the specified length, Oracle returns an error.

You must specify a maximum length for a VARCHAR2 column. This maximum must be at least 1 byte, although the actual length of the string stored is permitted to be zero. The maximum length of VARCHAR2 data is 4000 bytes. Oracle compares VARCHAR2 values using nonpadded comparison semantics.

> **See Also:** "Datatype Comparison Rules" on page 2-26 for information on comparison semantics

---

Note: To ensure proper data conversion between databases with different character sets, you must ensure that VARCHAR2 data consists of well-formed strings. See *Oracle8i National Language Support Guide* for more information on character set support.

VARCHAR Datatype

The VARCHAR datatype is currently synonymous with the VARCHAR2 datatype. Oracle recommends that you use VARCHAR2 rather than VARCHAR. In the future, VARCHAR might be defined as a separate datatype used for variable-length character strings compared with different comparison semantics.

NUMBER Datatype

The NUMBER datatype stores zero, positive, and negative fixed and floating-point numbers with magnitudes between 1.0×10^{-130} and $9.9...9 \times 10^{125}$ (38 nines followed by 88 zeroes) with 38 digits of precision. If you specify an arithmetic expression whose value has a magnitude greater than or equal to 1.0×10^{126}, Oracle returns an error.

Specify a fixed-point number using the following form:

```
NUMBER(p,s)
```

where:

- *p* is the **precision**, or the total number of digits. Oracle guarantees the portability of numbers with precision ranging from 1 to 38.
- *s* is the **scale**, or the number of digits to the right of the decimal point. The scale can range from -84 to 127.

Specify an integer using the following form:

- `NUMBER(p)` is a fixed-point number with precision p and scale 0. This is equivalent to `NUMBER(p,0)`.

Specify a floating-point number using the following form:

- `NUMBER` is a floating-point number with decimal precision 38. Note that a scale value is not applicable for floating-point numbers.

See Also: "Floating-Point Numbers" on page 2-12

Scale and Precision

Specify the scale and precision of a fixed-point number column for extra integrity checking on input. Specifying scale and precision does not force all values to a fixed length. If a value exceeds the precision, Oracle returns an error. If a value exceeds the scale, Oracle rounds it.

The following examples show how Oracle stores data using different precisions and scales.

Actual Data	Specified As	Stored As
7456123.89	`NUMBER`	7456123.89
7456123.89	`NUMBER(9)`	7456124
7456123.89	`NUMBER(9,2)`	7456123.89
7456123.89	`NUMBER(9,1)`	7456123.9
7456123.89	`NUMBER(6)`	exceeds precision
7456123.89	`NUMBER(7,-2)`	7456100
7456123.89	`NUMBER(-7,2)`	exceeds precision

Negative Scale

If the scale is negative, the actual data is rounded to the specified number of places to the left of the decimal point. For example, a specification of (10,-2) means to round to hundreds.

Scale Greater than Precision

You can specify a scale that is greater than precision, although it is uncommon. In this case, the precision specifies the maximum number of digits to the right of the decimal point. As with all number datatypes, if the value exceeds the precision, Oracle returns an error message. If the value exceeds the scale, Oracle rounds the

value. For example, a column defined as NUMBER(4,5) requires a zero for the first digit after the decimal point and rounds all values past the fifth digit after the decimal point. The following examples show the effects of a scale greater than precision:

Actual Data	Specified As	Stored As
.01234	NUMBER(4,5)	.01234
.00012	NUMBER(4,5)	.00012
.000127	NUMBER(4,5)	.00013
.0000012	NUMBER(2,7)	.0000012
.00000123	NUMBER(2,7)	.0000012

Floating-Point Numbers

Oracle allows you to specify floating-point numbers, which can have a decimal point anywhere from the first to the last digit or can have no decimal point at all. A scale value is not applicable to floating-point numbers, because the number of digits that can appear after the decimal point is not restricted.

You can specify floating-point numbers with the form discussed in "NUMBER Datatype" on page 2-10. Oracle also supports the ANSI datatype FLOAT. You can specify this datatype using one of these syntactic forms:

- FLOAT specifies a floating-point number with decimal precision 38, or binary precision 126.
- FLOAT(b) specifies a floating-point number with binary precision *b*. The precision *b* can range from 1 to 126. To convert from binary to decimal precision, multiply *b* by 0.30103. To convert from decimal to binary precision, multiply the decimal precision by 3.32193. The maximum of 126 digits of binary precision is roughly equivalent to 38 digits of decimal precision.

LONG Datatype

LONG columns store variable-length character strings containing up to 2 gigabytes, or 2^{31}-1 bytes. LONG columns have many of the characteristics of VARCHAR2 columns. You can use LONG columns to store long text strings. The length of LONG values may be limited by the memory available on your computer.

> **Note:** Oracle Corporation strongly recommends that you convert LONG columns to LOB columns. LOB columns are subject to far fewer restrictions than LONG columns. See "TO_LOB" on page 4-111 for more information.

You can reference LONG columns in SQL statements in these places:

- SELECT lists
- SET clauses of UPDATE statements
- VALUES clauses of INSERT statements

The use of LONG values is subject to some restrictions:

- A table cannot contain more than one LONG column.
- You cannot create an object type with a LONG attribute.
- LONG columns cannot appear in integrity constraints (except for NULL and NOT NULL constraints).
- LONG columns cannot be indexed.
- A stored function cannot return a LONG value.
- Within a single SQL statement, all LONG columns, updated tables, and locked tables must be located on the same database.

LONG columns cannot appear in certain parts of SQL statements:

- WHERE clauses, GROUP BY clauses, ORDER BY clauses, or CONNECT BY clauses or with the DISTINCT operator in SELECT statements
- The UNIQUE operator of a SELECT statement
- The column list of a CREATE CLUSTER statement
- The CLUSTER clause of a CREATE MATERIALIZED VIEW statement
- SQL functions (such as SUBSTR or INSTR)
- Expressions or conditions
- SELECT lists of queries containing GROUP BY clauses
- SELECT lists of subqueries or queries combined by the UNION, INTERSECT, or MINUS set operators

- SELECT lists of CREATE TABLE ... AS SELECT statements
- SELECT lists in subqueries in INSERT statements

Triggers can use the LONG datatype in the following manner:

- A SQL statement within a trigger can insert data into a LONG column.
- If data from a LONG column can be converted to a constrained datatype (such as CHAR and VARCHAR2), a LONG column can be referenced in a SQL statement within a trigger.
- Variables in triggers cannot be declared using the LONG datatype.
- :NEW and :OLD cannot be used with LONG columns.

You can use the Oracle Call Interface functions to retrieve a portion of a LONG value from the database.

See Also: *Oracle Call Interface Programmer's Guide*

DATE Datatype

The DATE datatype stores date and time information. Although date and time information can be represented in both CHAR and NUMBER datatypes, the DATE datatype has special associated properties. For each DATE value, Oracle stores the following information: century, year, month, day, hour, minute, and second.

If you specify a date value without a time component, the default time is 12:00:00 AM (midnight). If you specify a time value without a date, the default date is the first day of the current month. The date function SYSDATE returns the current date and time.

The default date format is specified by the initialization parameter NLS_DATE_FORMAT and is a string such as 'DD-MON-YY'. This example default date format includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year. Oracle automatically converts character values that are in the default date format into DATE values when they are used in date expressions.

To specify a date value that is not in the default format, you must convert a character or numeric value to a date value with the TO_DATE function. In this case, you must specify the nondefault date format model (sometimes called a "date mask") to tell Oracle how to interpret the character or numeric value. For example, the date format model for '17:45:29' is 'HH24:MI:SS'. The date format model for '11-NOV-1999' is 'DD-MON-YYYY'.

See Also:

- "Date Format Models" on page 2-47 for a listing of the elements of date format models
- "TO_DATE" on page 4-110 for information on converting character and numeric values into DATE values
- "TO_CHAR (date conversion)" on page 4-108 for information on converting DATE values into strings
- "SYSDATE" on page 4-106 for information on obtaining the current system date and time.

Date Arithmetic

You can add and subtract number constants as well as other dates from dates. Oracle interprets number constants in arithmetic date expressions as numbers of days. For example, SYSDATE + 1 is tomorrow. SYSDATE - 7 is one week ago. SYSDATE + (10 / 1440) is ten minutes from now. Subtracting the `hiredate` column of the `emp` table from SYSDATE returns the number of days since each employee was hired. You cannot multiply or divide DATE values.

Oracle provides functions for many common date operations. For example, the ADD_MONTHS function lets you add or subtract months from a date. The MONTHS_BETWEEN function returns the number of months between two dates. The fractional portion of the result represents that portion of a 31-day month.

Because each date contains a time component, most results of date operations include a fraction. This fraction means a portion of one day. For example, 1.5 days is 36 hours.

See Also: "Date Functions" on page 4-5 for more information on date functions

Using Julian Dates

A Julian date is the number of days since January 1, 4712 BC. Julian dates allow continuous dating from a common reference. You can use the date format model "J" with date functions TO_DATE and TO_CHAR to convert between Oracle DATE values and their Julian equivalents.

Example This statement returns the Julian equivalent of January 1, 1997:

```
SELECT TO_CHAR(TO_DATE('01-01-1997', 'MM-DD-YYYY'),'J')
    FROM DUAL;
```

```
TO_CHAR
--------
2450450
```

> **See Also:** "Selecting from the DUAL Table" on page 5-28 for a description of the DUAL table

RAW and LONG RAW Datatypes

The RAW and LONG RAW datatypes store data that is not to be interpreted (not explicitly converted when moving data between different systems) by Oracle. These datatypes are intended for binary data or byte strings. For example, you can use LONG RAW to store graphics, sound, documents, or arrays of binary data, for which the interpretation is dependent on the use.

> **Note:** Oracle Corporation strongly recommends that you convert LONG RAW columns to binary LOB (BLOB) columns. LOB columns are subject to far fewer restrictions than LONG columns. See TO_LOB on page 4-111 for more information.

RAW is a variable-length datatype like VARCHAR2, except that Net8 (which connects user sessions to the instance) and the Import and Export utilities do not perform character conversion when transmitting RAW or LONG RAW data. In contrast, Net8 and Import/Export automatically convert CHAR, VARCHAR2, and LONG data from the database character set to the user session character set (which you can set with the NLS_LANGUAGE parameter of the ALTER SESSION statement), if the two character sets are different.

When Oracle automatically converts RAW or LONG RAW data to and from CHAR data, the binary data is represented in hexadecimal form, with one hexadecimal character representing every four bits of RAW data. For example, one byte of RAW data with bits 11001011 is displayed and entered as 'CB'.

Large Object (LOB) Datatypes

The built-in LOB datatypes BLOB, CLOB, and NCLOB (stored internally), and the BFILE (stored externally), can store large and unstructured data such as text, image, video, and spatial data up to 4 gigabytes in size.

When creating a table, you can optionally specify different tablespace and storage characteristics for LOB columns or LOB object attributes from those specified for the table.

LOB columns contain LOB locators that can refer to out-of-line or in-line LOB values. Selecting a LOB from a table actually returns the LOB's locator and not the entire LOB value. The DBMS_LOB package and Oracle Call Interface (OCI) operations on LOBs are performed through these locators.

LOBs are similar to LONG and LONG RAW types, but differ in the following ways:

- LOBs can be attributes of a user-defined datatype (object).
- The LOB locator is stored in the table column, either with or without the actual LOB value. BLOB, NCLOB, and CLOB values can be stored in separate tablespaces. BFILE data is stored in an external file on the server.
- When you access a LOB column, the locator is returned.
- A LOB can be up to 4 gigabytes in size. BFILE maximum size is operating system dependent, but cannot exceed 4 gigabytes.
- LOBs permit efficient, random, piece-wise access to and manipulation of data.
- You can define more than one LOB column in a table.
- With the exception of NCLOB, you can define one or more LOB attributes in an object.
- You can declare LOB bind variables.
- You can select LOB columns and LOB attributes.
- You can insert a new row or update an existing row that contains one or more LOB columns and / or an object with one or more LOB attributes. (You can set the internal LOB value to NULL, empty, or replace the entire LOB with data. You can set the BFILE to NULL or make it point to a different file.)
- You can update a LOB row / column intersection or a LOB attribute with another LOB row / column intersection or LOB attribute.
- You can delete a row containing a LOB column or LOB attribute and thereby also delete the LOB value. Note that for BFILEs, the actual operating system file is not deleted.

You can access and populate rows of an internal LOB column (a LOB column stored in the database) simply by issuing an INSERT or UPDATE statement. However, to access and populate a LOB attribute that is part of an object type, you must first initialize the LOB attribute using the EMPTY_CLOB or EMPTY_BLOB function. You

can then select the empty LOB attribute and populate it using the DBMS_LOB package or some other appropriate interface.

> **See Also:** "EMPTY_[B | C]LOB" on page 4-37

The following example creates a table with LOB columns. (It assumes the existence of tablespace resumes).

```
CREATE TABLE  person_table (name CHAR(40),
                            resume  CLOB,
                            picture BLOB)
  LOB (resume) STORE AS
               ( TABLESPACE resumes
                  STORAGE (INITIAL 5M NEXT 5M) );
```

> **See Also:**
>
> - *Oracle8i Supplied PL/SQL Packages Reference* and *Oracle Call Interface Programmer's Guide* for more information about these interfaces and LOBs
> - *Oracle8i Application Developer's Guide - Large Objects (LOBs)* for information on creating temporary LOBs and on LOB restrictions
> - "TO_LOB" on page 4-111 for more information on converting LONG columns to LOB columns

BFILE Datatype

The BFILE datatype enables access to binary file LOBs that are stored in file systems outside the Oracle database. A BFILE column or attribute stores a BFILE locator, which serves as a pointer to a binary file on the server's file system. The locator maintains the directory alias and the filename.

Binary file LOBs do not participate in transactions and are not recoverable. Rather, the underlying operating system provides file integrity and durability. The maximum file size supported is 4 gigabytes.

The database administrator must ensure that the file exists and that Oracle processes have operating system read permissions on the file.

The BFILE datatype allows read-only support of large binary files. You cannot modify or replicate such a file. Oracle provides APIs to access file data. The primary interfaces that you use to access file data are the DBMS_LOB package and the OCI.

See Also:

- *Oracle8i Application Developer's Guide - Large Objects (LOBs)* and *Oracle Call Interface Programmer's Guide* for more information about LOBs.
- CREATE DIRECTORY on page 9-40

BLOB Datatype

The BLOB datatype stores unstructured binary large objects. BLOBs can be thought of as bitstreams with no character set semantics. BLOBs can store up to 4 gigabytes of binary data.

BLOBs have full transactional support. Changes made through SQL, the DBMS_LOB package, or the OCI participate fully in the transaction. BLOB value manipulations can be committed and rolled back. Note, however, that you cannot save a BLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

CLOB Datatype

The CLOB datatype stores single-byte character data. Both fixed-width and variable-width character sets are supported, and both use the CHAR database character set. CLOBs can store up to 4 gigabytes of character data.

CLOBs have full transactional support. Changes made through SQL, the DBMS_LOB package, or the OCI participate fully in the transaction. CLOB value manipulations can be committed and rolled back. Note, however, that you cannot save a CLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

NCLOB Datatype

The NCLOB datatype stores multibyte national character set character (NCHAR) data. Both fixed-width and variable-width character sets are supported. NCLOBs can store up to 4 gigabytes of character text data.

NCLOBs have full transactional support. Changes made through SQL, the DBMS_LOB package, or the OCI participate fully in the transaction. NCLOB value manipulations can be committed and rolled back. Note, however, that you cannot save an NCLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

ROWID Datatype

Each row in the database has an address. You can examine a row's address by querying the pseudocolumn ROWID. Values of this pseudocolumn are hexadecimal strings representing the address of each row. These strings have the datatype ROWID. You can also create tables and clusters that contain actual columns having the ROWID datatype. Oracle does not guarantee that the values of such columns are valid rowids.

> **See Also:** "Pseudocolumns" on page 2-59 for more information on the ROWID pseudocolumn

Restricted Rowids

Beginning with Oracle8, Oracle SQL incorporated an extended format for rowids to efficiently support partitioned tables and indexes and tablespace-relative data block addresses (DBAs) without ambiguity.

Character values representing rowids in Oracle7 and earlier releases are called **restricted** rowids. Their format is as follows:

```
block.row.file
```

where:

block is a hexadecimal string identifying the data block of the datafile containing the row. The length of this string depends on your operating system.

row is a four-digit hexadecimal string identifying the row in the data block. The first row of the block has a digit of 0.

file is a hexadecimal string identifying the database file containing the row. The first datafile has the number 1. The length of this string depends on your operating system.

Extended Rowids

The **extended** ROWID datatype stored in a user column includes the data in the restricted rowid plus a **data object number**. The data object number is an identification number assigned to every database segment. You can retrieve the data object number from data dictionary views USER_OBJECTS, DBA_OBJECTS, and ALL_OBJECTS. Objects that share the same segment (clustered tables in the same cluster, for example) have the same object number.

Extended rowids are stored as base 64 values that can contain the characters A-Z, a-z, 0-9, as well as the plus sign (+) and forward slash (/). Extended rowids are not available directly. You can use a supplied package, DBMS_ROWID, to interpret extended rowid contents. The package functions extract and provide information that would be available directly from a restricted rowid, as well as information specific to extended rowids.

> **See Also:** *Oracle8i Supplied PL/SQL Packages Reference* for information on the functions available with the DBMS_ROWID package and how to use them

Compatibility and Migration

The restricted form of a rowid is still supported in Oracle8*i* for backward compatibility, but all tables return rowids in the extended format.

> **See Also:** *Oracle8i Migration* for information regarding compatibility and migration issues

UROWID Datatype

Each row in a database has an address. However, the rows of some tables have addresses that are not physical or permanent, or were not generated by Oracle. For example, the row addresses of index-organized tables are stored in index leaves, which can move. Rowids of foreign tables (such as DB2 tables accessed through a gateway) are not standard Oracle rowids.

Oracle uses "universal rowids" (**urowids**) to store the addresses of index-organized and foreign tables. Index-organized tables have logical urowids and foreign tables have foreign urowids. Both types of urowid are stored in the ROWID pseudocolumn (as are the physical rowids of heap-organized tables).

Oracle creates logical rowids based on a table's primary key. The logical rowids do not change as long as the primary key does not change. The ROWID pseudocolumn of an index-organized table has a datatype of UROWID. You can access this pseudocolumn as you would the ROWID pseudocolumn of a heap-organized (that is, using the SELECT ROWID statement). If you wish to store the rowids of an index-organized table, you can define a column of type UROWID for the table and retrieve the value of the ROWID pseudocolumn into that column.

Note: Heap-organized tables have physical rowids. Oracle Corporation does not recommend that you specify a column of datatype UROWID for a heap-organized table.

See Also:

- *Oracle8i Concepts* and *Oracle8i Performance Guide and Reference* for more information on the UROWID datatype and how Oracle generates and manipulates universal rowids
- "ROWID Datatype" on page 2-20 for a discussion of the address of database rows

ANSI, DB2, and SQL/DS Datatypes

SQL statements that create tables and clusters can also use ANSI datatypes and datatypes from IBM's products SQL/DS and DB2. Oracle recognizes the ANSI or IBM datatype name and records it as the name of the datatype of the column, and then stores the column's data in an Oracle datatype based on the conversions shown in Table 2–2 and Table 2–3.

Table 2–2 ANSI Datatypes Converted to Oracle Datatypes

ANSI SQL Datatype	Oracle Datatype
CHARACTER(n)	CHAR(n)
CHAR(n)	
CHARACTER VARYING(n)	VARCHAR(n)
CHAR VARYING(n)	
NATIONAL CHARACTER(n)	NCHAR(n)
NATIONAL CHAR(n)	
NCHAR(n)	

Table 2–2 ANSI Datatypes Converted to Oracle Datatypes

NATIONAL CHARACTER VARYING(n)	NVARCHAR2(n)
NATIONAL CHAR VARYING(n)	
NCHAR VARYING(n)	
NUMERIC(p,s)	NUMBER(p,s)
DECIMAL(p,s)[a]	
INTEGER	NUMBER(38)
INT	
SMALLINT	
FLOAT(b)[b]	NUMBER
DOUBLE PRECISION[c]	
REAL[d]	

[a]The NUMERIC and DECIMAL datatypes can specify only fixed-point numbers. For these datatypes, s defaults to 0.

[b]The FLOAT datatype is a floating-point number with a binary precision b. The default precision for this datatype is 126 binary, or 38 decimal.

[c]The DOUBLE PRECISION datatype is a floating-point number with binary precision 126.

[d]The REAL datatype is a floating-point number with a binary precision of 63, or 18 decimal.

Table 2–3 SQL/DS and DB2 Datatypes Converted to Oracle Datatypes

SQL/DS or DB2 Datatype	Oracle Datatype
CHARACTER(n)	CHAR(n)
VARCHAR(n)	VARCHAR(n)
LONG VARCHAR(n)	LONG
DECIMAL(p,s)[a]	NUMBER(p,s)
INTEGER	NUMBER(38)
SMALLINT	
FLOAT(b)b	NUMBER

Table 2–3 SQL/DS and DB2 Datatypes Converted to Oracle Datatypes

[a]The `DECIMAL` datatype can specify only fixed-point numbers. For this datatype, *s* defaults to 0.

[b]The `FLOAT` datatype is a floating-point number with a binary precision *b*. The default precision for this datatype is 126 binary, or 38 decimal.

Do not define columns with these SQL/DS and DB2 datatypes, because they have no corresponding Oracle datatype:

- `GRAPHIC`
- `LONG VARGRAPHIC`
- `VARGRAPHIC`
- `TIME`
- `TIMESTAMP`

Note that data of type `TIME` and `TIMESTAMP` can also be expressed as Oracle `DATE` data.

User-Defined Type Categories

User-defined datatypes use Oracle built-in datatypes and other user-defined datatypes as the building blocks of types that model the structure and behavior of data in applications.

The sections that follow describe the various categories of user-defined types.

See Also:

- *Oracle8i Concepts* for information about Oracle built-in datatypes
- CREATE TYPE on page 10-80 and the CREATE TYPE BODY on page 10-93 for information about creating user-defined types
- *Oracle8i Application Developer's Guide - Fundamentals* for information about using user-defined types

Object Types

Object types are abstractions of the real-world entities, such as purchase orders, that application programs deal with. An object type is a schema object with three kinds of components:

- A **name**, which identifies the object type uniquely within that schema
- **Attributes**, which are built-in types or other user-defined types. Attributes model the structure of the real-world entity
- **Methods**, which are functions or procedures written in PL/SQL and stored in the database, or written in a language like C or Java and stored externally. Methods implement operations the application can perform on the real-world entity.

REFs

An **object identifier** (OID) uniquely identifies an object and enables you to reference the object from other objects or from relational tables. A datatype category called REF represents such references. A REF is a container for an object identifier. REFs are pointers to objects.

When a REF value points to a nonexistent object, the REF is said to be "dangling". A dangling REF is different from a null REF. To determine whether a REF is dangling or not, use the predicate IS [NOT] DANGLING. For example, given table dept with column mgr whose type is a REF to type emp_t, which has an attribute name:

```
SELECT t.mgr.name
   FROM dept t
   WHERE t.mgr IS NOT DANGLING;
```

Varrays

An array is an ordered set of data elements. All elements of a given array are of the same datatype. Each element has an **index**, which is a number corresponding to the element's position in the array.

The number of elements in an array is the size of the array. Oracle arrays are of variable size, which is why they are called **varrays**. You must specify a maximum size when you declare the array.

When you declare a varray, it does not allocate space. It defines a type, which you can use as:

- The datatype of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type

Oracle normally stores an array object either in line (that is, as part of the row data) or out of line (in a LOB), depending on its size. However, if you specify separate

storage characteristics for a varray, Oracle will store it out of line, regardless of its size.

> **See Also:** The *varray_storage_clause* of CREATE TABLE on page 10-32

Nested Tables

A nested table type models an unordered set of elements. The elements may be built-in types or user-defined types. You can view a nested table as a single-column table or, if the nested table is an object type, as a multicolumn table, with a column for each attribute of the object type.

A nested table definition does not allocate space. It defines a type, which you can use to declare:

- Columns of a relational table
- Object type attributes
- PL/SQL variables, parameters, and function return values

When a nested table appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table.

Datatype Comparison Rules

This section describes how Oracle compares values of each datatype.

Number Values

A larger value is considered greater than a smaller one. All negative numbers are less than zero and all positive numbers. Thus, -1 is less than 100; -100 is less than -1.

Date Values

A later date is considered greater than an earlier one. For example, the date equivalent of '29-MAR-1997' is less than that of '05-JAN-1998' and '05-JAN-1998 1:35pm' is greater than '05-JAN-1998 10:09am'.

Character String Values

Character values are compared using one of these comparison rules:

- blank-padded comparison semantics
- nonpadded comparison semantics

The following sections explain these comparison semantics. The results of comparing two character values using different comparison semantics may vary. The table below shows the results of comparing five pairs of character values using each comparison semantic. Usually, the results of blank-padded and nonpadded comparisons are the same. The last comparison in the table illustrates the differences between the blank-padded and nonpadded comparison semantics.

Blank-Padded	Nonpadded
`'ab' > 'aa'`	`'ab' > 'aa'`
`'ab' > 'a  '`	`'ab' > 'a  '`
`'ab' > 'a'`	`'ab' > 'a'`
`'ab' = 'ab'`	`'ab' = 'ab'`
'a ' = 'a'	**'a ' > 'a'**

Blank-Padded Comparison Semantics If the two values have different lengths, Oracle first adds blanks to the end of the shorter one so their lengths are equal. Oracle then compares the values character by character up to the first character that differs. The value with the greater character in the first differing position is considered greater. If two values have no differing characters, then they are considered equal. This rule means that two values are equal if they differ only in the number of trailing blanks. Oracle uses blank-padded comparison semantics only when both values in the comparison are either expressions of datatype CHAR, NCHAR, text literals, or values returned by the USER function.

Nonpadded Comparison Semantics Oracle compares two values character by character up to the first character that differs. The value with the greater character in that position is considered greater. If two values of different length are identical up to the end of the shorter one, the longer value is considered greater. If two values of equal length have no differing characters, then the values are considered equal. Oracle uses nonpadded comparison semantics whenever one or both values in the comparison have the datatype VARCHAR2 or NVARCHAR2.

Single Characters

Oracle compares single characters according to their numeric values in the database character set. One character is greater than another if it has a greater numeric value than the other in the character set. Oracle considers blanks to be less than any character, which is true in most character sets.

These are some common character sets:

- 7-bit ASCII (American Standard Code for Information Interchange)
- EBCDIC Code (Extended Binary Coded Decimal Interchange Code)
- ISO 8859 / 1 (International Standards Organization)
- JEUC Japan Extended UNIX

Portions of the ASCII and EBCDIC character sets appear in Table 2–4 and Table 2–5. Note that uppercase and lowercase letters are not equivalent. Also, note that the numeric values for the characters of a character set may not match the linguistic sequence for a particular language.

Table 2–4 ASCII Character Set

Symbol	Decimal value	Symbol	Decimal value
blank	32	;	59
!	33	<	60
"	34	=	61
#	35	>	62
$	36	?	63
%	37	@	64
&	38	A-Z	65-90
’	39	[	91
(	40	\	92
)	41	]	93
*	42	^	94
+	43	_	95
,	44	‘	96
-	45	a-z	97-122

Table 2–4 (Cont.) ASCII Character Set

Symbol	Decimal value	Symbol	Decimal value
.	46	{	123
/	47	|	124
0-9	48-57	}	125
:	58	~	126

Table 2–5 EBCDIC Character Set

Symbol	Decimal value	Symbol	Decimal value
blank	64	%	108
¢	74	_	109
.	75	>	110
<	76	?	111
(	77	:	122
+	78	#	123
|	79	@	124
&	80	’	125
!	90	=	126
$	91	"	127
*	92	a-i	129-137
)	93	j-r	145-153
;	94	s-z	162-169
ÿ	95	A-I	193-201
-	96	J-R	209-217
/	97	S-Z	226-233

Object Values

Object values are compared using one of two comparison functions: MAP and ORDER. Both functions compare object type instances, but they are quite different from one another. These functions must be specified as part of the object type.

> **See Also:** "CREATE TYPE" on page 10-80 and *Oracle8i Application Developer's Guide - Fundamentals* for a description of MAP and ORDER methods and the values they return

Varrays and Nested Tables

You cannot compare varrays and nested tables in Oracle8*i*.

Data Conversion

Generally an expression cannot contain values of different datatypes. For example, an expression cannot multiply 5 by 10 and then add 'JAMES'. However, Oracle supports both implicit and explicit conversion of values from one datatype to another.

Implicit Data Conversion

Oracle automatically converts a value from one datatype to another when such a conversion makes sense. Oracle performs conversions in these cases:

- When an INSERT or UPDATE statement assigns a value of one datatype to a column of another, Oracle converts the value to the datatype of the column.
- When you use a SQL function or operator with an argument with a datatype other than the one it accepts, Oracle converts the argument to the accepted datatype.
- When you use a comparison operator on values of different datatypes, Oracle converts one of the expressions to the datatype of the other.

Implicit Data Conversion Examples

Text Literal Example The text literal '10' has datatype CHAR. Oracle implicitly converts it to the NUMBER datatype if it appears in a numeric expression as in the following statement:

```
SELECT sal + '10'
    FROM emp;
```

Character and Number Values Example When a condition compares a character value and a NUMBER value, Oracle implicitly converts the character value to a NUMBER value, rather than converting the NUMBER value to a character value. In the following statement, Oracle implicitly converts '7936' to 7936:

```
SELECT ename
    FROM emp
    WHERE empno = '7936';
```

Date Example In the following statement, Oracle implicitly converts '12-MAR-1993' to a DATE value using the default date format 'DD-MON-YYYY':

```
SELECT ename
    FROM emp
    WHERE hiredate = '12-MAR-1993';
```

Rowid Example In the following statement, Oracle implicitly converts the text literal 'AAAAZ8AABAAABvlAAA' to a rowid value:

```
SELECT ename
    FROM emp
    WHERE ROWID = 'AAAAZ8AABAAABvlAAA';
```

Explicit Data Conversion

You can also explicitly specify datatype conversions using SQL conversion functions. Table 2–6 shows SQL functions that explicitly convert a value from one datatype to another.

Table 2–6 SQL Functions for Datatype Conversion

TO / FROM	CHAR	NUMBER	DATE	RAW	ROWID	LONG/ LONG RAW	LOB
CHAR	–	TO_NUMBER	TO_DATE	HEXTORAW	CHARTO-ROWID		
NUMBER	TO_CHAR	–	TO_DATE (number,'J')				
DATE	TO_CHAR	TO_CHAR (date,'J')	–				
RAW	RAWTOHEX			–			
ROWID	ROWID-TOCHAR				–		
LONG / LONG RAW						–	TO_LOB
LOB							–

Note: You cannot specify LONG and LONG RAW values in cases in which Oracle can perform implicit datatype conversion. For example, LONG and LONG RAW values cannot appear in expressions with functions or operators. For information on the limitations on LONG and LONG RAW datatypes, see "LONG Datatype" on page 2-12.

See Also: "Conversion Functions" on page 4-5

Implicit vs. Explicit Data Conversion

Oracle recommends that you specify explicit conversions rather than rely on implicit or automatic conversions for these reasons:

- SQL statements are easier to understand when you use explicit datatype conversion functions.
- Automatic datatype conversion can have a negative impact on performance, especially if the datatype of a column value is converted to that of a constant rather than the other way around.

- Implicit conversion depends on the context in which it occurs and may not work the same way in every case. For example, implicit conversion from a date value to a VARCHAR2 value may return an unexpected year depending on the value of the NLS_DATE_FORMAT parameter.
- Algorithms for implicit conversion are subject to change across software releases and among Oracle products. Behavior of explicit conversions is more predictable.

Literals

The terms **literal** and **constant value** are synonymous and refer to a fixed data value. For example, 'JACK', 'BLUE ISLAND', and '101' are all character literals; 5001 is a numeric literal. Note that character literals are enclosed in single quotation marks, which enable Oracle to distinguish them from schema object names.

This section contains these topics:

Many SQL statements and functions require you to specify character and numeric literal values. You can also specify literals as part of expressions and conditions. You can specify character literals with the '*text*' notation, national character literals with the *N'text*' notation, and numeric literals with the *integer* or *number* notation, depending on the context of the literal. The syntactic forms of these notations appear in the sections that follow.

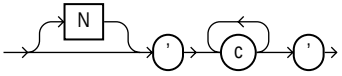
To specify a datetime or interval datatype as a literal, you must take into account any optional precisions included in the datatypes. Examples of specifying datetime and interval datatypes as literals are provided in the relevant sections of "Datatypes" on page 2-2.

Text

Text specifies a text or character literal. You must use this notation to specify values whenever *'text'* or *char* appear in expressions, conditions, SQL functions, and SQL statements in other parts of this reference.

The syntax of text is as follows:

text::=



where

- N specifies representation of the literal using the national character set. Text entered using this notation is translated into the national character set by Oracle when used.
- *c* is any member of the user's character set, except a single quotation mark (').
- ' ' are two single quotation marks that begin and end text literals. To represent one single quotation mark within a literal, enter two single quotation marks.

A text literal must be enclosed in single quotation marks. This reference uses the terms **text literal** and **character literal** interchangeably.

Text literals have properties of both the CHAR and VARCHAR2 datatypes:

- Within expressions and conditions, Oracle treats text literals as though they have the datatype CHAR by comparing them using blank-padded comparison semantics.
- A text literal can have a maximum length of 4000 bytes.

Here are some valid text literals:

```
'Hello'
'ORACLE.dbs'
'Jackie''s raincoat'
'09-MAR-98'
N'nchar literal'
```

See Also:

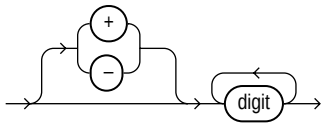
- "Expressions" on page 5-2 for the syntax description of *expr*
- "Blank-Padded Comparison Semantics" on page 2-27

Integer

You must use the integer notation to specify an integer whenever *integer* appears in expressions, conditions, SQL functions, and SQL statements described in other parts of this reference.

The syntax of *integer* is as follows:

integer::=



where *digit* is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

An integer can store a maximum of 38 digits of precision.

Here are some valid integers:

```
7
+255
```

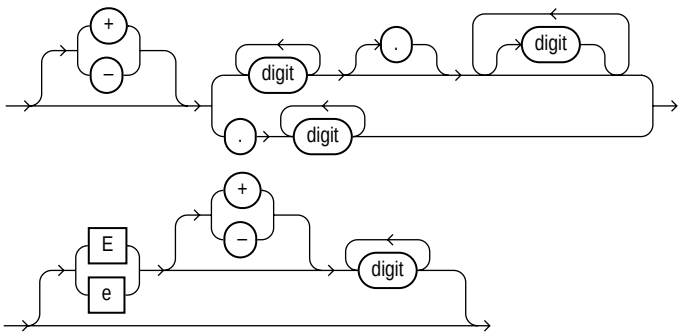
> **See Also:** "Expressions" on page 5-2 for the syntax description of *expr*

Number

You must use the number notation to specify values whenever *number* appears in expressions, conditions, SQL functions, and SQL statements in other parts of this reference.

The syntax of *number* is as follows:

number::=



where

- + or - indicates a positive or negative value. If you omit the sign, a positive value is the default.
- *digit* is one of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9.
- e or E indicates that the number is specified in scientific notation. The digits after the E specify the exponent. The exponent can range from -130 to 125.

A *number* can store a maximum of 38 digits of precision.

If you have established a decimal character other than a period (.) with the initialization parameter NLS_NUMERIC_CHARACTERS, you must specify numeric literals with *'text'* notation. In such cases, Oracle automatically converts the text literal to a numeric value.

For example, if the NLS_NUMERIC_CHARACTERS parameter specifies a decimal character of comma, specify the number 5.123 as follows:

```
'5,123'
```

> **See Also:** ALTER SESSION on page 7-105 and *Oracle8i Reference*

Here are some valid representations of *number*:

```
25
+6.34
0.5
25e-03
-1
```

> **See Also:** "Expressions" on page 5-2 for the syntax description of *expr*

Interval

An interval literal specifies a period of time. You can specify these differences in terms of years and months, or in terms of days, hours, minutes, and seconds. Oracle supports two types of interval literals, YEAR TO MONTH and DAY TO SECOND. Each type contains a leading field and may contain a trailing field. The leading field defines the basic unit of date or time being measured. The trailing field defines the smallest increment of the basic unit being considered. For example, a YEAR TO MONTH interval considers an interval of years to the nearest month. A DAY TO MINUTE interval considers an interval of days to the nearest minute.

If you have date data in numeric form, you can use the NUMTOYMINTERVAL or NUMTODSINTERVAL conversion function to convert the numeric data into interval literals.

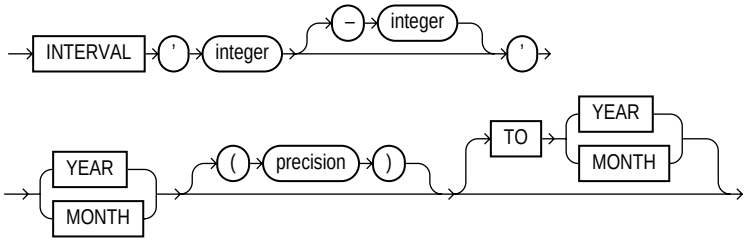
Interval literals are used primarily with analytic functions.

See Also:

- "Analytic Functions" on page 4-8 and *Oracle8i Data Warehousing Guide*
- "NUMTODSINTERVAL" on page 4-69 and "NUMTOYMINTERVAL" on page 4-70

INTERVAL YEAR TO MONTH

Specify YEAR TO MONTH interval literals using the following syntax:



where

- *'integer [-integer]'* specifies integer values for the leading and optional trailing field of the literal. If the leading field is YEAR and the trailing field is MONTH, the range of integer values for the month field is 0 to 11.
- *precision* is the number of digits in the leading field. The valid range of the leading field precision is 0 to 9 and its default value is 2.

Restriction: The leading field must be a larger time element than the trailing field. For example, INTERVAL '0-1' MONTH TO YEAR is not valid.

The following INTERVAL YEAR TO MONTH literal indicates an interval of 123 years, 2 months:

```
INTERVAL '123-2' YEAR(3) TO MONTH
```

Examples of the other forms of the literal follow, including some abbreviated versions:

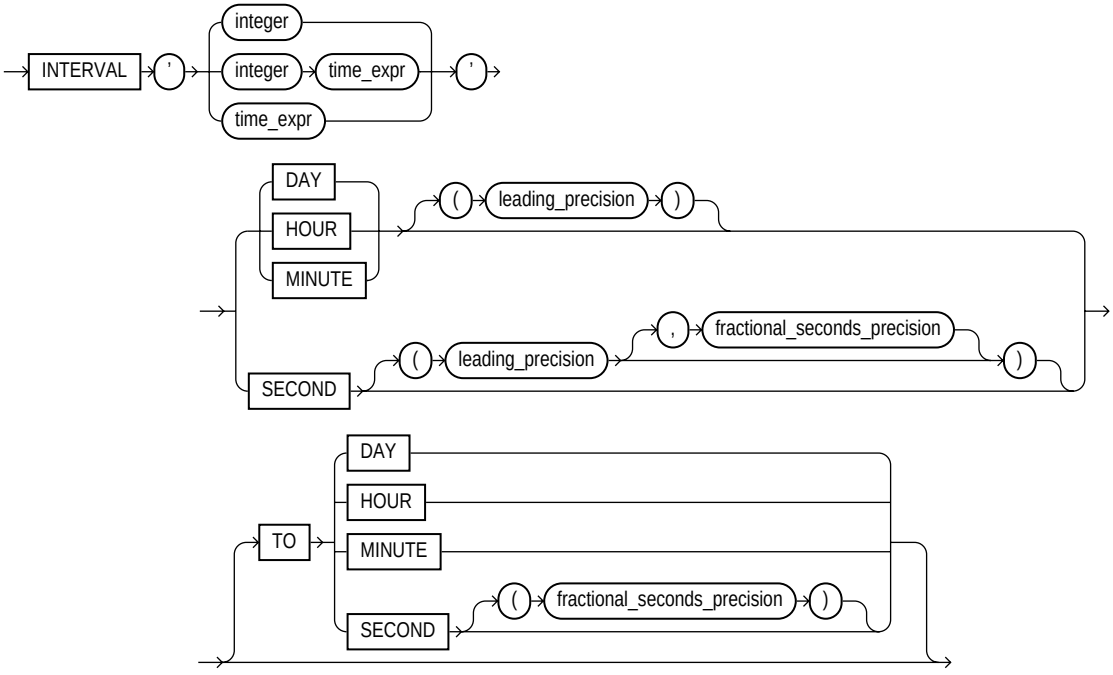
`INTERVAL '123-2' YEAR(3) TO MONTH`	indicates an interval of 123 years, 2 months. You must specify the leading field precision if it is greater than the default of 2 digits.
`INTERVAL '123' YEAR(3)`	indicates an interval of 123 years 0 months.
`INTERVAL '300' MONTH(3)`	indicates an interval of 300 months.
`INTERVAL '4' YEAR`	maps to `INTERVAL '4-0' YEAR TO MONTH` and indicates 4 years.
`INTERVAL '50' MONTH`	maps to `INTERVAL '4-2' YEAR TO MONTH` and indicates 50 months or 4 years 2 months.
`INTERVAL '123' YEAR`	returns an error, because the default precision is 2, and '123' has 3 digits.

You can add or subtract one `INTERVAL YEAR TO MONTH` literal to or from another to yield another `INTERVAL YEAR TO MONTH` literal. For example:

```
INTERVAL '5-3' YEAR TO MONTH + INTERVAL '20' MONTH TO MONTH =
INTERVAL '6-11' YEAR TO MONTH
```

INTERVAL DAY TO SECOND

Specify DAY TO SECOND interval literals using the following syntax:



where

- *integer* specifies the number of days. If this value contains more digits than the number specified by the leading precision, Oracle returns an error.
- *time_expr* specifies a time in the format HH[:MI[:SS[.n]]]or MI[:SS[.n]] or SS[.n], where *n* specifies the fractional part of a second. If *n* contains more digits than the number specified by *fractional_seconds_precision*, then *n* is rounded to the number of digits specified by the *fractional_seconds_precision* value. You can specify *time_expr* following an integer and a space only if the leading field is DAY.
- *leading_precision* is the number of digits in the leading field. Accepted values are 0 to 9. The default is 2.
- *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 1 to 9. The default is 6.

Restriction: The leading field must be a larger time element than the trailing field. For example, INTERVAL MINUTE TO DAY is not valid. As a result of this restriction, if SECOND is the leading field, the interval literal cannot have any trailing field.

The valid range of values for the trailing field are as follows:

HOUR	0 to 23
MINUTE	0 to 59
SECOND	0 to 59.999999999

Examples of the various forms of INTERVAL DAY TO SECOND literals follow, including some abbreviated versions:

INTERVAL '4 5:12:10.222' DAY(3) TO SECOND(3)	indicates 4 days, 5 hours, 12 minutes, 10 seconds, and 222 thousandths of a second.
INTERVAL '4 5:12' DAY TO MINUTE	indicates 4 days, 5 hours and 12 minutes.
INTERVAL '400 5' DAY(3) TO HOUR	indicates 400 days 5 hours.
INTERVAL '400' DAY(3)	indicates 400 days.
INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)	indicates 11 hours, 12 minutes, and 10.2222222 seconds.
INTERVAL '11:20' HOUR TO MINUTE	indicates 11 hours and 20 minutes.
INTERVAL '10' HOUR	indicates 10 hours.
INTERVAL '10:22' MINUTE TO SECOND	indicates 10 minutes 22 seconds.
INTERVAL '10' MINUTE	indicates 10 minutes.
INTERVAL '4' DAY	indicates 4 days.
INTERVAL '25' HOUR	indicates 25 hours.
INTERVAL '40' MINUTE	indicates 40 minutes.
INTERVAL '120' HOUR(3)	indicates 120 hours
INTERVAL '30.12345' SECOND(2,4)	indicates 30.1235 seconds. The fractional second '12345' is rounded to '1235' because the precision is 4.

You can add or subtract one DAY TO SECOND interval literal from another DAY TO SECOND literal. For example.

```
INTERVAL '20' DAY - INTERVAL '240' HOUR = INTERVAL '10' DAY
```

Format Models

A **format model** is a character literal that describes the format of DATE or NUMBER data stored in a character string. When you convert a character string into a date or number, a format model tells Oracle how to interpret the string. In SQL statements, you can use a format model as an argument of the TO_CHAR and TO_DATE functions:

- To specify the format for Oracle to use to return a value from the database
- To specify the format for a value you have specified for Oracle to store in the database

---

Note: A format model does not change the internal representation of the value in the database.

For example, the date format model for the string '17:45:29' is 'HH24:MI:SS'. The date format model for the string '11-Nov-1999' is 'DD-Mon-YYYY'. The number format model for the string '$2,304.25' is '$9,999.99'. For lists of date and number format model elements, see Table 2–7, "Number Format Elements" on page 2-44 and Table 2–9, "Datetime Format Elements" on page 2-49.

The values of some formats are determined by the value of initialization parameters. For such formats, you can specify the characters returned by these format elements implicitly using the initialization parameter NLS_TERRITORY. You can change the default date format for your session with the ALTER SESSION statement.

See Also:

- *Oracle8i Reference* and *Oracle8i National Language Support Guide* for information on these parameters
- ALTER SESSION on page 7-105 for information on changing the values of these parameters

Format of Return Values: Examples You can use a format model to specify the format for Oracle to use to return values from the database to you.

The following statement selects the commission values of the employees in Department 30 and uses the TO_CHAR function to convert these commissions into character values with the format specified by the number format model '$9,990.99':

```
SELECT ename employee, TO_CHAR(comm, '$9,990.99') commission
   FROM emp
   WHERE deptno = 30;

EMPLOYEE   COMMISSION
---------- ----------
ALLEN         $300.00
WARD          $500.00
MARTIN      $1,400.00
BLAKE
TURNER          $0.00
JAMES
```

Because of this format model, Oracle returns commissions with leading dollar signs, commas every three digits, and two decimal places. Note that TO_CHAR returns null for all employees with null in the comm column.

The following statement selects the date on which each employee from Department 20 was hired and uses the TO_CHAR function to convert these dates to character strings with the format specified by the date format model 'fmMonth DD, YYYY':

```
 SELECT ename, TO_CHAR(Hiredate,'fmMonth DD, YYYY') hiredate
    FROM emp
    WHERE deptno = 20;

ENAME      HIREDATE
---------- ------------------
SMITH      December 17, 1980
JONES      April 2, 1981
SCOTT      April 19, 1987
ADAMS      May 23, 1987
FORD       December 3, 1981
LEWIS      October 23, 1997
```

With this format model, Oracle returns the hire dates (as specified by "fm") without blank padding, two digits for the day, and the century included in the year.

> **See Also:** "Format Model Modifiers" on page 2-54 for a description of the fm format element

Supplying the Correct Format Model: Examples When you insert or update a column value, the datatype of the value that you specify must correspond to the column's datatype. You can use format models to specify the format of a value that you are converting from one datatype to another datatype required for a column.

For example, a value that you insert into a DATE column must be a value of the DATE datatype or a character string in the default date format (Oracle implicitly converts character strings in the default date format to the DATE datatype). If the value is in another format, you must use the TO_DATE function to convert the value to the DATE datatype. You must also use a format model to specify the format of the character string.

The following statement updates BAKER's hire date using the TO_DATE function with the format mask 'YYYY MM DD' to convert the character string '1998 05 20' to a DATE value:

```
UPDATE emp
  SET hiredate = TO_DATE('1998 05 20','YYYY MM DD')
  WHERE ename = 'BLAKE';
```

This remainder of this section describes how to use:

- Number Format Models
- Date Format Models
- Format Model Modifiers

> **See Also:** "TO_CHAR (date conversion)" on page 4-108, "TO_CHAR (number conversion)" on page 4-109, and "TO_DATE" on page 4-110

Number Format Models

You can use number format models:

- In the TO_CHAR function to translate a value of NUMBER datatype to VARCHAR2 datatype
- In the TO_NUMBER function to translate a value of CHAR or VARCHAR2 datatype to NUMBER datatype

All number format models cause the number to be rounded to the specified number of significant digits. If a value has more significant digits to the left of the decimal place than are specified in the format, pound signs (#) replace the value. If a positive value is extremely large and cannot be represented in the specified format, then the

infinity sign (~) replaces the value. Likewise, if a negative value is extremely small and cannot be represented by the specified format, then the negative infinity sign replaces the value (-~). This event typically occurs when you are using TO_CHAR with a restrictive number format string, causing a rounding operation.

Number Format Elements

A number format model is composed of one or more number format elements. Table 2–7 lists the elements of a number format model. Examples are shown in Table 2–8.

Negative return values automatically contain a leading negative sign and positive values automatically contain a leading space unless the format model contains the MI, S, or PR format element.

Table 2–7 Number Format Elements

Element	Example	Description
, (comma)	9,999	Returns a comma in the specified position. You can specify multiple commas in a number format model. **Restrictions:** ■ A comma element cannot begin a number format model. ■ A comma cannot appear to the right of a decimal character or period in a number format model.
. (period)	99.99	Returns a decimal point, which is a period (.) in the specified position. **Restriction:** You can specify only one period in a number format model.
$	$9999	Returns value with a leading dollar sign.
0	0999	Returns leading zeros.
	9990	Returns trailing zeros.
9	9999	Returns value with the specified number of digits with a leading space if positive or with a leading minus if negative. Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed-point number.
B	B9999	Returns blanks for the integer part of a fixed-point number when the integer part is zero (regardless of "0"s in the format model).
C	C999	Returns in the specified position the ISO currency symbol (the current value of the NLS_ISO_CURRENCY parameter).

Table 2–7 Number Format Elements

Element	Example	Description
D	99D99	Returns in the specified position the decimal character, which is the current value of the NLS_NUMERIC_CHARACTER parameter. The default is a period (.).
		Restriction: You can specify only one decimal character in a number format model.
EEEE	9.9EEEE	Returns a value using in scientific notation.
FM	FM90.9	Returns a value with no leading or trailing blanks.
G	9G999	Returns in the specified position the group separator (the current value of the NLS_NUMERIC_CHARACTER parameter). You can specify multiple group separators in a number format model.
		Restriction: A group separator cannot appear to the right of a decimal character or period in a number format model.
L	L999	Returns in the specified position the local currency symbol (the current value of the NLS_CURRENCY parameter).
MI	9999MI	Returns negative value with a trailing minus sign (-).
		Returns positive value with a trailing blank.
		Restriction: The MI format element can appear only in the last position of a number format model.
PR	9999PR	Returns negative value in <angle brackets>.
		Returns positive value with a leading and trailing blank.
		Restriction: The PR format element can appear only in the last position of a number format model.
RN	RN	Returns a value as Roman numerals in uppercase.
rn	rn	Returns a value as Roman numerals in lowercase.
		Value can be an integer between 1 and 3999.
S	S9999	Returns negative value with a leading minus sign (-).
		Returns positive value with a leading plus sign (+).
	9999S	Returns negative value with a trailing minus sign (-).
		Returns positive value with a trailing plus sign (+).
		Restriction: The S format element can appear only in the first or last position of a number format model.

Table 2–7 Number Format Elements

Element	Example	Description
TM	`TM`	"Text minimum". Returns (in decimal output) the smallest number of characters possible. This element is case-insensitive. The default is TM9, which returns the number in fixed notation unless the output exceeds 64 characters. If output exceeds 64 characters, Oracle automatically returns the number in scientific notation. **Restrictions**: ■ You cannot precede this element with any other element. ■ You can follow this element only with 9 or E (only one) or e (only one).
U	`U9999`	Returns in the specified position the "Euro" (or other) dual currency symbol (the current value of the `NLS_DUAL_CURRENCY` parameter).
V	`999V99`	Returns a value multiplied by 10^n (and if necessary, round it up), where *n* is the number of 9's after the "V".
X	`XXXX` `xxxx`	Returns the hexadecimal value of the specified number of digits. If the specified number is not an integer, Oracle rounds it to an integer. **Restrictions**: ■ This element accepts only positive values or 0. Negative values return an error. ■ You can precede this element only with 0 (which returns leading zeroes) or FM. Any other elements return an error. If you specify neither 0 nor FM with X, the return always has 1 leading blank.

Table 2–8 shows the results of the following query for different values of *number* and *'fmt'*:

```
SELECT TO_CHAR(number, 'fmt')
   FROM DUAL;
```

Table 2–8 Results of Example Number Conversions

number	'fmt'	Result
-1234567890	9999999999S	'1234567890-'
0	99.99	' .00'

Table 2–8 Results of Example Number Conversions (Cont.)

number	'fmt'	Result
+0.1	99.99	' .10'
-0.2	99.99	' -.20'
0	90.99	' 0.00'
+0.1	90.99	' 0.10'
-0.2	90.99	' -0.20'
0	9999	' 0'
1	9999	' 1'
0	B9999	' '
1	B9999	' 1'
0	B90.99	' '
+123.456	999.999	' 123.456'
-123.456	999.999	'-123.456'
+123.456	FM999.009	'123.456'
+123.456	9.9EEEE	' 1.2E+02'
+1E+123	9.9EEEE	' 1.0E+123'
+123.456	FM9.9EEEE	'1.2E+02'
+123.45	FM999.009	'123.45'
+123.0	FM999.009	'123.00'
+123.45	L999.99	' $123.45'
+123.45	FML999.99	'$123.45'
+1234567890	9999999999S	'1234567890+'

Date Format Models

You can use date format models:

- In the TO_DATE function to translate a character value that is in a format other than the default date format into a DATE value

- In the TO_CHAR function to translate a DATE value that is in a format other than the default date format into a string (for example, to print the date from an application)

The default date format is specified either explicitly with the initialization parameter NLS_DATE_FORMAT or implicitly with the initialization parameter NLS_TERRITORY. For information on these parameters, see *Oracle8i Reference*.

You can change the default date format for your session with the ALTER SESSION statement.

See Also: ALTER SESSION on page 7-105

The total length of a date format model cannot exceed 22 characters.

Date Format Elements

A date format model is composed of one or more date format elements as listed in Table 2–9.

- For input format models, format items cannot appear twice, and format items that represent similar information cannot be combined. For example, you cannot use 'SYYYY' and 'BC' in the same format string.
- Some of the date format elements cannot be used in the TO_DATE function, as noted in Table 2–9.

Capitalization of Date Format Elements Capitalization in a spelled-out word, abbreviation, or Roman numeral follows capitalization in the corresponding format element. For example, the date format model 'DAY' produces capitalized words like 'MONDAY'; 'Day' produces 'Monday'; and 'day' produces 'monday'.

Punctuation and Character Literals in Date Format Models You can also include these characters in a date format model:

- Punctuation such as hyphens, slashes, commas, periods, and colons
- Character literals, enclosed in double quotation marks

These characters appear in the return value in the same location as they appear in the format model.

Table 2–9 Datetime Format Elements

Element	Specify in TO_DATE?	Meaning
- / , . ; : "text"	Yes	Punctuation and quoted text is reproduced in the result.
AD or A.D.	Yes	AD indicator with or without periods. **Note:** The indicator with periods is supported only if the NLS_LANGUAGE parameter is set to AMERICAN.
AM A.M.	Yes	Meridian indicator with or without periods. **Note:** The indicator with periods is supported only if the NLS_LANGUAGE parameter is set to AMERICAN.
BC B.C.	Yes	BC indicator with or without periods. **Note:** The indicator with periods is supported only if the NLS_LANGUAGE parameter is set to AMERICAN.
CC SCC	No	The first two digits of the century of a four-digit year, for example, '19' from '1900' and '20' from '2001'. "S" prefixes BC dates with "-".
D	Yes	Day of week (1-7). This element is used *only* to validate a date specified in the TO_DATE function.
DAY	Yes	Name of day, padded with blanks to length of 9 characters. This element is used *only* to validate a date specified in the TO_DATE function.
DD	Yes	Day of month (1-31).
DDD	Yes	Day of year (1-366).
DY	Yes	Abbreviated name of day. This element is used *only* to validate a date specified in the TO_DATE function.
E	Yes	Abbreviated era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
EE	Yes	Full era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
HH	Yes	Hour of day (1-12).

Table 2–9 Datetime Format Elements

Element	Specify in TO_DATE?	Meaning
HH12	Yes	Hour of day (1-12).
HH24	Yes	Hour of day (0-23).
IW	No	Week of year (1-52 or 1-53) based on the ISO standard.
IYY IY I	No	Last 3, 2, or 1 digit(s) of ISO year.
IYYY	No	4-digit year based on the ISO standard.
J	Yes	Julian day; the number of days since January 1, 4712 BC. Number specified with 'J' must be integers.
MI	Yes	Minute (0-59).
MM	Yes	Two-digit numeric abbreviation of month (01-12; JAN = 01)
MON	Yes	Abbreviated name of month.
MONTH	Yes	Name of month, padded with blanks to length of 9 characters.
PM P.M.	No	Meridian indicator with or without periods. **Note:** The indicator with periods is supported only if the NLS_LANGUAGE parameter is set to AMERICAN.
Q	No	Quarter of year (1, 2, 3, 4; JAN-MAR = 1)
RM	Yes	Roman numeral month (I-XII; JAN = I).
RR	Yes	Given a year with 2 digits: ■ If the year is <50 and the last 2 digits of the current year are >=50, the first 2 digits of the returned year are 1 greater than the first two digits of the current year. ■ If the year is >=50 and the last 2 digits of the current year are <50, the first 2 digits of the returned year are the same as the first 2 digits of the current year.
RRRR	Yes	Round year. Accepts either 4-digit or 2-digit input. If 2-digit, provides the same return as RR. If you don't want this functionality, enter the 4-digit year.
SS	Yes	Second (0-59).

Table 2–9 Datetime Format Elements

Element	Specify in TO_DATE?	Meaning
SSSSS	Yes	Seconds past midnight (0-86399).
WW	No	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	No	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
Y,YYY	Yes	Year with comma in this position.
YEAR SYEAR	No	Year, spelled out. "S" prefixes BC dates with "-".
YYYY SYYYY	Yes	4-digit year. "S" prefixes BC dates with "-".
YYY YY Y	Yes	Last 3, 2, or 1 digit(s) of year.

Oracle returns an error if an alphanumeric character is found in the date string where punctuation character is found in the format string. For example:

```
TO_CHAR (TO_DATE('0297','MM/YY'), 'MM/YY')
```

returns an error.

Date Format Elements and National Language Support

The functionality of some date format elements depends on the country and language in which you are using Oracle. For example, these date format elements return spelled values:

- MONTH
- MON
- DAY
- DY
- BC or AD or B.C. or A.D.
- AM or PM or A.M or P.M.

The language in which these values are returned is specified either explicitly with the initialization parameter NLS_DATE_LANGUAGE or implicitly with the initialization parameter NLS_LANGUAGE. The values returned by the YEAR and SYEAR date format elements are always in English.

The date format element D returns the number of the day of the week (1-7). The day of the week that is numbered 1 is specified implicitly by the initialization parameter NLS_TERRITORY.

> **See Also:** *Oracle8i Reference* and *Oracle8i National Language Support Guide* for information on national language support initialization parameters

ISO Standard Date Format Elements

Oracle calculates the values returned by the date format elements IYYY, IYY, IY, I, and IW according to the ISO standard. For information on the differences between these values and those returned by the date format elements YYYY, YYY, YY, Y, and WW, see the discussion of national language support in *Oracle8i National Language Support Guide*.

The RR Date Format Element

The RR date format element is similar to the YY date format element, but it provides additional flexibility for storing date values in other centuries. The RR date format element allows you to store 21st century dates in the 20th century by specifying only the last two digits of the year. It will also allow you to store 20th century dates in the 21st century in the same way if necessary.

If you use the TO_DATE function with the YY date format element, the date value returned always has the same first 2 digits as the current year. If you use the RR date format element instead, the century of the return value varies according to the specified two-digit year and the last two digits of the current year. Table 2–10 summarizes the behavior of the RR date format element.

Table 2–10 The RR Date Element Format

		If the specified two-digit year is	
		0 - 49	50 - 99
If the last two digits of the current year are:	0-49	The return date has the same first 2 digits as the current date.	The first 2 digits of the return date are 1 less than the first 2 digits of the current date.
	50-99	The first 2 digits of the return date are 1 greater than the first 2 digits of the current date.	The return date has the same first 2 digits as the current date.

The following examples demonstrate the behavior of the RR date format element.

RR Date Format Examples

Assume these queries are issued between 1950 and 1999:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR') ,'YYYY') "Year"
     FROM DUAL;

Year
----
1998

SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR') ,'YYYY') "Year"
     FROM DUAL;

Year
----
2017
```

Now assume these queries are issued between 2000 and 2049:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR') ,'YYYY') "Year"
  FROM DUAL;

Year
----
1998

SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR') ,'YYYY') "Year"
     FROM DUAL;
```

```
Year
----
2017
```

Note that the queries return the same values regardless of whether they are issued before or after the year 2000. The RR date format element allows you to write SQL statements that will return the same values from years whose first two digits are different.

Date Format Element Suffixes

Table 2–11 lists suffixes that can be added to date format elements:

Table 2–11 Date Format Element Suffixes

Suffix	Meaning	Example Element	Example Value
TH	Ordinal Number	DDTH	4TH
SP	Spelled Number	DDSP	FOUR
SPTH or THSP	Spelled, ordinal number	DDSPTH	FOURTH

Restrictions:

- When you add one of these suffixes to a date format element, the return value is always in English.
- Date suffixes are valid only on output. You cannot use them to insert a date into the database.

Format Model Modifiers

The FM and FX modifiers, used in format models in the TO_CHAR function, control blank padding and exact format checking.

A modifier can appear in a format model more than once. In such a case, each subsequent occurrence toggles the effects of the modifier. Its effects are enabled for the portion of the model following its first occurrence, and then disabled for the portion following its second, and then reenabled for the portion following its third, and so on.

FM "Fill mode". This modifier suppresses blank padding in the return value of the TO_CHAR function:

- In a date format element of a TO_CHAR function, this modifier suppresses blanks in subsequent character elements (such as MONTH) and suppresses leading zeroes for subsequent number elements (such as MI) in a date format

model. Without FM, the result of a character element is always right padded with blanks to a fixed length, and leading zeroes are always returned for a number element. With FM, because there is no blank padding, the length of the return value may vary.

- In a number format element of a TO_CHAR function, this modifier suppresses blanks added to the left of the number, so that the result is left-justified in the output buffer. Without FM, the result is always right-justified in the buffer, resulting in blank-padding to the left of the number.

FX "Format exact". This modifier specifies exact matching for the character argument and date format model of a TO_DATE function:

- Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.
- The character argument cannot have extra blanks. Without FX, Oracle ignores extra blanks.
- Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without FX, numbers in the character argument can omit leading zeroes.

  When FX is enabled, you can disable this check for leading zeroes by using the FM modifier as well.

If any portion of the character argument violates any of these conditions, Oracle returns an error message.

Format Modifier Examples

The following statement uses a date format model to return a character expression:

```
SELECT TO_CHAR(SYSDATE, 'fmDDTH')||' of '||TO_CHAR
   (SYSDATE, 'fmMonth')||', '||TO_CHAR(SYSDATE, 'YYYY') "Ides"
    FROM DUAL;

Ides
------------------
3RD of April, 1998
```

Note that the statement above also uses the FM modifier. If FM is omitted, the month is blank-padded to nine characters:

```
SELECT TO_CHAR(SYSDATE, 'DDTH')||' of '||
   TO_CHAR(SYSDATE, 'Month')||', '||
```

```
   TO_CHAR(SYSDATE, 'YYYY') "Ides"
   FROM DUAL;

Ides
-----------------------
03RD of April    , 1998
```

The following statement places a single quotation mark in the return value by using a date format model that includes two consecutive single quotation marks:

```
SELECT TO_CHAR(SYSDATE, 'fmDay')||'''s Special' "Menu"
     FROM DUAL;

Menu
-----------------
Tuesday's Special
```

Two consecutive single quotation marks can be used for the same purpose within a character literal in a format model.

Table 2–12 shows whether the following statement meets the matching conditions for different values of char and 'fmt' using FX (the table named `table` has a column `date_column` of datatype DATE):

```
UPDATE table
  SET date_column = TO_DATE(char, 'fmt');
```

Table 2–12 Matching Character Data and Format Models with the FX Format Model Modifier

char	'fmt'	Match or Error?
'15/ JAN /1998'	'DD-MON-YYYY'	Match
' 15! JAN % /1998'	'DD-MON-YYYY'	Error
'15/JAN/1998'	'FXDD-MON-YYYY'	Error
'15-JAN-1998'	'FXDD-MON-YYYY'	Match
'1-JAN-1998'	'FXDD-MON-YYYY'	Error
'01-JAN-1998'	'FXDD-MON-YYYY'	Match
'1-JAN-1998'	'FXFMDD-MON-YYYY'	Match

String-to-Date Conversion Rules

The following additional formatting rules apply when converting string values to date values (*unless* you have used the FX or FXFM modifiers in the format model to control exact format checking):

- You can omit punctuation included in the format string from the date string if all the digits of the numerical format elements, including leading zeros, are specified. In other words, specify 02 and not 2 for two-digit format elements such as MM, DD, and YY.
- You can omit time fields found at the end of a format string from the date string.
- If a match fails between a date format element and the corresponding characters in the date string, Oracle attempts alternative format elements, as shown in Table 2–13.

Table 2–13 Oracle Format Matching

Original Format Element	Additional Format Elements to Try in Place of the Original
'MM'	'MON' and 'MONTH'
'MON	'MONTH'
'MONTH'	'MON'
'YY'	'YYYY'
'RR'	'RRRR'

Nulls

If a column in a row has no value, then the column is said to be **null**, or to contain a null. Nulls can appear in columns of any datatype that are not restricted by NOT NULL or PRIMARY KEY integrity constraints. Use a null when the actual value is not known or when a value would not be meaningful.

Do not use null to represent a value of zero, because they are not equivalent. (Oracle currently treats a character value with a length of zero as null. However, this may not continue to be true in future releases, and Oracle recommends that you do not treat empty strings the same as nulls.) Any arithmetic expression containing a null always evaluates to null. For example, null added to 10 is null. In fact, all operators (except concatenation) return null when given a null operand.

Nulls in SQL Functions

All scalar functions (except REPLACE, NVL, and CONCAT) return null when given a null argument. You can use the NVL function to return a value when a null occurs. For example, the expression NVL(COMM,0) returns 0 if COMM is null or the value of COMM if it is not null.

Most aggregate functions ignore nulls. For example, consider a query that averages the five values 1000, null, null, null, and 2000. Such a query ignores the nulls and calculates the average to be (1000+2000)/2 = 1500.

Nulls with Comparison Operators

To test for nulls, use only the comparison operators IS NULL and IS NOT NULL. If you use any other operator with nulls and the result depends on the value of the null, the result is UNKNOWN. Because null represents a lack of data, a null cannot be equal or unequal to any value or to another null. However, Oracle considers two nulls to be equal when evaluating a DECODE expression.

> **See Also:** "DECODE Expressions" on page 5-13 for syntax and additional information, see

Oracle also considers two nulls to be equal if they appear in compound keys. That is, Oracle considers identical two compound keys containing nulls if all the non-null components of the keys are equal.

Nulls in Conditions

A condition that evaluates to UNKNOWN acts almost like FALSE. For example, a SELECT statement with a condition in the WHERE clause that evaluates to UNKNOWN returns no rows. However, a condition evaluating to UNKNOWN differs from FALSE in that further operations on an UNKNOWN condition evaluation will evaluate to UNKNOWN. Thus, NOT FALSE evaluates to TRUE, but NOT UNKNOWN evaluates to UNKNOWN.

Table 2–14 shows examples of various evaluations involving nulls in conditions. If the conditions evaluating to UNKNOWN were used in a WHERE clause of a SELECT statement, then no rows would be returned for that query.

Table 2–14 Conditions Containing Nulls

If A is:	Condition	Evaluates to:
10	a IS NULL	FALSE
10	a IS NOT NULL	TRUE
NULL	a IS NULL	TRUE
NULL	a IS NOT NULL	FALSE
10	a = NULL	UNKNOWN
10	a != NULL	UNKNOWN
NULL	a = NULL	UNKNOWN
NULL	a != NULL	UNKNOWN
NULL	a = 10	UNKNOWN
NULL	a != 10	UNKNOWN

For the truth tables showing the results of logical expressions containing nulls, see Table 3–6 on page 3-12, as well as Table 3–7 and Table 3–8.

Pseudocolumns

A **pseudocolumn** behaves like a table column, but is not actually stored in the table. You can select from pseudocolumns, but you cannot insert, update, or delete their values. This section describes these pseudocolumns:

- CURRVAL and NEXTVAL
- LEVEL
- ROWID
- ROWNUM

CURRVAL and NEXTVAL

A **sequence** is a schema object that can generate unique sequential values. These values are often used for primary and unique keys. You can refer to sequence values in SQL statements with these pseudocolumns:

CURRVAL	The CURRVAL pseudocolumn returns the current value of a sequence.
NEXTVAL	The NEXTVAL pseudocolumn increments the sequence and returns the next value.

You must qualify CURRVAL and NEXTVAL with the name of the sequence:

```
sequence.CURRVAL
sequence.NEXTVAL
```

To refer to the current or next value of a sequence in the schema of another user, you must have been granted either SELECT object privilege on the sequence or SELECT ANY SEQUENCE system privilege, and you must qualify the sequence with the schema containing it:

```
schema.sequence.CURRVAL
schema.sequence.NEXTVAL
```

To refer to the value of a sequence on a remote database, you must qualify the sequence with a complete or partial name of a database link:

```
schema.sequence.CURRVAL@dblink
schema.sequence.NEXTVAL@dblink
```

> **See Also:** "Referring to Objects in Remote Databases" on page 2-90 for more information on referring to database links

Where to Use Sequence Values

You can use CURRVAL and NEXTVAL in:

- The SELECT list of a SELECT statement that is not contained in a subquery, materialized view, or view
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

Restrictions: You cannot use CURRVAL and NEXTVAL:

- A subquery in a DELETE, SELECT, or UPDATE statement
- A query of a view or of a materialized view
- A SELECT statement with the DISTINCT operator

- A SELECT statement with a GROUP BY clause or ORDER BY clause
- A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator
- The WHERE clause of a SELECT statement
- DEFAULT value of a column in a CREATE TABLE or ALTER TABLE statement
- The condition of a CHECK constraint

Also, within a single SQL statement that uses CURVAL or NEXTVAL, all referenced LONG columns, updated tables, and locked tables must be located on the same database.

How to Use Sequence Values

When you create a sequence, you can define its initial value and the increment between its values. The first reference to NEXTVAL returns the sequence's initial value. Subsequent references to NEXTVAL increment the sequence value by the defined increment and return the new value. Any reference to CURRVAL always returns the sequence's current value, which is the value returned by the last reference to NEXTVAL. Note that before you use CURRVAL for a sequence in your session, you must first initialize the sequence with NEXTVAL.

Within a single SQL statement, Oracle will increment the sequence only once per row. If a statement contains more than one reference to NEXTVAL for a sequence, Oracle increments the sequence once and returns the same value for all occurrences of NEXTVAL. If a statement contains references to both CURRVAL and NEXTVAL, Oracle increments the sequence and returns the same value for both CURRVAL and NEXTVAL regardless of their order within the statement.

A sequence can be accessed by many users concurrently with no waiting or locking.

> **See Also:** CREATE SEQUENCE on page 9-155 for information on sequences

Finding the current value of a sequence: Example This example selects the current value of the employee sequence:

```
SELECT empseq.currval
    FROM DUAL;
```

Inserting sequence values into a table: Example This example increments the employee sequence and uses its value for a new employee inserted into the employee table:

```
INSERT INTO emp
    VALUES (empseq.nextval, 'LEWIS', 'CLERK',
              7902, SYSDATE, 1200, NULL, 20);
```

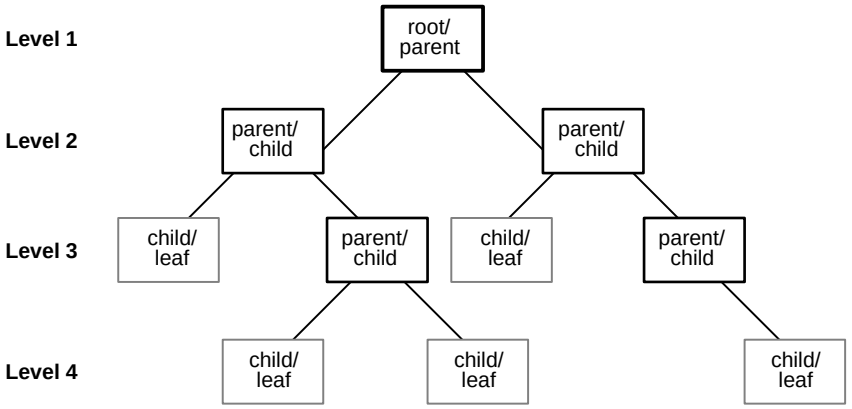
Reusing the current value of a sequence: Example This example adds a new order with the next order number to the master order table. It then adds suborders with this number to the detail order table:

```
INSERT INTO master_order(orderno, customer, orderdate)
    VALUES (orderseq.nextval, 'Al''s Auto Shop', SYSDATE);

INSERT INTO detail_order (orderno, part, quantity)
    VALUES (orderseq.currval, 'SPARKPLUG', 4);

INSERT INTO detail_order (orderno, part, quantity)
    VALUES (orderseq.currval, 'FUEL PUMP', 1);

INSERT INTO detail_order (orderno, part, quantity)
    VALUES (orderseq.currval, 'TAILPIPE', 2);
```

LEVEL

For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root node, 2 for a child of a root, and so on. A **root node** is the highest node within an inverted tree. A **child node** is any nonroot node. A **parent node** is any node that has children. A **leaf node** is any node without children. Figure 2–2 shows the nodes of an inverted tree with their LEVEL values.

Figure 2–2 Hierarchical Tree



To define a hierarchical relationship in a query, you must use the `START WITH` and `CONNECT BY` clauses.

See also:

- SELECT and subquery on page 11-88 for more information on using the `LEVEL` pseudocolumn
- "Hierarchical Queries" on page 5-22 for information on hierarchical queries in general

ROWID

For each row in the database, the `ROWID` pseudocolumn returns a row's address. Oracle8*i* rowid values contain information necessary to locate a row:

- The data object number of the object
- Which data block in the datafile
- Which row in the data block (first row is 0)
- Which datafile (first file is 1). The file number is relative to the tablespace.

Usually, a rowid value uniquely identifies a row in the database. However, rows in different tables that are stored together in the same cluster can have the same rowid.

Values of the `ROWID` pseudocolumn have the datatype `ROWID` or `UROWID`.

> **See Also:** "ROWID Datatype" on page 2-20 and "UROWID Datatype" on page 2-21

Rowid values have several important uses:

- They are the fastest way to access a single row.
- They can show you how a table's rows are stored.
- They are unique identifiers for rows in a table.

You should not use ROWID as a table's primary key. If you delete and reinsert a row with the Import and Export utilities, for example, its rowid may change. If you delete a row, Oracle may reassign its rowid to a new row inserted later.

Although you can use the ROWID pseudocolumn in the SELECT and WHERE clause of a query, these pseudocolumn values are not actually stored in the database. You cannot insert, update, or delete a value of the ROWID pseudocolumn.

Example This statement selects the address of all rows that contain data for employees in department 20:

```
SELECT ROWID, ename
   FROM emp
   WHERE deptno = 20;

ROWID              ENAME
------------------ ----------
AAAAqYAABAAAEPvAAA SMITH
AAAAqYAABAAAEPvAAD JONES
AAAAqYAABAAAEPvAAH SCOTT
AAAAqYAABAAAEPvAAK ADAMS
AAAAqYAABAAAEPvAAM FORD
```

ROWNUM

For each row returned by a query, the ROWNUM pseudocolumn returns a number indicating the order in which Oracle selects the row from a table or set of joined rows. The first row selected has a ROWNUM of 1, the second has 2, and so on.

You can use ROWNUM to limit the number of rows returned by a query, as in this example:

```
SELECT * FROM emp WHERE ROWNUM < 10;
```

If an ORDER BY clause follows ROWNUM in the same query, the rows will be reordered by the ORDER BY clause. The results can vary depending on the way the rows are accessed. For example, if the ORDER BY clause causes Oracle to use an index to access the data, Oracle may retrieve the rows in a different order than without the index. Therefore, the following statement will not have the same effect as the preceding example:

```
SELECT * FROM emp WHERE ROWNUM < 11 ORDER BY empno;
```

If you embed the ORDER BY clause in a subquery and place the ROWNUM condition in the top-level query, you can force the ROWNUM condition to be applied after the ordering of the rows. For example, the following query returns the 10 smallest employee numbers. This is sometimes referred to as a "top-N query":

```
SELECT * FROM
   (SELECT empno FROM emp ORDER BY empno)
   WHERE ROWNUM < 11;
```

In the preceding example, the ROWNUM values are those of the top-level SELECT statement, so they are generated after the rows have already been ordered by empno in the subquery.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* for more information about top-N queries

Conditions testing for ROWNUM values greater than a positive integer are always false. For example, this query returns no rows:

```
SELECT * FROM emp
    WHERE ROWNUM > 1;
```

The first row fetched is assigned a ROWNUM of 1 and makes the condition false. The second row to be fetched is now the first row and is also assigned a ROWNUM of 1 and makes the condition false. All rows subsequently fail to satisfy the condition, so no rows are returned.

You can also use ROWNUM to assign unique values to each row of a table, as in this example:

```
UPDATE tabx
    SET col1 = ROWNUM;
```

Note: Using ROWNUM in a query can affect view optimization. For more information, see *Oracle8i Concepts*.

Comments

You can associate comments with SQL statements and schema objects.

Comments Within SQL Statements

Comments within SQL statements do not affect the statement execution, but they may make your application easier for you to read and maintain. You may want to include a comment in a statement that describes the statement's purpose within your application.

A comment can appear between any keywords, parameters, or punctuation marks in a statement. You can include a comment in a statement using either of these means:

- Begin the comment with a slash and an asterisk (/*). Proceed with the text of the comment. This text can span multiple lines. End the comment with an asterisk and a slash (*/). The opening and terminating characters need not be separated from the text by a space or a line break.
- Begin the comment with -- (two hyphens). Proceed with the text of the comment. This text cannot extend to a new line. End the comment with a line break.

A SQL statement can contain multiple comments of both styles. The text of a comment can contain any printable characters in your database character set.

Note: You cannot use these styles of comments between SQL statements in a SQL script. Use the SQL*Plus REMARK command for this purpose. For information on these statements, see *SQL*Plus User's Guide and Reference*.

Example These statements contain many comments:

```
SELECT ename, sal + NVL(comm, 0), job, loc
/* Select all employees whose compensation is
greater than that of Jones.*/
  FROM emp, dept
       /*The DEPT table is used to get the department name.*/
```

```
  WHERE emp.deptno = dept.deptno
    AND sal + NVL(comm,0) >   /* Subquery:       */
   (SELECT sal + NLV(comm,0)
                              /* total compensation is sal + comm */
      FROM emp
      WHERE ename = 'JONES');

SELECT ename,                      -- select the name
    sal + NVL(comm, 0),            -- total compensation
    job,                           -- job
    loc                            -- and city containing the office
  FROM emp,                        -- of all employees
       dept
  WHERE emp.deptno = dept.deptno
    AND sal + NVL(comm, 0) >       -- whose compensation
                                   -- is greater than
      (SELECT sal + NVL(comm,0)    -- the compensation
    FROM emp
    WHERE ename = 'JONES');         -- of Jones.
```

Comments on Schema Objects

You can associate a comment with a table, view, materialized view, or column using the COMMENT command. Comments associated with schema objects are stored in the data dictionary.

> **See Also:** COMMENT on page 8-131 for a description of comments

Hints

You can use comments in a SQL statement to pass instructions, or *hints*, to the Oracle optimizer. The optimizer uses these hints as suggestions for choosing an execution plan for the statement.

A statement block can have only one comment containing hints, and that comment must follow the SELECT, UPDATE, INSERT, or DELETE keyword. The syntax below shows hints contained in both styles of comments that Oracle supports within a statement block.

```
{DELETE|INSERT|SELECT|UPDATE} /*+ hint [text] [hint[text]]... */
```

or

```
{DELETE|INSERT|SELECT|UPDATE} --+ hint [text] [hint[text]]...
```

where:

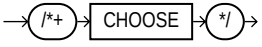
- DELETE, INSERT, SELECT, or UPDATE is a DELETE, INSERT, SELECT, or UPDATE keyword that begins a statement block. Comments containing hints can appear only after these keywords.
- + is a plus sign that causes Oracle to interpret the comment as a list of hints. The plus sign must follow immediately after the comment delimiter (no space is permitted).
- *hint* is one of the hints discussed in this section. The space between the plus sign and the hint is optional. If the comment contains multiple hints, separate the hints by at least one space.
- *text* is other commenting text that can be interspersed with the hints.

The syntax and a brief description of each hint appear below. Hints are divided into functional categories.

> **See Also:** *Oracle8i Performance Guide and Reference* and *Oracle8i Concepts* for more information on hints

Optimization Approaches and Goals Hints

The ALL_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption).



The ALL_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption).

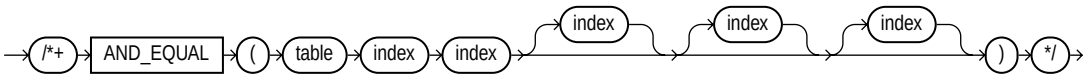
The FIRST_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best response time (minimum resource usage to return first row).

This hint causes the optimizer to make the following choices:

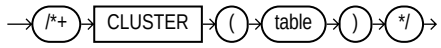
- If an index scan is available, then the optimizer might choose it over a full table scan.
- If an index scan is available, then the optimizer might choose a nested loops join over a sort-merge join whenever the associated table is the potential inner table of the nested loops.
- If an index scan is made available by an ORDER BY clause, then the optimizer might choose it to avoid a sort operation.

The RULE hint explicitly chooses rule-based optimization for a statement block. It also makes the optimizer ignore other hints specified for the statement block.

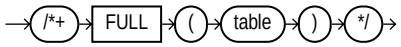
Access Method Hints



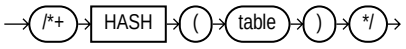
The AND_EQUAL hint explicitly chooses an execution plan that uses an access path that merges the scans on several single-column indexes.



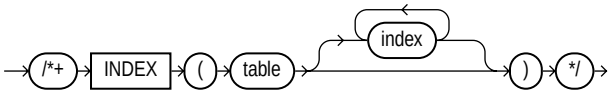
The CLUSTER hint explicitly chooses a cluster scan to access the specified table. It applies only to clustered objects.



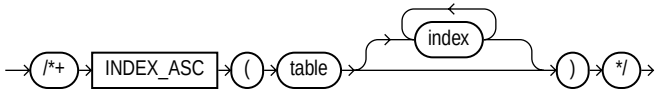
The FULL hint explicitly chooses a full table scan for the specified table.



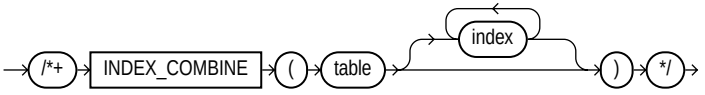
The HASH hint explicitly chooses a hash scan to access the specified table. It applies only to tables stored in a cluster.



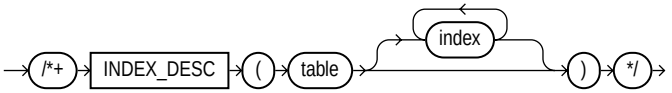
The INDEX hint explicitly chooses an index scan for the specified table. You can use the INDEX hint for domain, B*-tree, and bitmap indexes. However, Oracle recommends using INDEX_COMBINE rather than INDEX for bitmap indexes, because it is a more versatile hint.



The INDEX_ASC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, then Oracle scans the index entries in ascending order of their indexed values.

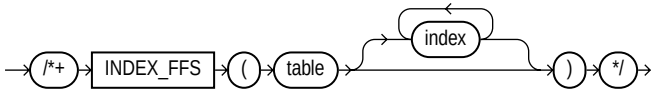


The INDEX_COMBINE hint explicitly chooses a bitmap access path for the table. If no indexes are given as arguments for the INDEX_COMBINE hint, then the optimizer uses whatever Boolean combination of bitmap indexes has the best cost estimate for the table. If certain indexes are given as arguments, then the optimizer tries to use some Boolean combination of those particular bitmap indexes.

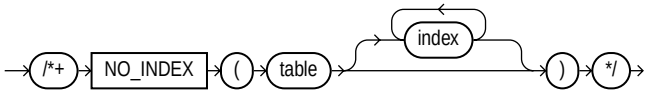


The INDEX_DESC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, then Oracle scans the index entries in

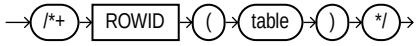
descending order of their indexed values. In a partitioned index, the results are in descending order within each partition.



The INDEX_FFS hint causes a fast full index scan to be performed rather than a full table scan.



The NO_INDEX hint explicitly disallows a set of indexes for the specified table.

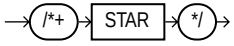


The ROWID hint explicitly chooses a table scan by rowid for the specified table.

Join Order Hints

The ORDERED hint causes Oracle to join tables in the order in which they appear in the FROM clause.

If you omit the ORDERED hint from a SQL statement performing a join, then the optimizer chooses the order in which to join the tables. You might want to use the ORDERED hint to specify a join order if you know something about the number of rows selected from each table that the optimizer does not. Such information lets you choose an inner and outer table better than the optimizer could.

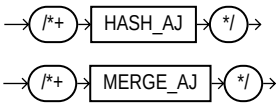


The STAR hint forces a star query plan to be used, if possible. A star plan has the largest table in the query last in the join order and joins it with a nested loops join on a concatenated index. The STAR hint applies when there are at least three tables,

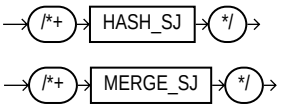
the large table's concatenated index has at least three columns, and there are no conflicting access or join method hints. The optimizer also considers different permutations of the small tables.

Join Operation Hints

The DRIVING_SITE hint forces query execution to be done at a different site than that selected by Oracle. This hint can be used with either rule-based or cost-based optimization.



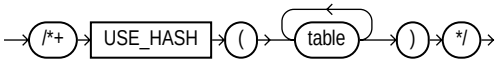
For a specific query, place the MERGE_AJ or HASH_AJ hints into the NOT IN subquery. MERGE_AJ uses a sort-merge anti-join and HASH_AJ uses a hash anti-join.



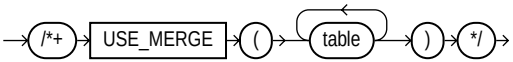
For a specific query, place the HASH_SJ or MERGE_SJ hint into the EXISTS subquery. HASH_SJ uses a hash semi-join and MERGE_SJ uses a sort merge semi-join.

The LEADING hint causes Oracle to use the specified table as the first table in the join order.

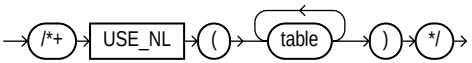
If you specify two or more LEADING hints on different tables, then all of them are ignored. If you specify the ORDERED hint, then it overrides all LEADING hints.



The USE_HASH hint causes Oracle to join each specified table with another row source with a hash join.



The USE_MERGE hint causes Oracle to join each specified table with another row source with a sort-merge join.

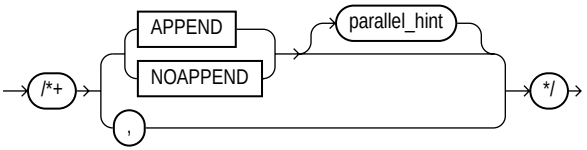


The USE_NL hint causes Oracle to join each specified table to another row source with a nested loops join using the specified table as the inner table.

Parallel Execution Hints

Note: Oracle ignores parallel hints on a temporary table.

See Also: CREATE TABLE on page 10-7 and *Oracle8i Concepts*

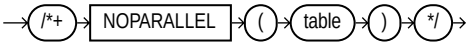


When you use the APPEND hint for INSERT, data is simply appended to a table. Existing free space in the blocks currently allocated to the table is not used.

If INSERT is parallelized using the PARALLEL hint or clause, then append mode is used by default. You can use NOAPPEND to override append mode. The APPEND hint applies to both serial and parallel insert.

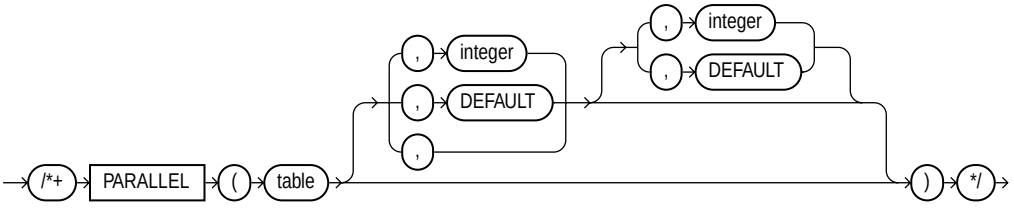
The append operation is performed in LOGGING or NOLOGGING mode, depending on whether the [NO] option is set for the table in question. Use the ALTER TABLE... [NO]LOGGING statement to set the appropriate value.

The NOAPPEND hint overrides append mode.



The NOPARALLEL hint overrides a PARALLEL specification in the table clause. In general, hints take precedence over table clauses.

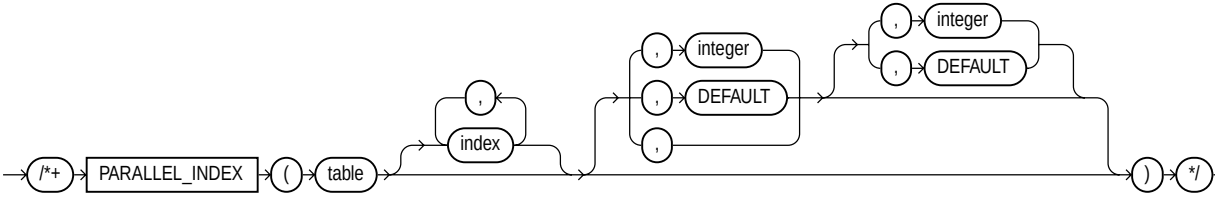
Restriction: You cannot parallelize a query involving a nested table.



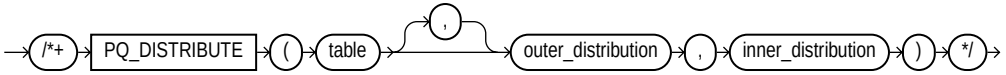
The PARALLEL hint lets you specify the desired number of concurrent servers that can be used for a parallel operation. The hint applies to the INSERT, UPDATE, and DELETE portions of a statement as well as to the table scan portion.

> **Note:** The number of servers that can be used is twice the value in the PARALLEL hint if sorting or grouping operations also take place.

If any parallel restrictions are violated, then the hint is ignored.



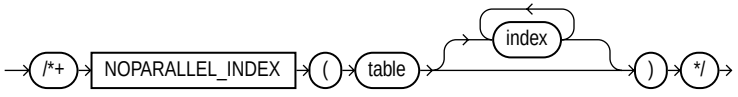
The PARALLEL_INDEX hint specifies the desired number of concurrent servers that can be used to parallelize index range scans for partitioned indexes.



The PQ_DISTRIBUTE hint improves parallel join operation performance. Do this by specifying how rows of joined tables should be distributed among producer and consumer query servers. Using this hint overrides decisions the optimizer would normally make.

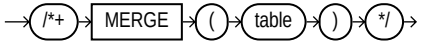
Use the EXPLAIN PLAN statement to identify the distribution chosen by the optimizer. The optimizer ignores the distribution hint if both tables are serial.

> **See Also:** *Oracle8i Performance Guide and Reference* for the permitted combinations of distributions for the outer and inner join tables



The NOPARALLEL_INDEX hint overrides a PARALLEL attribute setting on an index to avoid a parallel index scan operation.

Query Transformation Hints

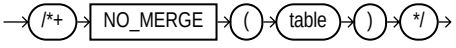


The MERGE hint lets you merge a view on a per-query basis.

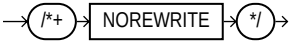
If a view's query contains a GROUP BY clause or DISTINCT operator in the SELECT list, then the optimizer can merge the view's query into the accessing statement only if complex view merging is enabled. Complex merging can also be used to merge an IN subquery into the accessing statement if the subquery is uncorrelated.

Complex merging is not cost-based--that is, the accessing query block must include the MERGE hint. Without this hint, the optimizer uses another approach.

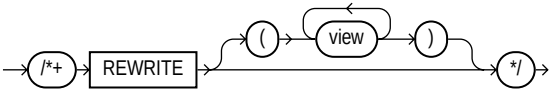
The NO_EXPAND hint prevents the cost-based optimizer from considering OR-expansion for queries having OR conditions or IN-lists in the WHERE clause. Usually, the optimizer considers using OR expansion and uses this method if it decides that the cost is lower than not using it.



The NO_MERGE hint causes Oracle not to merge mergeable views.

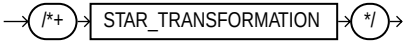


The NOREWRITE hint disables query rewrite for the query block, overriding the setting of the parameter QUERY_REWRITE_ENABLED. Use the NOREWRITE hint on any query block of a request.



The REWRITE hint forces the cost-based optimizer to rewrite a query in terms of materialized views, when possible, without cost consideration. Use the REWRITE hint with or without a view list. If you use REWRITE with a view list and the list contains an eligible materialized view, then Oracle uses that view regardless of its cost.

Oracle does not consider views outside of the list. If you do not specify a view list, then Oracle searches for an eligible materialized view and always uses it regardless of its cost.



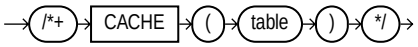
The STAR_TRANSFORMATION hint makes the optimizer use the best plan in which the transformation has been used. Without the hint, the optimizer could make a cost-based decision to use the best plan generated without the transformation, instead of the best plan for the transformed query.

Even if the hint is given, there is no guarantee that the transformation will take place. The optimizer only generates the subqueries if it seems reasonable to do so. If no subqueries are generated, then there is no transformed query, and the best plan for the untransformed query is used, regardless of the hint.

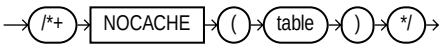
The USE_CONCAT hint forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator. Generally, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them.

The USE_CONCAT hint turns off IN-list processing and OR-expands all disjunctions, including IN-lists.

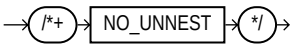
Other Hints



The CACHE hint specifies that the blocks retrieved for the table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This option is useful for small lookup tables.



The NOCACHE hint specifies that the blocks retrieved for the table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the normal behavior of blocks in the buffer cache.



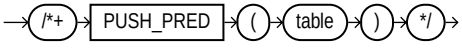
If you enabled subquery unnesting with the UNNEST_SUBQUERY parameter, then the NO_UNNEST hint turns it off for specific subquery blocks.

The ORDERED_PREDICATES hint forces the optimizer to preserve the order of predicate evaluation, except for predicates used as index keys. Use this hint in the WHERE clause of SELECT statements.

If you do not use the ORDERED_PREDICATES hint, then Oracle evaluates all predicates in the order specified by the following rules. Predicates:

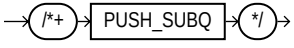
- Without user-defined functions, type methods, or subqueries are evaluated first, in the order specified in the WHERE clause.
- With user-defined functions and type methods that have user-computed costs are evaluated next, in increasing order of their cost.
- With user-defined functions and type methods without user-computed costs are evaluated next, in the order specified in the WHERE clause.
- Not specified in the WHERE clause (for example, predicates transitively generated by the optimizer) are evaluated next.
- With subqueries are evaluated last in the order specified in the WHERE clause.

Note: As mentioned, you cannot use the ORDERED_PREDICATES hint to preserve the order of predicate evaluation on index keys.



The PUSH_PRED hint forces pushing of a join predicate into the view.

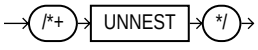
The NO_PUSH_PRED hint prevents pushing of a join predicate into the view.



The PUSH_SUBQ hint causes non-merged subqueries to be evaluated at the earliest possible place in the execution plan. Generally, subqueries that are not merged are

executed as the last step in the execution plan. If the subquery is relatively inexpensive and reduces the number of rows significantly, then it improves performance to evaluate the subquery earlier.

This hint has no effect if the subquery is applied to a remote table or one that is joined using a merge join.



Setting the UNNEST_SUBQUERY session parameter to TRUE enables subquery unnesting. Subquery unnesting unnests and merges the body of the subquery into the body of the statement that contains it, allowing the optimizer to consider them together when evaluating access paths and joins.

UNNEST_SUBQUERY first verifies if the statement is valid. If the statement is not valid, then subquery unnesting cannot proceed. The statement must then must pass a heuristic test.

The UNNEST hint checks the subquery block for validity only. If it is valid, then subquery unnesting is enabled without Oracle checking the heuristics.

Database Objects

Oracle recognizes objects that are associated with a particular schema and objects that are not associated with a particular schema, as described in the sections that follow.

Schema Objects

A **schema** is a collection of logical structures of data, or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema. Schema objects can be created and manipulated with SQL and include the following types of objects:

- Clusters
- Database links
- Database triggers
- Dimensions
- External procedure libraries

- Index-organized tables
- Indexes
- Indextypes
- Java classes, Java resources, Java sources
- Materialized views
- Materialized view logs
- Object tables
- Object types
- Object views
- Operators
- Packages
- Sequences
- Stored functions, stored procedures
- Synonyms
- Tables
- Views

Nonschema Objects

Other types of objects are also stored in the database and can be created and manipulated with SQL but are not contained in a schema:

- Contexts
- Directories
- Profiles
- Roles
- Rollback segments
- Tablespaces
- Users

In this reference, each type of object is briefly defined in Chapter 7 through Chapter 11, in the section describing the statement that creates the database object.

These statements begin with the keyword CREATE. For example, for the definition of a cluster, see CREATE CLUSTER on page 9-3.

See Also: *Oracle8i Concepts* for an overview of database objects

You must provide names for most types of schema objects when you create them. These names must follow the rules listed in the following sections.

Parts of Schema Objects

Some schema objects are made up of parts that you can or must name, such as:

- Columns in a table or view
- Index and table partitions and subpartitions
- Integrity constraints on a table
- Packaged procedures, packaged stored functions, and other objects stored within a package

Partitioned Tables and Indexes

Tables and indexes can be partitioned. When partitioned, these schema objects consist of a number of parts called **partitions**, all of which have the same logical attributes. For example, all partitions in a table share the same column and constraint definitions, and all partitions in an index share the same index columns.

When you partition a table or index using the range method, you specify a maximum value for the partitioning key column(s) for each partition. When you partition a table or index using the hash method, you instruct Oracle to distribute the rows of the table into partitions based on a system-defined hash function on the partitioning key column(s). When you partition a table or index using the composite-partitioning method, you specify ranges for the partitions, and Oracle distributes the rows in each partition into one or more hash subpartitions based on a hash function. Each subpartition of a table or index partitioned using the composite method has the same logical attributes.

Partition-Extended and Subpartition-Extended Table Names

Partition-extended and subpartition-extended table names let you perform some partition-level and subpartition-level operations, such as deleting all rows from a partition or subpartition, on only one partition or subpartition. Without extended table names, such operations would require that you specify a predicate (WHERE

clause). For range-partitioned tables, trying to phrase a partition-level operation with a predicate can be cumbersome, especially when the range partitioning key uses more than one column. For hash partitions and subpartitions, using a predicate is more difficult still, because these partitions and subpartitions are based on a system-defined hash function.

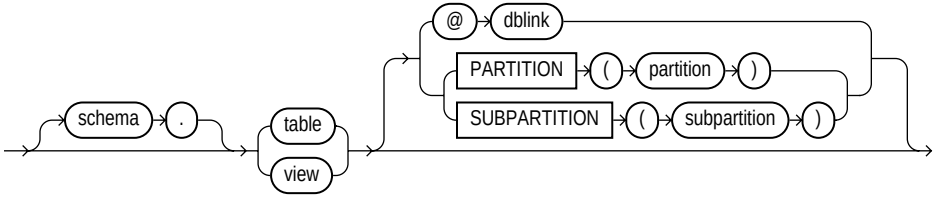
Partition-extended table names let you use partitions as if they were tables. An advantage of this method, which is most useful for range-partitioned tables, is that you can build partition-level access control mechanisms by granting (or revoking) privileges on these views to (or from) other users or roles.To use a partition as a table, create a view by selecting data from a single partition, and then use the view as a table.

You can specify partition-extended or subpartition-extended table names for the following DML statements:

- DELETE
- INSERT
- LOCK TABLE
- SELECT
- UPDATE

> **Note:** For application portability and ANSI syntax compliance, Oracle strongly recommends that you use views to insulate applications from this Oracle proprietary extension.

Syntax The basic syntax for using partition-extended and subpartition-extended table names is:



Restrictions Currently, the use of partition-extended and subpartition-extended table names has the following restrictions:

- No remote tables: A partition-extended or subpartition-extended table name cannot contain a database link (dblink) or a synonym that translates to a table with a dblink. To use remote partitions and subpartitions, create a view at the remote site that uses the extended table name syntax and then refer to the remote view.
- No synonyms: A partition or subpartition extension must be specified with a base table. You cannot use synonyms, views, or any other objects.

Example In the following statement, `sales` is a partitioned table with partition `jan97`. You can create a view of the single partition `jan97`, and then use it as if it were a table. This example deletes rows from the partition.

```
CREATE VIEW sales_jan97 AS
     SELECT * FROM sales PARTITION (jan97);
DELETE FROM sales_jan97 WHERE amount < 0;
```

Schema Object Names and Qualifiers

This section provides:

- Rules for naming schema objects and schema object location qualifiers
- Guidelines for naming schema objects and qualifiers

Schema Object Naming Rules

The following rules apply when naming schema objects:

1. Names must be from 1 to 30 bytes long with these exceptions:
   - Names of databases are limited to 8 bytes.
   - Names of database links can be as long as 128 bytes.
2. Names cannot contain quotation marks.
3. Names are not case sensitive.
4. A name must begin with an alphabetic character from your database character set unless surrounded by double quotation marks.
5. Names can contain only alphanumeric characters from your database character set and the underscore (_), dollar sign ($), and pound sign (#). Oracle strongly discourages you from using $ and #. Names of database links can also contain periods (.) and "at" signs (@).

If your database character set contains multibyte characters, Oracle recommends that each name for a user or a role contain at least one single-byte character.

Note: You cannot use special characters from European or Asian character sets in a database name, global database name, or database link names. For example, characters with an umlaut are not allowed.

---

6. A name cannot be an Oracle reserved word. , lists all Oracle reserved words.

   Depending on the Oracle product you plan to use to access a database object, names might be further restricted by other product-specific reserved words.

   See Also:

   - Appendix C, "Oracle Reserved Words" for a listing of all Oracle reserved words
   - The manual for the specific product, such as *PL/SQL User's Guide and Reference*, for a list of a product's reserved words

7. Do not use the word `DUAL` as a name for an object or part. `DUAL` is the name of a dummy table.

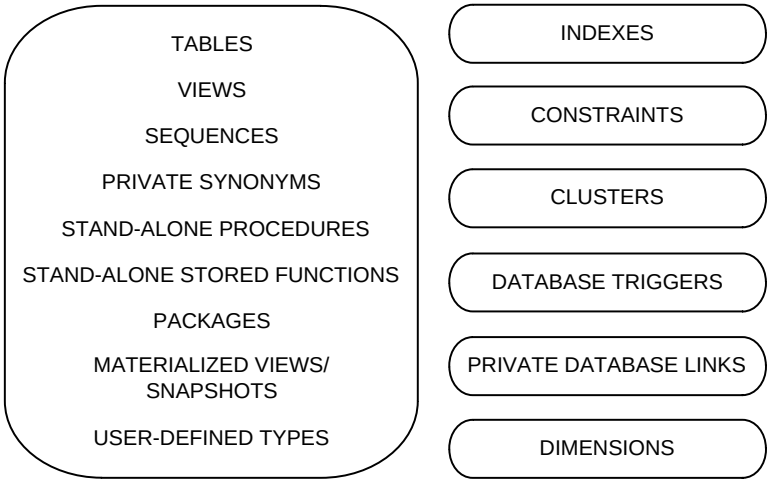
8. The Oracle SQL language contains other words that have special meanings. These words include datatypes, function names, and keywords (the uppercase words in SQL statements, such as `DIMENSION`, `SEGMENT`, `ALLOCATE`, DISABLE, and so forth). These words are not reserved. However, Oracle uses them internally. Therefore, if you use these words as names for objects and object parts, your SQL statements may be more difficult to read and may lead to unpredictable results.

   In particular, do not use words beginning with "`SYS_`" as schema object names, and do not use the names of SQL built-in functions for the names of schema objects or user-defined functions.

   See Also: "Datatypes" on page 2-2 and "SQL Functions" on page 4-2

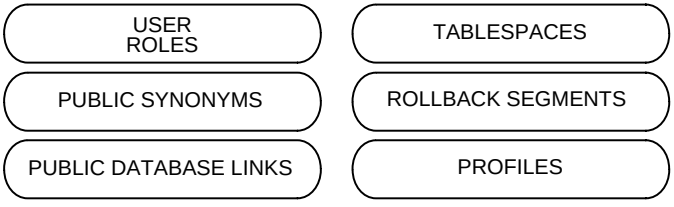
9. Within a namespace, no two objects can have the same name.

The following figure shows the namespaces for schema objects. Each box is a namespace. Tables and views are in the same namespace. Therefore, a table and a view in the same schema cannot have the same name. However, tables and indexes are in different namespaces. Therefore, a table and an index in the same schema can have the same name.



Each schema in the database has its own namespaces for the objects it contains. This means, for example, that two tables in different schemas are in different namespaces and can have the same name.

The following figure shows the namespaces for nonschema objects. Because the objects in these namespaces are not contained in schemas, these namespaces span the entire database.



10. Columns in the same table or view cannot have the same name. However, columns in different tables or views can have the same name.

11. Procedures or functions contained in the same package can have the same name, provided that their arguments are not of the same number and datatypes. Creating multiple procedures or functions with the same name in the same package with different arguments is called **overloading** the procedure or function.

12. A name can be enclosed in double quotation marks. Such names can contain any combination of characters, including spaces, ignoring rules 3 through 7 in this list. This exception is allowed for portability, but Oracle recommends that you do not break rules 3 through 7.

    If you give a schema object a name enclosed in double quotation marks, you must use double quotation marks whenever you refer to the object.

    Enclosing a name in double quotes allows it to:

    - Contain spaces
    - Be case sensitive
    - Begin with a character other than an alphabetic character, such as a numeric character
    - Contain characters other than alphanumeric characters and _, $, and #
    - Be a reserved word

    By enclosing names in double quotation marks, you can give the following names to different objects in the same namespace:

    ```
    emp
    "emp"
    "Emp"
    "EMP "
    ```

    Note that Oracle interprets the following names the same, so they cannot be used for different objects in the same namespace:

    ```
    emp
    EMP
    "EMP"
    ```

    If you give a user or password a quoted name, the name cannot contain lowercase letters.

    Database link names cannot be quoted.

Schema Object Naming Examples

The following examples are valid schema object names:

```
ename
horse
scott.hiredate
"EVEN THIS & THAT!"
a_very_long_and_valid_name
```

Although column aliases, table aliases, usernames, and passwords are not objects or parts of objects, they must also follow these naming rules with these exceptions:

- Column aliases and table aliases exist only for the execution of a single SQL statement and are not stored in the database, so rule 12 does not apply to them.
- Passwords do not have namespaces, so rule 9 does not apply to them.
- Do not use quotation marks to make usernames and passwords case sensitive.

> **See Also:** CREATE USER on page 10-99 for additional rules for naming users and passwords

Schema Object Naming Guidelines

Here are several helpful guidelines for naming objects and their parts:

- Use full, descriptive, pronounceable names (or well-known abbreviations).
- Use consistent naming rules.
- Use the same name to describe the same entity or attribute across tables.

When naming objects, balance the objective of keeping names short and easy to use with the objective of making names as descriptive as possible. When in doubt, choose the more descriptive name, because the objects in the database may be used by many people over a period of time. Your counterpart ten years from now may have difficulty understanding a database with a name like `pmdd` instead of `payment_due_date`.

Using consistent naming rules helps users understand the part that each table plays in your application. One such rule might be to begin the names of all tables belonging to the `FINANCE` application with `fin_`.

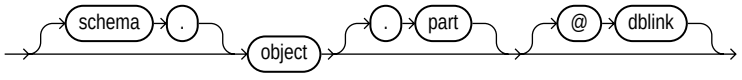
Use the same names to describe the same things across tables. For example, the department number columns of the sample `employees` and `departments` tables are both named `deptno`.

Syntax for Schema Objects and Parts in SQL Statements

This section tells you how to refer to schema objects and their parts in the context of a SQL statement. This section shows you:

- The general syntax for referring to an object
- How Oracle resolves a reference to an object
- How to refer to objects in schemas other than your own
- How to refer to objects in remote databases

The following diagram shows the general syntax for referring to an object or a part:



where:

- *object* is the name of the object.
- *schema* is the schema containing the object. The schema qualifier allows you to refer to an object in a schema other than your own. You must be granted privileges to refer to objects in other schemas. If you omit *schema*, Oracle assumes that you are referring to an object in your own schema.

  Only schema objects can be qualified with *schema*. Schema objects are shown with list item 9 on page 2-84. Nonschema objects, also shown with list item 9 on page 2-84, cannot be qualified with *schema* because they are not schema objects. (An exception is public synonyms, which can optionally be qualified with "PUBLIC". The quotation marks are required.)
- *part* is a part of the object. This identifier allows you to refer to a part of a schema object, such as a column or a partition of a table. Not all types of objects have parts.
- *dblink* applies only when you are using Oracle's distributed functionality. This is the name of the database containing the object. The *dblink* qualifier lets you refer to an object in a database other than your local database. If you omit *dblink*, Oracle assumes that you are referring to an object in your local database. Not all SQL statements allow you to access objects on remote databases.

You can include spaces around the periods separating the components of the reference to the object, but it is conventional to omit them.

How Oracle Resolves Schema Object References

When you refer to an object in a SQL statement, Oracle considers the context of the SQL statement and locates the object in the appropriate namespace. After locating the object, Oracle performs the statement's operation on the object. If the named object cannot be found in the appropriate namespace, Oracle returns an error.

The following example illustrates how Oracle resolves references to objects within SQL statements. Consider this statement that adds a row of data to a table identified by the name `dept`:

```
INSERT INTO dept
   VALUES (50, 'SUPPORT', 'PARIS');
```

Based on the context of the statement, Oracle determines that `dept` can be:

- A table in your own schema
- A view in your own schema
- A private synonym for a table or view
- A public synonym

Oracle always attempts to resolve an object reference within the namespaces in your own schema before considering namespaces outside your schema. In this example, Oracle attempts to resolve the name `dept` as follows:

1. First, Oracle attempts to locate the object in the namespace in your own schema containing tables, views, and private synonyms. If the object is a private synonym, Oracle locates the object for which the synonym stands. This object could be in your own schema, another schema, or on another database. The object could also be another synonym, in which case Oracle locates the object for which this synonym stands.
2. If the object is in the namespace, Oracle attempts to perform the statement on the object. In this example, Oracle attempts to add the row of data to `dept`. If the object is not of the correct type for the statement, Oracle returns an error. In this example, `dept` must be a table, view, or a private synonym resolving to a table or view. If `dept` is a sequence, Oracle returns an error.
3. If the object is not in any namespace searched in thus far, Oracle searches the namespace containing public synonyms. If the object is in that namespace, Oracle attempts to perform the statement on it. If the object is not of the correct type for the statement, Oracle returns an error. In this example, if `dept` is a public synonym for a sequence, Oracle returns an error.

Referring to Objects in Other Schemas

To refer to objects in schemas other than your own, prefix the object name with the schema name:

```
schema.object
```

For example, this statement drops the `emp` table in the schema `scott`:

```
DROP TABLE scott.emp
```

Referring to Objects in Remote Databases

To refer to objects in databases other than your local database, follow the object name with the name of the database link to that database. A database link is a schema object that causes Oracle to connect to a remote database to access an object there. This section tells you:

- How to create database links
- How to use database links in your SQL statements

Creating Database Links

You create a database link with the statement CREATE DATABASE LINK on page 9-28. The statement allows you to specify this information about the database link:

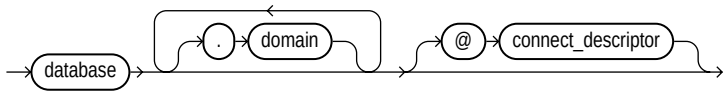
- The name of the database link
- The database connect string to access the remote database
- The username and password to connect to the remote database

Oracle stores this information in the data dictionary.

Database Link Names When you create a database link, you must specify its name. Database link names are different from names of other types of objects. They can be as long as 128 bytes and can contain periods (.) and the "at" sign (@).

The name that you give to a database link must correspond to the name of the database to which the database link refers and the location of that database in the hierarchy of database names. The following syntax diagram shows the form of the name of a database link:

dblink::=



where:

- *database* should specify *name* portion of the *global name* of the remote database to which the database link connects. This global name is stored in the data dictionary of the remote database; you can see this name in the GLOBAL_NAME view.
- *domain* should specify the *domain* portion of the global name of the remote database to which the database link connects. If you omit *domain* from the name of a database link, Oracle qualifies the database link name with the domain of your local database as it currently exists in the data dictionary.
- *connect_descriptor* allows you to further qualify a database link. Using connect descriptors, you can create multiple database links to the same database. For example, you can use connect descriptors to create multiple database links to different instances of the Oracle Parallel Server that access the same database.

The combination *database.domain* is sometimes called the "service name".

> **See Also:** *Net8 Administrator's Guide*

Username and Password Oracle uses the username and password to connect to the remote database. The username and password for a database link are optional.

Database Connect String The database connect string is the specification used by Net8 to access the remote database. For information on writing database connect strings, see the Net8 documentation for your specific network protocol. The database string for a database link is optional.

Referring to Database Links

Database links are available only if you are using Oracle's distributed functionality. When you issue a SQL statement that contains a database link, you can specify the database link name in one of these forms:

- *complete* is the complete database link name as stored in the data dictionary, including the *database*, *domain*, and optional *connect_descriptor* components.
- *partial* is the *database* and optional *connect_descriptor* components, but not the *domain* component.

Oracle performs these tasks before connecting to the remote database:

1. If the database link name specified in the statement is partial, Oracle expands the name to contain the domain of the local database as found in the global database name stored in the data dictionary. (You can see the current global database name in the GLOBAL_NAME data dictionary view.)
2. Oracle first searches for a private database link in your own schema with the same name as the database link in the statement. Then, if necessary, it searches for a public database link with the same name.
   - Oracle always determines the username and password from the first matching database link (either private or public). If the first matching database link has an associated username and password, Oracle uses it. If it does not have an associated username and password, Oracle uses your current username and password.
   - If the first matching database link has an associated database string, Oracle uses it. If not, Oracle searches for the next matching (public) database link. If no matching database link is found, or if no matching link has an associated database string, Oracle returns an error.
3. Oracle uses the database string to access the remote database. After accessing the remote database, if the value of the GLOBAL_NAMES parameter is true, Oracle verifies that the *database.domain* portion of the database link name matches the complete global name of the remote database. If this condition is true, Oracle proceeds with the connection, using the username and password chosen in Step 2. If not, Oracle returns an error.
4. If the connection using the database string, username, and password is successful, Oracle attempts to access the specified object on the remote database using the rules for resolving object references and referring to objects in other schemas discussed earlier in this section.

You can disable the requirement that the *database.domain* portion of the database link name must match the complete global name of the remote database by setting to false the initialization parameter GLOBAL_NAMES or the GLOBAL_NAMES parameter of the ALTER SYSTEM or ALTER SESSION statement.

> **See Also:** *Oracle8i Distributed Database Systems* for more information on remote name resolution

Referencing Object Type Attributes and Methods

To reference object type attributes or methods in a SQL statement, you must fully qualify the reference with a table alias. Consider the following example:

```
CREATE TYPE person AS OBJECT
   (ssno VARCHAR(20),
    name VARCHAR (10));

CREATE TABLE emptab (pinfo person);
```

In a SQL statement, reference to the ssno attribute must be fully qualified using a table alias, as illustrated below:

```
SELECT e.pinfo.ssno FROM emptab e;

UPDATE emptab e SET e.pinfo.ssno = '510129980'
   WHERE e.pinfo.name = 'Mike';
```

To reference an object type's member method that does not accept arguments, you must provide "empty" parentheses. For example, assume that age is a method in the person type that does not take arguments. In order to call this method in a SQL statement, you must provide empty parentheses as shows in this example:

```
SELECT e.pinfo.age() FROM emptab e
   WHERE e.pinfo.name = 'Mike';
```

> **See Also:** *Oracle8i Concepts* for more information on user-defined datatypes

3

Operators

An operator manipulates individual data items and returns a result. The data items are called **operands** or **arguments**. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*) and the operator that tests for nulls is represented by the keywords `IS NULL`.

This chapter contains these sections:

Unary and Binary Operators

The two general classes of operators are:

unary — A unary operator operates on only one operand. A unary operator typically appears with its operand in this format:

```
operator operand
```

binary — A binary operator operates on two operands. A binary operator appears with its operands in this format:

```
operand1 operator operand2
```

Other operators with special formats accept more than two operands. If an operator is given a null operand, the result is always null. The only operator that does not follow this rule is concatenation (||).

Precedence

Precedence is the order in which Oracle evaluates different operators in the same expression. When evaluating an expression containing multiple operators, Oracle evaluates operators with higher precedence before evaluating those with lower precedence. Oracle evaluates operators with equal precedence from left to right within an expression.

Table 3–1 lists the levels of precedence among SQL operators from high to low. Operators listed on the same line have the same precedence.

Table 3–1 SQL Operator Precedence

Operator	Operation
`+, -`	identity, negation
`*, /`	multiplication, division
`+, -, ||`	addition, subtraction, concatenation
`=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN`	comparison
`NOT`	exponentiation, logical negation
`AND`	conjunction
`OR`	disjunction

Precedence Example In the following expression, multiplication has a higher precedence than addition, so Oracle first multiplies 2 by 3 and then adds the result to 1.

```
1+2*3
```

You can use parentheses in an expression to override operator precedence. Oracle evaluates expressions inside parentheses before evaluating those outside.

SQL also supports set operators (`UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`), which combine sets of rows returned by queries, rather than individual data items. All set operators have equal precedence.

Arithmetic Operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of the operation is also a numeric value. Some of these operators are also used in date arithmetic. Table 3–2 lists arithmetic operators.

Table 3–2 Arithmetic Operators

Operator	Purpose	Example

+ -	When these denote a positive or negative expression, they are unary operators.	`SELECT * FROM orders`
		`WHERE qtysold = -1;`
		`SELECT * FROM emp`
		`WHERE -sal < 0;`

	When they add or subtract, they are binary operators.	`SELECT sal + comm FROM emp`
		`WHERE SYSDATE - hiredate`
		`> 365;`

* /	Multiply, divide. These are binary operators.	`UPDATE emp` `SET sal = sal * 1.1;`

Do not use two consecutive minus signs (--) in arithmetic expressions to indicate double negation or the subtraction of a negative value. The characters -- are used to begin comments within SQL statements. You should separate consecutive minus signs with a space or a parenthesis.

See Also: "Comments" on page 2-66 for more information on comments within SQL statements

Concatenation Operator

The concatenation operator manipulates character strings. Table 3–3 describes the concatenation operator.

Table 3–3 Concatenation Operator

Operator	Purpose	Example
`||`	Concatenates character strings.	`SELECT 'Name is ' || ename FROM emp;`

The result of concatenating two character strings is another character string. If both character strings are of datatype CHAR, the result has datatype CHAR and is limited to 2000 characters. If either string is of datatype VARCHAR2, the result has datatype VARCHAR2 and is limited to 4000 characters. Trailing blanks in character strings are preserved by concatenation, regardless of the strings' datatypes.

On most platforms, the concatenation operator is two solid vertical bars, as shown in Table 3–3. However, some IBM platforms use broken vertical bars for this operator. When moving SQL script files between systems having different character sets, such as between ASCII and EBCDIC, vertical bars might not be translated into the vertical bar required by the target Oracle environment. Oracle provides the CONCAT character function as an alternative to the vertical bar operator for cases when it is difficult or impossible to control translation performed by operating system or network utilities. Use this function in applications that will be moved between environments with differing character sets.

Although Oracle treats zero-length character strings as nulls, concatenating a zero-length character string with another operand always results in the other operand, so null can result only from the concatenation of two null strings. However, this may not continue to be true in future versions of Oracle. To concatenate an expression that might be null, use the NVL function to explicitly convert the expression to a zero-length string.

See Also: "Character Datatypes" on page 2-7 for more information on the differences between the CHAR and VARCHAR2 datatypes

Example This example creates a table with both CHAR and VARCHAR2 columns, inserts values both with and without trailing blanks, and then selects these values and concatenates them. Note that for both CHAR and VARCHAR2 columns, the trailing blanks are preserved.

```
CREATE TABLE tab1 (col1 VARCHAR2(6), col2 CHAR(6),
        col3 VARCHAR2(6), col4 CHAR(6) );

Table created.

INSERT INTO tab1 (col1,  col2,     col3,     col4)
        VALUES   ('abc', 'def   ', 'ghi   ', 'jkl');

1 row created.

SELECT col1||col2||col3||col4 "Concatenation"
        FROM tab1;

Concatenation
------------------------
abcdef   ghi   jkl
```

Comparison Operators

Comparison operators compare one expression with another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN.

> **See Also:** "Conditions" on page 5-15 for information on conditions

Table 3–4 lists comparison operators.

Table 3–4 Comparison Operators

Operator	Purpose	Example
=	Equality test.	`SELECT * FROM emp WHERE sal = 1500;`
!= ^= < > ¬=	Inequality test. Some forms of the inequality operator may be unavailable on some platforms.	`SELECT * FROM emp WHERE sal != 1500;`

Table 3–4 (Cont.) Comparison Operators

Operator	Purpose	Example
>	"Greater than" and "less than" tests.	`SELECT * FROM emp WHERE sal > 1500;`
<		`SELECT * FROM emp WHERE sal < 1500;`
>=	"Greater than or equal to" and "less than or equal to" tests.	`SELECT * FROM emp WHERE sal >= 1500;`
<=		`SELECT * FROM emp WHERE sal <= 1500;`
IN	"Equal to any member of" test. Equivalent to "= ANY".	`SELECT * FROM emp WHERE job IN ('CLERK','ANALYST');` `SELECT * FROM emp WHERE sal IN (SELECT sal FROM emp WHERE deptno = 30);`
NOT IN	Equivalent to "!=ALL". Evaluates to FALSE if any member of the set is NULL.	`SELECT * FROM emp WHERE sal NOT IN (SELECT sal FROM emp WHERE deptno = 30);` `SELECT * FROM emp WHERE job NOT IN ('CLERK', 'ANALYST');`
ANY SOME	Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to FALSE if the query returns no rows.	`SELECT * FROM emp WHERE sal = ANY (SELECT sal FROM emp WHERE deptno = 30);`
ALL	Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to TRUE if the query returns no rows.	`SELECT * FROM emp WHERE sal >= ALL ( 1400, 3000);`
[NOT] BETWEEN x AND y	[Not] greater than or equal to *x* and less than or equal to *y*.	`SELECT * FROM emp WHERE sal BETWEEN 2000 AND 3000;`

Table 3–4 (Cont.) Comparison Operators

Operator	Purpose	Example
EXISTS	TRUE if a subquery returns at least one row.	`SELECT ename, deptno FROM dept WHERE EXISTS (SELECT * FROM emp WHERE dept.deptno = emp.deptno);`
x [NOT] LIKE y [ESCAPE 'z']	TRUE if *x* does [not] match the pattern *y*. Within *y*, the character "%" matches any string of zero or more characters except null. The character "_" matches any single character. Any character, excepting percent (%) and underbar (_) may follow ESCAPE. A wildcard character is treated as a literal if preceded by the character designated as the escape character. **See Also:** "LIKE Operator" on page 3-8	`SELECT * FROM tab1 WHERE col1 LIKE 'A_C/%E%' ESCAPE '/';`
IS [NOT] NULL	Tests for nulls. This is the only operator that you should use to test for nulls. **See Also:** "Nulls" on page 2-57.	`SELECT ename, deptno FROM emp WHERE comm IS NULL;`

Additional information on the NOT IN and LIKE operators appears in the sections that follow.

NOT IN Operator

If any item in the list following a NOT IN operation is null, all rows evaluate to UNKNOWN (and no rows are returned). For example, the following statement returns the string 'TRUE' for each row:

```
SELECT 'TRUE'
    FROM emp
    WHERE deptno NOT IN (5,15);
```

However, the following statement returns no rows:

```
SELECT 'TRUE'
    FROM emp
    WHERE deptno NOT IN (5,15,null);
```

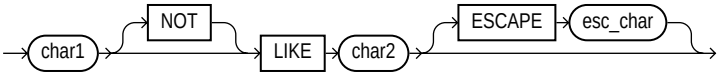
The above example returns no rows because the WHERE clause condition evaluates to:

```
deptno != 5 AND deptno != 15 AND deptno != null
```

Because all conditions that compare a null result in a null, the entire expression results in a null. This behavior can easily be overlooked, especially when the NOT IN operator references a subquery.

LIKE Operator

The LIKE operator is used in character string comparisons with pattern matching. The syntax for a condition using the LIKE operator is shown in this diagram:



char1 — Specify a value to be compared with a pattern. This value can have datatype CHAR or VARCHAR2.

NOT — The NOT keyword logically inverts the result of the condition, returning FALSE if the condition evaluates to TRUE and TRUE if it evaluates to FALSE.

char2 — Specify the pattern to which *char1* is compared. The pattern is a value of datatype CHAR or VARCHAR2 and can contain the special pattern matching characters % and _.

ESCAPE — Specify for *esc_char* a single character as the escape character. The escape character can be used to cause Oracle to interpret % or _ literally, rather than as a special character.

If you wish to search for strings containing an escape character, you must specify this character twice. For example, if the escape character is '/', to search for the string 'client/server', you must specify, 'client//server'.

Whereas the equal (=) operator exactly matches one character value to another, the LIKE operator matches a portion of one character value to another by searching the

first value for the pattern specified by the second. Note that blank padding is **not** used for LIKE comparisons.

With the LIKE operator, you can compare a value to a pattern rather than to a constant. The pattern must appear after the LIKE keyword. For example, you can issue the following query to find the salaries of all employees with names beginning with 'SM':

```
SELECT sal
    FROM emp
    WHERE ename LIKE 'SM%';
```

The following query uses the = operator, rather than the LIKE operator, to find the salaries of all employees with the name 'SM%':

```
SELECT sal
    FROM emp
    WHERE ename = 'SM%';
```

The following query finds the salaries of all employees with the name 'SM%'. Oracle interprets 'SM%' as a text literal, rather than as a pattern, because it precedes the LIKE operator:

```
SELECT sal
    FROM emp
    WHERE 'SM%' LIKE ename;
```

Patterns typically use special characters that Oracle matches with different characters in the value:

- An underscore (_) in the pattern matches exactly one character (as opposed to one byte in a multibyte character set) in the value.
- A percent sign (%) in the pattern can match zero or more characters (as opposed to bytes in a multibyte character set) in the value. Note that the pattern '%' cannot match a null.

Case Sensitivity and Pattern Matching Case is significant in all conditions comparing character expressions including the LIKE and equality (=) operators. You can use the UPPER function to perform a case-insensitive match, as in this condition:

```
UPPER(ename) LIKE 'SM%'
```

Pattern Matching on Indexed Columns When LIKE is used to search an indexed column for a pattern, Oracle can use the index to improve the statement's

performance if the leading character in the pattern is not "%" or "_". In this case, Oracle can scan the index by this leading character. If the first character in the pattern is "%" or "_", the index cannot improve the query's performance because Oracle cannot scan the index.

LIKE Operator Examples This condition is true for all ename values beginning with "MA":

```
ename LIKE 'MA%'
```

All of these ename values make the condition TRUE:

```
MARTIN, MA, MARK, MARY
```

Case is significant, so ename values beginning with "Ma," "ma," and "mA" make the condition FALSE.

Consider this condition:

```
ename LIKE 'SMITH_'
```

This condition is true for these ename values:

```
SMITHE, SMITHY, SMITHS
```

This condition is false for 'SMITH', since the special character "_" must match exactly one character of the ename value.

To search for employees with the pattern 'A_B' in their name:

```
SELECT ename
    FROM emp
    WHERE ename LIKE '%A\_B%' ESCAPE '\';
```

The ESCAPE option identifies the backslash (\) as the escape character. In the pattern, the escape character precedes the underscore (_). This causes Oracle to interpret the underscore literally, rather than as a special pattern matching character.

ESCAPE Option Example You can include the actual characters "%" or "_" in the pattern by using the ESCAPE option. The ESCAPE option identifies the escape character. If the escape character appears in the pattern before the character "%" or "_" then Oracle interprets this character literally in the pattern, rather than as a special pattern matching character.

Patterns Without % If a pattern does not contain the "%" character, the condition can be TRUE only if both operands have the same length.

Example: Consider the definition of this table and the values inserted into it:

```
CREATE TABLE freds (f CHAR(6), v VARCHAR2(6));
INSERT INTO freds VALUES ('FRED', 'FRED');
```

Because Oracle blank-pads CHAR values, the value of f is blank-padded to 6 bytes. v is not blank-padded and has length 4.

Logical Operators: NOT, AND, OR

A logical operator combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. Table 3–5 lists logical operators.

Table 3–5 Logical Operators

Operator	Function	Example

NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	`SELECT *`
		`  FROM emp`
		`  WHERE NOT (job IS NULL);`
		`SELECT *`
		`  FROM emp`
		`  WHERE NOT`
		`  (sal BETWEEN 1000 AND 2000);`

AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE. Otherwise returns UNKNOWN.	`SELECT *`
		`  FROM emp`
		`  WHERE job = 'CLERK'`
		`  AND deptno = 10;`

OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	`SELECT *`
		`  FROM emp`
		`  WHERE job = 'CLERK'`
		`  OR deptno = 10;`

For example, in the WHERE clause of the following SELECT statement, the AND logical operator is used to ensure that only those hired before 1984 and earning more than $1000 a month are returned:

```
SELECT *
```

```
FROM emp
WHERE hiredate < TO_DATE('01-JAN-1984', 'DD-MON-YYYY')
    AND sal > 1000;
```

NOT Operator

Table 3–6 shows the result of applying the NOT operator to a condition.

Table 3–6 NOT Truth Table

	TRUE	FALSE	UNKNOWN
NOT	FALSE	TRUE	UNKNOWN

AND Operator

Table 3–7 shows the results of combining two expressions with AND.

Table 3–7 AND Truth Table

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR Operator

Table 3–8 shows the results of combining two expressions with OR.

Table 3–8 OR Truth Table

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Set Operators: UNION [ALL], INTERSECT, MINUS

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries. Table 3–9 lists SQL set operators.

Table 3–9 Set Operators

Operator	Returns
UNION	All rows selected by either query.
UNION ALL	All rows selected by either query, including all duplicates.
INTERSECT	All distinct rows selected by both queries.
MINUS	All distinct rows selected by the first query but not the second.

All set operators have equal precedence. If a SQL statement contains multiple set operators, Oracle evaluates them from the left to right if no parentheses explicitly specify another order.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and datatype. If component queries select character data, the datatype of the return values are determined as follows:

- If both queries select values of datatype CHAR, the returned values have datatype CHAR.
- If either or both of the queries select values of datatype VARCHAR2, the returned values have datatype VARCHAR2.

Set Operator Examples Consider these two queries and their results:

```
SELECT part
    FROM orders_list1;

PART
----------
SPARKPLUG
FUEL PUMP
FUEL PUMP
TAILPIPE

SELECT part
    FROM orders_list2;

PART
----------
CRANKSHAFT
TAILPIPE
TAILPIPE
```

The following examples combine the two query results with each of the set operators.

UNION Example The following statement combines the results with the UNION operator, which eliminates duplicate selected rows. This statement shows that you must match datatype (using the TO_DATE and TO_NUMBER functions) when columns do not exist in one or the other table:

```
SELECT part, partnum, to_date(null) date_in
    FROM orders_list1
UNION
SELECT part, to_number(null), date_in
    FROM orders_list2;

PART       PARTNUM DATE_IN
---------- ------- --------
SPARKPLUG  3323165
SPARKPLUG          10/24/98
FUEL PUMP  3323162
FUEL PUMP          12/24/99
TAILPIPE   1332999
TAILPIPE           01/01/01
CRANKSHAFT 9394991
CRANKSHAFT         09/12/02

SELECT part
    FROM orders_list1
UNION
SELECT part
    FROM orders_list2;

PART
----------
SPARKPLUG
FUEL PUMP
TAILPIPE
CRANKSHAFT
```

UNION ALL Example The following statement combines the results with the UNION ALL operator, which does not eliminate duplicate selected rows:

```
SELECT part
    FROM orders_list1
UNION ALL
```

```
SELECT part
    FROM orders_list2;

PART
----------
SPARKPLUG
FUEL PUMP
FUEL PUMP
TAILPIPE
CRANKSHAFT
TAILPIPE
TAILPIPE
```

Note that the UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. A part value that appears multiple times in either or both queries (such as 'FUEL PUMP') is returned only once by the UNION operator, but multiple times by the UNION ALL operator.

INTERSECT Example The following statement combines the results with the INTERSECT operator, which returns only those rows returned by both queries:

```
SELECT part
    FROM orders_list1
INTERSECT
SELECT part
    FROM orders_list2;

PART
----------
TAILPIPE
```

MINUS Example The following statement combines results with the MINUS operator, which returns only rows returned by the first query but not by the second:

```
SELECT part
    FROM orders_list1
MINUS
SELECT part
    FROM orders_list2;

PART
----------
SPARKPLUG
FUEL PUMP
```

Other Built-In Operators

Table 3–10 lists other SQL operators.

Table 3–10 Other SQL Operators

Operator	Purpose	Example

(+)	Indicates that the preceding column is the outer join column in a join.
	
	See Also: "Outer Joins" on page 5-25.
		`SELECT ename, dname`
		`  FROM emp, dept`
		`  WHERE dept.deptno =`
		`  emp.deptno(+);`

PRIOR	Evaluates the following expression for the parent row of the current row in a hierarchical, or tree-structured, query. In such a query, you must use this operator in the `CONNECT BY` clause to define the relationship between parent and child rows. You can also use this operator in other parts of a `SELECT` statement that performs a hierarchical query. The `PRIOR` operator is a unary operator and has the same precedence as the unary + and - arithmetic operators.
	
	See Also: "Hierarchical Queries" on page 5-22.
		`SELECT empno, ename,`
		`mgr`
		`  FROM emp`
		`  CONNECT BY`
		`  PRIOR empno =`
		`mgr;`

User-Defined Operators

Like built-in operators, user-defined operators take a set of operands as input and return a result. However, you create them with the `CREATE OPERATOR` statement, and they are identified by names (e.g., `MERGE`). They reside in the same namespace as tables, views, types, and stand-alone functions.

Once you have defined a new operator, you can use it in SQL statements like any other built-in operator. For example, you can use user-defined operators in the select list of a `SELECT` statement, the condition of a `WHERE` clause, or in `ORDER BY` clauses and `GROUP BY` clauses. However, you must have `EXECUTE` privilege on the operator to do so, because it is a user-defined object.

For example, if you define an operator `CONTAINS`, which takes as input a text document and a keyword and returns 1 if the document contains the specified keyword, you can then write the following SQL query:

```
SELECT * FROM emp WHERE contains (resume, 'Oracle and UNIX') = 1;
```

See Also: CREATE OPERATOR on page 9-115 and *Oracle8i Data Cartridge Developer's Guide* for more information on user-defined operators

4

Functions

Functions are similar to operators in that they manipulate data items and return a result. Functions differ from operators in the format of their arguments. This format allows them to operate on zero, one, two, or more arguments:

```
function(argument, argument, ...)
```

This chapter contains these sections:

SQL Functions

SQL functions are built into Oracle and are available for use in various appropriate SQL statements. Do not confuse SQL functions with user functions written in PL/SQL.

If you call a SQL function with an argument of a datatype other than the datatype expected by the SQL function, Oracle implicitly converts the argument to the expected datatype before performing the SQL function. If you call a SQL function with a null argument, the SQL function automatically returns null. The only SQL functions that do not necessarily follow this behavior are CONCAT, NVL, and REPLACE.

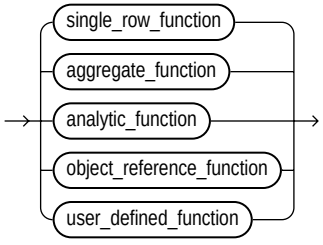
In the syntax diagrams for SQL functions, arguments are indicated by their datatypes. When the parameter "function" appears in SQL syntax, replace it with one of the functions described in this section. Functions are grouped by the datatypes of their arguments and their return values.

See Also:

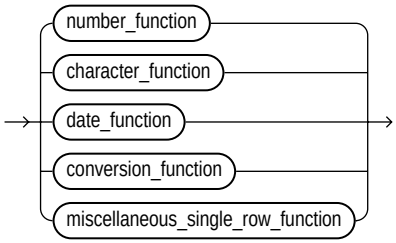
- "User-Defined Functions" on page 4-128 for information on user functions
- *Oracle interMedia Audio, Image, and Video User's Guide and Reference* for information on functions used with Oracle interMedia
- "Data Conversion" on page 2-30 for implicit conversion of datatypes
- "Syntax Diagrams and Notation" on page -xxi

The general syntax is as follows:

function::=



single_row_function::=



The sections that follow list the built-in SQL functions in each of the groups illustrated above except user-defined functions. All of the built-in SQL functions are then described in alphabetical order. User-defined functions are described at the end of this chapter.

The examples provided with the function descriptions use the `emp` and `dept` tables that are part of the `scott` schema in your sample Oracle database. Many examples also use a `sales` table, which has the following contents:

```
REGION PRODUCT  S_DAY    S_MONTH     S_YEAR   S_AMOUNT   S_PROFIT
------ ------- ------ ---------- ---------- ----------   --------
200          1     10          6       1998      77586        586
200          1     26          8       1998      62109        509
200          1     11         11       1998      46632        432
200          1     14          4       1999      15678        278
201          1      9          6       1998      77972        587
201          1     25          8       1998      62418        510
201          1     10         11       1998      46864        433
201          1     13          4       1999      15756        279
200          2      9          6       1998      39087      293.5
200          2     25          8       1998      31310        255
200          2     10         11       1998      23533      216.5
200          2     13          4       1999       7979      139.5
201          2      9         11       1998    23649.5        217
201          2     12          4       1999     8018.5        140
200          3      9         11       1998      15834     144.67
200          3     12          4       1999    5413.33      93.33
201          3     11          4       1999       5440      93.67
200          4     11          4       1999       4131      70.25
201          4     10          4       1999    4151.25       70.5
200          5     10          4       1999       3362       56.4
201          5      5          6       1998      16068      118.2
201          5     21          8       1998    12895.6      102.8
201          5      9          4       1999     3378.4       56.6
```

Single-Row Functions

Single-row functions return a single result row for every row of a queried table or view. These functions can appear in select lists, WHERE clauses, START WITH clauses, and CONNECT BY clauses.

Number Functions

Number functions accept numeric input and return numeric values. Most of these functions return values that are accurate to 38 decimal digits. The transcendental functions COS, COSH, EXP, LN, LOG, SIN, SINH, SQRT, TAN, and TANH are accurate to 36 decimal digits. The transcendental functions ACOS, ASIN, ATAN, and ATAN2 are accurate to 30 decimal digits. The number functions are:

ABS	COSH	SIGN
ACOS	EXP	SIN
ADD_MONTHS	FLOOR	SINH
ATAN	LN	SQRT
ATAN2	LOG	TAN
BITAND	MOD	TANH
CEIL	POWER	TRUNC (number function)
COS	ROUND (number function)	

Character Functions Returning Character Values

Character functions that return character values, unless otherwise noted, return values with the datatype VARCHAR2 and are limited in length to 4000 bytes. Functions that return values of datatype CHAR are limited in length to 2000 bytes. If the length of the return value exceeds the limit, Oracle truncates it and returns the result without an error message. The character functions that return character values are:

CHR	NLS_LOWER	SUBSTR
CONCAT	NLSSORT	SUBSTRB
INITCAP	NLS_UPPER	TRANSLATE
LOWER	REPLACE	TRIM
LPAD	RPAD	UPPER
LTRIM	RTRIM	
NLS_INITCAP	SOUNDEX	

Character Functions Returning Number Values

The character functions that return number values are:

ASCII	INSTRB	LENGTHB
INSTR	LENGTH	

Date Functions

Date functions operate on values of the DATE datatype. All date functions return a value of DATE datatype, except the MONTHS_BETWEEN function, which returns a number. The date functions are:

ADD_MONTHS	NEW_TIME	SYSDATE
LAST_DAY	NEXT_DAY	TRUNC (date function)
MONTHS_BETWEEN	ROUND (date function)	

Conversion Functions

Conversion functions convert a value from one datatype to another. Generally, the form of the function names follows the convention *datatype* TO *datatype*. The first datatype is the input datatype. The second datatype is the output datatype. The SQL conversion functions are:

CHARTOROWID	ROWIDTOCHAR	TO_LOB
CONVERT	TO_CHAR (date conversion)	TO_MULTI_BYTE
HEXTORAW		TO_NUMBER
NUMTODSINTERVAL	TO_CHAR (number conversion)	TO_SINGLE_BYTE
NUMTOYMINTERVAL	TO_DATE	TRANSLATE ... USING
RAWTOHEX		

Miscellaneous Single-Row Functions

The following single-row functions do not fall into any of the other single-row function categories.

BFILENAME	NLS_CHARSET_ID	SYS_GUID
DUMP	NLS_CHARSET_NAME	UID
EMPTY_[B | C]LOB	NVL	USER
GREATEST	NVL2	USERENV
LEAST	SYS_CONTEXT	VSIZE
NLS_CHARSET_DECL_LEN		

Aggregate Functions

Aggregate functions return a single result row based on groups of rows, rather than on single rows. Aggregate functions can appear in select lists and in ORDER BY and HAVING clauses. They are commonly used with the GROUP BY clause in a SELECT statement, where Oracle divides the rows of a queried table or view into groups. In a query containing a GROUP BY clause, the elements of the select list can be aggregate functions, GROUP BY expressions, constants, or expressions involving one of these. Oracle applies the aggregate functions to each group of rows and returns a single result row for each group.

If you omit the GROUP BY clause, Oracle applies aggregate functions in the select list to all the rows in the queried table or view. You use aggregate functions in the HAVING clause to eliminate groups from the output based on the results of the aggregate functions, rather than on the values of the individual rows of the queried table or view.

> **See Also:** "GROUP BY Examples" on page 11-105 and the HAVING clause on page 11-100 for more information on the GROUP BY clause and HAVING clauses in queries and subqueries

Many (but not all) aggregate functions that take a single argument accept these options:

- DISTINCT causes an aggregate function to consider only distinct values of the argument expression.
- ALL causes an aggregate function to consider all values, including all duplicates.

For example, the DISTINCT average of 1, 1, 1, and 3 is 2. The ALL average is 1.5. If you specify neither option, the default is ALL.

All aggregate functions except COUNT(*) and GROUPING ignore nulls. You can use the NVL function in the argument to an aggregate function to substitute a value for a null. COUNT never returns null, but returns either a number or zero. For all the remaining aggregate functions, if the data set contains no rows, or contains only rows with nulls as arguments to the aggregate function, then the function returns null.

You can nest aggregate functions. For example, the following example calculates the average of the maximum salaries of all the departments in the scott schema:

```
SELECT AVG(MAX(sal)) FROM emp GROUP BY deptno;

AVG(MAX(SAL))
-------------
   3616.66667
```

This calculation evaluates the inner aggregate (MAX(sal)) for each group defined by the GROUP BY clause (deptno), and aggregates the results again.

The aggregate functions are:

AVG	MAX	STDDEV_SAMP
CORR	MIN	SUM
COUNT	REGR_ (linear regression) functions	VAR_POP
COVAR_POP	STDDEV	VAR_SAMP
COVAR_SAMP	STDDEV_POP	VARIANCE
GROUPING		

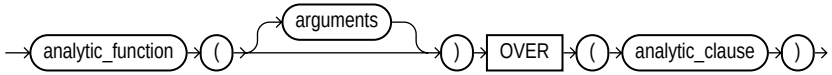
Analytic Functions

Analytic functions compute an aggregate value based on a group of rows. The group of rows is called a **window** and is defined by the analytic clause. For each row, a "sliding" window of rows is defined. The window determines the range of rows used to perform the calculations for the "current row". Window sizes can be based on either a physical number of rows or a logical interval such as time.

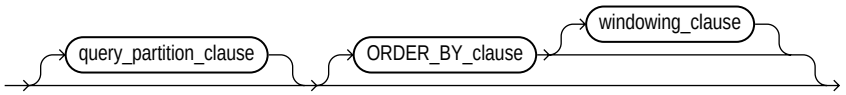
Analytic functions are the last set of operations performed in a query except for the final `ORDER BY` clause. All joins and all `WHERE`, `GROUP BY`, and `HAVING` clauses are completed before the analytic functions are processed. Therefore, analytic functions can appear only in the select list or `ORDER BY` clause.

Analytic functions are commonly used to compute cumulative, moving, centered, and reporting aggregates.

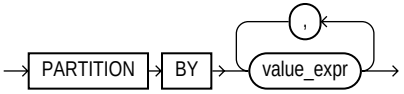
`analytic_function`::=



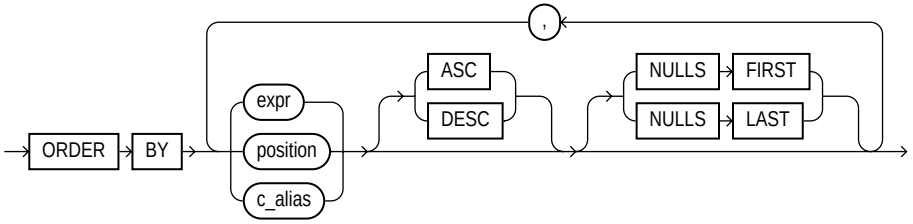
`analytic_clause`::=



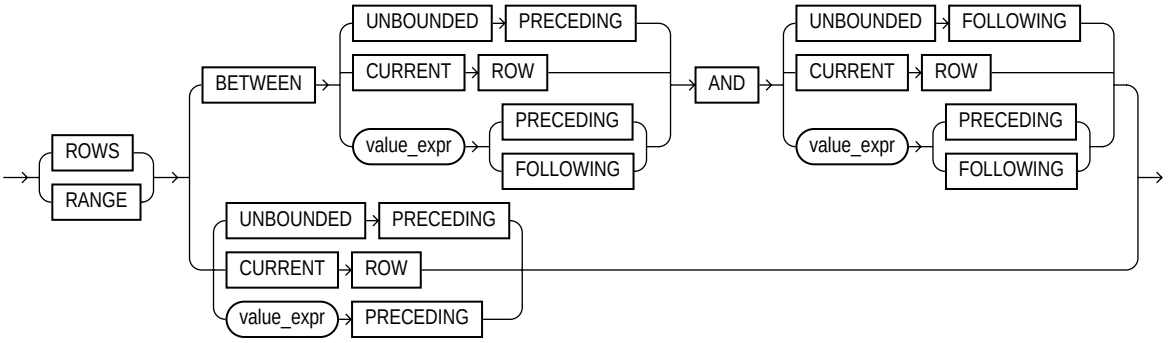
`query_partition_clause`::=



`ORDER_BY_clause`::=



windowing_clause::=



The keywords and parameters of this syntax are:

analytic_function

Specify the name of an analytic function (see the listings of different types of analytic functions following this table).

arguments

Analytic functions take 0 to 3 arguments.

analytic_clause

Use *analytic_clause* OVER clause to indicate that the function operates on a query result set. That is, it is computed after the FROM, WHERE, GROUP BY, and HAVING clauses. You can specify analytic functions with this clause in the select list or ORDER BY clause. To filter the results of a query based on an analytic function, nest these functions within the parent query, and then filter the results of the nested subquery.

Note: You cannot specify any analytic function in any part of the *analytic_clause*. That is, you cannot nest analytic functions. However, you can specify an analytic function in a subquery and compute another analytic function over it.

---

query_partition_clause

PARTITION BY — Use the PARTITION BY clause to partition the query result set into groups based on one or more *value_expr*. If you omit this clause, the function treats all rows of the query result set as a single group.

You can specify multiple analytic functions in the same query, each with the same or different PARTITION BY keys.

> **Note:** If the objects being queried have the parallel attribute, and if you specify an analytic function with the *query_partition_clause*, then the function computations are parallelized as well.

value_expr — Valid value expressions are constants, columns, nonanalytic functions, function expressions, or expressions involving any of these.

ORDER_BY_clause

Use the ORDER BY clause to specify how data is ordered within a partition. You can order the values in a partition on multiple keys, each defined by a *value_expr* and each qualified by an ordering sequence.

Within each function, you can specify multiple ordering expressions. Doing so is especially useful when using functions that rank values, because the second expression can resolve ties between identical values for the first expression.

Restriction: When used in an analytic function, the *ORDER_BY_clause* must take an expression (*expr*). Position (*position*) and column aliases (*c_alias*) are invalid. Otherwise this *ORDER_BY_clause* is the same as that used to order the overall query or subquery.

> **Note:** Analytic functions always operate on rows in the order specified in the *ORDER_BY_clause* of the function. However, the *ORDER_BY_clause* of the function does not guarantee the order of the result. Use the *ORDER_BY_clause* of the query to guarantee the final result ordering.

See Also: *order_by_clause* of "SELECT and Subqueries" on page 11-102 for more information on this clause

ASC | DESC — Specify the ordering sequence (ascending or descending). ASC is the default.

NULLS FIRST | NULLS LAST — Specify whether returned rows containing null values should appear first or last in the ordering sequence.

NULLS LAST is the default for ascending order, and NULLS FIRST is the default for descending order.

windowing_clause

ROWS | RANGE — These keywords define for each row a "window" (a physical or logical set of rows) used for calculating the function result. The function is then applied to all the rows in the window. The window "slides" through the query result set or partition from top to bottom.

- ROWS specifies the window in physical units (rows).
- RANGE specifies the window as a logical offset.

You cannot specify this clause unless you have specified the *ORDER_BY_clause*.

Note: The value returned by an analytic function with a logical offset is always deterministic. However, the value returned by an analytic function with a physical offset may produce nondeterministic results unless the ordering expression results in a unique ordering. You may have to specify multiple columns in the *ORDER_BY_clause* to achieve this unique ordering.

BETWEEN ... AND — Use the BETWEEN ... AND clause to specify a start point and end point for the window. The first expression (before AND) defines the start point and the second expression (after AND) defines the end point.

If you omit BETWEEN and specify only one end point, Oracle considers it the start point, and the end point defaults to the current row.

UNBOUNDED PRECEDING Specify UNBOUNDED PRECEDING to indicate that the window starts at the first row of the partition. This is the start point specification and cannot be used as an end point specification.

UNBOUNDED FOLLOWING Specify UNBOUNDED FOLLOWING to indicate that the window ends at the last row of the partition. This is the end point specification and cannot be used as a start point specification.

CURRENT ROW As a start point, CURRENT ROW specifies that the window begins at the current row or value (depending on whether you have specified ROW or RANGE, respectively). In this case the end point cannot be *value_expr* PRECEDING.

As an end point, CURRENT ROW specifies that the window ends at the current row or value (depending on whether you have specified ROW or RANGE, respectively). In this case the start point cannot be *value_expr* FOLLOWING.

***value_expr* PRECEDING**

***value_expr* FOLLOWING**

For RANGE or ROW:

- If *value_expr* FOLLOWING is the start point, then the end point must be *value_expr* FOLLOWING.
- If *value_expr* PRECEDING is the end point, then the start point must be *value_expr* PRECEDING.

If you are defining a logical window defined by an interval of time in numeric format, you may need to use conversion functions.

See Also: NUMTOYMINTERVAL on page 4-70 and NUMTODSINTERVAL on page 4-69 for information on converting numeric times into interval literals

If you specified ROWS:

- *value_expr* is a physical offset. It must be a constant or expression and must evaluate to a positive numeric value.
- If *value_expr* is part of the start point, it must evaluate to a row before the end point.

If you specified RANGE:

- *value_expr* is a logical offset. It must be a constant or expression that evaluates to a positive numeric value or an interval literal.

  See Also: "Literals" on page 2-33 for information on interval literals.

- You can specify only one expression in the *ORDER_BY_clause*
- If *value_expr* evaluates to a numeric value, the ORDER BY *expr* must be a NUMBER or DATE datatype.
- If *value_expr* evaluates to an interval value, the ORDER BY *expr* must be a DATE datatype.

If you omit the *windowing_clause* entirely, the default is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

Analytic functions are commonly used in data warehousing environments. The analytic functions are:

AVG	LEAD	STDDEV
CORR	MAX	STDDEV_POP
COVAR_POP	MIN	STDDEV_SAMP
COVAR_SAMP	NTILE	SUM
COUNT	PERCENT_RANK	VAR_POP
CUME_DIST	RATIO_TO_REPORT	VAR_SAMP
DENSE_RANK	RANK	VARIANCE
LAG	REGR_ (linear regression) functions	
FIRST_VALUE	ROW_NUMBER	
LAST_VALUE		

See Also: *Oracle8i Data Warehousing Guide* for more information on these functions, and for scenarios illustrating their use

Object Reference Functions

Object functions manipulate REFs, which are references to objects of specified object types. The object reference functions are:

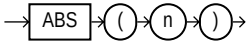
DEREF	REF	VALUE
MAKE_REF	REFTOHEX	

See Also: *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals* for more information about REFs

Alphabetical Listing of SQL Functions

ABS

Syntax



Purpose

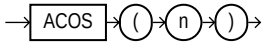
ABS returns the absolute value of *n*.

Example

```
SELECT ABS(-15) "Absolute" FROM DUAL;

  Absolute
----------
        15
```

ACOS

Syntax



Purpose

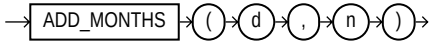
ACOS returns the arc cosine of *n*. Inputs are in the range of -1 to 1, and outputs are in the range of 0 to π and are expressed in radians.

Example

```
SELECT ACOS(.3)"Arc_Cosine" FROM DUAL;

Arc_Cosine
----------
1.26610367
```

ADD_MONTHS

Syntax



Purpose

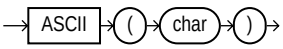
ADD_MONTHS returns the date *d* plus *n* months. The argument *n* can be any integer. If *d* is the last day of the month or if the resulting month has fewer days than the day component of *d*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *d*.

Example

```
SELECT TO_CHAR(
     ADD_MONTHS(hiredate,1),
     'DD-MON-YYYY') "Next month"
     FROM emp
     WHERE ename = 'SMITH';

Next Month
-----------
17-JAN-1981
```

ASCII

Syntax



Purpose

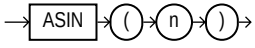
ASCII returns the decimal representation in the database character set of the first character of *char*. If your database character set is 7-bit ASCII, this function returns an ASCII value. If your database character set is EBCDIC Code, this function returns an EBCDIC value. There is no corresponding EBCDIC character function.

Example

```
SELECT ASCII('Q') FROM DUAL;

ASCII('Q')
----------
        81
```

ASIN

Syntax



Purpose

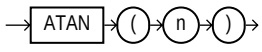
ASIN returns the arc sine of *n*. Inputs are in the range of -1 to 1, and outputs are in the range of $-\pi/2$ to $\pi/2$ and are expressed in radians.

Example

```
SELECT ASIN(.3) "Arc_Sine" FROM DUAL;

 Arc_Sine
----------
.304692654
```

ATAN

Syntax



Purpose

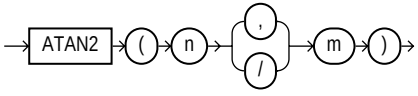
ATAN returns the arc tangent of *n*. Inputs are in an unbounded range, and outputs are in the range of $-\pi/2$ to $\pi/2$ and are expressed in radians.

Example

```
SELECT ATAN(.3) "Arc_Tangent" FROM DUAL;

Arc_Tangent
----------
.291456794
```

ATAN2

Syntax



Purpose

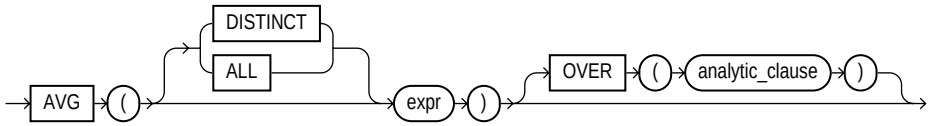
ATAN2 returns the arc tangent of *n* and *m*. Inputs are in an unbounded range, and outputs are in the range of $-\pi$ to π, depending on the signs of *n* and *m*, and are expressed in radians. ATAN2(*n*, *m*) is the same as ATAN2(*n*/*m*)

Example

```
SELECT ATAN2(.3, .2) "Arc_Tangent2" FROM DUAL;

Arc_Tangent2
------------
  .982793723
```

AVG

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

AVG returns average value of *expr*. You can use it as an aggregate or analytic function.

If you specify DISTINCT, you can specify only the *query_partition_clause* of the *analytic_clause*. The *ORDER_BY_clause* and *windowing_clause* are not allowed.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the average salary of all employees in the emp table:

```
SELECT AVG(sal) "Average" FROM emp;

   Average
----------
2077.21429
```

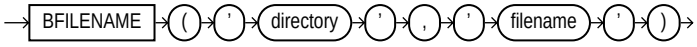
Analytic Example

The following example calculates, for each employee in the emp table, the average salary of the employees reporting to the same manager who were hired in the range just before through just after the employee:

```
SELECT mgr, ename, hiredate, sal,
   AVG(sal) OVER (PARTITION BY mgr ORDER BY hiredate
   ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS c_mavg
   FROM emp;

       MGR ENAME      HIREDATE          SAL     C_MAVG
---------- ---------- --------- ---------- ----------
      7566 FORD       03-DEC-81       3000       3000
      7566 SCOTT      19-APR-87       3000       3000
      7698 ALLEN      20-FEB-81       1600       1425
      7698 WARD       22-FEB-81       1250       1450
      7698 TURNER     08-SEP-81       1500 1333.33333
      7698 MARTIN     28-SEP-81       1250 1233.33333
      7698 JAMES      03-DEC-81        950       1100
      7782 MILLER     23-JAN-82       1300       1300
      7788 ADAMS      23-MAY-87       1100       1100
      7839 JONES      02-APR-81       2975     2912.5
      7839 BLAKE      01-MAY-81       2850 2758.33333
      7839 CLARK      09-JUN-81       2450       2650
      7902 SMITH      17-DEC-80        800        800
           KING       17-NOV-81       5000       5000
```

BFILENAME

Syntax



Purpose

BFILENAME returns a BFILE locator that is associated with a physical LOB binary file on the server's file system. A directory is an alias for a full pathname on the server's file system where the files are actually located, and 'filename' is the name of the file in the server's file system.

Neither '*directory*' nor '*filename*' needs to point to an existing object on the file system at the time you specify BFILENAME. However, you must associate a BFILE value with a physical file before performing subsequent SQL, PL/SQL, DBMS_LOB package, or OCI operations.

See Also:

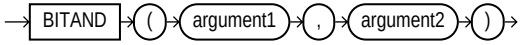
- *Oracle8i Application Developer's Guide - Large Objects (LOBs)* and *Oracle Call Interface Programmer's Guide* for more information on LOBs
- CREATE DIRECTORY on page 9-40

Example

```
INSERT INTO file_tbl
   VALUES (BFILENAME ('lob_dir1', 'image1.gif'));
```

BITAND

Syntax



Purpose

BITAND computes an AND operation on the bits of *argument1* and *argument2*, both of which must resolve to nonnegative integers, and returns an integer. This function is commonly used with the DECODE expression, as illustrated in the example that follows.

Example

Consider the following table named cars:

```
MANUFACTURER    MODEL         OPTIONS
--------------- ---------- ----------
TOYOTA          CAMRY               3
TOYOTA          COROLLA             5
NISSAN          MAXIMA              6
```

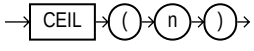
The following example represents each option in each car by individual bits:

```
SELECT manufacturer, model,
   DECODE(BITAND(options, 1), 1, 'Automatic', 'Stick-shift'),
   DECODE(BITAND(options, 2), 2, 'CD', 'Radio'),
   DECODE(BITAND(options, 4), 4, 'ABS', 'No-ABS')
FROM cars;
```

```
MANUFACTURER    MODEL      DECODE(BITA DECOD DECODE
--------------- ---------- ----------- ----- ------
TOYOTA          CAMRY      Automatic   CD    No-ABS
TOYOTA          COROLLA    Automatic   Radio ABS
NISSAN          MAXIMA     Stick-shift CD    ABS
```

CEIL

Syntax



Purpose

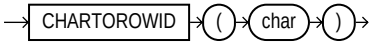
CEIL returns smallest integer greater than or equal to *n*.

Example

```
SELECT CEIL(15.7) "Ceiling" FROM DUAL;

   Ceiling
----------
        16
```

CHARTOROWID

Syntax



Purpose

CHARTOROWID converts a value from CHAR or VARCHAR2 datatype to ROWID datatype.

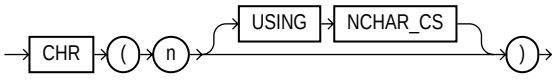
Example

```
SELECT ename FROM emp
   WHERE ROWID = CHARTOROWID('AAAAfZAABAAACp8AAO');
```

```
ENAME
----------
LEWIS
```

CHR

Syntax



Purpose

CHR returns the character having the binary equivalent to *n* in either the database character set or the national character set.

If USING NCHAR_CS is not specified, this function returns the character having the binary equivalent to *n* as a VARCHAR2 value in the database character set.

If USING NCHAR_CS is specified, this function returns the character having the binary equivalent to *n* as a NVARCHAR2 value in the national character set.

> **Note:** Use of the CHR function (either with or without the optional USING NCHAR_CS clause) results in code that is not portable between ASCII- and EBCDIC-based machine architectures.

Examples

The following example is run on an ASCII-based machine with the database character set defined as WE8ISO8859P1:

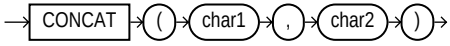
```
SELECT CHR(67)||CHR(65)||CHR(84) "Dog" FROM DUAL;

Dog
---
CAT

SELECT CHR(16705 USING NCHAR_CS) FROM DUAL;

C
-
A
```

To produce the same results on an EBCDIC-based machine with the WE8EBCDIC1047 character set, the first example above would have to be modified as follows:

```
SELECT CHR(195)||CHR(193)||CHR(227) "Dog"
   FROM DUAL;

Dog
---
CAT
```

CONCAT

Syntax



Purpose

CONCAT returns *char1* concatenated with *char2*. This function is equivalent to the concatenation operator (||).

> **See Also:** "Concatenation Operator" on page 3-4 for information on the CONCAT operator

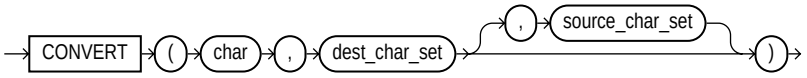
Example

This example uses nesting to concatenate three character strings:

```
SELECT CONCAT(CONCAT(ename, ' is a '), job) "Job"
FROM emp
WHERE empno = 7900;

Job
-----------------
JAMES is a CLERK
```

CONVERT

Syntax



Purpose

CONVERT converts a character string from one character set to another.

- The *char* argument is the value to be converted.
- The *dest_char_set* argument is the name of the character set to which *char* is converted.
- The *source_char_set* argument is the name of the character set in which *char* is stored in the database. The default value is the database character set.

Both the destination and source character set arguments can be either literals or columns containing the name of the character set.

For complete correspondence in character conversion, it is essential that the destination character set contains a representation of all the characters defined in the source character set. Where a character does not exist in the destination character set, a replacement character appears. Replacement characters can be defined as part of a character set definition.

Example

```
SELECT CONVERT('Groß', 'US7ASCII', 'WE8HP')
   "Conversion" FROM DUAL;

Conversion
----------
Gross
```

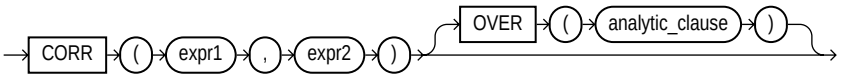
Common character sets include:

- US7ASCII: US 7-bit ASCII character set
- WE8DECDEC: West European 8-bit character set
- WE8HP: HP West European Laserjet 8-bit character set

- F7DEC: DEC French 7-bit character set
- WE8EBCDIC500: IBM West European EBCDIC Code Page 500
- WE8PC850: IBM PC Code Page 850
- WE8ISO8859P1: ISO 8859-1 West European 8-bit character set

CORR

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

CORR returns the coefficient of correlation of a set of number pairs. You can use it as an aggregate or analytic function.

Both *expr1* and *expr2* are number expressions. Oracle applies the function to the set of (*expr1* , *expr2*) after eliminating the pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
COVAR_POP(expr1, expr2) / (STDDEV_POP(expr1) * STDDEV_POP(expr2))
```

The function returns a value of type NUMBER. If the function is applied to an empty set, it returns null.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the coefficient of correlation between the salaries and commissions of the employees whose manager is 7698 from the emp table:

```
SELECT mgr, CORR(sal, comm) FROM EMP
   GROUP BY mgr
   HAVING mgr = 7698;

       MGR CORR(SAL,COMM)
---------- --------------
      7698     -.69920974
```

Analytic Example

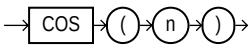
The following example returns the cumulative coefficient of correlation of monthly sales and monthly profits from the `sales` table for year 1998:

```
SELECT s_month, CORR(SUM(s_amount), SUM(s_profit))
   OVER (ORDER BY s_month) AS CUM_CORR
   FROM sales
   WHERE s_year=1998
   GROUP BY s_month
   ORDER BY s_month;

 S_MONTH    CUM_CORR
---------- ----------
         6
         8          1
        11 .860554259
```

Correlation functions require more than one row on which to operate, so the first row in the preceding example has no value calculated for it.

COS

Syntax



Purpose

COS returns the cosine of *n* (an angle expressed in radians).

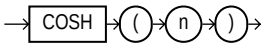
Example

```
SELECT COS(180 * 3.14159265359/180)
"Cosine of 180 degrees" FROM DUAL;
```

```
Cosine of 180 degrees
---------------------
                   -1
```

COSH

Syntax



Purpose

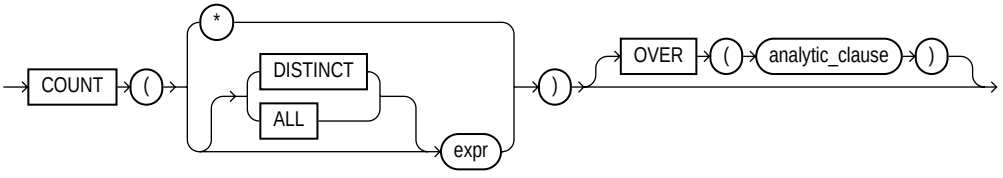
COSH returns the hyperbolic cosine of *n*.

Example

```
SELECT COSH(0) "Hyperbolic cosine of 0" FROM DUAL;

Hyperbolic cosine of 0
----------------------
                     1
```

COUNT

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

COUNT returns the number of rows in the query. You can use it as an aggregate or analytic function.

If you specify DISTINCT, you can specify only the *query_partition_clause* of the *analytic_clause*. The *ORDER_BY_clause* and *windowing_clause* are not allowed.

If you specify *expr*, COUNT returns the number of rows where *expr* is not null. You can count either all rows, or only distinct values of *expr*.

If you specify the asterisk (*), this function returns all rows, including duplicates and nulls. COUNT never returns null.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Examples

```
SELECT COUNT(*) "Total" FROM emp;

Total
----------
        14

SELECT COUNT(*) "Allstars" FROM emp
   WHERE comm > 0;

Allstars
--------
       3

SELECT COUNT(mgr) "Count" FROM emp;

Count
----------
        13

SELECT COUNT(DISTINCT mgr) "Managers" FROM emp;

Managers
----------
         6
```

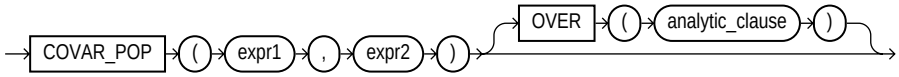
Analytic Example

The following example calculates, for each employee in the emp table, the moving count of employees earning salaries in the range $50 less than through $150 greater than the employee's salary.

```
SELECT ename, sal,
   COUNT(*) OVER (ORDER BY sal RANGE BETWEEN 50 PRECEDING
      AND 150 FOLLOWING) AS mov_count
   FROM emp;

ENAME              SAL  MOV_COUNT
---------- ---------- ----------
SMITH              800          2
JAMES              950          2
ADAMS             1100          3
WARD              1250          3
MARTIN            1250          3
MILLER            1300          3
TURNER            1500          2
ALLEN             1600          1
CLARK             2450          1
BLAKE             2850          4
JONES             2975          3
SCOTT             3000          3
FORD              3000          3
KING              5000          1
```

COVAR_POP

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

COVAR_POP returns the population covariance of a set of number pairs. You can use it as an aggregate or analytic function.

Both *expr1* and *expr2* are number expressions. Oracle applies the function to the set of (*expr1* , *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
(SUM(expr1 * expr2) - SUM(expr2) * SUM(expr1) / n) / n
```

where *n* is the number of (*expr1* , *expr2*) pairs where neither *expr1* nor *expr2* is null.

The function returns a value of type NUMBER. If the function is applied to an empty set, it returns null.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the population covariance for the amount of sales and sale profits for each year from the table sales.

```
SELECT s_year,
   COVAR_POP(s_amount, s_profit) AS COVAR_POP,
   COVAR_SAMP(s_amount, s_profit) AS COVAR_SAMP
   FROM sales GROUP BY s_year;

S_YEAR      COVAR_POP COVAR_SAMP
---------- ---------- ----------
      1998 3747965.53 4060295.99
      1999 360536.162 400595.736
```

Analytic Example

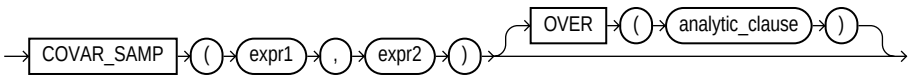
The following example calculates cumulative sample covariance of the amount of sales and sale profits in 1998.

```
SELECT s_year, s_month, s_day,
   COVAR_POP(s_amount, s_profit)
      OVER (ORDER BY s_month, s_day) AS CUM_COVP,
   COVAR_SAMP(s_amount, s_profit)
      OVER (ORDER BY s_month, s_day) AS CUM_COVS
   FROM sales
   WHERE s_year=1998
   ORDER BY s_year, s_month, s_day;
```

```
S_YEAR     S_MONTH    S_DAY      CUM_COVP   CUM_COVS
---------- ---------- ---------- ---------- ----------
      1998          6          5          0
      1998          6          9  4940952.6  7411428.9
      1998          6          9  4940952.6  7411428.9
      1998          6         10 5281752.33 7042336.44
      1998          8         21 6092799.46 7615999.32
      1998          8         25 4938283.61 5761330.88
      1998          8         25 4938283.61 5761330.88
      1998          8         26 4612074.09 5270941.82
      1998         11          9 4556799.53 5063110.59
      1998         11          9 4556799.53 5063110.59
      1998         11         10 4014833.65 4379818.52
      1998         11         10 4014833.65 4379818.52
      1998         11         11 3747965.53 4060295.99
```

COVAR_SAMP

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

COVAR_SAMP returns the sample covariance of a set of number pairs. You can use it as an aggregate or analytic function.

Both *expr1* and *expr2* are number expressions. Oracle applies the function to the set of (*expr1* , *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
(SUM(expr1 * expr2) - SUM(expr1) * SUM(expr2) / n) / (n-1)
```

where *n* is the number of (*expr1* , *expr2*) pairs where neither *expr1* nor *expr2* is null.

The function returns a value of type NUMBER. If the function is applied to an empty set, it returns null.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the population covariance for the amount of sales and sale profits for each year from the table sales.

```
SELECT s_year,
   COVAR_POP(s_amount, s_profit) AS COVAR_POP,
   COVAR_SAMP(s_amount, s_profit) AS COVAR_SAMP
   FROM sales GROUP BY s_year;

S_YEAR      COVAR_POP COVAR_SAMP
---------- ---------- ----------
      1998 3747965.53 4060295.99
      1999 360536.162 400595.736
```

Analytic Example

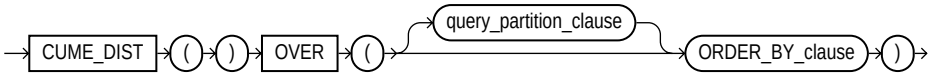
The following example calculates cumulative sample covariance of the amount of sales and sale profits in 1998.

```
SELECT s_year, s_month, s_day,
   COVAR_POP(s_amount, s_profit)
      OVER (ORDER BY s_month, s_day) AS CUM_COVP,
   COVAR_SAMP(s_amount, s_profit)
      OVER (ORDER BY s_month, s_day) AS CUM_COVS
   FROM sales
   WHERE s_year=1998
   ORDER BY s_year, s_month, s_day;

S_YEAR     S_MONTH    S_DAY      CUM_COVP   CUM_COVS
---------- ---------- ---------- ---------- ----------
      1998          6          5          0
      1998          6          9  4940952.6  7411428.9
      1998          6          9  4940952.6  7411428.9
      1998          6         10 5281752.33 7042336.44
      1998          8         21 6092799.46 7615999.32
      1998          8         25 4938283.61 5761330.88
```

```
1998          8         25 4938283.61 5761330.88
1998          8         26 4612074.09 5270941.82
1998         11          9 4556799.53 5063110.59
1998         11          9 4556799.53 5063110.59
1998         11         10 4014833.65 4379818.52
1998         11         10 4014833.65 4379818.52
1998         11         11 3747965.53 4060295.99
```

CUME_DIST

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

CUME_DIST (cumulative distribution) is an analytic function. It computes the relative position of a specified value in a group of values. For a row R, assuming ascending ordering, the CUME_DIST of R is the number of rows with values lower than or equal to the value of R, divided by the number of rows being evaluated (the entire query result set or a partition). The range of values returned by CUME_DIST is >0 to <=1. Tie values always evaluate to the same cumulative distribution value.

Example

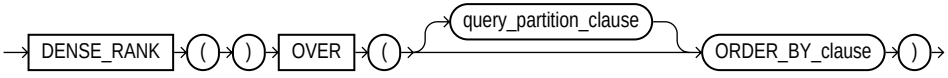
The following example calculates the salary percentile for each employee within each job category excluding job categories PRESIDENT and MANAGER. For example, 50% of clerks have salaries less than or equal to James.

```
SELECT job, ename, sal, CUME_DIST()
   OVER (PARTITION BY job ORDER BY sal) AS cume_dist
   FROM emp
   WHERE job NOT IN ('MANAGER', 'PRESIDENT');

JOB       ENAME             SAL  CUME_DIST
--------- ---------- ---------- ----------
ANALYST   SCOTT            3000          1
ANALYST   FORD             3000          1
```

```
CLERK     SMITH             800        .25
CLERK     JAMES             950         .5
CLERK     ADAMS            1100        .75
CLERK     MILLER           1300          1
SALESMAN  WARD             1250         .5
SALESMAN  MARTIN           1250         .5
SALESMAN  TURNER           1500        .75
SALESMAN  ALLEN            1600          1
```

DENSE_RANK

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

DENSE_RANK is an analytic function. It computes the rank of each row returned from a query with respect to the other rows, based on the values of the *value_exprs* in the *ORDER_BY_clause*. Rows with equal values for the ranking criteria receive the same rank. The ranks are consecutive integers beginning with 1. The largest rank value is the number of unique values returned by the query. Rank values are not skipped in the event of ties.

Example

The following statement selects the department name, employee name, and salary of all employees who work in the RESEARCH or SALES department, and then computes a rank for each unique salary in each of the two departments. The salaries that are equal receive the same rank. Compare this example with the example for RANK on page 4-74.

```
SELECT dname, ename, sal, DENSE_RANK()
   OVER (PARTITION BY dname ORDER BY sal) as drank
   FROM emp, dept
   WHERE emp.deptno = dept.deptno
   AND dname IN ('SALES', 'RESEARCH');
```

```
DNAME          ENAME              SAL      DRANK
-------------- ---------- ---------- ----------
RESEARCH       SMITH              800          1
RESEARCH       ADAMS             1100          2
RESEARCH       JONES             2975          3
RESEARCH       FORD              3000          4
RESEARCH       SCOTT             3000          4
SALES          JAMES              950          1
SALES          MARTIN            1250          2
SALES          WARD              1250          2
SALES          TURNER            1500          3
SALES          ALLEN             1600          4
SALES          BLAKE             2850          5
```

DEREF

Syntax

Purpose

DEREF returns the object reference of argument *expr*, where *expr* must return a REF to an object. If you do not use this function in a query, Oracle returns the object ID of the REF instead, as shown in the example that follows.

See Also: MAKE_REF on page 4-55

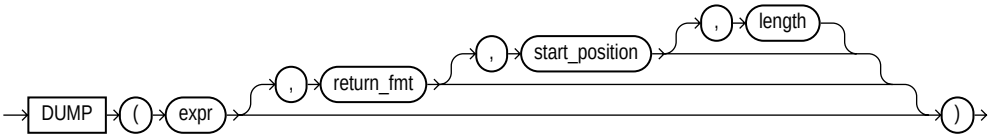
Example

```
CREATE TYPE emp_type AS OBJECT
   (eno NUMBER, ename VARCHAR2(20), salary NUMBER);
CREATE TABLE emp_table OF emp_type
   (primary key (eno, ename));
CREATE TABLE dept_table
   (dno NUMBER, mgr REF emp_type SCOPE IS emp_table);
INSERT INTO emp_table VALUES (10, 'jack', 50000);
INSERT INTO dept_table SELECT 10, REF(e) FROM emp_table e;

SELECT mgr FROM dept_table;

MGR
--------------------------------------------------------------------------------
```

```
00002202085928CB5CDF7B61CAE03400400B40DCB15928C35861E761BCE03400400B40DCB1

SELECT DEREF(mgr) from dept_table;

DEREF(MGR)(ENO, ENAME, SALARY)
------------------------------------------------------
EMP_TYPE(10, 'jack', 50000)
```

DUMP

Syntax



Purpose

DUMP returns a VARCHAR2 value containing the datatype code, length in bytes, and internal representation of *expr*. The returned result is always in the database character set. For the datatype corresponding to each code, see Table 2–1 on page 2-6.

The argument *return_fmt* specifies the format of the return value and can have any of the following values:

- 8 returns result in octal notation.
- 10 returns result in decimal notation.
- 16 returns result in hexadecimal notation.
- 17 returns result as single characters.

By default, the return value contains no character set information. To retrieve the character set name of *expr*, specify any of the format values above, plus 1000. For example, a *return_fmt* of 1008 returns the result in octal, plus provides the character set name of *expr*.

The arguments *start_position* and *length* combine to determine which portion of the internal representation to return. The default is to return the entire internal representation in decimal notation.

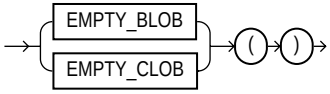
If *expr* is null, this function returns a null.

Examples

```
SELECT DUMP('abc', 1016)
   FROM DUAL;

DUMP('ABC',1016)
----------------------------------------
Typ=96 Len=3 CharacterSet=WE8DEC: 61,62,63


SELECT DUMP(ename, 8, 3, 2) "OCTAL"
   FROM emp
   WHERE ename = 'SCOTT';

OCTAL
-----------------------------
Type=1 Len=5: 117,124


SELECT DUMP(ename, 10, 3, 2) "ASCII"
   FROM emp
   WHERE ename = 'SCOTT';

ASCII
-----------------------------
Type=1 Len=5: 79,84
```

EMPTY_[B | C]LOB

Syntax



Purpose

EMPTY_BLOB and EMPTY_CLOB returns an empty LOB locator that can be used to initialize a LOB variable or in an INSERT or UPDATE statement to initialize a LOB

column or attribute to EMPTY. EMPTY means that the LOB is initialized, but not populated with data.

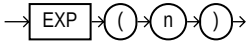
You cannot use the locator returned from this function as a parameter to the DBMS_LOB package or the OCI.

Example

```
INSERT INTO lob_tab1 VALUES (EMPTY_BLOB());
UPDATE lob_tab1
   SET clob_col = EMPTY_BLOB();
```

EXP

Syntax



Purpose

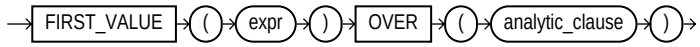
EXP returns e raised to the *n*th power, where e = 2.71828183 ...

Example

```
SELECT EXP(4) "e to the 4th power" FROM DUAL;

e to the 4th power
------------------
          54.59815
```

FIRST_VALUE

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

FIRST_VALUE is an analytic function. It returns the first value in an ordered set of values.

You cannot use FIRST_VALUE or any other analytic function for *expr*. That is, you can use other built-in function expressions for *expr*, but you cannot nest analytic functions.

> **See Also:** "Expressions" on page 5-2 for information on valid forms of *expr*

Examples

The following example selects, for each employee in Department 20, the name of the employee with the highest salary.

```
SELECT deptno, ename, sal, FIRST_VALUE(ename)
   OVER (ORDER BY sal DESC ROWS UNBOUNDED PRECEDING) AS rich_emp
   FROM (SELECT * FROM emp WHERE deptno = 20 ORDER BY empno);

    DEPTNO ENAME             SAL RICH_EMP
---------- ---------- ---------- ----------
        20 SCOTT            3000 SCOTT
        20 FORD             3000 SCOTT
        20 JONES            2975 SCOTT
        20 ADAMS            1100 SCOTT
        20 SMITH             800 SCOTT
```

The example illustrates the nondeterministic nature of the FIRST_VALUE function. Scott and Ford have the same salary, so are in adjacent rows. Scott appears first because the rows returned by the subquery are ordered by empno. However, if the rows returned by the subquery are ordered by empno in descending order, as in the next example, the function returns a different value:

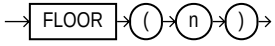
```
SELECT deptno, ename, sal, FIRST_VALUE(ename)
   OVER (ORDER BY sal DESC ROWS UNBOUNDED PRECEDING) AS fv
   FROM (SELECT * FROM emp WHERE deptno = 20 ORDER BY empno desc);

    DEPTNO ENAME             SAL FV
---------- ---------- ---------- ----------
        20 FORD             3000 FORD
        20 SCOTT            3000 FORD
        20 JONES            2975 FORD
        20 ADAMS            1100 FORD
        20 SMITH             800 FORD
```

The following example shows how to make the FIRST_VALUE function deterministic by ordering on a unique key.

```
SELECT deptno, ename, sal, hiredate, FIRST_VALUE(ename)
   OVER (ORDER BY sal DESC, hiredate ROWS UNBOUNDED PRECEDING) AS fv
   FROM (SELECT * FROM emp WHERE deptno = 20 ORDER BY empno desc);

DEPTNO     ENAME      SAL        HIREDATE  FV
---------- ---------- ---------- --------- ----------
        20 FORD             3000 03-DEC-81 FORD
        20 SCOTT            3000 19-APR-87 FORD
        20 JONES            2975 02-APR-81 FORD
        20 ADAMS            1100 23-MAY-87 FORD
        20 SMITH             800 17-DEC-80 FORD
```

FLOOR

Syntax



Purpose

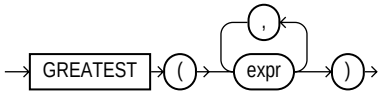
FLOOR returns largest integer equal to or less than *n*.

Example

```
SELECT FLOOR(15.7) "Floor" FROM DUAL;

     Floor
----------
        15
```

GREATEST

Syntax



Purpose

GREATEST returns the greatest of the list of *exprs*. All *exprs* after the first are implicitly converted to the datatype of the first *expr* before the comparison. Oracle compares the *exprs* using nonpadded comparison semantics. Character comparison is based on the value of the character in the database character set. One character is greater than another if it has a higher character set value. If the value returned by this function is character data, its datatype is always VARCHAR2.

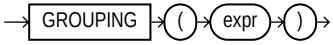
See Also: "Datatype Comparison Rules" on page 2-26

Example

```
SELECT GREATEST ('HARRY', 'HARRIOT', 'HAROLD')
   "Greatest" FROM DUAL;

Greatest
--------
HARRY
```

GROUPING

Syntax



Purpose

The GROUPING function is applicable only in a SELECT statement that contains a GROUP BY extension, such as ROLLUP or CUBE. These operations produce superaggregate rows that contain nulls representing the set of all values. You can use the GROUPING function to distinguish a null that represents the set of all values in a superaggregate row from an actual null.

The *expr* in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 1 if the value of *expr* in the row is a null representing the set of all values. Otherwise, it returns zero. The datatype of the value returned by the GROUPING function is Oracle NUMBER.

See Also: *group_by_clause* of the SELECT statement on page 11-99 for a discussion of these terms

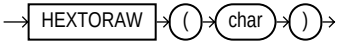
Example

In the following example, if the GROUPING function returns 1 (indicating a superaggregate row rather than a data row from the table), the string "All Jobs" appears instead of the null that would otherwise appear:

```
SELECT DECODE(GROUPING(dname), 1, 'All Departments',
         dname) AS dname,
   DECODE(GROUPING(job), 1, 'All Jobs', job) AS job,
   COUNT(*) "Total Empl", AVG(sal) * 12 "Average Sal"
   FROM emp, dept
   WHERE dept.deptno = emp.deptno
   GROUP BY ROLLUP (dname, job);

DNAME           JOB       Total Empl Average Sa
--------------- --------- ---------- ----------
ACCOUNTING      CLERK              1      15600
ACCOUNTING      MANAGER            1      29400
ACCOUNTING      PRESIDENT          1      60000
ACCOUNTING      All Jobs           3      35000
RESEARCH        ANALYST            2      36000
RESEARCH        CLERK              2      11400
RESEARCH        MANAGER            1      35700
RESEARCH        All Jobs           5      26100
SALES           CLERK              1      11400
SALES           MANAGER            1      34200
SALES           SALESMAN           4      16800
SALES           All Jobs           6      18800
All Departments All Jobs          14 24878.5714
```

HEXTORAW

Syntax



Purpose

HEXTORAW converts *char* containing hexadecimal digits to a raw value.

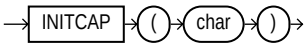
Example

```
INSERT INTO graphics (raw_column)
  SELECT HEXTORAW('7D') FROM DUAL;
```

See Also: "RAW and LONG RAW Datatypes" on page 2-16 and RAWTOHEX on page 4-76

INITCAP

Syntax



Purpose

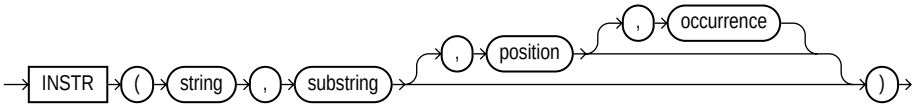
INITCAP returns *char*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

Example

```
SELECT INITCAP('the soap') "Capitals" FROM DUAL;

Capitals
---------
The Soap
```

INSTR

Syntax



Purpose

INSTR searches *string* for *substring*.

- *position* is an integer indicating the character of *string* where Oracle begins the search. If *position* is negative, Oracle counts and searches backward from the end of *string*.

- *occurrence* is an integer indicating which occurrence of *string* Oracle should search for. The value of *occurrence* must be positive.

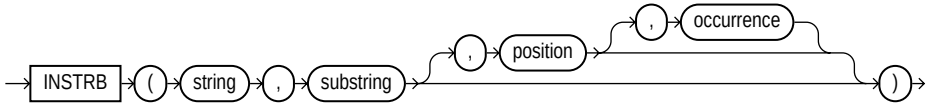
The function returns an integer indicating the position of the character in *string* that is the first character of this occurrence. The default values of both *position* and *occurrence* are 1, meaning Oracle begins searching at the first character of *string* for the first occurrence of *substring*. The return value is relative to the beginning of *string*, regardless of the value of *position*, and is expressed in characters. If the search is unsuccessful (if *substring* does not appear *occurrence* times after the *position* character of *string*) the return value is 0.

Examples

```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2)
  "Instring" FROM DUAL;

  Instring
----------
        14

SELECT INSTR('CORPORATE FLOOR','OR', -3, 2)
"Reversed Instring"
     FROM DUAL;

Reversed Instring
-----------------
                2
```

INSTRB

Syntax



Purpose

INSTRB is the same as INSTR, except that *position* and the return value are expressed in bytes, rather than in characters. For a single-byte database character set, INSTRB is equivalent to INSTR.

See Also: INSTR on page 4-43

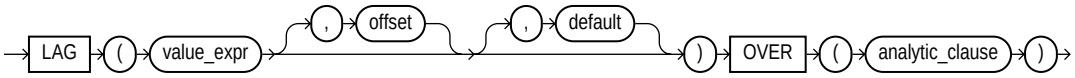
Example

This example assumes a double-byte database character set.

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2)
"Instring in bytes"
FROM DUAL;

Instring in bytes
-----------------
               27
```

LAG

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

LAG is an analytic function. It provides access to more than one row of a table at the same time without a self-join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset prior to that position.

If you do not specify *offset*, its default is 1. The optional *default* value is returned if the offset goes beyond the scope of the window. If you do not specify *default*, its default value is null.

You cannot use LAG or any other analytic function for *value_expr*. That is, you can use other built-in function expressions for *expr*, but you cannot nest analytic functions.

See Also: "Expressions" on page 5-2 for information on valid forms of *expr*

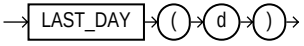
Example

The following example provides, for each salesperson in the emp table, the salary of the employee hired just before:

```
SELECT ename, hiredate, sal,
   LAG(sal, 1, 0) OVER (ORDER BY hiredate) as prev_sal
   FROM emp
   WHERE job = 'SALESMAN';

ENAME      HIREDATE          SAL   PREV_SAL
---------- --------- ---------- ----------
ALLEN      20-FEB-81       1600          0
WARD       22-FEB-81       1250       1600
TURNER     08-SEP-81       1500       1250
MARTIN     28-SEP-81       1250       1500
```

LAST_DAY

Syntax



Purpose

LAST_DAY returns the date of the last day of the month that contains *d*. You might use this function to determine how many days are left in the current month.

Examples

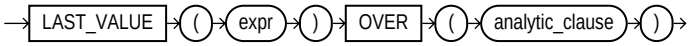
```
SELECT SYSDATE,
   LAST_DAY(SYSDATE) "Last",
   LAST_DAY(SYSDATE) - SYSDATE "Days Left"
   FROM DUAL;

SYSDATE   Last       Days Left
--------- --------- ----------
23-OCT-97 31-OCT-97          8
```

The following example adds 5 months to the hiredate of each employee to give an evaluation date:

```
SELECT ename, hiredate, TO_CHAR(
   ADD_MONTHS(LAST_DAY(hiredate), 5)) "Eval Date"
   FROM emp;

ENAME      HIREDATE  Eval Date
---------- --------- ---------
SMITH      17-DEC-80 31-MAY-81
ALLEN      20-FEB-81 31-JUL-81
WARD       22-FEB-81 31-JUL-81
JONES      02-APR-81 30-SEP-81
MARTIN     28-SEP-81 28-FEB-82
BLAKE      01-MAY-81 31-OCT-81
CLARK      09-JUN-81 30-NOV-81
SCOTT      19-APR-87 30-SEP-87
KING       17-NOV-81 30-APR-82
TURNER     08-SEP-81 28-FEB-82
ADAMS      23-MAY-87 31-OCT-87
JAMES      03-DEC-81 31-MAY-82
FORD       03-DEC-81 31-MAY-82
MILLER     23-JAN-82 30-JUN-82
```

LAST_VALUE

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

LAST_VALUE is an analytic function. It returns the last value in an ordered set of values.

You cannot use LAST_VALUE or any other analytic function for *expr*. That is, you can use other built-in function expressions for *expr*, but you cannot nest analytic functions.

See Also: "Expressions" on page 5-2 for information on valid forms of *expr*

Examples

The following example returns the hiredate of the employee earning the highest salary.

```
SELECT ename, sal, hiredate, LAST_VALUE(hiredate) OVER
   (ORDER BY sal
      ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS lv
FROM (SELECT * FROM emp WHERE deptno=20 ORDER BY hiredate);

ENAME             SAL HIREDATE  LV
---------- ---------- --------- ---------
SMITH              800 17-DEC-80 19-APR-87
ADAMS             1100 23-MAY-87 19-APR-87
JONES             2975 02-APR-81 19-APR-87
FORD              3000 03-DEC-81 19-APR-87
SCOTT             3000 19-APR-87 19-APR-87
```

This example illustrates the nondeterministic nature of the LAST_VALUE function. Ford and Scott have the same salary, so they are in adjacent rows. Ford appears first because the rows in the subquery are ordered by `hiredate`. However, if the rows are ordered by hiredate in descending order, as in the next example, the function returns a different value:

```
SELECT ename, sal, hiredate, LAST_VALUE(hiredate) OVER
   (ORDER BY sal
      ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS lv
FROM (SELECT * FROM emp WHERE deptno=20 ORDER BY hiredate DESC);

ENAME             SAL HIREDATE  LV
---------- ---------- --------- ---------
SMITH              800 17-DEC-80 03-DEC-81
ADAMS             1100 23-MAY-87 03-DEC-81
JONES             2975 02-APR-81 03-DEC-81
SCOTT             3000 19-APR-87 03-DEC-81
FORD              3000 03-DEC-81 03-DEC-81
```

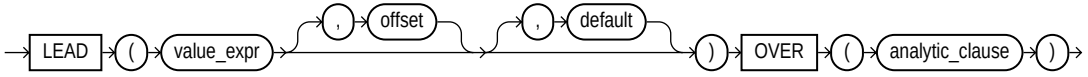
The following two examples show how to make the LAST_VALUE function deterministic by ordering on a unique key. By ordering within the function by both salary and hiredate, you can ensure the same result regardless of the ordering in the subquery.

```
SELECT ename, sal, hiredate, LAST_VALUE(hiredate) OVER
   (ORDER BY sal, hiredate
      ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS lv
FROM (SELECT * FROM emp WHERE deptno=20 ORDER BY hiredate);
```

```
ENAME             SAL HIREDATE  LV
---------- ---------- --------- ---------
SMITH              800 17-DEC-80 19-APR-87
ADAMS             1100 23-MAY-87 19-APR-87
JONES             2975 02-APR-81 19-APR-87
FORD              3000 03-DEC-81 19-APR-87
SCOTT             3000 19-APR-87 19-APR-87

SELECT ename, sal, hiredate, LAST_VALUE(hiredate) OVER
   (ORDER BY sal, hiredate
      ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS lv
FROM (SELECT * FROM emp WHERE deptno=20 ORDER BY hiredate DESC);

ENAME             SAL HIREDATE  LV
---------- ---------- --------- ---------
SMITH              800 17-DEC-80 19-APR-87
ADAMS             1100 23-MAY-87 19-APR-87
JONES             2975 02-APR-81 19-APR-87
FORD              3000 03-DEC-81 19-APR-87
SCOTT             3000 19-APR-87 19-APR-87
```

LEAD

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

LEAD is an analytic function. It provides access to more than one row of a table at the same time without a self-join. Given a series of rows returned from a query and a position of the cursor, LEAD provides access to a row at a given physical offset beyond that position.

If you do not specify *offset*, its default is 1. The optional *default* value is returned if the offset goes beyond the scope of the table. If you do not specify *default*, its default value is null.

You cannot use LEAD or any other analytic function for *value_expr*. That is, you can use other built-in function expressions for *value_expr*, but you cannot nest analytic functions.

> **See Also:** "Expressions" on page 5-2 for information on valid forms of *expr*

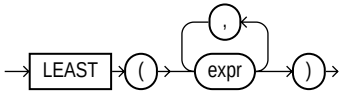
Example

The following example provides, for each employee in the emp table, the hiredate of the employee hired just after:

```
SELECT ename, hiredate,
   LEAD(hiredate, 1) OVER (ORDER BY hiredate) AS "NextHired"
   FROM emp;

ENAME      HIREDATE  NextHired
---------- --------- ---------
SMITH      17-DEC-80 20-FEB-81
ALLEN      20-FEB-81 22-FEB-81
WARD       22-FEB-81 02-APR-81
JONES      02-APR-81 01-MAY-81
BLAKE      01-MAY-81 09-JUN-81
CLARK      09-JUN-81 08-SEP-81
TURNER     08-SEP-81 28-SEP-81
MARTIN     28-SEP-81 17-NOV-81
KING       17-NOV-81 03-DEC-81
JAMES      03-DEC-81 03-DEC-81
FORD       03-DEC-81 23-JAN-82
MILLER     23-JAN-82 19-APR-87
SCOTT      19-APR-87 23-MAY-87
ADAMS      23-MAY-87
```

LEAST

Syntax



Purpose

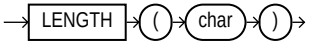
LEAST returns the least of the list of *expr*s. All *expr*s after the first are implicitly converted to the datatype of the first *expr* before the comparison. Oracle compares the *expr*s using nonpadded comparison semantics. If the value returned by this function is character data, its datatype is always VARCHAR2.

Example

```
SELECT LEAST('HARRY','HARRIOT','HAROLD') "LEAST"
     FROM DUAL;

LEAST
------
HAROLD
```

LENGTH

Syntax



Purpose

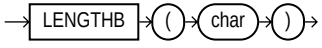
LENGTH returns the length of *char* in characters. If *char* has datatype CHAR, the length includes all trailing blanks. If *char* is null, this function returns null.

Example

```
SELECT LENGTH('CANDIDE') "Length in characters"
FROM DUAL;

Length in characters
--------------------
                   7
```

LENGTHB

Syntax



Purpose

LENGTHB returns the length of *char* in bytes. If *char* is null, this function returns null. For a single-byte database character set, LENGTHB is equivalent to LENGTH.

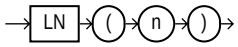
Example

This example assumes a double-byte database character set.

```
SELECT LENGTHB ('CANDIDE') "Length in bytes"
     FROM DUAL;

Length in bytes
---------------
             14
```

LN

Syntax



Purpose

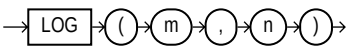
LN returns the natural logarithm of *n*, where *n* is greater than 0.

Example

```
SELECT LN(95) "Natural log of 95" FROM DUAL;

Natural log of 95
-----------------
       4.55387689
```

LOG

Syntax



Purpose

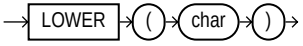
LOG returns the logarithm, base *m*, of *n*. The base *m* can be any positive number other than 0 or 1 and *n* can be any positive number.

Example

```
SELECT LOG(10,100) "Log base 10 of 100" FROM DUAL;

Log base 10 of 100
------------------
                 2
```

LOWER

Syntax



Purpose

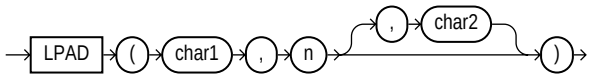
LOWER returns *char*, with all letters lowercase. The return value has the same datatype as the argument *char* (CHAR or VARCHAR2).

Example

```
SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase"
   FROM DUAL;

Lowercase
--------------------
mr. scott mcmillan
```

LPAD

Syntax



Purpose

LPAD returns *char1*, left-padded to length *n* with the sequence of characters in *char2*; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* that fits in *n*.

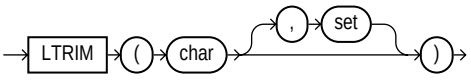
The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Example

```
SELECT LPAD('Page 1',15,'*.') "LPAD example"
     FROM DUAL;

LPAD example
---------------
*.*.*.*.*Page 1
```

LTRIM

Syntax



Purpose

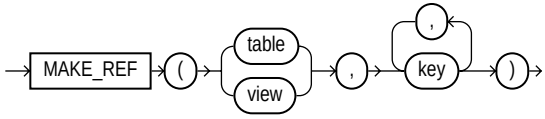
LTRIM removes characters from the left of *char*, with all the leftmost characters that appear in *set* removed; *set* defaults to a single blank. If *char* is a character literal, you must enclose it in single quotes. Oracle begins scanning *char* from its first character and removes all characters that appear in set until reaching a character not in set and then returns the result.

Example

```
SELECT LTRIM('xyxXxyLAST WORD','xy') "LTRIM example"
     FROM DUAL;

LTRIM example
------------
XxyLAST WORD
```

MAKE_REF

Syntax



Purpose

MAKE_REF creates a REF to a row of an object view or a row in an object table whose object identifier is primary key based.

See Also:

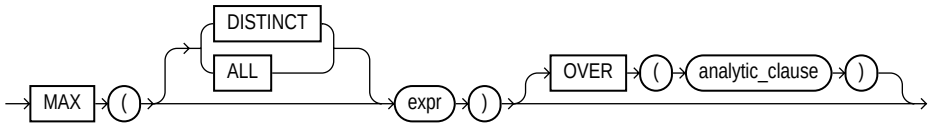
- *Oracle8i Application Developer's Guide - Fundamentals* for more information about object views
- DEREF on page 4-35

Example

```
CREATE TABLE employee (eno NUMBER, ename VARCHAR2(20),
   salary NUMBER, PRIMARY KEY (eno, ename));
CREATE TYPE emp_type AS OBJECT
   (eno NUMBER, ename CHAR(20), salary NUMBER);
CREATE VIEW emp_view OF emp_type
   WITH OBJECT IDENTIFIER (eno, ename)
   AS SELECT * FROM emp;
SELECT MAKE_REF(emp_view, 1, 'jack') FROM DUAL;

MAKE_REF(EMP_VIEW,1,'JACK')
----------------------------------------------------------
000067030A0063420D06E06F3C00C1E03400400B40DCB10000001C26010001000200
2900000000000F06008101001401000002A0007000A8401FE0000001F02C102146A61
636B20202020202020202020202020202020000000000000000000000000000000000
00000000
```

MAX

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

MAX returns maximum value of *expr*. You can use it as an aggregate or analytic function.

If you specify DISTINCT, you can specify only the *query_partition_clause* of the *analytic_clause*. The *ORDER_BY_clause* and *windowing_clause* are not allowed.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

```
SELECT MAX(sal) "Maximum" FROM emp;

   Maximum
----------
      5000
```

Analytic Example

The following example calculates, for each employee, the highest salary of the employees reporting to the same manager as the employee.

```
SELECT mgr, ename, sal,
   MAX(sal) OVER (PARTITION BY mgr) AS mgr_max
   FROM emp;
```

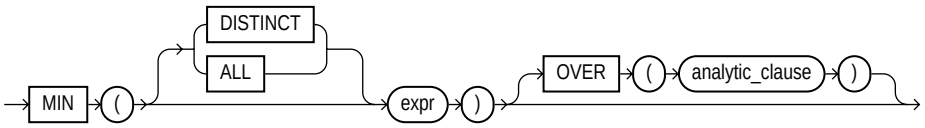
```
      MGR  ENAME      SAL         MGR_MAX
---------- ---------- ---------- ----------
      7566 SCOTT            3000       3000
      7566 FORD             3000       3000
      7698 ALLEN            1600       1600
      7698 WARD             1250       1600
      7698 JAMES             950       1600
      7698 TURNER           1500       1600
      7698 MARTIN           1250       1600
      7782 MILLER           1300       1300
      7788 ADAMS            1100       1100
      7839 JONES            2975       2975
      7839 CLARK            2450       2975
      7839 BLAKE            2850       2975
      7902 SMITH             800        800
           KING             5000       5000
```

If you enclose this query in the parent query with a predicate, you can determine the employee who makes the highest salary in each department:

```
SELECT mgr, ename, sal
   FROM (SELECT mgr, ename, sal,
      MAX(sal) OVER (PARTITION BY mgr) AS rmax_sal
      FROM emp)
      WHERE sal = rmax_sal;

       MGR ENAME             SAL
---------- ---------- ----------
      7566 SCOTT            3000
      7566 FORD             3000
      7698 ALLEN            1600
      7782 MILLER           1300
      7788 ADAMS            1100
      7839 JONES            2975
      7902 SMITH             800
           KING             5000
```

MIN

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

`MIN` returns minimum value of *`expr`*. You can use it as an aggregate or analytic function.

If you specify `DISTINCT`, you can specify only the *`query_partition_clause`* of the *`analytic_clause`*. The *`ORDER_BY_clause`* and *`windowing_clause`* are not allowed.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

```
SELECT MIN(hiredate) "Earliest" FROM emp;

Earliest
---------
17-DEC-80
```

Analytic Example

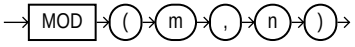
The following example determines, for each employee, the employees who were hired on or before the same date as the employee. It then determines the subset of employees reporting to the same manager as the employee, and returns the lowest salary in that subset.

```
SELECT mgr, ename, hiredate, sal,
   MIN(sal) OVER(PARTITION BY mgr ORDER BY hiredate
```

```
   RANGE UNBOUNDED PRECEDING) as p_cmin
   FROM emp;

       MGR ENAME      HIREDATE          SAL     P_CMIN
---------- ---------- --------- ---------- ----------
      7566 FORD       03-DEC-81       3000       3000
      7566 SCOTT      19-APR-87       3000       3000
      7698 ALLEN      20-FEB-81       1600       1600
      7698 WARD       22-FEB-81       1250       1250
      7698 TURNER     08-SEP-81       1500       1250
      7698 MARTIN     28-SEP-81       1250       1250
      7698 JAMES      03-DEC-81        950        950
      7782 MILLER     23-JAN-82       1300       1300
      7788 ADAMS      23-MAY-87       1100       1100
      7839 JONES      02-APR-81       2975       2975
      7839 BLAKE      01-MAY-81       2850       2850
      7839 CLARK      09-JUN-81       2450       2450
      7902 SMITH      17-DEC-80        800        800
           KING       17-NOV-81       5000       5000
```

MOD

Syntax



Purpose

MOD returns remainder of *m* divided by *n*. Returns *m* if *n* is 0.

Example

```
SELECT MOD(11,4) "Modulus" FROM DUAL;

   Modulus
----------
         3
```

This function behaves differently from the classical mathematical modulus function when *m* is negative. The classical modulus can be expressed using the MOD function with this formula:

```
m - n * FLOOR(m/n)
```

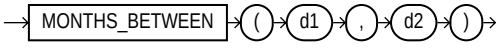
The following statement illustrates the difference between the MOD function and the classical modulus:

```
SELECT m, n, MOD(m, n),
m - n * FLOOR(m/n) "Classical Modulus"
  FROM test_mod_table;

         M          N   MOD(M,N) Classical Modulus
---------- ---------- ---------- -----------------
        11          4          3                 3
        11         -4          3                -1
       -11          4         -3                 1
       -11         -4         -3                -3
```

See Also: FLOOR on page 4-40

MONTHS_BETWEEN

Syntax



Purpose

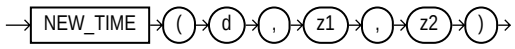
MONTHS_BETWEEN returns number of months between dates *d1* and *d2*. If *d1* is later than *d2*, result is positive; if earlier, negative. If *d1* and *d2* are either the same days of the month or both last days of months, the result is always an integer. Otherwise Oracle calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *d1* and *d2*.

Example

```
SELECT MONTHS_BETWEEN
   (TO_DATE('02-02-1995','MM-DD-YYYY'),
    TO_DATE('01-01-1995','MM-DD-YYYY') ) "Months"
    FROM DUAL;

    Months
----------
1.03225806
```

NEW_TIME

Syntax



Purpose

NEW_TIME returns the date and time in time zone *z2* when date and time in time zone *z1* are *d*. Before using this function, you must set the NLS_DATE_FORMAT parameter to display 24-hour time.

The arguments *z1* and *z2* can be any of these text strings:

- AST, ADT: Atlantic Standard or Daylight Time
- BST, BDT: Bering Standard or Daylight Time
- CST, CDT: Central Standard or Daylight Time
- EST, EDT: Eastern Standard or Daylight Time
- GMT: Greenwich Mean Time
- HST, HDT: Alaska-Hawaii Standard Time or Daylight Time.
- MST, MDT: Mountain Standard or Daylight Time
- NST: Newfoundland Standard Time
- PST, PDT: Pacific Standard or Daylight Time
- YST, YDT: Yukon Standard or Daylight Time

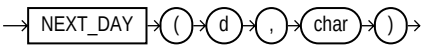
Example

The following example returns an Atlantic Standard time, given the Pacific Standard time equivalent:

```
ALTER SESSION SET NLS_DATE_FORMAT =
   'DD-MON-YYYY HH24:MI:SS';

SELECT NEW_TIME(TO_DATE(
   '11-10-99 01:23:45', 'MM-DD-YY HH24:MI:SS'),
   'AST', 'PST') "New Date and Time" FROM DUAL;

New Date and Time
--------------------
09-NOV-1999 21:23:45
```

NEXT_DAY

Syntax



Purpose

NEXT_DAY returns the date of the first weekday named by *char* that is later than the date *d*. The argument *char* must be a day of the week in the date language of your session, either the full name or the abbreviation. The minimum number of letters required is the number of letters in the abbreviated version. Any characters immediately following the valid abbreviation are ignored. The return value has the same hours, minutes, and seconds component as the argument *d*.

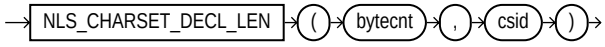
Example

This example returns the date of the next Tuesday after March 15, 1998.

```
SELECT NEXT_DAY('15-MAR-98','TUESDAY') "NEXT DAY"
     FROM DUAL;

NEXT DAY
---------
16-MAR-98
```

NLS_CHARSET_DECL_LEN

Syntax



Purpose

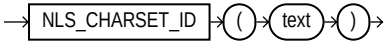
NLS_CHARSET_DECL_LEN returns the declaration width (in number of characters) of an NCHAR column. The *bytecnt* argument is the width of the column. The *csid* argument is the character set ID of the column.

Example

```
SELECT NLS_CHARSET_DECL_LEN
  (200, nls_charset_id('ja16eucfixed'))
   FROM DUAL;

NLS_CHARSET_DECL_LEN(200,NLS_CHARSET_ID('JA16EUCFIXED'))
--------------------------------------------------------
                     100
```

NLS_CHARSET_ID

Syntax



Purpose

NLS_CHARSET_ID returns the NLS character set ID number corresponding to NLS character set name, *text*. The *text* argument is a run-time VARCHAR2 value. The *text* value 'CHAR_CS' returns the database character set ID number of the server. The *text* value 'NCHAR_CS' returns the national character set ID number of the server.

Invalid character set names return null.

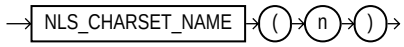
Example

```
SELECT NLS_CHARSET_ID('ja16euc')
  FROM DUAL;

NLS_CHARSET_ID('JA16EUC')
-------------------------
                      830
```

See Also: *Oracle8i National Language Support Guide* for a list of character set names

NLS_CHARSET_NAME

Syntax



Purpose

NLS_CHARSET_NAME returns the name of the NLS character set corresponding to ID number *n*. The character set name is returned as a VARCHAR2 value in the database character set.

If *n* is not recognized as a valid character set ID, this function returns null.

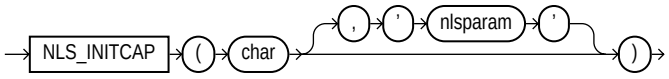
Example

```
SELECT NLS_CHARSET_NAME(2)
  FROM DUAL;

NLS_CH
------
WE8DEC
```

See Also: *Oracle8i National Language Support Guide* for a list of character set IDs

NLS_INITCAP

Syntax



Purpose

NLS_INITCAP returns *char*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric. The value of '*nlsparam*' can have this form:

```
'NLS_SORT = sort'
```

where *sort* is either a linguistic sort sequence or BINARY. The linguistic sort sequence handles special linguistic requirements for case conversions. These requirements can result in a return value of a different length than the *char*. If you omit '*nlsparam*', this function uses the default sort sequence for your session.

Example

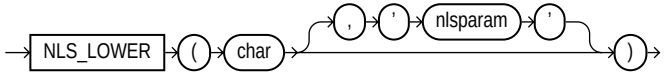
The following examples show how the linguistic sort sequence results in a different return value from the function:

```
SELECT NLS_INITCAP
   ('ijsland') "InitCap" FROM DUAL;

InitCap
-------
Ijsland

SELECT NLS_INITCAP
   ('ijsland', 'NLS_SORT = XDutch') "InitCap"
   FROM DUAL;

InitCap
-------
IJsland
```

> **See Also:** *Oracle8i National Language Support Guide* for information on sort sequences

NLS_LOWER

Syntax



Purpose

NLS_LOWER returns *char*, with all letters lowercase. The '*nlsparam*' can have the same form and serve the same purpose as in the NLS_INITCAP function.

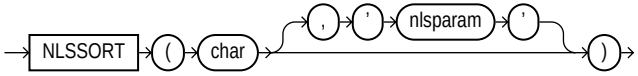
Example

```
SELECT NLS_LOWER
```

```
   ('CITTA''', 'NLS_SORT = XGerman') "Lowercase"
   FROM DUAL;

Lower
-----
cittá
```

NLSSORT

Syntax



Purpose

NLSSORT returns the string of bytes used to sort *char*. The value of '*nlsparams*' can have the form

```
'NLS_SORT = sort'
```

where *sort* is a linguistic sort sequence or BINARY. If you omit '*nlsparams*', this function uses the default sort sequence for your session. If you specify BINARY, this function returns *char*.

Example

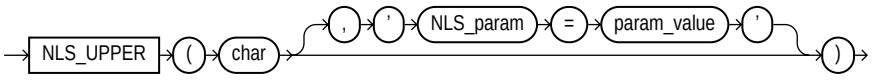
This function can be used to specify comparisons based on a linguistic sort sequence rather than on the binary value of a string:

```
SELECT ename FROM emp
   WHERE NLSSORT (ename, 'NLS_SORT = German')
   > NLSSORT ('S', 'NLS_SORT = German') ORDER BY ename;

ENAME
----------
SCOTT
SMITH
TURNER
WARD
```

> **See Also:** *Oracle8i National Language Support Guide* for information on sort sequences

NLS_UPPER

Syntax



Purpose

NLS_UPPER returns *char*, with all letters uppercase. The '*nlsparam*' can have the same form and serve the same purpose as in the NLS_INITCAP function.

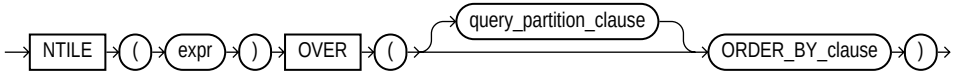
Example

```
SELECT NLS_UPPER
   ('große', 'NLS_SORT = XGerman') "Uppercase"
     FROM DUAL;

Upper
-----
GROSS
```

See Also: NLS_INITCAP on page 4-64

NTILE

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

NTILE is an analytic function. It divides an ordered dataset into a number of buckets indicated by *expr* and assigns the appropriate bucket number to each row. The buckets are numbered 1 through *expr*, and *expr* must resolve to a positive constant for each partition.

The number of rows in the buckets can differ by at most 1. The remainder values (the remainder of number of rows divided by buckets) are distributed 1 per bucket, starting with bucket 1.

If *expr* is greater than the number of rows, a number of buckets equal to the number of rows will be filled, and the remaining buckets will be empty.

You cannot use `NTILE` or any other analytic function for *expr*. That is, you can use other built-in function expressions for *expr*, but you cannot nest analytic functions.

> **See Also:** "Expressions" on page 5-2 for information on valid forms of *expr*

Example

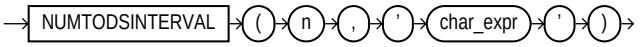
The following example divides the values in the SAL column into 4 buckets. The SAL column has 14 values, so the two extra values (the remainder of 14 / 4) are allocated to buckets 1 and 2, which therefore have one more value than buckets 3 or 4.

```
SELECT ename, sal, NTILE(4) OVER (ORDER BY sal DESC) AS quartile
   FROM emp;

ENAME              SAL   QUARTILE
---------- ---------- ----------
KING             5000          1
SCOTT            3000          1
FORD             3000          1
JONES            2975          1
BLAKE            2850          2
CLARK            2450          2
ALLEN            1600          2
TURNER           1500          2
MILLER           1300          3
WARD             1250          3
MARTIN           1250          3
ADAMS            1100          4
JAMES             950          4
SMITH             800          4
```

NUMTODSINTERVAL

> **Note:** This function is restricted to use with analytic functions. It accepts only numbers as arguments, and returns interval literals. See "Analytic Functions" on page 4-8 and "Interval" on page 2-36.

Syntax



Purpose

NUMTODSINTERVAL converts *n* to an INTERVAL DAY TO SECOND literal. *n* can be a number or an expression resolving to a number. The value for *char_expr* specifies the unit of *n* and must resolve to one of the following string values:

- 'DAY'
- 'HOUR'
- 'MINUTE'
- 'SECOND'

char_expr is case insensitive. Leading and trailing values within the parentheses are ignored. By default, precision of the return is 9.

Example

The following example calculates for each employee, the number of employees hired, by the same manager, within the last 100 days from his / her hiredate:

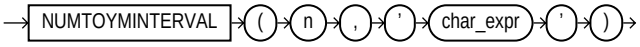
```
SELECT mgr, ename, hiredate,
   COUNT(*) OVER (PARTITION BY mgr ORDER BY hiredate
   RANGE NUMTODSINTERVAL(100, 'day') PRECEDING) AS t_count
   FROM emp;

       MGR ENAME      HIREDATE     T_COUNT
---------- ---------- --------- ----------
      7566 FORD       03-DEC-81          1
      7566 SCOTT      19-APR-87          1
      7698 ALLEN      20-FEB-81          1
      7698 WARD       22-FEB-81          2
      7698 TURNER     08-SEP-81          1
```

```
7698 MARTIN     28-SEP-81          2
7698 JAMES      03-DEC-81          3
7782 MILLER     23-JAN-82          1
7788 ADAMS      23-MAY-87          1
7839 JONES      02-APR-81          1
7839 BLAKE      01-MAY-81          2
7839 CLARK      09-JUN-81          3
7902 SMITH      17-DEC-80          1
     KING       17-NOV-81          1
```

NUMTOYMINTERVAL

Note: This function is restricted to use with analytic functions. It accepts only numbers as arguments, and returns interval literals. See "Analytic Functions" on page 4-8 and "Interval" on page 2-36.

Syntax



Purpose

NUMTOYMINTERVAL converts number *n* to an INTERVAL YEAR TO MONTH literal. *n* can be a number or an expression resolving to a number. The value for *char_expr* specifies the unit of *n*, and must resolve to one of the following string values:

- 'YEAR'
- 'MONTH'

char_expr is case insensitive. Leading and trailing values within the parentheses are ignored. By default, precision of the return is 9.

Example

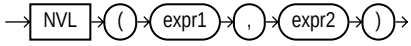
The following example calculates, for each employee, the total salary of employees hired in the past one year from his/her hiredate.

```
SELECT ename, hiredate, sal, SUM(sal) OVER (ORDER BY hiredate
   RANGE NUMTOYMINTERVAL(1,'year') PRECEDING) AS t_sal
   FROM emp;

ENAME      HIREDATE         SAL      T_SAL
```

```
---------- --------- ---------- ----------
SMITH      17-DEC-80        800        800
ALLEN      20-FEB-81       1600       2400
WARD       22-FEB-81       1250       3650
JONES      02-APR-81       2975       6625
BLAKE      01-MAY-81       2850       9475
CLARK      09-JUN-81       2450      11925
TURNER     08-SEP-81       1500      13425
MARTIN     28-SEP-81       1250      14675
KING       17-NOV-81       5000      19675
JAMES      03-DEC-81        950      23625
FORD       03-DEC-81       3000      23625
MILLER     23-JAN-82       1300      24125
SCOTT      19-APR-87       3000       3000
ADAMS      23-MAY-87       1100       4100
```

NVL

Syntax



Purpose

If *expr1* is null, NVL returns *expr2*; if *expr1* is not null, NVL returns *expr1*. The arguments *expr1* and *expr2* can have any datatype. If their datatypes are different, Oracle converts *expr2* to the datatype of *expr1* before comparing them. The datatype of the return value is always the same as the datatype of *expr1*, unless *expr1* is character data, in which case the return value's datatype is VARCHAR2.

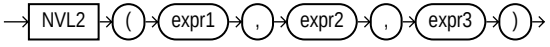
Example

```
SELECT ename, NVL(TO_CHAR(COMM), 'NOT APPLICABLE')
   "COMMISSION" FROM emp
   WHERE deptno = 30;

ENAME      COMMISSION
---------- ------------------------------------
ALLEN      300
WARD       500
MARTIN     1400
```

```
BLAKE      NOT APPLICABLE
TURNER     0
JAMES      NOT APPLICABLE
```

NVL2

Syntax



Purpose

If *expr1* is not null, NVL2 returns *expr2*. If *expr1* is null, NVL2 returns *expr3*. The argument *expr1* can have any datatype. The arguments *expr2* and *expr3* can have any datatypes except LONG.

If the datatypes of *expr2* and *expr3* are different, Oracle converts *expr3* to the datatype of *expr2* before comparing them unless *expr3* is a null constant. In that case, a datatype conversion is not necessary.

The datatype of the return value is always the same as the datatype of *expr2*, unless *expr2* is character data, in which case the return value's datatype is VARCHAR2.

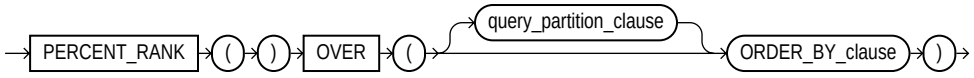
Example

The following example shows whether the income of each employee in department 30 is made up of salary plus commission, or just salary, depending on whether the comm column of emp is null or not.

```
SELECT ename, NVL2(TO_CHAR(COMM), 'SAL & COMM', 'SAL') income
   FROM emp WHERE deptno = 30;

ENAME      INCOME
---------- ----------
ALLEN      SAL & COMM
WARD       SAL & COMM
MARTIN     SAL & COMM
BLAKE      SAL
TURNER     SAL & COMM
JAMES      SAL
```

PERCENT_RANK

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

PERCENT_RANK is an analytic function, and is similar to the CUME_DIST (cumulative distribution) function. For a row R, PERCENT_RANK calculates the rank of R minus 1, divided by 1 less than the number of rows being evaluated (the entire query result set or a partition). The range of values returned by PERCENT_RANK is 0 to 1, inclusive. The first row in any set has a PERCENT_RANK of 0.

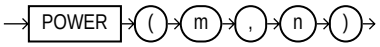
Example

The following example calculates, for each employee, the percent rank of the employee's salary within the department:

```
SELECT deptno, ename, sal,
   PERCENT_RANK() OVER (PARTITION BY deptno ORDER BY sal DESC) AS pr
   FROM emp;

    DEPTNO ENAME             SAL         PR
---------- ---------- ---------- ----------
        10 KING             5000          0
        10 CLARK            2450         .5
        10 MILLER           1300          1
        20 SCOTT            3000          0
        20 FORD             3000          0
        20 JONES            2975         .5
        20 ADAMS            1100        .75
        20 SMITH             800          1
        30 BLAKE            2850          0
        30 ALLEN            1600         .2
        30 TURNER           1500         .4
        30 WARD             1250         .6
        30 MARTIN           1250         .6
        30 JAMES             950          1
```

POWER

Syntax



Purpose

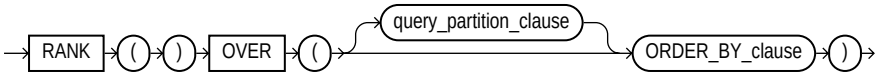
POWER returns *m* raised to the *n*th power. The base *m* and the exponent *n* can be any numbers, but if *m* is negative, *n* must be an integer.

Example

```
SELECT POWER(3,2) "Raised" FROM DUAL;

    Raised
----------
         9
```

RANK

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

RANK is an analytic function. It computes the rank of each row returned from a query with respect to the other rows returned by the query, based on the values of the *value_exprs* in the *ORDER_BY_clause*. Rows with equal values for the ranking criteria receive the same rank. Oracle then adds the number of tied rows to the tied rank to calculate the next rank. Therefore, the ranks may not be consecutive numbers.

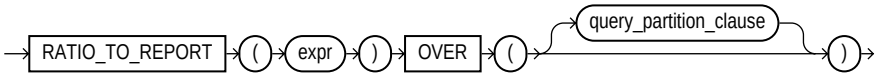
Example

The following statement ranks the employees within each department based on their salary and commission. Identical salary values receive the same rank and cause nonconsecutive ranks. Compare this example with the example for DENSE_RANK on page 4-34.

```
SELECT deptno, ename, sal, comm,
   RANK() OVER (PARTITION BY deptno ORDER BY sal DESC, comm) as rk
   FROM emp;

    DEPTNO ENAME              SAL       COMM         RK
---------- ---------- ---------- ---------- ----------
        10 KING             5000                    1
        10 CLARK            2450                    2
        10 MILLER           1300                    3
        20 SCOTT            3000                    1
        20 FORD             3000                    1
        20 JONES            2975                    3
        20 ADAMS            1100                    4
        20 SMITH             800                    5
        30 BLAKE            2850                    1
        30 ALLEN            1600        300         2
        30 TURNER           1500          0         3
        30 WARD             1250        500         4
        30 MARTIN           1250       1400         5
        30 JAMES             950                    6
```

RATIO_TO_REPORT

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

RATIO_TO_REPORT is an analytic function. It computes the ratio of a value to the sum of a set of values. If *expr* evaluates to null, the ratio-to-report value also evaluates to null.

The set of values is determined by the *query_partition_clause*. If you omit that clause, the ratio-to-report is computed over all rows returned by the query.

You cannot use RATIO_TO_REPORT or any other analytic function for *expr*. That is, you can use other built-in function expressions for *expr*, but you cannot nest analytic functions.

> **See Also:** "Expressions" on page 5-2 for information on valid forms of *expr*

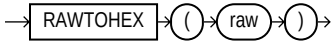
Example

The following example calculates the ratio-to-report of each salesperson's salary to the total of all salespeople's salaries:

```
SELECT ename, sal, RATIO_TO_REPORT(sal) OVER () AS rr
   FROM emp
   WHERE job = 'SALESMAN';

ENAME              SAL         RR
---------- ---------- ----------
ALLEN            1600 .285714286
WARD             1250 .223214286
MARTIN           1250 .223214286
TURNER           1500 .267857143
```

RAWTOHEX

Syntax



Purpose

RAWTOHEX converts *raw* to a character value containing its hexadecimal equivalent.

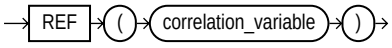
Example

```
SELECT RAWTOHEX(raw_column) "Graphics"
   FROM graphics;

Graphics
--------
7D
```

See Also: "RAW and LONG RAW Datatypes" on page 2-16 and HEXTORAW on page 4-42

REF

Syntax



Purpose

In a SQL statement, REF takes as its argument a correlation variable (table alias) associated with a row of an object table or an object view. A REF value is returned for the object instance that is bound to the variable or row.

Example

```
CREATE TYPE emp_type AS OBJECT
   (eno NUMBER, ename VARCHAR2(20), salary NUMBER);
CREATE TABLE emp_table OF emp_type
   (primary key (eno, ename));
INSERT INTO emp_table VALUES (10, 'jack', 50000);
SELECT REF(e) FROM emp_table e;

REF(E)
----------------------------------------------------
0000280209420D2FEABD9400C3E03400400B40DCB1420D2FEABD9300C3E03400400B
40DCB1004049EE0000
```

See Also: *Oracle8i Concepts*

REFTOHEX

Syntax

Purpose

REFTOHEX converts argument *expr* to a character value containing its hexadecimal equivalent. *expr* must return a REF.

Example

```
CREATE TYPE emp_type AS OBJECT
   (eno NUMBER, ename VARCHAR2(20), salary NUMBER);
CREATE TABLE emp_table OF emp_type
   (primary key (eno, ename));
CREATE TABLE dept
   (dno NUMBER, mgr REF emp_type SCOPE IS emp);
INSERT INTO emp_table VALUES (10, 'jack', 50000);
INSERT INTO dept SELECT 10, REF(e) FROM emp_table e;
SELECT REFTOHEX(mgr) FROM dept;

REFTOHEX(MGR)
--------------------------------------------------------
0000220208420D2FEABD9400C3E03400400B40DCB1420D2FEABD9300C3E03400400B
40DCB1
```

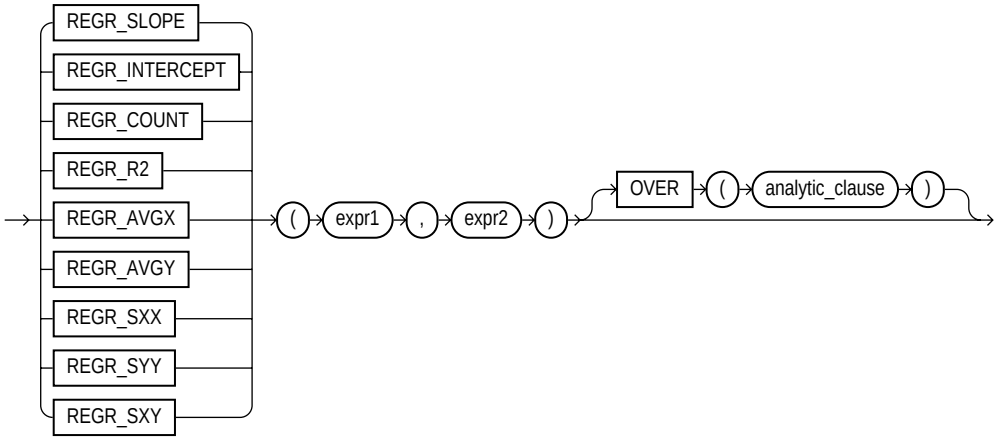
REGR_ (linear regression) functions

The linear regression functions are:

- REGR_SLOPE
- REGR_INTERCEPT
- REGR_COUNT
- REGR_R2
- REGR_AVGX
- REGR_AVGY
- REGR_SXX

- REGR_SYY
- REGR_SXY

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

The linear regression functions fit an ordinary-least-squares regression line to a set of number pairs. You can use them as both aggregate and analytic functions.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Oracle applies the function to the set of (*expr1*, *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Oracle computes all the regression functions simultaneously during a single pass through the data.

expr1 is interpreted as a value of the dependent variable (a "y value"), and *expr2* is interpreted as a value of the independent variable (an "x value"). Both expressions must be numbers.

- REGR_SLOPE returns the slope of the line. The return value is a number and can be null. After the elimination of null (*expr1*, *expr2*) pairs, it makes the following computation:

  ```
  COVAR_POP(expr1, expr2) / VAR_POP(expr2)
  ```

- REGR_INTERCEPT returns the y-intercept of the regression line. The return value is a number and can be null. After the elimination of null (*expr1*, *expr2*) pairs, it makes the following computation:

  ```
  AVG(expr1) - REGR_SLOPE(expr1, expr2) * AVG(expr2)
  ```

- REGR_COUNT returns an integer that is the number of non-null number pairs used to fit the regression line.

- REGR_R2 returns the coefficient of determination (also called "R-squared" or "goodness of fit") for the regression. The return value is a number and can be null. VAR_POP(*expr1*) and VAR_POP(*expr2*) are evaluated after the elimination of null pairs. The return values are:

  ```
                          NULL if VAR_POP(expr2)  = 0

                             1 if VAR_POP(expr1)  = 0 and
                                  VAR_POP(expr2) != 0

  POWER(CORR(expr1,expr),2) if VAR_POP(expr1)  > 0 and
                                  VAR_POP(expr2  != 0
  ```

All of the remaining regression functions return a number and can be null:

- REGR_AVGX evaluates the average of the independent variable (*expr2*) of the regression line. It makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  AVG(expr2)
  ```

- REGR_AVGY evaluates the average of the dependent variable (*expr1*) of the regression line. It makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  AVG(expr1)
  ```

REGR_SXY, REGR_SXX, REGR_SYY are auxiliary functions that are used to compute various diagnostic statistics.

- REGR_SXX makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

```
REGR_COUNT(expr1, expr2) * VAR_POP(expr2)
```

- REGR_SYY makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * VAR_POP(expr1)
  ```

- REGR_SXY makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * COVAR_POP(expr1, expr2)
  ```

The following examples are based on the `sales` table, described in COVAR_POP on page 4-29.

REGR_SLOPE and REGR_INTERCEPT Examples

The following example determines the slope and intercept of the regression line for the amount of sales and sale profits for each year.

```
SELECT s_year,
   REGR_SLOPE(s_amount, s_profit),
   REGR_INTERCEPT(s_amount, s_profit)
FROM sales GROUP BY s_year;

 S_YEAR     REGR_SLOPE REGR_INTER
---------- ---------- ----------
      1998 128.401558 -2277.5684
      1999  55.618655 226.855296
```

The following example determines the cumulative slope and cumulative intercept of the regression line for the amount of sales and sale profits for each day in 1998:

```
SELECT s_year, s_month, s_day,
   REGR_SLOPE(s_amount, s_profit)
      OVER (ORDER BY s_month, s_day) AS CUM_SLOPE,
   REGR_INTERCEPT(s_amount, s_profit)
      OVER (ORDER BY s_month, s_day) AS CUM_ICPT
   FROM sales
   WHERE s_year=1998
   ORDER BY s_month, s_day;

S_YEAR     S_MONTH    S_DAY      CUM_SLOPE  CUM_ICPT
---------- ---------- ---------- ---------- ----------
      1998          6          5
```

```
      1998          6          9 132.093066 401.884833
      1998          6          9 132.093066 401.884833
      1998          6         10 131.829612  450.65349
      1998          8         21 132.963737  -153.5413
      1998          8         25 130.681718 -451.47349
      1998          8         25 130.681718 -451.47349
      1998          8         26  128.76502 -236.50096
      1998         11          9 131.499934 -1806.7535
      1998         11          9 131.499934 -1806.7535
      1998         11         10 130.190972 -2323.3056
      1998         11         10 130.190972 -2323.3056
      1998         11         11 128.401558 -2277.5684
```

REGR_COUNT Examples

The following example returns the number of sales transactions in the `sales` table that resulted in a profit. (None of the rows for containing a sales amount have a null in the `s_profit` column, so the function returns the total number of rows in the `sales` table.)

```
SELECT REGR_COUNT(s_amount, s_profit) FROM sales;

REGR_COUNT
----------
        23
```

The following example computes, for each day, the cumulative number of transactions within each month for the year 1998:

```
SELECT s_month, s_day,
   REGR_COUNT(s_amount,s_profit)
      OVER (PARTITION BY s_month ORDER BY s_day)
FROM SALES
WHERE S_YEAR=1998
ORDER BY S_MONTH;

S_MONTH    S_DAY      REGR_COUNT
---------- ---------- ----------
         6          5          1
         6          9          3
         6          9          3
         6         10          4
         8         21          1
         8         25          3
         8         25          3
         8         26          4
```

```
        11          9          2
        11          9          2
        11         10          4
        11         10          4
        11         11          5
```

REGR_R2 Examples

The following example computes the coefficient of determination of the regression line for amount of sales and sale profits:

```
SELECT REGR_R2(s_amount, s_profit) FROM sales;

REGR_R2(S_
----------
.942435028
```

The following example computes the cumulative coefficient of determination of the regression line for monthly sales and monthly profits for each month in 1998:

```
SELECT s_month,
       REGR_R2(SUM(s_amount), SUM(s_profit))
              OVER (ORDER BY s_month)
FROM SALES
WHERE s_year=1998
GROUP BY s_month
ORDER BY s_month;

S_MONTH    REGR_R2(SU
---------- ----------
         6
         8          1
        11 .740553632
```

REGR_AVGY and REGR_AVGX Examples

The following example calculates the regression average for the amount of sales and sale profits for each year:

```
SELECT s_year,
   REGR_AVGY(s_amount, s_profit),
   REGR_AVGX(s_amount, s_profit)
FROM sales GROUP BY s_year;

 S_YEAR     REGR_AVGY( REGR_AVGX(
---------- ---------- ----------
```

```
      1998 41227.5462 338.820769
      1999   7330.748    127.725
```

The following example calculates the cumulative averages for the amount of sales and sale profits in 1998:

```
SELECT s_year, s_month, s_day,
   REGR_AVGY(s_amount, s_profit)
      OVER (ORDER BY s_month, s_day) AS CUM_AMOUNT,
   REGR_AVGX(s_amount, s_profit)
      OVER (ORDER BY s_month, s_day) AS CUM_PROFIT
   FROM sales
   WHERE s_year=1998
   ORDER BY s_month, s_day;

S_YEAR     S_MONTH    S_DAY      CUM_AMOUNT CUM_PROFIT
---------- ---------- ---------- ---------- ----------
      1998          6          5      16068      118.2
      1998          6          9 44375.6667      332.9
      1998          6          9 44375.6667      332.9
      1998          6         10   52678.25    396.175
      1998          8         21   44721.72      337.5
      1998          8         25    45333.8 350.357143
      1998          8         25    45333.8 350.357143
      1998          8         26    47430.7   370.1875
      1998         11          9   41892.91    332.317
      1998         11          9   41892.91    332.317
      1998         11         10  40777.175 331.055833
      1998         11         10  40777.175 331.055833
      1998         11         11 41227.5462 338.820769
```

REGR_SXY, REGR_SXX, and REGR_SYY Examples

The following example computes the REGR_SXY, REGR_SXX, and REGR_SYY values for the regression analysis of amount of sales and sale profits for each year:

```
SELECT s_year,
   REGR_SXY(s_amount, s_profit),
       REGR_SYY(s_amount, s_profit),
       REGR_SXX(s_amount, s_profit)
FROM sales GROUP BY s_year;

S_YEAR     REGR_SXY(S REGR_SYY(S REGR_SXX(S
---------- ---------- ---------- ----------
      1998 48723551.8 6423698688 379462.311
```

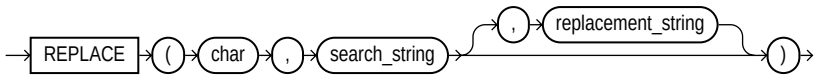
```
      1999 3605361.62  200525751 64822.8841
```

The following example computes the cumulative REGR_SXY, REGR_SXX, and REGR_SYY statistics for amount of sales and sale profits for each month-day value in 1998:

```
SELECT s_year, s_month, s_day,
       REGR_SXY(s_amount, s_profit)
              OVER (ORDER BY s_month, s_day) AS CUM_SXY,
       REGR_SYY(s_amount, s_profit)
              OVER (ORDER BY s_month, s_day) AS CUM_SXY,
       REGR_SXX(s_amount, s_profit)
              OVER (ORDER BY s_month, s_day) AS CUM_SXX
   FROM sales
   WHERE s_year=1998
   ORDER BY s_month, s_day;

S_YEAR     S_MONTH    S_DAY      CUM_SXY    CUM_SXY    CUM_SXX
---------- ---------- ---------- ---------- ---------- ----------
      1998          6          5          0          0          0
      1998          6          9 14822857.8 1958007601  112215.26
      1998          6          9 14822857.8 1958007601  112215.26
      1998          6         10 21127009.3 2785202281 160259.968
      1998          8         21 30463997.3 4051329674  229115.08
      1998          8         25 34567985.3 4541739739 264520.437
      1998          8         25 34567985.3 4541739739 264520.437
      1998          8         26 36896592.7 4787971157 286542.049
      1998         11          9 45567995.3 6045196901 346524.854
      1998         11          9 45567995.3 6045196901 346524.854
      1998         11         10 48178003.8 6392056557 370056.411
      1998         11         10 48178003.8 6392056557 370056.411
      1998         11         11 48723551.8 6423698688 379462.311
```

REPLACE

Syntax



Purpose

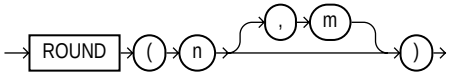
REPLACE returns *char* with every occurrence of *search_string* replaced with *replacement_string*. If *replacement_string* is omitted or null, all occurrences of *search_string* are removed. If *search_string* is null, *char* is returned. This function provides a superset of the functionality provided by the TRANSLATE function. TRANSLATE provides single-character, one-to-one substitution. REPLACE lets you substitute one string for another as well as to remove character strings.

Example

```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
     FROM DUAL;

Changes
--------------
BLACK and BLUE
```

ROUND (number function)

Syntax



Purpose

ROUND returns *n* rounded to *m* places right of the decimal point. If *m* is omitted, *n* is rounded to 0 places. *m* can be negative to round off digits left of the decimal point. *m* must be an integer.

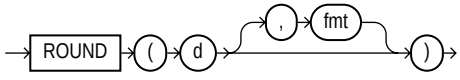
Examples

```
SELECT ROUND(15.193,1) "Round" FROM DUAL;

     Round
----------
      15.2


SELECT ROUND(15.193,-1) "Round" FROM DUAL;
```

```
     Round
----------
        20
```

ROUND (date function)

Syntax



Purpose

ROUND returns *d* rounded to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is rounded to the nearest day.

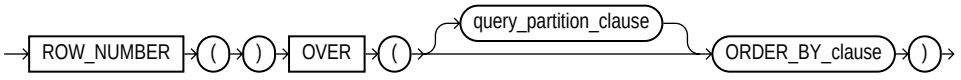
> **See Also:** "ROUND and TRUNC Date Functions" on page 4-127 for the permitted format models to use in *fmt*

Example

```
SELECT ROUND (TO_DATE ('27-OCT-92'),'YEAR')
   "New Year" FROM DUAL;

New Year
---------
01-JAN-93
```

ROW_NUMBER

Syntax



> **See Also:** "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

ROW_NUMBER is an analytic function. It assigns a unique number to each row to which it is applied (either each row in the partition or each row returned by the query), in the ordered sequence of rows specified in the *ORDER_BY_clause*, beginning with 1.

You cannot use ROW_NUMBER or any other analytic function for *expr*. That is, you can use other built-in function expressions for *expr*, but you cannot nest analytic functions.

> **See Also:** "Expressions" on page 5-2 for information on valid forms of *expr*

Example

For each department in the emp table, the following example assigns numbers to each row in order of employee's hire date:

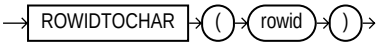
```
SELECT deptno, ename, hiredate, ROW_NUMBER()
   OVER (PARTITION BY deptno ORDER BY hiredate) AS emp_id
   FROM emp;

    DEPTNO ENAME      HIREDATE      EMP_ID
---------- ---------- --------- ----------
        10 CLARK      09-JUN-81          1
        10 KING       17-NOV-81          2
        10 MILLER     23-JAN-82          3
        20 SMITH      17-DEC-80          1
        20 JONES      02-APR-81          2
        20 FORD       03-DEC-81          3
        20 SCOTT      19-APR-87          4
        20 ADAMS      23-MAY-87          5
        30 ALLEN      20-FEB-81          1
        30 WARD       22-FEB-81          2
        30 BLAKE      01-MAY-81          3
        30 TURNER     08-SEP-81          4
        30 MARTIN     28-SEP-81          5
        30 JAMES      03-DEC-81          6
```

ROW_NUMBER is a nondeterministic function. However, hiredate is a unique key, so the results of this application of the function are deterministic.

> **See Also:** FIRST_VALUE on page 4-38 and LAST_VALUE on page 4-47 for examples of nondeterministic behavior

ROWIDTOCHAR

Syntax



Purpose

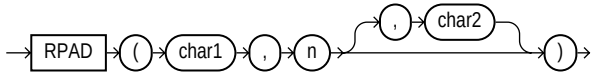
ROWIDTOCHAR converts a rowid value to VARCHAR2 datatype. The result of this conversion is always 18 characters long.

Example

```
SELECT ROWID
     FROM offices
     WHERE
     ROWIDTOCHAR(ROWID) LIKE '%Br1AAB%';

ROWID
------------------
AAAAZ6AABAAABr1AAB
```

RPAD

Syntax



Purpose

RPAD returns *char1*, right-padded to length *n* with *char2*, replicated as many times as necessary; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* that fits in *n*.

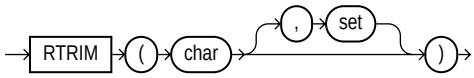
The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Example

```
SELECT RPAD('MORRISON',12,'ab') "RPAD example"
     FROM DUAL;

RPAD example
-----------------
MORRISONabab
```

RTRIM

Syntax



Purpose

`RTRIM` returns *`char`*, with all the rightmost characters that appear in *`set`* removed; *`set`* defaults to a single blank. If *`char`* is a character literal, you must enclose it in single quotes. `RTRIM` works similarly to `LTRIM`.

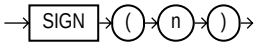
Example

```
SELECT RTRIM('BROWNINGyxXxy','xy') "RTRIM e.g."
     FROM DUAL;

RTRIM e.g
-------------
BROWNINGyxX
```

See Also: LTRIM on page 4-54

SIGN

Syntax



Purpose

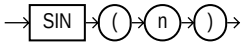
If *n*<0, SIGN returns -1. If *n*=0, the function returns 0. If *n*>0, SIGN returns 1.

Example

```
SELECT SIGN(-15) "Sign" FROM DUAL;

      Sign
----------
        -1
```

SIN

Syntax



Purpose

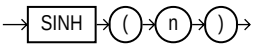
SIN returns the sine of *n* (an angle expressed in radians).

Example

```
SELECT SIN(30 * 3.14159265359/180)
 "Sine of 30 degrees" FROM DUAL;

Sine of 30 degrees
------------------
                .5
```

SINH

Syntax



Purpose

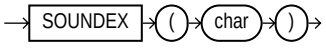
SINH returns the hyperbolic sine of *n*.

Example

```
SELECT SINH(1) "Hyperbolic sine of 1" FROM DUAL;

Hyperbolic sine of 1
--------------------
          1.17520119
```

SOUNDEX

Syntax



Purpose

SOUNDEX returns a character string containing the phonetic representation of *char*. This function allows you to compare words that are spelled differently, but sound alike in English.

The phonetic representation is defined in *The Art of Computer Programming*, Volume 3: Sorting and Searching, by Donald E. Knuth, as follows:

- Retain the first letter of the string and remove all other occurrences of the following letters: a, e, h, i, o, u, w, y.
- Assign numbers to the remaining letters (after the first) as follows:

  ```
  b, f, p, v = 1
  c, g, j, k, q, s, x, z = 2
  d, t = 3
  l = 4
  m, n = 5
  r = 6
  ```

- If two or more letters with the same number were adjacent in the original name (before step 1), or adjacent except for any intervening h and w, omit all but the first.
- Return the first four bytes padded with 0.

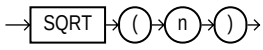
Example

```
SELECT ename
     FROM emp
```

```
     WHERE SOUNDEX(ename)
         = SOUNDEX('SMYTHE');

ENAME
----------
SMITH
```

SQRT

Syntax



Purpose

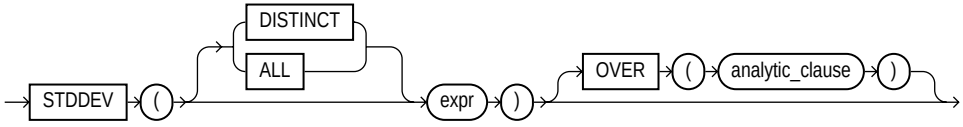
SQRT returns square root of *n*. The value *n* cannot be negative. SQRT returns a "real" result.

Example

```
SELECT SQRT(26) "Square root" FROM DUAL;

Square root
-----------
5.09901951
```

STDDEV

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

STDDEV returns sample standard deviation of *expr*, a set of numbers. You can use it as both an aggregate and analytic function. It differs from STDDEV_SAMP in that STDDEV returns zero when it has only 1 row of input data, whereas STDDEV_SAMP returns a null.

Oracle calculates the standard deviation as the square root of the variance defined for the VARIANCE aggregate function.

If you specify DISTINCT, you can specify only the *query_partition_clause* of the *analytic_clause*. The *ORDER_BY_clause* and *windowing_clause* are not allowed.

See Also:

- "Aggregate Functions" on page 4-6, VARIANCE on page 4-125, and STDDEV_SAMP on page 4-96
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

```
SELECT STDDEV(sal) "Deviation"
   FROM emp;

 Deviation
----------
1182.50322
```

Analytic Example

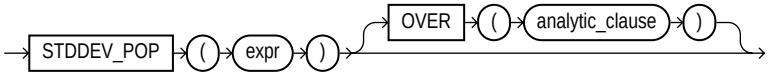
The query in the following example returns the cumulative standard deviation of salary values in Department 30 ordered by hiredate:

```
SELECT ENAME, SAL, STDDEV(SAL) OVER (ORDER BY HIREDATE)
   FROM EMP
   WHERE DEPTNO=30;

ENAME      SAL        STDDEV(SAL
---------- ---------- ----------
ALLEN            1600          0
WARD             1250 247.487373
BLAKE            2850 841.130192
TURNER           1500 715.308791
MARTIN           1250 666.520817
```

```
JAMES                950 668.331255
```

STDDEV_POP

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

STDDEV_POP computes the population standard deviation and returns the square root of the population variance. You can use it as both an aggregate and analytic function.

The *expr* is a number expression, and the function returns a value of type NUMBER. This function is same as the square root of the VAR_POP function. When VAR_POP returns null, this function returns null.

See Also:

- "Aggregate Functions" on page 4-6 and VAR_POP on page 4-122
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

The following example returns the population and sample standard deviations of profit from sales in the SALES table.

```
SELECT STDDEV_POP(s_profit), STDDEV_SAMP(s_profit) FROM sales;

STDDEV_POP STDDEV_SAM
---------- ----------
173.975774 177.885831
```

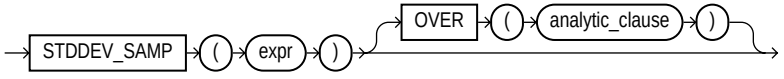
Analytic Example

The following example returns the population standard deviations of salaries in the emp table by department:

```
SELECT deptno, ename, sal,
   STDDEV_POP(sal) OVER (PARTITION BY deptno) AS pop_std
   FROM emp;

    DEPTNO ENAME              SAL    POP_STD
---------- ---------- ---------- ----------
        10 CLARK            2450 1546.14215
        10 KING             5000 1546.14215
        10 MILLER           1300 1546.14215
        20 SMITH             800 1004.73877
        20 ADAMS            1100 1004.73877
        20 FORD             3000 1004.73877
        20 SCOTT            3000 1004.73877
        20 JONES            2975 1004.73877
        30 ALLEN            1600 610.100174
        30 BLAKE            2850 610.100174
        30 MARTIN           1250 610.100174
        30 JAMES             950 610.100174
        30 TURNER           1500 610.100174
        30 WARD             1250 610.100174
```

STDDEV_SAMP

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

STDDEV_SAMP computes the cumulative sample standard deviation and returns the square root of the sample variance. You can use it as both an aggregate and analytic function.

The *expr* is a number expression, and the function returns a value of type NUMBER. This function is same as the square root of the VAR_SAMP function. When VAR_SAMP returns null, this function returns null.

See Also:

- "Aggregate Functions" on page 4-6 and VAR_SAMP on page 4-123
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

The following example returns the population and sample standard deviations of profit from sales in the SALES table.

```
SELECT STDDEV_POP(s_profit), STDDEV_SAMP(s_profit) FROM sales;

STDDEV_POP STDDEV_SAM
---------- ----------
173.975774 177.885831
```

Analytic Example

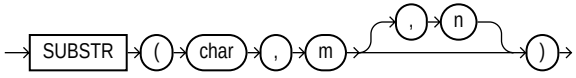
The following example returns the sample standard deviation of salaries in the EMP table by department:

```
SELECT deptno, ename, hiredate, sal,
   STDDEV_SAMP(sal) OVER (PARTITION BY deptno ORDER BY hiredate
   ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cum_sdev
   FROM emp;

    DEPTNO ENAME      HIREDATE         SAL   CUM_SDEV
---------- ---------- --------- ---------- ----------
        10 CLARK      09-JUN-81       2450
        10 KING       17-NOV-81       5000 1803.12229
        10 MILLER     23-JAN-82       1300 1893.62967
        20 SMITH      17-DEC-80        800
        20 JONES      02-APR-81       2975 1537.95725
        20 FORD       03-DEC-81       3000 1263.01557
        20 SCOTT      19-APR-87       3000  1095.8967
        20 ADAMS      23-MAY-87       1100  1123.3321
        30 ALLEN      20-FEB-81       1600
        30 WARD       22-FEB-81       1250 247.487373
        30 BLAKE      01-MAY-81       2850 841.130192
```

```
        30 TURNER     08-SEP-81       1500 715.308791
        30 MARTIN     28-SEP-81       1250 666.520817
        30 JAMES      03-DEC-81        950 668.331255
```

SUBSTR

Syntax



Purpose

SUBSTR returns a portion of *char*, beginning at character *m*, *n* characters long.

- If *m* is 0, it is treated as 1.
- If *m* is positive, Oracle counts from the beginning of *char* to find the first character.
- If *m* is negative, Oracle counts backwards from the end of *char*.
- If *n* is omitted, Oracle returns all characters to the end of *char*. If n is less than 1, a null is returned.

Floating-point numbers passed as arguments to SUBSTR are automatically converted to integers.

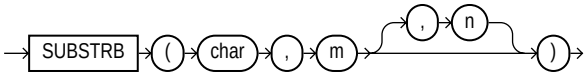
```
SELECT SUBSTR('ABCDEFG',3,4) "Substring"
     FROM DUAL;

Substring
---------
CDEF
```

Example 2

```
SELECT SUBSTR('ABCDEFG',-5,4) "Substring"
     FROM DUAL;

Substring
---------
CDEF
```

SUBSTRB

Syntax



Purpose

SUBSTRB is the same as SUBSTR, except that the arguments *m* and *n* are expressed in bytes, rather than in characters. For a single-byte database character set, SUBSTRB is equivalent to SUBSTR.

Floating-point numbers passed as arguments to SUBSTRB are automatically converted to integers.

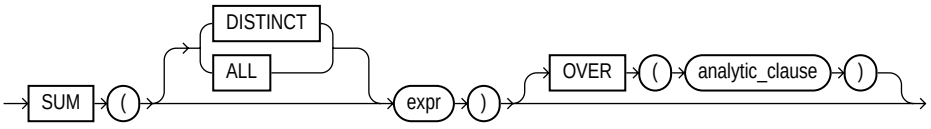
Example

Assume a double-byte database character set:

```
SELECT SUBSTRB('ABCDEFG',5,4.2)
     "Substring with bytes"
     FROM DUAL;

Substring with bytes
--------------------
CD
```

SUM

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

SUM returns sum of values of *expr*. You can use it as an aggregate or analytic function.

If you specify DISTINCT, you can specify only the *query_partition_clause* of the *analytic_clause*. The *ORDER_BY_clause* and *windowing_clause* are not allowed.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the sum of all salaries in the emp table:

```
SELECT SUM(sal) "Total"
     FROM emp;

     Total
----------
     29025
```

Analytic Example

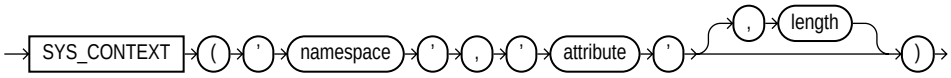
The following example calculates, for each manager, a cumulative total of salaries of employees who answer to that manager that are equal to or less than the current salary:

```
SELECT mgr, ename, sal,
   SUM(sal) OVER (PARTITION BY mgr ORDER BY sal
   RANGE UNBOUNDED PRECEDING) l_csum
   FROM emp;

       MGR ENAME              SAL     L_CSUM
---------- ---------- ---------- ----------
      7566 SCOTT            3000       6000
      7566 FORD             3000       6000
      7698 JAMES             950        950
      7698 WARD             1250       3450
      7698 MARTIN           1250       3450
      7698 TURNER           1500       4950
      7698 ALLEN            1600       6550
      7782 MILLER           1300       1300
```

```
7788 ADAMS            1100       1100
7839 CLARK            2450       2450
7839 BLAKE            2850       5300
7839 JONES            2975       8275
7902 SMITH             800        800
     KING             5000       5000
```

SYS_CONTEXT

Syntax



Purpose

SYS_CONTEXT returns the value of *attribute* associated with the context *namespace*. You can use this function in both SQL and PL/SQL statements. The context *namespace* must already have been created, and the associated *attribute* and its value must also have been set using the DBMS_SESSION.set_context procedure. The *namespace* must be a valid SQL identifier. The *attribute* name can be any string, and it is not case sensitive, but it cannot exceed 30 bytes in length.

The datatype of the return value is VARCHAR2. The default maximum size of the return value is 256 bytes. You can override this default by specifying the optional *length* parameter. The valid range of values is 1 to 4000 bytes. (If you specify an invalid value, Oracle ignores it and uses the default.)

Oracle8*i* provides a built-in namespace called USERENV, which describes the current session. The predefined attributes of namespace USERENV are listed Table 4–1 on page 4-102, along with the lengths of their return strings.

See Also:

- *Oracle8i Application Developer's Guide - Fundamentals* for information on using the application context feature in your application development
- CREATE CONTEXT on page 9-13 for information on creating user-defined context namespaces
- *Oracle8i Supplied PL/SQL Packages Reference* for information on the DBMS_SESSION.set_context procedure

Examples

The following statement returns the name of the user who logged onto the database:

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER')
   FROM DUAL;

SYS_CONTEXT ('USERENV', 'SESSION_USER')
----------------------------------------------------
SCOTT
```

The following example returns the group number that was set as the value for the attribute group_no in the PL/SQL package that was associated with the context hr_apps when hr_apps was created:

```
SELECT SYS_CONTEXT ('hr_apps', 'group_no') "User Group"
   FROM DUAL;

User Group
----------
Sales
```

Table 4–1 Predefined Attributes of Namespace USERENV

Attribute	Return Value	Return Length (bytes)
AUTHENTICATION_DATA	Data being used to authenticate the login user. For X.503 certificate authenticated sessions, this field returns the context of the certificate in HEX2 format.	256

Table 4–1 Predefined Attributes of Namespace USERENV

Attribute	Return Value	Return Length (bytes)
	Note: You can change the return value of the AUTHENTICATION_DATA attribute using the *length* parameter of the syntax. Values of up to 4000 are accepted. This is the only attribute of USERENV for which Oracle implements such a change.	
AUTHENTICATION_TYPE	How the user was authenticated: ■ DATABASE: username / password authentication ■ OS: operating system external user authentication ■ NETWORK: network protocol or ANO authentication ■ PROXY: OCI proxy connection authentication	30
BG_JOB_ID	Job ID of the current session if it was established by an Oracle background process. Null if the session was not established by a background process.	30
CLIENT_INFO	Returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package.	64
CURRENT_SCHEMA	Name of the default schema being used in the current schema. This value can be changed during the session with an ALTER SESSION SET CURRENT_SCHEMA statement.	30
CURRENT_SCHEMAID	Identifier of the default schema being used in the current session.	30
CURRENT_USER	The name of the user whose privilege the current session is under.	30
CURRENT_USERID	User ID of the user whose privilege the current session is under	30
DB_DOMAIN	Domain of the database as specified in the DB_DOMAIN initialization parameter.	256
DB_NAME	Name of the database as specified in the DB_NAME initialization parameter	30

Table 4–1 Predefined Attributes of Namespace USERENV

Attribute	Return Value	Return Length (bytes)
ENTRYID	The available auditing entry identifier. You cannot use this option in distributed SQL statements. To use this keyword in USERENV, the initialization parameter AUDIT_TRAIL must be set to true.	30
EXTERNAL_NAME	External name of the database user. For SSL authenticated sessions using v.503 certificates, this field returns the distinguished name (DN) stored in the user certificate.	256
FG_JOB_ID	Job ID of the current session if it was established by a client foreground process. Null if the session was not established by a foreground process.	30
HOST	Name of the host machine from which the client has connected.	54
INSTANCE	The instance identification number of the current instance.	30
IP_ADDRESS	IP address of the machine from which the client is connected.	30
ISDBA	TRUE if you currently have the DBA role enabled and FALSE if you do not.	30
LANG	The ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.	62
LANGUAGE	The language and territory currently used by your session, along with the database character set, in this form: language_territory.characterset	52
NETWORK_PROTOCOL	Network protocol being used for communication, as specified in the 'PROTOCOL=*protocol*' portion of the connect string.	256
NLS_CALENDAR	The current calendar of the current session.	62
NLS_CURRENCY	The currency of the current session.	62
NLS_DATE_FORMAT	The date format for the session.	62
NLS_DATE_LANGUAGE	The language used for expressing dates.	62
NLS_SORT	BINARY or the linguistic sort basis.	62

Table 4–1 Predefined Attributes of Namespace USERENV

Attribute	Return Value	Return Length (bytes)
NLS_TERRITORY	The territory of the current session.	62
OS_USER	Operating system username of the client process that initiated the database session	30
PROXY_USER	Name of the database user who opened the current session on behalf of SESSION_USER.	30
PROXY_USERID	Identifier of the database user who opened the current session on behalf of SESSION_USER.	30
SESSION_USER	Database user name by which the current user is authenticated. This value remains the same throughout the duration of the session.	30
SESSION_USERID	Identifier of the database user name by which the current user is authenticated.	30
SESSIONID	The auditing session identifier. You cannot use this option in distributed SQL statements.	30
TERMINAL	The operating system identifier for the client of the current session. In distributed SQL statements, this option returns the identifier for your local session. In a distributed environment, this is supported only for remote SELECT statements, not for remote INSERT, UPDATE, or DELETE operations. (The return length of this parameter may vary by operating system.)	10

SYS_GUID

Syntax

Purpose

SYS_GUID generates and returns a globally unique identifier (RAW value) made up of 16 bytes. On most platforms, the generated identifier consists of a host identifier and a process or thread identifier of the process or thread invoking the function, and a nonrepeating value (sequence of bytes) for that process or thread.

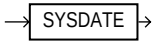
Example

The following examples return the 32-character hexadecimal representation of the 16-byte raw value of the global unique identifier:

```
CREATE TABLE mytable (col1 VARCHAR2(10), col2 RAW(32));
INSERT INTO mytable VALUES ('BOB', SYS_GUID());
SELECT * FROM mytable;

COL1       COL2
---------- --------------------------------------------------
BOB        5901B85D996C570CE03400400B40DCB1

SELECT SYS_GUID() FROM DUAL;

SYS_GUID()
--------------------------------
5901B85D996D570CE03400400B40DCB1
```

SYSDATE

Syntax



Purpose

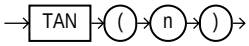
SYSDATE returns the current date and time. Requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a CHECK constraint.

Example

```
SELECT TO_CHAR
    (SYSDATE, 'MM-DD-YYYY HH24:MI:SS')"NOW"
     FROM DUAL;

NOW
-------------------
10-29-1999 20:27:11
```

TAN

Syntax



Purpose

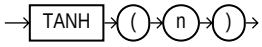
TAN returns the tangent of *n* (an angle expressed in radians).

Example

```
SELECT TAN(135 * 3.14159265359/180)
"Tangent of 135 degrees"  FROM DUAL;

Tangent of 135 degrees
----------------------
                    - 1
```

TANH

Syntax



Purpose

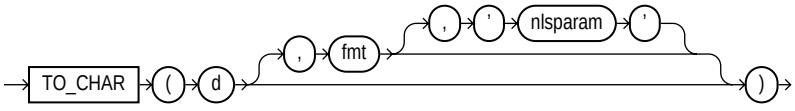
TANH returns the hyperbolic tangent of *n*.

Example

```
SELECT TANH(.5) "Hyperbolic tangent of .5"
   FROM DUAL;

Hyperbolic tangent of .5
------------------------
               .462117157
```

TO_CHAR (date conversion)

Syntax



Purpose

TO_CHAR converts *d* of DATE datatype to a value of VARCHAR2 datatype in the format specified by the date format *fmt*. If you omit *fmt*, *d* is converted to a VARCHAR2 value in the default date format.

The '*nlsparams*' specifies the language in which month and day names and abbreviations are returned. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit *nlsparams*, this function uses the default date language for your session.

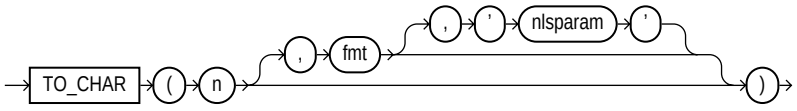
> **See Also:** "Format Models" on page 2-41 for information on date formats

Example

```
SELECT TO_CHAR(HIREDATE, 'Month DD, YYYY')
    "New date format" FROM emp
    WHERE ename = 'BLAKE';

New date format
------------------
May       01, 1981
```

TO_CHAR (number conversion)

Syntax



Purpose

TO_CHAR converts *n* of NUMBER datatype to a value of VARCHAR2 datatype, using the optional number format *fmt*. If you omit *fmt*, *n* is converted to a VARCHAR2 value exactly long enough to hold its significant digits.

The '*nlsparams*' specifies these characters that are returned by number format elements:

- Decimal character
- Group separator
- Local currency symbol
- International currency symbol

This argument can have this form:

```
'NLS_NUMERIC_CHARACTERS = ''dg''
   NLS_CURRENCY = ''text''
   NLS_ISO_CURRENCY = territory '
```

The characters *d* and *g* represent the decimal character and group separator, respectively. They must be different single-byte characters. Note that within the quoted string, you must use two single quotation marks around the parameter values. Ten characters are available for the currency symbol.

If you omit '*nlsparams*' or any one of the parameters, this function uses the default parameter values for your session.

> **See Also:** "Format Models" on page 2-41 for information on number formats

Examples

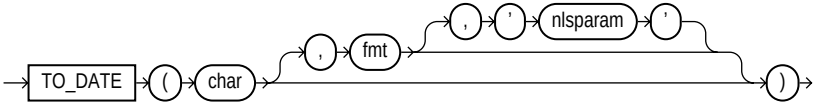
In this example, the output is blank padded to the left of the currency symbol.

```
SELECT TO_CHAR(-10000,'L99G999D99MI') "Amount"
     FROM DUAL;

Amount
--------------
  $10,000.00-

SELECT TO_CHAR(-10000,'L99G999D99MI',
   'NLS_NUMERIC_CHARACTERS = '',.''
   NLS_CURRENCY = ''AusDollars'' ') "Amount"
     FROM DUAL;

Amount
-------------------
AusDollars10.000,00-
```

Note: In the optional number format *fmt*, L designates local currency symbol and MI designates a trailing minus sign. See Table 2–7 on page 2-44 for a complete listing of number format elements.

TO_DATE

Syntax



Purpose

TO_DATE converts *char* of CHAR or VARCHAR2 datatype to a value of DATE datatype. The *fmt* is a date format specifying the format of *char*. If you omit *fmt*, *char* must be in the default date format. If *fmt* is 'J', for Julian, then *char* must be an integer.

The '*nlsparams*' has the same purpose in this function as in the TO_CHAR function for date conversion.

Do not use the TO_DATE function with a DATE value for the *char* argument. The first 2 digits of the returned DATE value can differ from the original *char*, depending on *fmt* or the default date format.

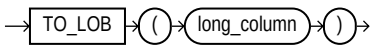
> **See Also:** "Date Format Models" on page 2-47

Example

```
INSERT INTO bonus (bonus_date)
  SELECT TO_DATE(
    'January 15, 1989, 11:00 A.M.',
    'Month dd, YYYY, HH:MI A.M.',
     'NLS_DATE_LANGUAGE = American')
     FROM DUAL;
```

TO_LOB

Syntax



Purpose

TO_LOB converts LONG or LONG RAW values in the column *long_column* to LOB values. You can apply this function only to a LONG or LONG RAW column, and only in the SELECT list of a subquery in an INSERT statement.

Before using this function, you must create a LOB column to receive the converted LONG values. To convert LONGs, create a CLOB column. To convert LONG RAWs, create a BLOB column.

> **See Also:** INSERT on page 11-51 for information on the subquery of an INSERT statement

Example

Given the following tables:

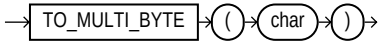
```
CREATE TABLE long_table (n NUMBER, long_col LONG);
CREATE TABLE lob_table (n NUMBER, lob_col CLOB);
```

use this function to convert LONG to LOB values as follows:

```
INSERT INTO lob_table
  SELECT n, TO_LOB(long_col) FROM long_table;
```

TO_MULTI_BYTE

Syntax

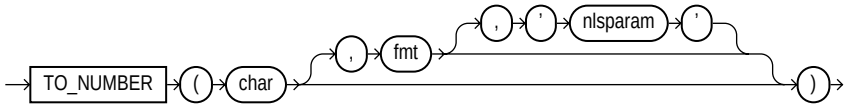


Purpose

TO_MULTI_BYTE returns *char* with all of its single-byte characters converted to their corresponding multibyte characters. Any single-byte characters in *char* that have no multibyte equivalents appear in the output string as single-byte characters. This function is useful only if your database character set contains both single-byte and multibyte characters.

TO_NUMBER

Syntax



Purpose

TO_NUMBER converts *char*, a value of CHAR or VARCHAR2 datatype containing a number in the format specified by the optional format model *fmt*, to a value of NUMBER datatype.

Examples

```
UPDATE emp SET sal = sal +
   TO_NUMBER('100.00', '9G999D99')
   WHERE ename = 'BLAKE';
```

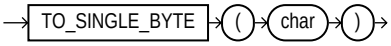
The '*nlsparams*' string in this function has the same purpose as it does in the TO_CHAR function for number conversions.

See Also: "TO_CHAR (number conversion)" on page 4-109

```
SELECT TO_NUMBER('-AusDollars100','L9G999D99',
   ' NLS_NUMERIC_CHARACTERS = '',.''
     NLS_CURRENCY            = ''AusDollars''
   ') "Amount"
     FROM DUAL;

    Amount
----------
      -100
```

TO_SINGLE_BYTE

Syntax

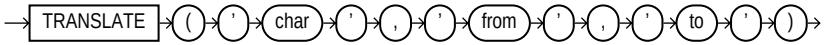


Purpose

TO_SINGLE_BYTE returns *char* with all of its multibyte characters converted to their corresponding single-byte characters. Any multibyte characters in *char* that have no single-byte equivalents appear in the output as multibyte characters. This function is useful only if your database character set contains both single-byte and multibyte characters.

TRANSLATE

Syntax



Purpose

TRANSLATE returns *char* with all occurrences of each character in *from* replaced by its corresponding character in *to*. Characters in *char* that are not in *from* are not replaced. The argument *from* can contain more characters than *to*. In this case, the extra characters at the end of *from* have no corresponding characters in *to*. If these extra characters appear in *char*, they are removed from the return value. You

cannot use an empty string for *to* to remove all characters in *from* from the return value. Oracle interprets the empty string as null, and if this function has a null argument, it returns null.

Examples

The following statement translates a license number. All letters 'ABC...Z' are translated to 'X' and all digits '012 . . . 9' are translated to '9':

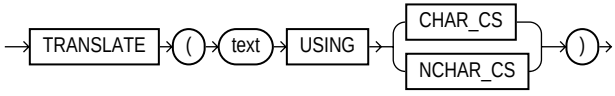
```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'9999999999XXXXXXXXXXXXXXXXXXXXXXXXXX') "License"
     FROM DUAL;

License
--------
9XXX999
```

The following statement returns a license number with the characters removed and the digits remaining:

```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', '0123456789')
"Translate example"
     FROM DUAL;

Translate example
-----------------
2229
```

TRANSLATE ... USING

Syntax



Purpose

TRANSLATE ... USING converts *text* into the character set specified for conversions between the database character set and the national character set.

The *text* argument is the expression to be converted.

Specifying the USING CHAR_CS argument converts *text* into the database character set. The output datatype is VARCHAR2.

Specifying the USING NCHAR_CS argument converts *text* into the national character set. The output datatype is NVARCHAR2.

This function is similar to the Oracle CONVERT function, but must be used instead of CONVERT if either the input or the output datatype is being used as NCHAR or NVARCHAR2.

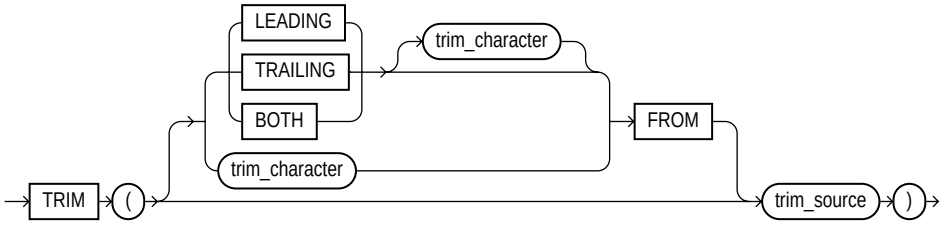
Examples

The examples below use the following table and table values:

```
CREATE TABLE t1 (char_col  CHAR(20),
                 nchar_col nchar(20));
INSERT INTO t1
  VALUES ('Hi', N'Bye');
SELECT * FROM t1;

CHAR_COL     NCHAR_COL
--------     ---------
Hi           Bye

UPDATE t1 SET
  nchar_col = TRANSLATE(char_col USING NCHAR_CS);
UPDATE t1 SET
  char_col = TRANSLATE(nchar_col USING CHAR_CS);
SELECT * FROM t1;

CHAR_COL     NCHAR_COL
--------     ---------
Hi           Hi

UPDATE t1 SET
  nchar_col = TRANSLATE('deo' USING NCHAR_CS);
UPDATE t1 SET
  char_col = TRANSLATE(N'deo' USING CHAR_CS);
SELECT * FROM t1;

CHAR_COL     NCHAR_COL
--------     ---------
deo          deo
```

TRIM

Syntax



Purpose

`TRIM` enables you to trim leading or trailing characters (or both) from a character string. If *`trim_character`* or *`trim_source`* is a character literal, you must enclose it in single quotes.

- If you specify `LEADING`, Oracle removes any leading characters equal to *`trim_character`*.
- If you specify `TRAILING`, Oracle removes any trailing characters equal to *`trim_character`*.
- If you specify `BOTH` or none of the three, Oracle removes leading and trailing characters equal to *`trim_character`*.
- If you do not specify *`trim_character`*, the default value is a blank space.
- If you specify only *`trim_source`*, Oracle removes leading and trailing blank spaces.
- The function returns a value with datatype `VARCHAR2`. The maximum length of the value is the length of *`trim_source`*.
- If either *`trim_source`* or *`trim_character`* is a null value, then the `TRIM` function returns a null value.

This example trims leading and trailing zeroes from a number:

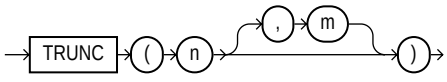
Example

```
SELECT TRIM (0 FROM 0009872348900) "TRIM Example"
   FROM DUAL;

TRIM example
```

```
------------
    98723489
```

TRUNC (number function)

Syntax



Purpose

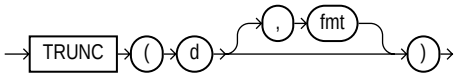
TRUNC returns *n* truncated to *m* decimal places. If *m* is omitted, *n* is truncated to 0 places. *m* can be negative to truncate (make zero) *m* digits left of the decimal point.

Example

```
SELECT TRUNC(15.79,1) "Truncate" FROM DUAL;

  Truncate
----------
      15.7


SELECT TRUNC(15.79,-1) "Truncate" FROM DUAL;

  Truncate
----------
        10
```

TRUNC (date function)

Syntax



Purpose

TRUNC returns *d* with the time portion of the day truncated to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is truncated to the nearest day.

> **See Also:** "ROUND and TRUNC Date Functions" on page 4-127 for the permitted format models to use in *fmt*

Example

```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')
  "New Year" FROM DUAL;

New Year
---------
01-JAN-92
```

UID

Syntax

Purpose

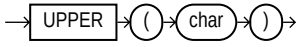
UID returns an integer that uniquely identifies the session user (the user who logged on).

Example

```
SELECT UID FROM DUAL;

       UID
----------
        19
```

UPPER

Syntax



Purpose

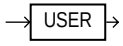
UPPER returns *char*, with all letters uppercase. The return value has the same datatype as the argument *char*.

Example

```
SELECT UPPER('Large') "Uppercase"
     FROM DUAL;

Upper
-----
LARGE
```

USER

Syntax



Purpose

USER returns the name of the session user (the user who logged on) with the datatype VARCHAR2. Oracle compares values of this function with blank-padded comparison semantics.

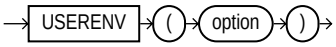
In a distributed SQL statement, the UID and USER functions identify the user on your local database. You cannot use these functions in the condition of a CHECK constraint.

Example

```
SELECT USER, UID FROM DUAL;

USER                                UID
------------------------------ ----------
SCOTT                                  19
```

USERENV

Syntax



Purpose

USERENV returns information of VARCHAR2 datatype about the current session. This information can be useful for writing an application-specific audit trail table or for determining the language-specific characters currently used by your session. You cannot use USERENV in the condition of a CHECK constraint. Table 4–2 describes the values for the *option* argument.

Table 4–2 USERENV Options

Option	Return Value
'CLIENT_INFO'	CLIENT_INFO returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package. **Caution:** Some commercial applications may be using this context value. Check the applicable documentation for those applications to determine what restrictions they may impose on use of this context area. Oracle recommends that you use the application context feature or the SYS_CONTEXT function with the USERENV option. These alternatives are more secure and flexible. **See Also:** - *Oracle8i Concepts* for information on application context - CREATE CONTEXT on page 9-13 and SYS_CONTEXT on page 4-101
'ENTRYID'	ENTRYID returns available auditing entry identifier. You cannot use this option in distributed SQL statements. To use this keyword in USERENV, the initialization parameter AUDIT_TRAIL must be set to true.
'INSTANCE'	INSTANCE returns the instance identification number of the current instance.

Table 4–2 (Cont.) USERENV Options

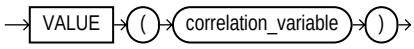
Option	Return Value
'ISDBA'	ISDBA returns 'TRUE' if you currently have the ISDBA role enabled and 'FALSE' if you do not.
'LANG'	LANG returns the ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.
'LANGUAGE'	LANGUAGE returns the language and territory currently used by your session along with the database character set in this form: language_territory.characterset
'SESSIONID'	SESSIONID returns your auditing session identifier. You cannot use this option in distributed SQL statements.
'TERMINAL'	TERMINAL returns the operating system identifier for your current session's terminal. In distributed SQL statements, this option returns the identifier for your local session. In a distributed environment, this is supported only for remote SELECT statements, not for remote INSERT, UPDATE, or DELETE operations.

Example

```
SELECT USERENV('LANGUAGE') "Language" FROM DUAL;

Language
----------------------------------
AMERICAN_AMERICA.WE8DEC
```

VALUE

Syntax



Purpose

In a SQL statement, VALUE takes as its argument a correlation variable (table alias) associated with a row of an object table and returns object instances stored in the object table. The type of the object instances is the same type as the object table.

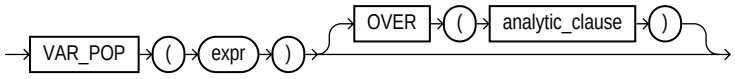
Example

```
CREATE TYPE emp_type AS OBJECT
```

```
   (eno NUMBER, ename VARCHAR2(20), salary NUMBER);
CREATE TABLE emp_table OF emp_type
   (primary key (eno, ename));
INSERT INTO emp_table VALUES (10, 'jack', 50000);
SELECT VALUE(e) FROM emp_table e;

VALUE(E)(ENO, ENAME, SALARY)
--------------------------------------------------
EMP_TYPE(10, 'jack', 50000)
```

VAR_POP

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

VAR_POP returns the population variance of a set of numbers after discarding the nulls in this set. You can use it as both an aggregate and analytic function.

The *expr* is a number expression, and the function returns a value of type NUMBER. If the function is applied to an empty set, it returns null. The function makes the following calculation:

$(SUM(expr^2) - SUM(expr)^2 / COUNT(expr)) / COUNT(expr)$

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

The following example returns the population variance of the salaries in the EMP table:

```
SELECT VAR_POP(sal) FROM emp;

VAR_POP(SAL)
------------
  1298434.31
```

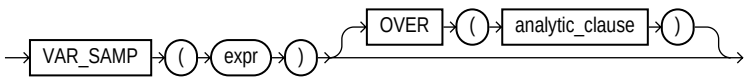
Analytic Example

The following example calculates the cumulative population and sample variances of the monthly sales in 1998:

```
SELECT s_month, VAR_POP(SUM(s_amount)) OVER (ORDER BY s_month),
   VAR_SAMP(SUM(s_amount)) OVER (ORDER BY s_month)
   FROM sales
   WHERE s_year =1998
   GROUP BY s_month;

S_MONTH    VAR_POP(SU VAR_SAMP(S
---------- ---------- ----------
         6          0
         8  440588496  881176992
        11  538819892  808229838
```

VAR_SAMP

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

VAR_SAMP returns the sample variance of a set of numbers after discarding the nulls in this set. You can use it as both an aggregate and analytic function.

The *expr* is a number expression, and the function returns a value of type NUMBER. If the function is applied to an empty set, it returns null. The function makes the following calculation:

$(SUM(expr^2) - SUM(expr)^2 / COUNT(expr)) / (COUNT(expr) - 1)$

This function is similar to VARIANCE, except that given an input set of one element, VARIANCE returns 0 and VAR_SAMP returns null.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

The following example returns the sample variance of the salaries in the emp table.

```
SELECT VAR_SAMP(sal) FROM emp;

VAR_SAMP(SAL)
-------------
   1398313.87
```

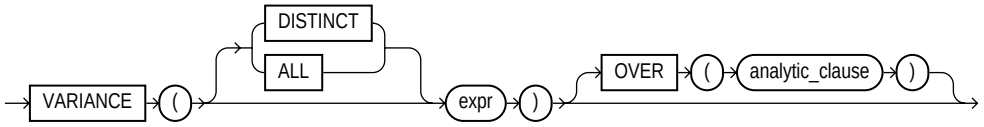
Analytic Example

The following example calculates the cumulative population and sample variances of the monthly sales in 1998:

```
SELECT s_month, VAR_POP(SUM(s_amount)) OVER (ORDER BY s_month),
   VAR_SAMP(SUM(s_amount)) OVER (ORDER BY s_month)
   FROM sales
   WHERE s_year =1998
   GROUP BY s_month;

S_MONTH    VAR_POP(SU VAR_SAMP(S
---------- ---------- ----------
         6          0
         8  440588496  881176992
        11  538819892  808229838
```

VARIANCE

Syntax



See Also: "Analytic Functions" on page 4-8 for information on syntax, semantics, and restrictions

Purpose

VARIANCE returns variance of *expr*. You can use it as an aggregate or analytic function.

Oracle calculates the variance of *expr* as follows:

- 0 if the number of rows in *expr* = 1
- VAR_SAMP if the number of rows in *expr* > 1

If you specify DISTINCT, you can specify only the *query_partition_clause* of the *analytic_clause*. The *ORDER_BY_clause* and *windowing_clause* are not allowed.

See Also:

- "Aggregate Functions" on page 4-6
- "Expressions" on page 5-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the variance of all salaries in the emp table:

```
SELECT VARIANCE(sal) "Variance"
     FROM emp;

Variance
----------
1389313.87
```

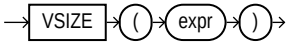
Analytic Example

The query returns the cumulative variance of salary values in Department 30 ordered by hiredate.

```
SELECT ename, sal, VARIANCE(sal) OVER (ORDER BY hiredate)
   FROM emp
   WHERE deptno=30;

ENAME      SAL        VARIANCE(S
---------- ---------- ----------
ALLEN            1600          0
WARD             1250      61250
BLAKE            2850     707500
TURNER           1500 511666.667
MARTIN           1250     444250
JAMES             950 446666.667
```

VSIZE

Syntax



Purpose

VSIZE returns the number of bytes in the internal representation of *expr*. If *expr* is null, this function returns null.

Example

```
SELECT ename, VSIZE (ename) "BYTES"
  FROM emp
  WHERE deptno = 10;

ENAME           BYTES
---------- ----------
CLARK               5
KING                4
MILLER              6
```

ROUND and TRUNC Date Functions

Table 4–3 lists the format models you can use with the ROUND and TRUNC date functions and the units to which they round and truncate dates. The default model, 'DD', returns the date rounded or truncated to the day with a time of midnight.

Table 4–3 Date Format Models for the ROUND and TRUNC Date Functions

Format Model	Rounding or Truncating Unit
CC SCC	One greater than the first two digits of a four-digit year.
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (rounds up on July 1)
IYYY IY IY I	ISO Year
Q	Quarter (rounds up on the sixteenth day of the second month of the quarter)
MONTH MON MM RM	Month (rounds up on the sixteenth day)
WW	Same day of the week as the first day of the year.

Table 4–3 (Cont.) Date Format Models for the ROUND and TRUNC Date Functions

Format Model	Rounding or Truncating Unit
IW	Same day of the week as the first day of the ISO year.
W	Same day of the week as the first day of the month.
DDD DD J	Day
DAY DY D	Starting day of the week
HH HH12 HH24	Hour
MI	Minute

The starting day of the week used by the format models DAY, DY, and D is specified implicitly by the initialization parameter NLS_TERRITORY.

> **See Also:** *Oracle8i Reference* and *Oracle8i National Language Support Guide* for information on this parameter

User-Defined Functions

You can write user-defined functions in PL/SQL or Java to provide functionality that is not available in SQL or SQL functions. User functions can appear in a SQL statement anywhere SQL functions can appear, that is, wherever an expression can occur.

For example, user functions can be used in the following:

- The select list of a SELECT statement
- The condition of a WHERE clause
- CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

See Also:

- CREATE FUNCTION on page 9-43 for information on creating functions, including restrictions on user-defined functions
- *Oracle8i Application Developer's Guide - Fundamentals* for a complete description on the creation and use of user functions

Prerequisites

User functions must be created as top-level functions or declared with a package specification before they can be named within a SQL statement.

To use a user function in a SQL expression, you must own or have EXECUTE privilege on the user function. To query a view defined with a user function, you must have SELECT privileges on the view. No separate EXECUTE privileges are needed to select from the view.

See Also:

- CREATE FUNCTION on page 9-43 for information on creating top-level functions
- CREATE PACKAGE on page 9-122 for information on specifying packaged functions

Name Precedence

Within a SQL statement, the names of database columns take precedence over the names of functions with no parameters. For example, if user `scott` creates the following two objects in his own schema:

```
CREATE TABLE emp(new_sal NUMBER, ...);
CREATE FUNCTION new_sal RETURN NUMBER IS BEGIN ... END;
```

then in the following two statements, the reference to NEW_SAL refers to the column `emp.new_sal`:

```
SELECT new_sal FROM emp;
SELECT emp.new_sal FROM emp;
```

To access the function `new_sal`, you would enter:

```
SELECT scott.new_sal FROM emp;
```

Here are some sample calls to user functions that are allowed in SQL expressions:

```
circle_area (radius)
payroll.tax_rate (empno)
scott.payroll.tax_rate (dependent, empno)@ny
```

Example To call the `tax_rate` user function from schema `scott`, execute it against the `ss_no` and `sal` columns in `tax_table`, and place the results in the variable `income_tax`, specify the following:

```
SELECT scott.tax_rate (ss_no, sal)
    INTO income_tax
    FROM tax_table
    WHERE ss_no = tax_id;
```

Naming Conventions

If only one of the optional schema or package names is given, the first identifier can be either a schema name or a package name. For example, to determine whether PAYROLL in the reference PAYROLL.TAX_RATE is a schema or package name, Oracle proceeds as follows:

1. Check for the PAYROLL package in the current schema.
2. If a PAYROLL package is not found, look for a schema name PAYROLL that contains a top-level TAX_RATE function. If no such function is found, return an error.
3. If the PAYROLL package is found in the current schema, look for a TAX_RATE function in the PAYROLL package. If no such function is found, return an error.

You can also refer to a stored top-level function using any synonym that you have defined for it.

5

Expressions, Conditions, and Queries

This chapter describes how to combine the values, operators, and functions described in earlier chapters evaluate to a value.

This chapter includes these sections:

Expressions

An **expression** is a combination of one or more values, operators, and SQL functions that evaluate to a value. An expression generally assumes the datatype of its components.

This simple expression evaluates to 4 and has datatype NUMBER (the same datatype as its components):

```
2*2
```

The following expression is an example of a more complex expression that uses both functions and operators. The expression adds seven days to the current date, removes the time component from the sum, and converts the result to CHAR datatype:

```
TO_CHAR(TRUNC(SYSDATE+7))
```

You can use expressions in:

- The select list of the SELECT statement
- A condition of the WHERE clause and HAVING clause
- The CONNECT BY, START WITH, and ORDER BY clauses
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement

For example, you could use an expression in place of the quoted string 'smith' in this UPDATE statement SET clause:

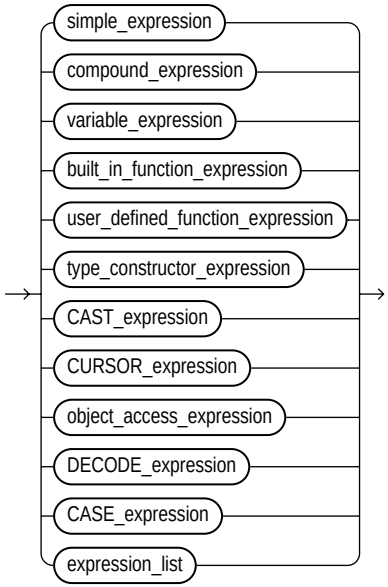
```
SET ename = 'smith';
```

This SET clause has the expression LOWER(ename) instead of the quoted string 'smith':

```
SET ename = LOWER(ename);
```

Expressions have several forms, as shown in the following syntax:

expr::=



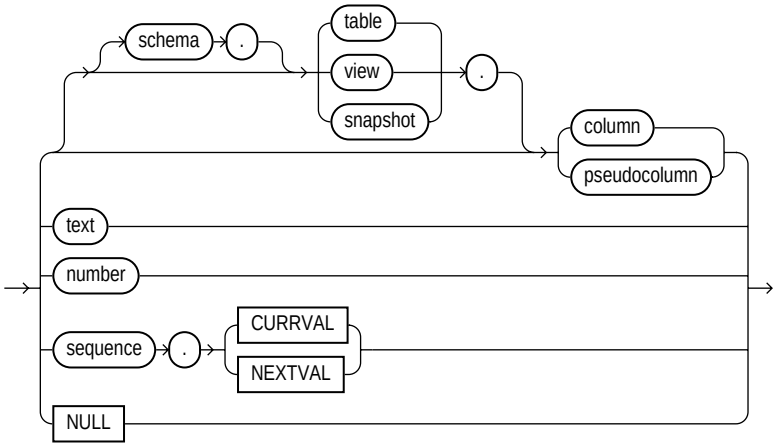
Oracle does not accept all forms of expressions in all parts of all SQL statements. You must use appropriate expression notation whenever *expr* appears in conditions, SQL functions, or SQL statements in other parts of this reference. The sections that follow describe and provide examples of the various forms of expressions.

> **See Also:** The individual SQL statements in Chapter 7 through Chapter 11 for information on restrictions on the expressions in that statement

Simple Expressions

A simple expression specifies column, pseudocolumn, constant, sequence number, or null.

simple_expression::=



In addition to the schema of a user, *schema* can also be "PUBLIC" (double quotation marks required), in which case it must qualify a public synonym for a table, view, or materialized view. Qualifying a public synonym with "PUBLIC" is supported only in data manipulation language (DML) statements, not data definition language (DDL) statements.

The *pseudocolumn* can be either LEVEL, ROWID, or ROWNUM. You can use a pseudocolumn only with a table, not with a view or materialized view. NCHAR and NVARCHAR2 are not valid pseudocolumn datatypes.

> **See Also:** "Pseudocolumns" on page 2-59 for more information on pseudocolumns

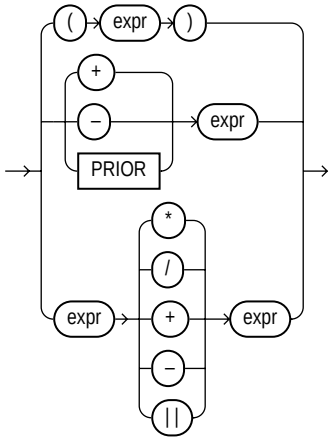
Some valid simple expressions are:

```
emp.ename
'this is a text string'
10
N'this is an NCHAR string'
```

Compound Expressions

A compound expression specifies a combination of other expressions.

compound_expression::=



Note that some combinations of functions are inappropriate and are rejected. For example, the LENGTH function is inappropriate within an aggregate function.

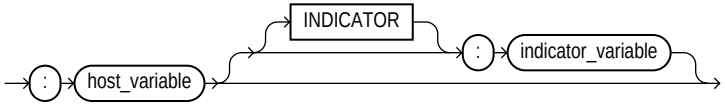
Some valid compound expressions are:

```
('CLARK' || 'SMITH')
LENGTH('MOOSE') * 57
SQRT(144) + 72
my_fun(TO_CHAR(sysdate,'DD-MMM-YY')
```

Variable Expressions

A variable expression specifies a host variable with an optional indicator variable. Note that this form of expression can appear only in embedded SQL statements or SQL statements processed in an Oracle Call Interface (OCI) program.

variable_expression::=



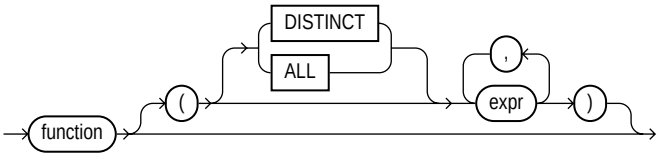
Some valid variable expressions are:

```
:employee_name INDICATOR :employee_name_indicator_var
:department_location
```

Built-In Function Expressions

A built-in function expression specifies a call to a single-row SQL function.

built_in_function_expression::=



Some valid built-in function expressions are:

```
LENGTH('BLAKE')
ROUND(1234.567*43)
SYSDATE
```

> **See Also:** "SQL Functions" on page 4-2 and "Aggregate Functions" on page 4-6 for information on built-in functions

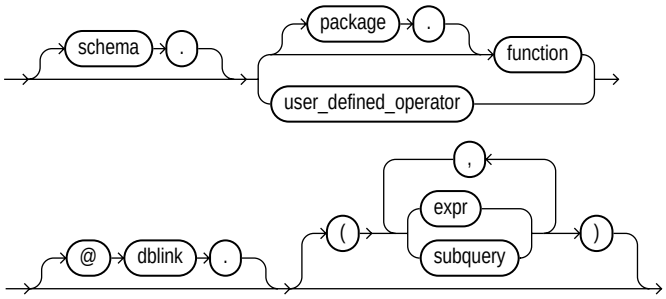
Function Expressions

A function expression specifies a call to

- A SQL built-in function (see Chapter 4, "Functions")
- A function in an Oracle-supplied package (see *Oracle8i Supplied PL/SQL Packages Reference*).
- A function in a user-defined package or in a standalone user-defined function (see "User-Defined Functions" on page 4-128)
- A user-defined operator (see CREATE OPERATOR on page 9-115 and *Oracle8i Data Cartridge Developer's Guide*)

The optional expression / subquery list must match attributes of the function, package, or operator. Only scalar subqueries are supported.

function_expression::=



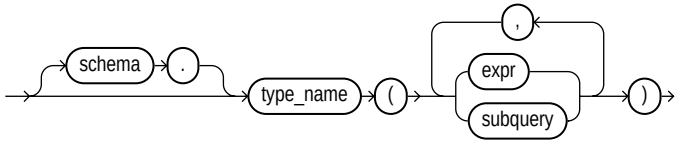
Some valid user-defined function expressions are:

```
circle_area(radius)
payroll.tax_rate(empno)
scott.payrol.tax_rate(dependents, empno)@ny
DBMS_LOB.getlength(column_name)
```

Type Constructor Expressions

A type constructor expression specifies a call to a type constructor. The argument to the type constructor is any expression or subquery. Only scalar subqueries are supported.

type_constructor_expression::=



If *type_name* is an **object type**, then the expression/subquery list must be an ordered list, where the first argument is a value whose type matches the first attribute of the object type, the second argument is a value whose type matches the second attribute of the object type, and so on. The total number of arguments to the constructor must match the total number of attributes of the object type.

If *type_name* is a **varray or nested table type**, then the expression/subquery list can contain zero or more arguments. Zero arguments implies construction of an empty collection. Otherwise, each argument corresponds to an element value whose type is the element type of the collection type.

If *type_name* is an **object type, a varray, or a nested table type**, the maximum number of arguments it can contain is 1000 minus some overhead.

Expression Example This example shows the use of an expression in the call to a type constructor.

```
CREATE TYPE address_t AS OBJECT
  (no NUMBER, street CHAR(31), city CHAR(21), state CHAR(3), zip NUMBER);
CREATE TYPE address_book_t AS TABLE OF address_t;
DECLARE
  /* Object Type variable initialized via Object Type Constructor */
  myaddr address_t = address_t(500, 'Oracle Parkway', 'Redwood Shores', 'CA',  94065);
  /* nested table variable initialized to an empty table via a constructor*/
  alladdr address_book_t = address_book_t();
BEGIN
  /* below is an example of a nested table constructor with two elements
     specified, where each element is specified as an object type constructor. */
  insert into employee values (666999, address_book_t(address_t(500,
     'Oracle  Parkway', 'Redwood Shores', 'CA', 94065), address_t(400,
     'Mission Street', 'Fremont', 'CA', 94555)));
END;
```

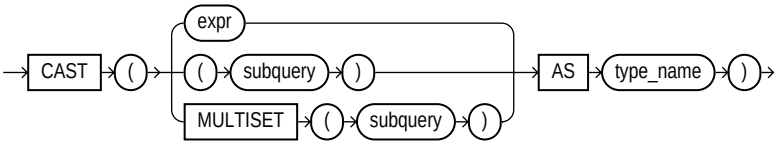
Subquery Example This example illustrates the use of a subquery in the call to the type constructor.

```
CREATE TYPE employee AS OBJECT (
   empno NUMBER,
   ename VARCHAR2(20));
CREATE TABLE emptbl of EMPLOYEE;
INSERT INTO emptbl VALUES(7377, 'JOHN');
CREATE TYPE project AS OBJECT (
   pname VARCHAR2(25),
   empref REF employee);
CREATE TABLE depttbl (dno number, proj project);
INSERT INTO depttbl values(10, project('SQL Extensions',
                                       (SELECT REF(p) FROM emptbl p
                                        WHERE ename='JOHN')));
```

CAST Expressions

A CAST expression converts one built-in datatype or collection-typed value into another built-in datatype or collection-typed value.

CAST_expression::=



CAST allows you to convert built-in datatypes or collection-typed values of one type into another built-in datatype or collection type. You can cast an unnamed operand (such as a date or the result set of a subquery) or a named collection (such as a varray or a nested table) into a type-compatible datatype or named collection. The *type_name* must be the name of a built-in datatype or collection type and the *operand* must be a built-in datatype or must evaluate to a collection value.

For the operand, *expr* can be either a built-in datatype or a collection type, and *subquery* must return a single value of collection type or built-in type. MULTISET informs Oracle to take the result set of the subquery and return a collection value. Table 5–1 shows which built-in datatypes can be cast into which other built-in datatypes. (CAST does not support LONG, LONG RAW, or any of the LOB datatypes.)

Table 5–1 Casting Built-In Datatypes

From/ To	CHAR, VARCHAR2	NUMBER	DATE	RAW	ROWID, UROWID	NCHAR, NVARCHAR2
CHAR, VARCHAR2	X	X	X	X	X	
NUMBER	X	X				
DATE	X		X			
RAW	X			X		
ROWID, UROWID	X				X[a]	
NCHAR, NVARCHAR2		X	X	X	X	X

[a] You cannot cast a UROWID to a ROWID if the UROWID contains the value of a ROWID of an index-organized table.

To cast a named collection type into another named collection type, the elements of both collections must be of the same type.

If the result set of *subquery* can evaluate to multiple rows, you must specify the MULTISET keyword. The rows resulting from the subquery form the elements of the collection value into which they are cast. Without the MULTISET keyword, the subquery is treated as a scalar subquery, which is not supported in the CAST expression. In other words, scalar subqueries as arguments of the CAST operator are not valid in Oracle8*i*.

Built-In Datatype Examples

```
SELECT CAST ('1997-10-22' AS DATE) FROM DUAL;
SELECT * FROM t1 WHERE CAST (ROWID AS VARCHAR2) = '01234';
```

Collection Examples The CAST examples that follow use the following user-defined types and tables:

```
CREATE TYPE address_t AS OBJECT
      (no NUMBER, street CHAR(31), city CHAR(21), state CHAR(2));
CREATE TYPE address_book_t AS TABLE OF address_t;
CREATE TYPE address_array_t AS VARRAY(3) OF address_t;
CREATE TABLE emp_address (empno NUMBER, no NUMBER, street CHAR(31),
                          city  CHAR(21), state CHAR(2));
CREATE TABLE employees (empno NUMBER, name CHAR(31));
CREATE TABLE dept (dno NUMBER, addresses address_array_t);
```

This example casts a subquery:

```
SELECT e.empno, e.name, CAST(MULTISET(SELECT ea.no, ea.street,
                                             ea.city, ea.state
                                      FROM emp_address ea
                                      WHERE ea.empno = e.empno)
                        AS address_book_t)
  FROM employees e;
```

CAST converts a varray type column into a nested table:

```
SELECT CAST(d.addresses AS address_book_t)
   FROM dept d
   WHERE d.dno = 111;
```

The following example casts a MULTISET expression with an ORDER BY clause:

```
CREATE TABLE projects (empid NUMBER, projname VARCHAR2(10));
CREATE TABLE employees (empid NUMBER, ename VARCHAR2(10));
CREATE TYPE projname_table_type AS TABLE OF VARCHAR2(10);
```

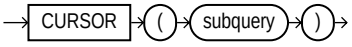
An example of a MULTISET expression with the above schema is:

```
SELECT e.ename, CAST(MULTISET(SELECT p.projname
                              FROM projects p
                              WHERE p.empid=e.empid
                              ORDER BY p.projname)
  AS projname_table_type)
   FROM employees e;
```

CURSOR Expressions

A CURSOR expression returns a nested cursor. This form of expression is similar to the PL/SQL REF cursor.

CURSOR_expression::=



A nested cursor is implicitly opened when the containing row is fetched from the parent cursor. The nested cursor is closed only when:

- The nested cursor is explicitly closed by the user
- The parent cursor is reexecuted
- The parent cursor is closed
- The parent cursor is cancelled
- An error arises during fetch on one of its parent cursors (it is closed as part of the clean-up)

Restrictions: The following restrictions apply to the CURSOR expression:

- Nested cursors can appear only in a SELECT statement that is not nested in any other query expression, except when it is a subquery of the CURSOR expression itself.
- Nested cursors can appear only in the outermost SELECT list of the query specification.
- Nested cursors cannot appear in views.
- You cannot perform BIND and EXECUTE operations on nested cursors.

Example

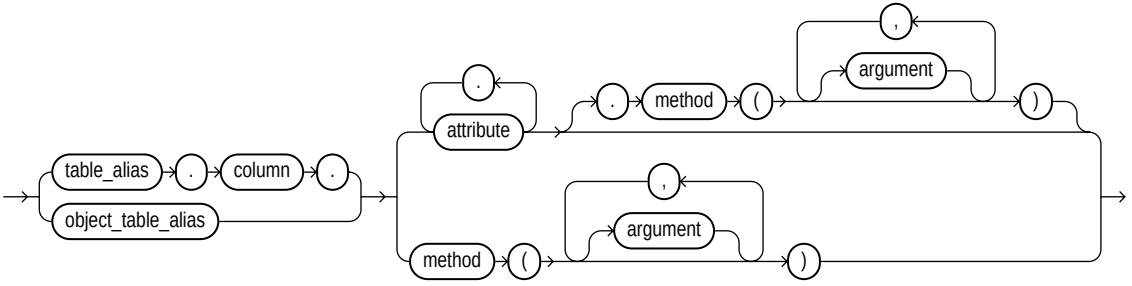
```
SELECT d.deptno, CURSOR(SELECT e.empno, CURSOR(SELECT p.projnum,
                                                      p.projname
```

```
                                                FROM   projects p
                                                WHERE  p.empno = e.empno)
                                  FROM TABLE(d.employees) e)
  FROM dept d
  WHERE d.dno = 605;
```

Object Access Expressions

An object access expression specifies attribute reference and method invocation.

object_access_expression::=



The column parameter can be an object or REF column.

When a type's member function is invoked in the context of a SQL statement, if the SELF argument is null, Oracle returns null and the function is not invoked.

Examples in this section use the following user-defined types and tables:

```
CREATE OR REPLACE TYPE employee_t AS OBJECT
   (empid NUMBER,
    name VARCHAR2(31),
    birthdate DATE,
    MEMBER FUNCTION age RETURN NUMBER,
    PRAGMA RESTRICT_REFERENCES (age, RNPS, WNPS, WNDS)
    );

CREATE OR REPLACE TYPE BODY employee_t AS
    MEMBER FUNCTION age RETURN NUMBER IS
       var NUMBER;
       BEGIN
          var := TRUNC(MONTHS_BETWEEN(SYSDATE, birthdate) /12);
          RETURN(var);
       END;
    END;
```

```
CREATE TABLE department (dno NUMBER, manager EMPLOYEE_T);
```

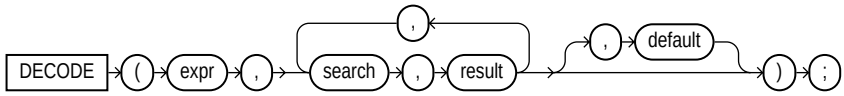
Examples The following examples update and select from the object columns and method defined above.

```
UPDATE department d
  SET d.manager.empid = 100;
SELECT d.manager.name, d.manager.age()
  FROM department d;
```

DECODE Expressions

A DECODE expression uses the special DECODE syntax:

DECODE_expression::=



To evaluate this expression, Oracle compares *expr* to each *search* value one by one. If *expr* is equal to a *search*, Oracle returns the corresponding *result*. If no match is found, Oracle returns *default*, or, if *default* is omitted, returns null. If *expr* and *search* contain character data, Oracle compares them using nonpadded comparison semantics.

The *search*, *result*, and *default* values can be derived from expressions. Oracle evaluates each *search* value only before comparing it to *expr*, rather than evaluating all *search* values before comparing any of them with *expr*. Consequently, Oracle never evaluates a *search* if a previous *search* is equal to *expr*.

Oracle automatically converts *expr* and each *search* value to the datatype of the first *search* value before comparing. Oracle automatically converts the return value to the same datatype as the first *result*. If the first *result* has the datatype CHAR or if the first *result* is null, then Oracle converts the return value to the datatype VARCHAR2.

In a DECODE expression, Oracle considers two nulls to be equivalent. If *expr* is null, Oracle returns the *result* of the first *search* that is also null.

The maximum number of components in the DECODE expression, including *expr*, *search*es, *results*, and *default* is 255.

See Also:

- "Datatype Comparison Rules" on page 2-26 for information on comparison semantics
- "Data Conversion" on page 2-30 for information on datatype conversion in general
- "Implicit vs. Explicit Data Conversion" on page 2-32 for information on the drawbacks of implicit conversion

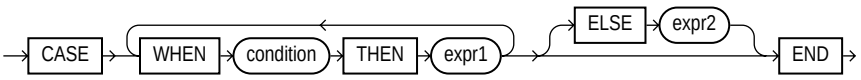
Example This expression decodes the value `deptno`. If `deptno` is 10, the expression evaluates to 'ACCOUNTING'; if `deptno` is 20, it evaluates to 'RESEARCH'; etc. If `deptno` is not 10, 20, 30, or 40, the expression returns 'NONE'.

```
DECODE (deptno,10, 'ACCOUNTING',
               20, 'RESEARCH',
               30, 'SALES',
               40, 'OPERATION',
                   'NONE')
```

CASE Expressions

CASE expressions let you use IF ... THEN ... ELSE logic in SQL statements without having to invoke procedures. The syntax is:

CASE_expression::=



Oracle searches for the first WHEN ... THEN pair for which *condition* is true.

- If Oracle finds such a pair, then the result of the CASE expression is *expr1*.
- If Oracle does not find such a pair,
  - If an ELSE clause is specified, the result of the CASE expression is *expr2*.
  - If no ELSE clause is specified, the result of the CASE expression in null.

At least one occurrence of *expr1* or *expr2* must be non-null.

Note: The maximum number of arguments in a CASE expression is 255, and each WHEN ... THEN pair counts as two arguments. To avoid exceeding the limit of 128 choices, you can nest CASE expressions. That is *expr1* can itself be a CASE expression.

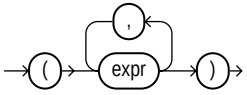
Example The following statement finds the average salary of all employees in the EMP table. If an employee's salary is less than $2000, the CASE expression uses $2000 instead.

```
SELECT AVG(CASE WHEN e.sal > 2000 THEN e.sal ELSE 2000 END) from emp e;
```

Expression List

An expression list is a series of expressions separated by a comma. The entire series is enclosed in parentheses.

expression_list::=



An expression list can contain up to 1000 expressions. Some valid expression lists are:

```
(10, 20, 40)
('SCOTT', 'BLAKE', 'TAYLOR')
(LENGTH('MOOSE') * 57, -SQRT(144) + 72, 69)
```

Conditions

A condition specifies a combination of one or more expressions and logical operators that evaluates to either TRUE, FALSE, or unknown. You must use this syntax whenever *condition* appears in SQL statements.

You can use a condition in the WHERE clause of these statements:

- DELETE
- SELECT
- UPDATE

You can use a condition in any of these clauses of the SELECT statement:

- WHERE
- START WITH
- CONNECT BY
- HAVING

A condition could be said to be of the "logical" datatype, although Oracle does not formally support such a datatype.

The following simple condition always evaluates to TRUE:

```
1 = 1
```

The following more complex condition adds the sal value to the comm value (substituting the value 0 for null) and determines whether the sum is greater than the number constant 2500:

```
NVL(sal, 0) + NVL(comm, 0) > 2500
```

Logical operators can combine multiple conditions into a single condition. For example, you can use the AND operator to combine two conditions:

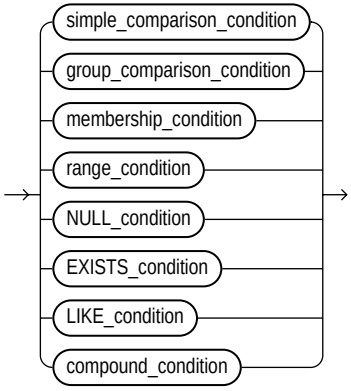
```
(1 = 1) AND (5 < 7)
```

Here are some valid conditions:

```
name = 'SMITH'
emp.deptno = dept.deptno
hiredate > '01-JAN-88'
job IN ('PRESIDENT', 'CLERK', 'ANALYST')
sal BETWEEN 500 AND 1000
comm IS NULL AND sal = 2000
```

Conditions can have several forms, as shown in the following syntax.

condition::=



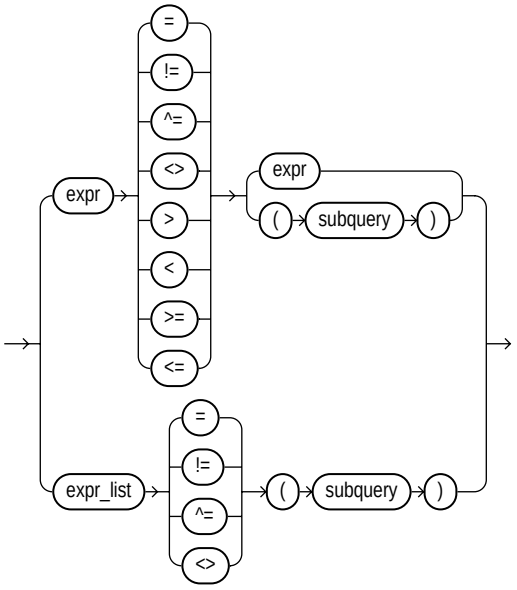
The sections that follow describe the various forms of conditions.

> **See Also:** The description of each statement in Chapter 7 through Chapter 11 for the restrictions on the conditions in that statement

Simple Comparison Conditions

A simple comparison condition specifies a comparison with expressions or subquery results.

simple_comparison_condition::=

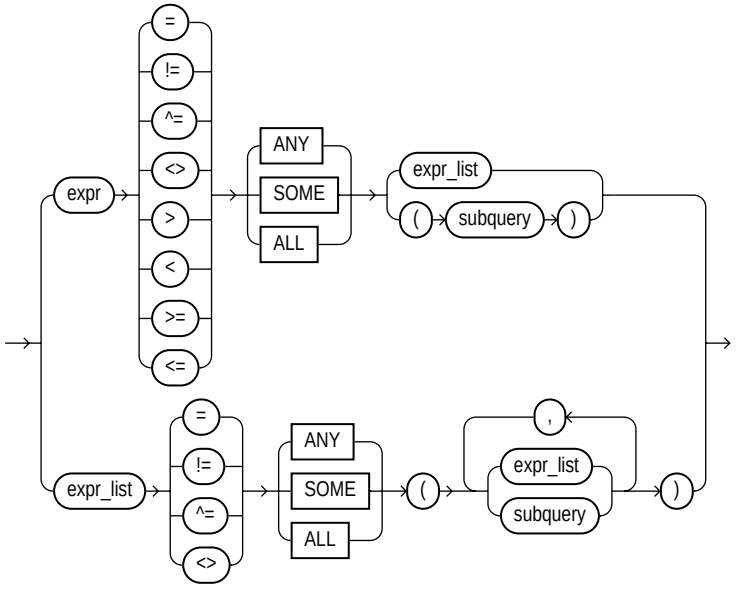


See Also: "Comparison Operators" on page 3-5 for information on comparison operators

Group Comparison Conditions

A group comparison condition specifies a comparison with any or all members in a list or subquery.

group_comparison_condition::=

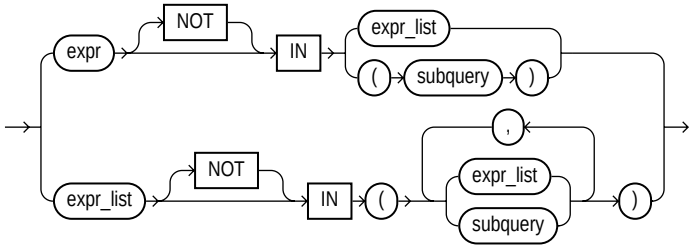


See Also: SELECT and subquery on page 11-88

Membership Conditions

A membership condition tests for membership in a list or subquery.

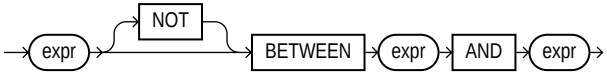
membership_condition::=



Range Conditions

A range condition tests for inclusion in a range.

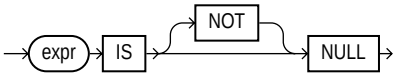
range_condition::=



NULL Conditions

A `NULL` condition tests for nulls.

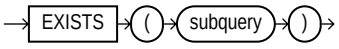
NULL_condition::=



EXISTS Conditions

An `EXISTS` condition tests for existence of rows in a subquery.

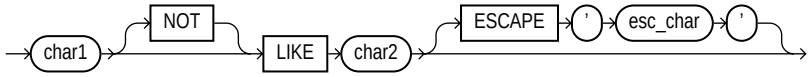
EXISTS_condition::=



LIKE Conditions

A `LIKE` condition specifies a test involving pattern matching.

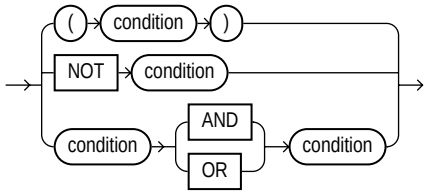
LIKE_condition::=



Compound Conditions

A compound condition specifies a combination of other conditions.

compound_condition::=



Queries and Subqueries

A **query** is an operation that retrieves data from one or more tables or views. In this reference, a top-level SELECT statement is called a **query**, and a query nested within another SQL statement is called a **subquery**.

This section describes some types of queries and subqueries and how to use them.

> **See Also:** SELECT and subquery on page 11-88 for the full syntax of all the clauses and the semantics of the keywords and parameters

Creating Simple Queries

The list of expressions that appears after the SELECT keyword and before the FROM clause is called the **select list**. Each expression *expr* becomes the name of one column in the set of returned rows, and each *table.** becomes a set of columns, one for each column in the table in the order they were defined when the table was created. The datatype and length of each expression is determined by the elements of the expression.

If two or more tables have some column names in common, you must qualify column names with names of tables. Otherwise, fully qualified column names are optional. However, it is always a good idea to qualify table and column references explicitly. Oracle often does less work with fully qualified table and column names.

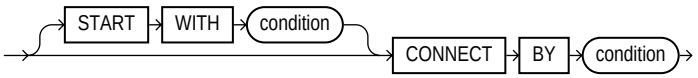
You can use a column alias, *c_alias*, to label the preceding expression in the select list so that the column is displayed with a new heading. The alias effectively renames the select list item for the duration of the query. The alias can be used in the ORDER BY clause, but not other clauses in the query.

You can use comments in a SELECT statement to pass instructions, or **hints**, to the Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement.

> **See Also:** "Hints" on page 2-67 and *Oracle8i Performance Guide and Reference* for more information on hints

Hierarchical Queries

If a table contains hierarchical data, you can select rows in a hierarchical order using the hierarchical query clause:

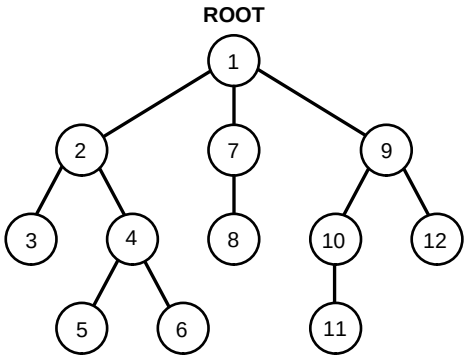


- START WITH specifies the root row(s) of the hierarchy.
- CONNECT BY specifies the relationship between parent rows and child rows of the hierarchy. Some part of *condition* must use the PRIOR operator to refer to the parent row. See the PRIOR operator on page 3-16.
- WHERE restricts the rows returned by the query without affecting other rows of the hierarchy.

Oracle uses the information from the hierarchical query clause clause to form the hierarchy using the following steps:

1. Oracle selects the root row(s) of the hierarchy—those rows that satisfy the START WITH condition.
2. Oracle selects the child rows of each root row. Each child row must satisfy the condition of the CONNECT BY condition with respect to one of the root rows.
3. Oracle selects successive generations of child rows. Oracle first selects the children of the rows returned in step 2, and then the children of those children, and so on. Oracle always selects children by evaluating the CONNECT BY condition with respect to a current parent row.
4. If the query contains a WHERE clause, Oracle eliminates all rows from the hierarchy that do not satisfy the condition of the WHERE clause. Oracle evaluates this condition for each row individually, rather than removing all the children of a row that does not satisfy the condition.
5. Oracle returns the rows in the order shown in Figure 5–1. In the diagram children appear below their parents.

Figure 5–1 Hierarchical Queries



To find the children of a parent row, Oracle evaluates the PRIOR expression of the CONNECT BY condition for the parent row and the other expression for each row in the table. Rows for which the condition is true are the children of the parent. The CONNECT BY condition can contain other conditions to further filter the rows selected by the query. The CONNECT BY condition cannot contain a subquery.

If the CONNECT BY condition results in a loop in the hierarchy, Oracle returns an error. A loop occurs if one row is both the parent (or grandparent or direct ancestor) and a child (or a grandchild or a direct descendent) of another row.

> **See Also:** "LEVEL" on page 2-62 for a discussion of how the LEVEL pseudocolumn operates in a hierarchical query

Sorting Query Results

You can use the ORDER BY clause to order the rows selected by a query. Sorting by position is useful in the following cases:

- To order by a lengthy select list expression, you can specify its position, rather than duplicate the entire expression, in the ORDER BY clause.
- For compound queries (containing set operators UNION, INTERSECT, MINUS, or UNION ALL), the ORDER BY clause must use positions, rather than explicit expressions. Also, the ORDER BY clause can appear only in the last component query. The ORDER BY clause orders all rows returned by the entire compound query.

The mechanism by which Oracle sorts values for the ORDER BY clause is specified either explicitly by the NLS_SORT initialization parameter or implicitly by the NLS_LANGUAGE initialization parameter. For information on these parameters, see *Oracle8i National Language Support Guide*. You can change the sort mechanism dynamically from one linguistic sort sequence to another using the ALTER SESSION statement. You can also specify a specific sort sequence for a single query by using the NLSSORT function with the NLS_SORT parameter in the ORDER BY clause.

Joins

A **join** is a query that combines rows from two or more tables, views, or materialized views ("snapshots"). Oracle performs a join whenever multiple tables appear in the query's FROM clause. The query's select list can select any columns from any of these tables. If any two of these tables have a column name in common, you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

Join Conditions

Most join queries contain WHERE clause conditions that compare two columns, each from a different table. Such a condition is called a **join condition**. To execute a join, Oracle combines pairs of rows, each containing one row from each table, for which the join condition evaluates to TRUE. The columns in the join conditions need not also appear in the select list.

To execute a join of three or more tables, Oracle first joins two of the tables based on the join conditions comparing their columns and then joins the result to another table based on join conditions containing columns of the joined tables and the new table. Oracle continues this process until all tables are joined into the result. The optimizer determines the order in which Oracle joins tables based on the join conditions, indexes on the tables, and, in the case of the cost-based optimization approach, statistics for the tables.

In addition to join conditions, the WHERE clause of a join query can also contain other conditions that refer to columns of only one table. These conditions can further restrict the rows returned by the join query.

Equijoins

An **equijoin** is a join with a join condition containing an equality operator. An equijoin combines rows that have equivalent values for the specified columns. Depending on the internal algorithm the optimizer chooses to execute the join, the total size of the columns in the equijoin condition in a single table may be limited to

the size of a data block minus some overhead. The size of a data block is specified by the initialization parameter DB_BLOCK_SIZE.

See Also: "Equijoin Examples" on page 11-108

Self Joins

A **self join** is a join of a table to itself. This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition. To perform a self join, Oracle combines and returns rows of the table that satisfy the join condition.

See Also: "Self Join Example" on page 11-110

Cartesian Products

If two tables in a join query have no join condition, Oracle returns their **Cartesian product**. Oracle combines each row of one table with each row of the other. A Cartesian product always generates many rows and is rarely useful. For example, the Cartesian product of two tables, each with 100 rows, has 10,000 rows. Always include a join condition unless you specifically need a Cartesian product. If a query joins three or more tables and you do not specify a join condition for a specific pair, the optimizer may choose a join order that avoids producing an intermediate Cartesian product.

Outer Joins

An outer join extends the result of a simple join. An **outer join** returns all rows that satisfy the join condition and those rows from one table for which no rows from the other satisfy the join condition. Such rows are not returned by a simple join. To write a query that performs an outer join of tables A and B and returns all rows from A, apply the outer join operator (+) to all columns of B in the join condition. For all rows in A that have no matching rows in B, Oracle returns null for any select list expressions containing columns of B.

Outer join queries are subject to the following rules and restrictions:

- The (+) operator can appear only in the WHERE clause or, in the context of left-correlation (that is, when specifying the TABLE clause) in the FROM clause, and can be applied only to a column of a table or view.
- If A and B are joined by multiple join conditions, you must use the (+) operator in all of these conditions. If you do not, Oracle will return only the rows

resulting from a simple join, but without a warning or error to advise you that you do not have the results of an outer join.

- The (+) operator can be applied only to a column, not to an arbitrary expression. However, an arbitrary expression can contain a column marked with the (+) operator.
- A condition containing the (+) operator cannot be combined with another condition using the OR logical operator.
- A condition cannot use the IN comparison operator to compare a column marked with the (+) operator with an expression.
- A condition cannot compare any column marked with the (+) operator with a subquery.

If the WHERE clause contains a condition that compares a column from table B with a constant, the (+) operator must be applied to the column so that Oracle returns the rows from table A for which it has generated NULLS for this column. Otherwise Oracle will return only the results of a simple join.

In a query that performs outer joins of more than two pairs of tables, a single table can be the null-generated table for only one other table. For this reason, you cannot apply the (+) operator to columns of B in the join condition for A and B and the join condition for B and C.

> **See Also:** SELECT and subquery on page 11-88 for the syntax for an outer join

Using Subqueries

A **subquery** answers multiple-part questions. For example, to determine who works in Taylor's department, you can first use a subquery to determine the department in which Taylor works. You can then answer the original question with the parent SELECT statement. A subquery in the FROM clause of a SELECT statement is also called an **inline view**. A subquery in the WHERE clause of a SELECT statement is also called a **nested subquery**.

A subquery can contain another subquery. Oracle imposes no limit on the number of subquery levels in the FROM clause of the top-level query. You can nest up to 255 levels of subqueries in the WHERE clause.

If tables in a subquery have the same name as tables in the containing statement, you must prefix any reference to the column of the table from the containing statement with the table name or alias. To make your statements easier for you to

read, always qualify the columns in a subquery with the name or alias of the table, view, or materialized view.

Oracle performs a **correlated subquery** when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT, UPDATE, or DELETE statement.

A correlated subquery answers a multiple-part question whose answer depends on the value in each row processed by the parent statement. For example, you can use a correlated subquery to determine which employees earn more than the average salaries for their departments. In this case, the correlated subquery specifically computes the average salary for each department.

See Also: "Correlated Subquery Examples" on page 11-118

A **scalar subquery** returns exactly one column value from one row. You can use a scalar subquery in place of an expression to specify a value in the VALUES clause of an INSERT statement or to provide an argument of a type constructor expression or user-defined function expression.

Use subqueries for the following purposes:

- To define the set of rows to be inserted into the target table of an INSERT or CREATE TABLE statement
- To define the set of rows to be included in a view or materialized view ("snapshot) in a CREATE VIEW or CREATE MATERIALIZED VIEW statement
- To define one or more values to be assigned to existing rows in an UPDATE statement
- To provide values for conditions in a WHERE clause, HAVING clause, or START WITH clause of SELECT, UPDATE, and DELETE statements
- To provide a value for a specified column in an INSERT ... VALUES list (scalar subqueries only)
- To provide values for arguments of a type constructor expression or a user-defined function expression (scalar subqueries only)
- To define a table to be operated on by a containing query.

  You do this by placing the subquery in the FROM clause of the containing query as you would a table name. You may use subqueries in place of tables in this way as well in INSERT, UDPATE, and DELETE statements.

Subqueries so used can employ correlation variables, but only those defined within the subquery itself, not outer references. Outer references ("left-correlated subqueries") are allowed only in the FROM clause of a SELECT statement.

See Also: *table_collection_expression* on page 11-96

Unnesting of Nested Subqueries

Subqueries are "nested" when they appear in the WHERE clause of the parent statement. When Oracle evaluates a statement with a nested subquery, it must evaluate the subquery portion multiple times and may overlook some efficient access paths or joins.

Subquery unnesting unnests and merges the body of the subquery into the body of the statement that contains it, allowing the optimizer to consider them together when evaluating access paths and joins. The optimizer can unnest most subqueries, with some exceptions. Those exceptions include subqueries that contain a CONNECT BY or START WITH clause, a ROWNUM pseudocolumn, one of the set operators, a nested aggregate function, or a correlated reference to a query block that is not the subquery's immediate outer query block.

Assuming no restrictions exist, the optimizer automatically unnests some (but not all) of the following nested subqueries:

- Uncorrelated IN subqueries
- IN and EXISTS correlated subqueries as long, as they do not contain aggregate functions or a GROUP BY clause

You can enable **extended subquery unnesting** by instructing the optimizer to unnest additional types of subqueries:

- You can unnest an uncorrelated NOT IN subquery by specifying the HASH_AJ or MERGE_AJ hint in the subquery.
- You can unnest other subqueries by specifying the UNNEST hint in the subquery

See Also: Chapter 2, "Basic Elements of Oracle SQL" for information on hints

Selecting from the DUAL Table

DUAL is a table automatically created by Oracle along with the data dictionary. DUAL is in the schema of the user SYS, but is accessible by the name DUAL to all

users. It has one column, DUMMY, defined to be VARCHAR2(1), and contains one row with a value 'X'. Selecting from the DUAL table is useful for computing a constant expression with the SELECT statement. Because DUAL has only one row, the constant is returned only once. Alternatively, you can select a constant, pseudocolumn, or expression from any table, but the value will be returned as many times as there are rows in the table.

> **See Also:** "SQL Functions" on page 4-2 for many examples of selecting a constant value from DUAL

Distributed Queries

Oracle's distributed database management system architecture allows you to access data in remote databases using Net8 and an Oracle server. You can identify a remote table, view, or materialized view by appending *@dblink* to the end of its name. The *dblink* must be a complete or partial name for a database link to the database containing the remote table, view, or materialized view.

> **See Also:** "Referring to Objects in Remote Databases" on page 2-90 for more information on referring to database links

Restrictions on Distributed Queries

Distributed queries are currently subject to the restriction that all tables locked by a FOR UPDATE clause and all tables with LONG columns selected by the query must be located on the same database. For example, the following statement will raise an error:

```
SELECT emp_ny.*
    FROM emp_ny@ny, dept
    WHERE emp_ny.deptno = dept.deptno
    AND dept.dname = 'ACCOUNTING'
    FOR UPDATE OF emp_ny.sal;
```

The following statement fails because it selects long_column, a LONG value, from the emp_review table on the ny database and locks the emp table on the local database:

```
SELECT emp.empno, review.long_column, emp.sal
    FROM emp, emp_review@ny review
    WHERE emp.empno = emp_review.empno
    FOR UPDATE OF emp.sal;
```

In addition, Oracle currently does not support distributed queries that select user-defined types or object REFs on remote tables.

6

About SQL Statements

This chapter describes the various types of Oracle SQL statements, and provides guidelines for finding the right SQL statement for your task.

This chapter contains these sections:

Summary of SQL Statements

The tables in the following sections provide a functional summary of SQL statements and are divided into these categories:

- Data Definition Language (DDL) Statements
- Data Manipulation Language (DML) Statements
- Transaction Control Statements
- Session Control Statements
- System Control Statements

Data Definition Language (DDL) Statements

Data definition language (DDL) statements enable you to perform these tasks:

- Create, alter, and drop schema objects
- Grant and revoke privileges and roles
- Analyze information on a table, index, or cluster
- Establish auditing options
- Add comments to the data dictionary

The `CREATE`, `ALTER`, and `DROP` commands require exclusive access to the specified object. For example, an `ALTER TABLE` statement fails if another user has an open transaction on the specified table.

The `GRANT`, `REVOKE`, `ANALYZE`, `AUDIT`, and `COMMENT` commands do not require exclusive access to the specified object. For example, you can analyze a table while other users are updating the table.

Oracle implicitly commits the current transaction before and after every DDL statement.

Many DDL statements may cause Oracle to recompile or reauthorize schema objects. For information on how Oracle recompiles and reauthorizes schema objects and the circumstances under which a DDL statement would cause this, see *Oracle8i Concepts*.

DDL statements are supported by PL/SQL with the use of the `DBMS_SQL` package.

See Also: *Oracle8i Supplied PL/SQL Packages Reference*

Table 6–1 lists the DDL statements.

Table 6–1 Data Definition Language Statements

ALTER CLUSTER	CREATE DIMENSION	DROP DATABASE LINK
ALTER DATABASE	CREATE DIRECTORY	DROP DIMENSION
ALTER DIMENSION	CREATE FUNCTION	DROP DIRECTORY
ALTER FUNCTION	CREATE INDEX	DROP FUNCTION
ALTER INDEX	CREATE INDEXTYPE	DROP INDEX
ALTER MATERIALIZED VIEW / SNAPSHOT	CREATE LIBRARY	DROP INDEXTYPE
ALTER MATERIALIZED VIEW / SHAPSHOT LOG	CREATE MATERIALIZED VIEW / SHAPSHOT	DROP LIBRARY
ALTER PACKAGE	CREATE MATERIALIZED VIEW / SNAPSHOT LOG	DROP MATERIALIZED VIEW / SNAPSHOT
ALTER PROCEDURE	CREATE OPERATOR	DROP MATERIALIZED VIEW / SNAPSHOT LOG
ALTER PROFILE	CREATE PACKAGE	DROP OPERATOR
ALTER RESOURCE COST	CREATE PACKAGE BODY	DROP PACKAGE
ALTER ROLE	CREATE PROCEDURE	DROP PROCEDURE
ALTER ROLLBACK SEGMENT	CREATE PROFILE	DROP PROFILE
ALTER SEQUENCE	CREATE ROLE	DROP ROLE
ALTER SNAPSHOT	CREATE ROLLBACK SEGMENT	DROP ROLLBACK SEGMENT
ALTER SHAPSHOT LOG	CREATE SCHEMA	DROP SEQUENCE
ALTER TABLE	CREATE SEQUENCE	DROP SNAPSHOT
ALTER TABLESPACE	CREATE SHAPSHOT	DROP SNAPSHOT LOG
ALTER TRIGGER	CREATE SNAPSHOT LOG	DROP SYNONYM
ALTER TYPE	CREATE SYNONYM	DROP TABLE
ALTER USER	CREATE TABLE	DROP TABLESPACE
ALTER VIEW	CREATE TABLESPACE	DROP TRIGGER
ANALYZE	CREATE TEMPORARY TABLESPACE	DROP TYPE
ASSOCIATE STATISTICS	CREATE TRIGGER	DROP USER
AUDIT	CREATE TYPE	DROP VIEW
COMMENT	CREATE USER	GRANT
CREATE CLUSTER	CREATE VIEW	NOAUDIT
CREATE CONTEXT	DISASSOCIATE STATISTICS	RENAME
CREATE CONTROLFILE	DROP CLUSTER	REVOKE
CREATE DATABASE	DROP CONTEXT	TRUNCATE
CREATE DATABASE LINK		

Data Manipulation Language (DML) Statements

Data manipulation language (DML) statements query and manipulate data in existing schema objects. These statements do not implicitly commit the current transaction.

Table 6–2 Data Manipulation Language Statements

Statement
CALL
DELETE
EXPLAIN PLAN
INSERT
LOCK TABLE
SELECT
UPDATE

The CALL and EXPLAIN PLAN statements are supported in PL/SQL only when executed dynamically. All other DML statements are fully supported in PL/SQL.

Transaction Control Statements

Transaction control statements manage changes made by DML statements.

Table 6–3 Transaction Control Statements

Statement
COMMIT
ROLLBACK
SAVEPOINT
SET TRANSACTION

All transaction control statements except certain forms of the COMMIT and ROLLBACK commands are supported in PL/SQL. For information on the restrictions, see COMMIT on page 8-133and ROLLBACK on page 11-83.

Session Control Statements

Session control statements dynamically manage the properties of a user session. These statements do not implicitly commit the current transaction.

PL/SQL does not support session control statements.

Table 6–4 Session Control Statements

Statement
ALTER SESSION
SET ROLE

System Control Statement

The single system control statement dynamically manages the properties of an Oracle instance. This statement does not implicitly commit the current transaction.

ALTER SYSTEM is not supported in PL/SQL.

Table 6–5 System Control Statement

Statement
ALTER SYSTEM

Embedded SQL Statements

Embedded SQL statements place DDL, DML, and transaction control statements within a procedural language program. Embedded SQL is supported by the Oracle precompilers and is documented in the following books:

- *Pro*COBOL Precompiler Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide*
- *SQL*Module for Ada Programmer's Guide*

Finding the SQL Statement for a Database Task

The particular SQL statement you use to accomplish a given database task is sometimes obvious and sometimes difficult to predict. For example, you create a table with the CREATE TABLE statement. However, you don't enable a constraint with the ENABLE CONSTRAINT statement, because such a statement doesn't exist. Rather, you modify the column options using the ALTER TABLE statement.

This section lists, by database object and task, the appropriate SQL statement to use to accomplish various database tasks. You can then refer to Chapter 7 through Chapter 11, for the syntax and semantics of each SQL statement.

> **Note:** Your ability to use the SQL statements listed in this section depends on the version and edition of Oracle you are using, as well as the options you have installed. Be sure to read the detailed descriptions in Chapter 7 through Chapter 11, before using these statements.

Database Object / Task	Operation	SQL Statement
application	allowing to connect as a user	`ALTER USER` *`proxy_clause`*
application server	allowing to connect as a user	`ALTER USER` *`proxy_clause`*
auditing	of database events	`CREATE TRIGGER`
call	limit CPU time for	`CPU_PER_CALL` parameter
	limit data blocks read	`LOGICAL_READS_PER_CALL` parameter
checkpoint	perform explicitly	`ALTER SYSTEM CHECKPOINT`
clone database	mount	`ALTER DATABASE MOUNT`
cluster	cluster key, change columns of	prohibited
	extent, allocate for	`ALTER CLUSTER` *`allocate_extent_clause`*
	migrated or chained rows, identify	`ANALYZE`
	parallelism of, change	`ALTER CLUSTER` *`parallel_clause`*
	rename	prohibited
	storage characteristics of, change	`ALTER CLUSTER` *`physical_attributes_clause`*
	tablespace of, change	prohibited
	unused space in, release	`ALTER CLUSTER` *`deallocate_unused_clause`*
column	add to a table or modify	`ALTER TABLE` *`add_column_options`*, *`modify_column_options`*
	define	`CREATE TABLE`

Database Object / Task	Operation	SQL Statement
	drop from a table	`ALTER TABLE drop_column_clause`
	generate derived values automatically	`CREATE TRIGGER`
	organization of, define	`CREATE TABLE`
commit operation	prevent procedure or function from issuing	`ALTER SESSION`
compilation	avoid run-time of	`ALTER FUNCTION ... COMPILE`
constraint	add to a table or modify	`ALTER TABLE add_column_options, modify_column_options`
	business, enforce	`CREATE TRIGGER`
	enable, disable, or drop	`ALTER TABLE enable_disable_clause, drop_constraint_clause`
	specify	`CREATE TABLE`
control file	back up	`ALTER DATABASE controlfile_clauses`
	standby, create	`ALTER DATABASE CREATE STANDBY CONTROLFILE`
currency symbol	reset for session	`ALTER SESSION SET NLS_CURRENCY`
data	frequently used, caching	`ALTER TABLE cache_clause`
	specify as temporary or permanent	`CREATE TABLE`
data dictionary	convert from Oracle7 to Oracle8i	`ALTER DATABASE CONVERT`
data independence	provide	`CREATE SYNONYM`
database	character set of, change	`ALTER DATABASE CHARACTER SET`
	create script for	`ALTER DATABASE controlfile_clauses`
	database character set for, specify	`CREATE DATABASE`
	datafiles for, specify	`CREATE DATABASE`
	datafiles of, modify	`ALTER DATABASE`
	datafiles, establish number of	`CREATE DATABASE`
	downgrade to an earlier release	`ALTER DATABASE RESET COMPATIBILITY`
	global name of, change	`ALTER DATABASE RENAME GLOBAL_NAME`

Database Object / Task	Operation	SQL Statement
	global name resolution, enable for the session	`ALTER SESSION SET GLOBAL_NAMES`
	instances, establish number of	`CREATE DATABASE`
	media recovery, design	`ALTER DATABASE` *`general_recovery_clause`*
	media recovery, perform ongoing	`ALTER DATABASE` *`managed_recovery_clause`*
	mount	`ALTER DATABASE MOUNT`
	move a subset to a different Oracle database	`ALTER TABLE` *`exchange_partition_clause`*
	national character set for, specify	`CREATE DATABASE`
	national character set of, change	`ALTER DATABASE CHARACTER SET`
	open	`ALTER DATABASE OPEN`
	parallelize recovery of	`ALTER DATABASE` *`parallel_clause`*
	place in read-only mode	`ALTER DATABASE OPEN`
	place in read-write mode	`ALTER DATABASE OPEN`
	place in sustained standby recovery mode	`ALTER DATABASE` *`general_recovery_clause`*
	prepare to re-create	`ALTER DATABASE` *`controlfile_clauses`*
	recover	`ALTER DATABASE` *`recover_clauses`*
	redo log file groups, establish number of	`CREATE DATABASE`
	redo log files for, specify	`CREATE DATABASE`
	redo log files of, create or modify	`ALTER DATABASE`
	redo log files, establish number of	`CREATE DATABASE`
	redo log, choose mode for	`CREATE DATABASE`
	upgrade to Oracle8i	`ALTER DATABASE`
database character set	specify for a database	`CREATE DATABASE`
database events	transparent logging of	`CREATE TRIGGER`
database link	close	`ALTER SESSION`

Database Object / Task	Operation	SQL Statement
database security	enforce authorizations	`CREATE TRIGGER`
datafile	automatic extension of, allow	`ALTER DATABASE DATAFILE` *`autoextend_clause`*
	create	`ALTER DATABASE CREATE DATAFILE`
	put online	`ALTER DATABASE DATAFILE ONLINE`
	reconstruct damaged	`ALTER DATABASE` *`general_recovery_clause`*
	reconstruct lost or damaged	`ALTER DATABASE CREATE DATAFILE`
	recover specified	`ALTER DATABASE` *`general_recovery_clause`*
	replace an old, for recovery	`ALTER DATABASE CREATE DATAFILE`
	resize	`ALTER DATABASE DATAFILE RESIZE`
	take offline	`ALTER DATABASE DATAFILE ONLINE/ OFFLINE`
	begin or end backup of	`ALTER TABLESPACE ... BACKUP`
	number of, establish for a database	`CREATE DATABASE`
	online, update instance information on	`ALTER SYSTEM` *`check_datafiles_clause`*
	specify for a database	`CREATE DATABASE`
dates	format of	See Table 2–9, "Date Format Elements" on page 2-48.
decimal character	reset for session	`ALTER SESSION SET NLS_NUMERIC_CHARACTERS`
dimension	add a level, hierarchy, or attribute to	`ALTER DIMENSION ... ADD`
	change the relationships of	`ALTER DIMENSION`
	drop a level, hierarchy, or attribute from	`ALTER DIMENSION ... DROP`
	explicitly compile	`ALTER DIMENSION ... COMPILE`
dispatcher processes	multi-threaded server, manage	`MTS_` parameters of `ALTER SYSTEM`

Database Object / Task	Operation	SQL Statement
domain index	alter	`ALTER INDEX ... PARAMETERS`
	rebuild	`ALTER INDEX` *`rebuild_clause`*
dump file	limit the size of	`ALTER SESSION SET MAX_DUMP_FILE_SIZE`
error messages	language in which displayed, change	`ALTER SESSION SET NLS_LANGUAGE`
function	allow to or prevent from committing a transaction	`ALTER SESSION`
	declaration of, change	`CREATE OR REPLACE FUNCTION`
	definition of, change	`CREATE OR REPLACE FUNCTION`
	recompile explicitly	`ALTER FUNCTION`
function-based index	disable	`ALTER INDEX ... [`*`rebuild_clause`*`] DISABLE`
	disabled, re-enable	`ALTER INDEX ... [`*`rebuild_clause`*`] ENABLE`
global names	enforce resolution of	`GLOBAL_NAMES` parameter of `ALTER SYSTEM`
hash join operations	data blocks for, allocate	`ALTER SESSION SET HASH_MULTIBLOCK_IO_COUNT`
	in queries, enable or disable	`ALTER SESSION SET HASH_JOIN_ENABLED ...`
	memory for, allocate	`ALTER SESSION SET HASH_AREA_SIZE`
index	allow DML operations during rebuilding of	`ALTER INDEX` *`rebuild_clause`*
	based on a function; see "function-based index"	`CREATE INDEX ...` *`column_expression`*
	based on an indextype; see "domain index"	`CREATE INDEX` *`domain_index_clause`*
	collect statistics during rebuilding of	`ALTER INDEX` *`rebuild_clause`*
	default attribute values of, change	`ALTER INDEX` *`partitioning_clauses`*
	degree of parallelism for, change	`ALTER INDEX` *`parallel_clause`*
	direct-load INSERT operations, write to a log	`ALTER INDEX` *`physical_attributes_clause`*

Database Object / Task	Operation	SQL Statement
	extent for, allocate new	ALTER INDEX *allocate_extent_clause*
	key compression, enable	ALTER INDEX *rebuild_clause*
	key values, eliminate repetition of	ALTER INDEX *rebuild_clause*
	merge block contents of	ALTER INDEX *rebuild_clause*
	physical attributes of a partition of, change	ALTER INDEX *physical_attributes_clause*
	physical attributes of a subpartition of, change the	ALTER INDEX *physical_attributes_clause*
	physical attributes of, change	ALTER INDEX *physical_attributes_clause*
	re-create	ALTER INDEX *rebuild_clause*
	rebuild operations, write to a log	ALTER INDEX *rebuild_clause*
	SQL*Loader operations against, write to a log	ALTER INDEX *physical_attributes_clause*
	store bytes in reverse order	ALTER INDEX *rebuild_clause*
	tablespace for, specify	ALTER INDEX *rebuild_clause*
	tell Oracle not to use	ALTER INDEX ... [*rebuild_clause*] UNUSABLE
	unused space, release	ALTER INDEX *deallocate_unused_clause*
	rename	ALTER INDEX *rebuild_clause*
index partition	create-time attributes, change	ALTER INDEX *rebuild_clause*
	log direct-load INSERT operations	ALTER INDEX *physical_attributes_clause*
	log SQL*Loader operations against	ALTER INDEX *physical_attributes_clause*
	move to a different tablespace	ALTER INDEX *rebuild_clause*
	physical attributes of, change	ALTER INDEX *physical_attributes_clause*
	physical, logging, or storage characteristics of, change	ALTER INDEX *partitioning_clauses*
	re-create	ALTER INDEX *rebuild_clause*

Database Object / Task	Operation	SQL Statement
	remove from the database	ALTER INDEX *partitioning_clauses*
	specify a tablespace for	ALTER INDEX *rebuild_clause*
	split into two partitions	ALTER INDEX *partitioning_clauses*
	tell Oracle not to use	ALTER INDEX ... UNUSABLE
index subpartition	change a create-time attributes, change	ALTER INDEX *rebuild_clause*
	log direct-load INSERT operations	ALTER INDEX *physical_attributes_clause*
	log SQL*Loader operations against	ALTER INDEX *physical_attributes_clause*
	move to a different tablespace	ALTER INDEX *rebuild_clause*
	physical attributes, change	ALTER INDEX *physical_attributes_clause*
	physical, logging, or storage characteristics, change	ALTER INDEX *partitioning_clauses*
	re-create	ALTER INDEX *rebuild_clause*
	tablespace for, specify	ALTER INDEX *rebuild_clause*
	tell Oracle not to use	ALTER INDEX ... UNUSABLE
index-organized table	characteristics, change	ALTER TABLE
indexes	on a cluster	CREATE INDEX
	on a nested table storage table	CREATE INDEX
	on a partitioned table	CREATE INDEX
	on an index-organized table	CREATE INDEX
	on columns of a table	CREATE INDEX
	on scalar typed object attributes	CREATE INDEX
instance	dynamically modify	ALTER SYSTEM
	make an index extent available to	ALTER INDEX *allocate_extent_clause*
	switch to a different	ALTER SESSION SET INSTANCE
instance recovery	continue after interruption	ALTER DATABASE *general_recovery_clause*

Database Object / Task	Operation	SQL Statement
instances	number of, establish for a database	`CREATE DATABASE`
Java class	force resolution of	`ALTER JAVA`
Java resource	force compilation of	`ALTER JAVA`
Java source	force compilation of	`ALTER JAVA`
licensing	changing limits or thresholds	`LICENSE_` parameters of `ALTER SYSTEM`
LOB columns	add to a table or modify	`ALTER TABLE` *`add_column_options, modify_column_options, LOB_storage_clause`*
location transparency	provide	`CREATE SYNONYM`
materialized view	automatic refresh, change the mode or timing of	`ALTER MATERIALIZED VIEW` *`refresh_clause`*
	change from rowid-based to primary-key-based	`ALTER MATERIALIZED VIEW` `ALTER MATERIALIZED VIEW LOG`
	degree of parallelism, specify or change	`ALTER MATERIALIZED VIEW` *`parallel_clause`*
	divide into partitions	`ALTER MATERIALIZED VIEW` *`partitioning_clauses`*
	LOB storage characteristics, change	`ALTER MATERIALIZED VIEW` *`modify_LOB_storage_clause`*
	LOB storage characteristics, specify	`ALTER MATERIALIZED VIEW` *`LOB_storage_clause`*
	log changes to	`ALTER MATERIALIZED VIEW ... LOGGING`
	make eligible for query rewrite	`ALTER MATERIALIZED VIEW ... QUERY REWRITE` `ALTER SESSION SET QUERY_REWRITE_ENABLED`
	make frequently accessed data accessible	`ALTER MATERIALIZED VIEW ... CACHE`
	revalidate	`ALTER MATERIALIZED VIEW ... COMPILE`
	storage characteristics, change	`ALTER MATERIALIZED VIEW` *`physical_attributes_clause`*

Database Object / Task	Operation	SQL Statement
materialized view log	automatic refresh, change the mode and timing of	`ALTER MATERIALIZED VIEW LOG`
	change from rowid-based to primary-key-based	`ALTER MATERIALIZED VIEW LOG`
	divide into partitions	`ALTER MATERIALIZED VIEW LOG` *`partitioning_clauses`*
	physical and storage characteristics, change	`ALTER MATERIALIZED VIEW LOG ...` *`physical_attributes_clause`*
	save both old and new values	`ALTER MATERIALIZED VIEW LOG ...NEW VALUES`
	store primary key of changed rows	`ALTER MATERIALIZED VIEW LOG ... ADD`
	store rowid of changed rows	`ALTER MATERIALIZED VIEW LOG ... ADD`
media recovery	avoid on startup	`ALTER DATABASE DATAFILE END BACKUP`
	from specified redo log file	`ALTER DATABASE` *`general_recovery_clause`*
	prepare for	`ALTER DATABASE ARCHIVELOG`
national character set	specify for a database	`CREATE DATABASE`
national language support	change settings for the session	`ALTER SESSION SET NLS_` parameters
nested table	update in a view	create an `INSTEAD OF` trigger
nested table columns	indexing	`CREATE INDEX`
numbers	format	See Table 2–7, "Number Format Elements" on page 2-44.
object references. See REFs		
online redo log	reinitialize	`ALTER DATABASE CLEAR LOGFILE`
outline	assign to a different category	`ALTER OUTLINE ... CHANGE CATEGORY TO`
	recompile	`ALTER OUTLINE ... REBUILD`
	rename	`ALTER OUTLINE ... RENAME`
	automatically create and store	`ALTER SESSION SET CREATE_STORED_OUTLINES`

Database Object / Task	Operation	SQL Statement
	use to generate execution plans	`ALTER SESSION SET USE_STORED_OUTLINES`
package	avoid run-time compilation	`ALTER PACKAGE`
	compile explicitly	`ALTER PACKAGE`
package body	avoid run-time compilation	`ALTER PACKAGE`
	recompile explicitly	`ALTER PACKAGE`
parallelism	specify for a table	`CREATE TABLE`
	specify for DML on a table	`CREATE TABLE`
parameter, initialization	change the setting for the current session	`ALTER SESSION` *`set_clause`*
parameter, session	set or change the setting of	`ALTER SESSION` *`set_clause`*
partition	add to a table or modify	`ALTER TABLE`
	default attributes, change	`ALTER TABLE` *`modify_default_attributes_clause`*
	logging characteristics, change	`ALTER TABLE` *`logging_clause`*
	merge with another partition	`ALTER TABLE` *`merge_partitions_clause`*
	point to data in a nonpartitioned table	`ALTER TABLE` *`exchange_partition_clause`*
	real attributes, change	`ALTER TABLE` *`modify_partition_clause`*
password	complexity of, guarantee	`PASSWORD_VERIFY_FUNCTION` parameter
	make unavailable	`PASSWORD_REUSE_TIME` parameter
	number of days account will be locked after failed login attempts, specify	`PASSWORD_LOCK_TIME` parameter
	number of days before reuse, limit	`PASSWORD_REUSE_TIME` parameter
	number of days in grace period, specify	`PASSWORD_GRACE_TIME` parameter
	number of days usable, limit	`PASSWORD_LIFE_TIME` parameter
	number of times reused, limit	`PASSWORD_REUSE_MAX` parameter
	special characters in, allow	`PASSWORD_VERIFY_FUNCTION` parameter

Database Object / Task	Operation	SQL Statement
performance	optimize for index access path	`ALTER SESSION SET OPTIMIZER_INDEX_COST_ADJ`
	optimize for nested loop joins	`ALTER SESSION SET OPTIMIZER_INDEX_CACHING`
	specify the optimizer approach for the session	`ALTER SESSION SET OPTIMIZER_MODE`
procedure	allow to or prevent from committing a transaction	`ALTER SESSION`
	avoid run-time compilation	`ALTER PROCEDURE`
	recompile explicitly	`ALTER PROCEDURE`
profile	resource limit, add to	`ALTER PROFILE`
	resource limit, change	`ALTER PROFILE`
	resource limit, drop from	`ALTER PROFILE`
recovery	distributed, enable or disable	`ALTER SYSTEM` *`distributed_recovery_clause`*
recovery data	discard	`ALTER DATABASE RESETLOGS`
redo log	remove changes from	`ALTER DATABASE OPEN RESETLOGS`
	reset sequence of	`ALTER DATABASE OPEN RESETLOGS`
	specify mode of	`CREATE DATABASE`
redo log file	add	`ALTER DATABASE ADD LOGFILE MEMBER`
	automatically generates names for	`ALTER DATABASE` *`general_recovery_clause`*
	clear	`ALTER DATABASE CLEAR LOGFILE`
	drop	`ALTER DATABASE DROP LOGFILE`
	enable or disable thread	`ALTER DATABASE ENABLE THREAD`
	rename	`ALTER DATABASE RENAME FILE`
	number of, establish for a database	`CREATE DATABASE`
	archive manually or automatically	`ALTER SYSETM` *`archive_log_clause`*
	number of, establish for a database	`CREATE DATABASE`

Database Object / Task	Operation	SQL Statement
	specify a path for	`ALTER SESSION SET LOG_ARCHIVE_DEST_`*`n`*
	switch manually	`ALTER SYSTEM` *`switch_logfile_clause`*
REFS	validate and update	`ANALYZE`
role	change authorization required	`ALTER ROLE`
rollback segment	bring online	`ALTER ROLLBACK SEGMENT`
	reduce in size	`ALTER ROLLBACK SEGMENT`
	storage characteristics, change	`ALTER ROLLBACK SEGMENT`
	take offline	`ALTER ROLLBACK SEGMENT`
rowid	examine	query the `ROWID` pseudocolumn
	extended, interpreting contents	`DBMS_ROWID` package; see *Oracle8i Supplied PL/SQL Packages Reference*
schema	change during the session	`ALTER SESSION SET CURRENT_SCHEMA`
schema object	reference without referencing its location	`CREATE SYNONYM`
	reference without referencing its owner	`CREATE SYNONYM`
	specify another name for	`CREATE SYNONYM`
	validate structure of	`ANALYZE`
sequence	cached sequence values, change number of	`ALTER SEQUENCE` *`cache_clause`*
	consecutive order of values, guarantee	`CREATE SEQUENCE ... ORDER` `ALTER SEQUENCE ... ORDER`
	create	`CREATE SEQUENCE`
	determine current value of	See "CURRVAL and NEXTVAL" on page 2-59.
	increment value, set	`CREATE SEQUENCE ... INCREMENT BY` `ALTER SEQUENCE ... INCREMENT BY`
	maximum or minimum value, eliminate	`ALTER SEQUENCE`
	minimum or maximum value, set	`CREATE SEQUENCE` `ALTER SEQUENCE`

Database Object / Task	Operation	SQL Statement
	preallocate values for faster access	CREATE SEQUENCE
		ALTER SEQUENCE
	restart after a predefined limit	CREATE SEQUENCE ... CYCLE
		ALTER SEQUENCE ... CYCLE
	starting value, set	CREATE SEQUENCE
server processes	multi-threaded server, manage	MTS_ parameters of ALTER SYSTEM
session	CPU time for, limit	CPU_PER_SESSION parameter
	data blocks read, limit	LOGICAL_READS_PER_SESSION parameter
	enable or disable parallel transactions in	ALTER SESSION
	inactive period duration, limit	IDLE_TIME parameter
	private SGA space for, limit	PRIVATE_SGA parameter
	resource costs allowed, change	ALTER RESOURCE COST
	restrict to privileged users	ALTER SYSTEM *restricted_session_clause*
	terminate	ALTER SYSTEM *kill_session_clause*
	total elapsed time, limit	CONNECT_TIME parameter
	total resources for, limit	COMPOSITE_LIMIT parameter
SGA	flush data from shared pool	ALTER SYSTEM *flush_shared_pool_clause*
shared pool	flush	ALTER SYSTEM *flush_shared_pool_clause*
snapshot. *See* "**materialized view**".		
sort operations	linguistic sequence, change	ALTER SESSION SET NLS_SORT
standby database	activate	ALTER DATABASE ACTIVATE STANDBY DATABASE
	recover	ALTER DATABASE *recover_clauses*
statistics	on a schema object, collect	ANALYZE
	on a schema object, delete	ANALYZE
	on scalar object attributes, collect	ANALYZE
subpartition	add to a table or modify	ALTER TABLE

Database Object / Task	Operation	SQL Statement
	default attributes, change	ALTER TABLE *modify_default_attributes_clause, modify_partition_clause*
	logging characteristics, change	ALTER TABLE *logging_clause*
	real attributes, change	ALTER TABLE *modify_subpartition_clause*
system resources	enable or disable	RESOURCE_LIMITS parameter of ALTER SYSTEM
table	allocate space for	ALTER TABLE *allocate_extent_clause*
	characteristics, change	ALTER TABLE *physical_attributes_clause, modify_storage_clauses*
	column, drop from table	ALTER TABLE *drop_column_clause*
	degree of parallelism, change	ALTER TABLE *parallel_clause*
	logging characteristics, change	ALTER TABLE *logging_clause*
	make read-only, read-write	ALTER TABLE
	migrated or chained rows, identify	ANALYZE
	organization, define	CREATE TABLE
	partition, point to the contents of another table	ALTER TABLE *exchange_partition_clause*
	partitioning, specify	CREATE TABLE
	rename	ALTER TABLE
	unused space of, release	ALTER TABLE *deallocate_unused_clause*
	heap or index organized	CREATE TABLE
	include in a cluster	CREATE TABLE
	replicate asynchronous, maintain	CREATE TRIGGER
	storage characteristics of, set	CREATE TABLE
tablespace	allow or disallow writing to	ALTER TABLESPACE READ WRITE/ONLY
	datafiles, add or rename	ALTER TABLESPACE datafile/ *tempfile_clauses*

Database Object / Task	Operation	SQL Statement
	logging characteristics, change	`ALTER TABLESPACE`
	minimum extent length, change	`ALTER TABLESPACE`
	reconstruct damaged	`ALTER DATABASE` *`general_recovery_clause`*
	reconstruct lost or damaged	`ALTER DATABASE CREATE DATAFILE`
	recover specified	`ALTER DATABASE` *`general_recovery_clause`*
	specifying for a table	`CREATE TABLE`
	storage characteristics, change	`ALTER TABLESPACE`
	take online or offline	`ALTER TABLESPACE`
	user quota on, change	`ALTER USER`
	assign to a user	`CREATE USER`
	space quota for a user, allocate	`CREATE USER`
tempfile	allow for automatic extension of	`ALTER DATABASE TEMPFILE`
	resize	`ALTER DATABASE TEMPFILE`
transaction	distributed, force commit of	`ALTER SESSION`
	distributed, force rollback of	`ALTER SESSION`
trigger	enable or disable	`ALTER TABLE`
user	authentication, change	`ALTER USER`
	database resources limits, change	`ALTER USER` *`profile_clause`*
	default roles, change	`ALTER USER`
	failed attempts to log in, limit	`FAILED_LOGIN_ATTEMPTS` parameter
	number of sessions, limit	`SESSIONS_PER_USER` parameter
	password, change	`ALTER USER`
	resource limits, set	`CREATE USER`

Database Object / Task	Operation	SQL Statement
	restrict access to Oracle	`ALTER SYSTEM` *`restricted_session_clause`*
	tablespace quota, allocate	`CREATE USER`
	tablespaces, assign	`CREATE USER`

7

SQL Statements: ALTER CLUSTER to ALTER SYSTEM

All SQL statements in this chapter, as well as in Chapters 8 through 11, are organized into the following sections:

Syntax — The syntax diagrams show the keywords and parameters that make up the statement.

> **Caution:** Not all keywords and parameters are valid in all circumstances. Be sure to refer to the "Keywords and Parameters" section of each statement and clause to learn about any restrictions on the syntax.

Purpose — The "Purpose" section describes the basic uses of the statement.

Prerequisites — The "Prerequisites" section lists privileges you must have and steps that you must take before using the statement. In addition to the prerequisites listed, most statements also require that the database be opened by your instance, unless otherwise noted.

Keywords and Parameters — The "Keywords and Parameters" section describes the purpose of each keyword and parameter. (The conventions for keywords and parameters used in this chapter are explained in the Preface of this reference.) Restrictions and usage notes also appear in this section.

Examples — The "Examples" section shows how to use various clauses and parameters of the statement.

This chapter contains the following SQL statements:

- ALTER CLUSTER
- ALTER DATABASE
- ALTER DIMENSION
- ALTER FUNCTION
- ALTER INDEX
- ALTER JAVA
- ALTER MATERIALIZED VIEW
- ALTER MATERIALIZED VIEW LOG
- ALTER OUTLINE
- ALTER PACKAGE
- ALTER PROCEDURE
- ALTER PROFILE
- ALTER RESOURCE COST
- ALTER ROLE
- ALTER ROLLBACK SEGMENT
- ALTER SEQUENCE
- ALTER SESSION
- ALTER SYSTEM

ALTER CLUSTER

Purpose

Use the ALTER CLUSTER statement to redefine storage and parallelism characteristics of a cluster.

> **Note:** You cannot use this statement to change the number or the name of columns in the cluster key, and you cannot change the tablespace in which the cluster is stored.

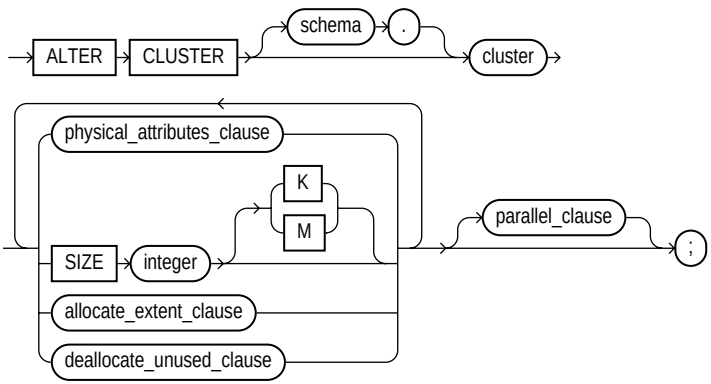
See Also:

- CREATE CLUSTER on page 9-3 for information on creating a cluster
- DROP CLUSTER on page 10-126and DROP TABLE on page 11-7 for information on removing tables from a cluster

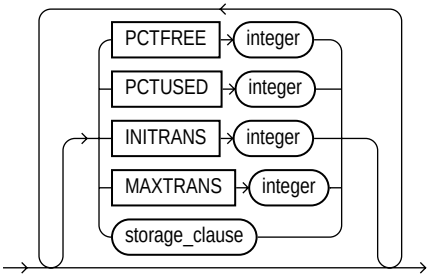
Prerequisites

The cluster must be in your own schema or you must have ALTER ANY CLUSTER system privilege.

Syntax

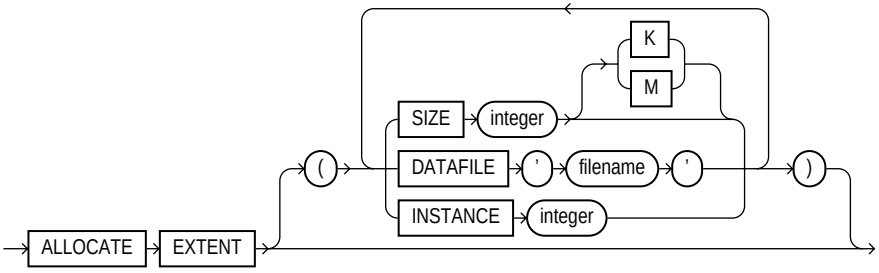


physical_attributes_clause:=

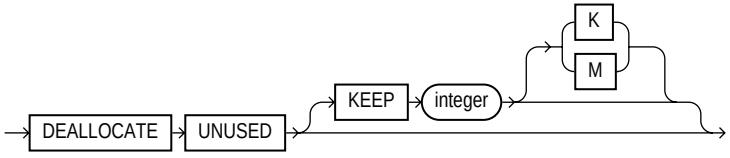


storage_clause: See storage_clause on page 11-129.

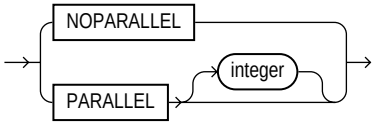
allocate_extent_clause::=



deallocate_unused_clause::=



parallel_clause::=



Keywords and Parameters

schema

Specify the schema containing the cluster. If you omit *schema*, Oracle assumes the cluster is in your own schema.

cluster

Specify the name of the cluster to be altered.

physical_attributes_clause

Use this clause to change the values of the PCTUSED, PCTFREE, INITRANS, and MAXTRANS parameters of the cluster.

> **See Also:** CREATE CLUSTER on page 9-3 for a description of these parameters

storage_clause

Use the STORAGE clause to change the storage characteristics for the cluster.

Restriction: You cannot change the values of the storage parameters INITIAL and MINEXTENTS for a cluster.

> **See Also:** *storage_clause* on page 11-129

SIZE *integer*

Use the SIZE clause to specify the number of cluster keys that will be stored in data blocks allocated to the cluster.

Restriction: You can change the SIZE parameter only for an indexed cluster, not for a hash cluster.

> **See Also:** CREATE CLUSTER on page 9-3 for a description of the SIZE parameter

allocate_extent_clause

Specify the ALLOCATE EXTENT clause to explicitly allocate a new extent for the cluster.

Restriction: You can allocate a new extent only for an indexed cluster, not for a hash cluster.

`SIZE` *integer* Use the `SIZE` parameter to specify the size of the extent in bytes. Use K or M to specify the extent size in kilobytes or megabytes.

When you explicitly allocate an extent with this clause, Oracle does not evaluate the cluster's storage parameters and determine a new size for the next extent to be allocated (as it does when you create a table). Therefore, specify `SIZE` if you do not want Oracle to use a default value.

`DATAFILE` *'filename'* Use the `DATAFILE` parameter to specify one of the datafiles in the cluster's tablespace to contain the new extent. If you omit this parameter, Oracle chooses the datafile.

`INSTANCE` *integer* Use the `INSTANCE` parameter to make the new extent available to the specified instance. An instance is identified by the value of its initialization parameter `INSTANCE_NUMBER`. If you omit `INSTANCE`, the extent is available to all instances.

Note: Use this parameter only if you are using Oracle with the Parallel Server option in parallel mode.

---

deallocate_unused_clause

Specify the `DEALLOCATE UNUSED` clause to explicitly deallocate unused space at the end of the cluster and makes the freed space available for other segments. Only unused space above the high water mark can be freed.

`KEEP` *integer* Use the `KEEP` parameter to specify the number of bytes above the high water mark that the cluster will have after deallocation. If the number of remaining extents is less than `MINEXTENTS`, then `MINEXTENTS` is set to the current number of extents. If the initial extent becomes smaller than `INITIAL`, then `INITIAL` is set to the value of the current initial extent. If you omit `KEEP`, all unused space is freed.

See Also: ALTER TABLE on page 8-2 for a more complete description of this clause

parallel_clause

Specify the *parallel_clause* to change the default degree of parallelism for queries and `DML` on the cluster.

Restriction: If the tables in *cluster* contain any columns of LOB or user-defined object type, this statement as well as subsequent INSERT, UPDATE, or DELETE operations on *cluster* are executed serially without notification.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL — Specify NOPARALLEL for serial execution. This is the default.

PARALLEL — Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

See Also: "Notes on the parallel_clause" for CREATE TABLE on page 10-41

Examples

Modifying a Cluster Example The following statement alters the CUSTOMER cluster in the schema scott:

```
ALTER CLUSTER scott.customer
    SIZE 512
    STORAGE (MAXEXTENTS 25);
```

Oracle allocates 512 bytes for each cluster key value. Assuming a data block size of 2 kilobytes, future data blocks within this cluster contain 4 cluster keys per data block, or 2 kilobytes divided by 512 bytes. The cluster can have a maximum of 25 extents.

Deallocating Unused Space Example The following statement deallocates unused space from the CUSTOMER cluster, keeping 30 kilobytes of unused space for future use:

```
ALTER CLUSTER scott.customer
    DEALLOCATE UNUSED KEEP 30 K;
```

ALTER DATABASE

Purpose

Use the ALTER DATABASE statement to modify, maintain, or recover an existing database.

See Also:

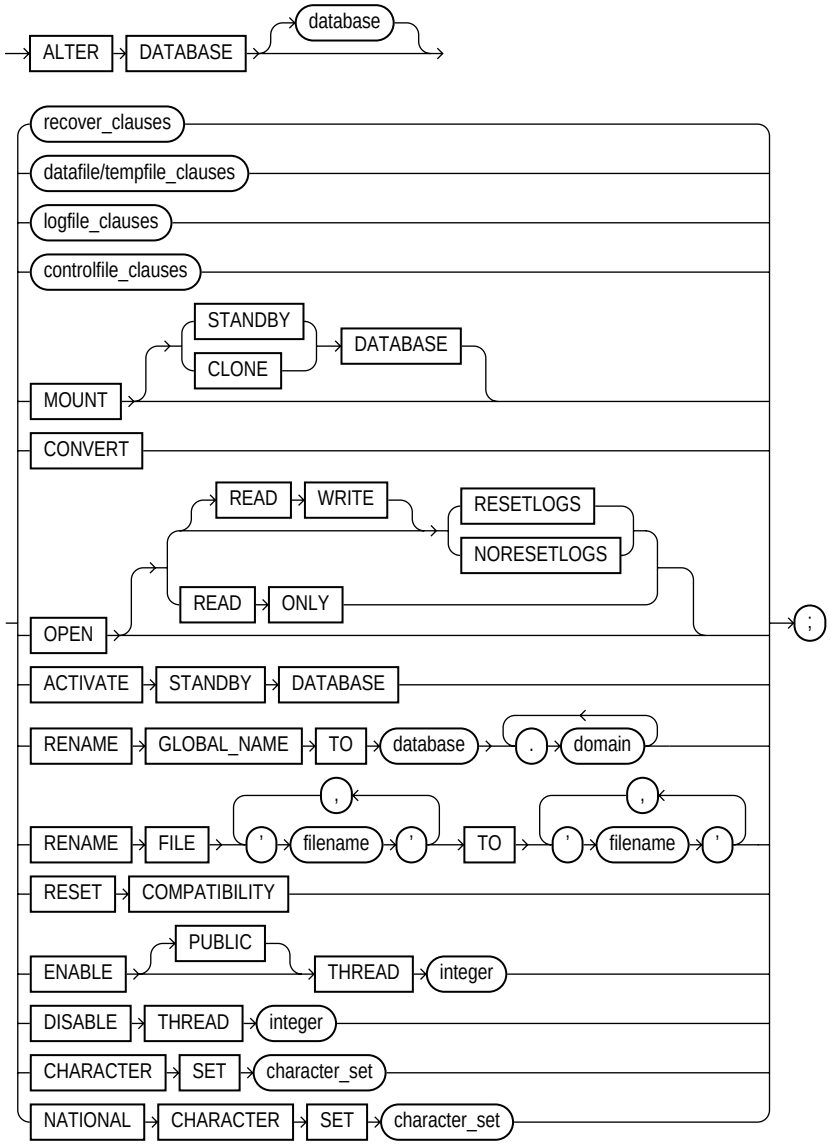
- *Oracle8i Administrator's Guide* for more information on using the ALTER DATABASE statement for database maintenance
- *Oracle8i Administrator's Guide*, *Oracle8i Recovery Manager User's Guide and Reference,* and *Oracle8i Backup and Recovery Guide* for examples of performing media recovery
- CREATE DATABASE on page 9-21 for information on creating a database

Prerequisites

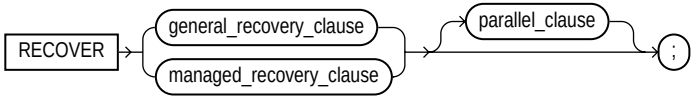
You must have ALTER DATABASE system privilege.

To specify the RECOVER clause, you must also have the OSDBA role enabled.

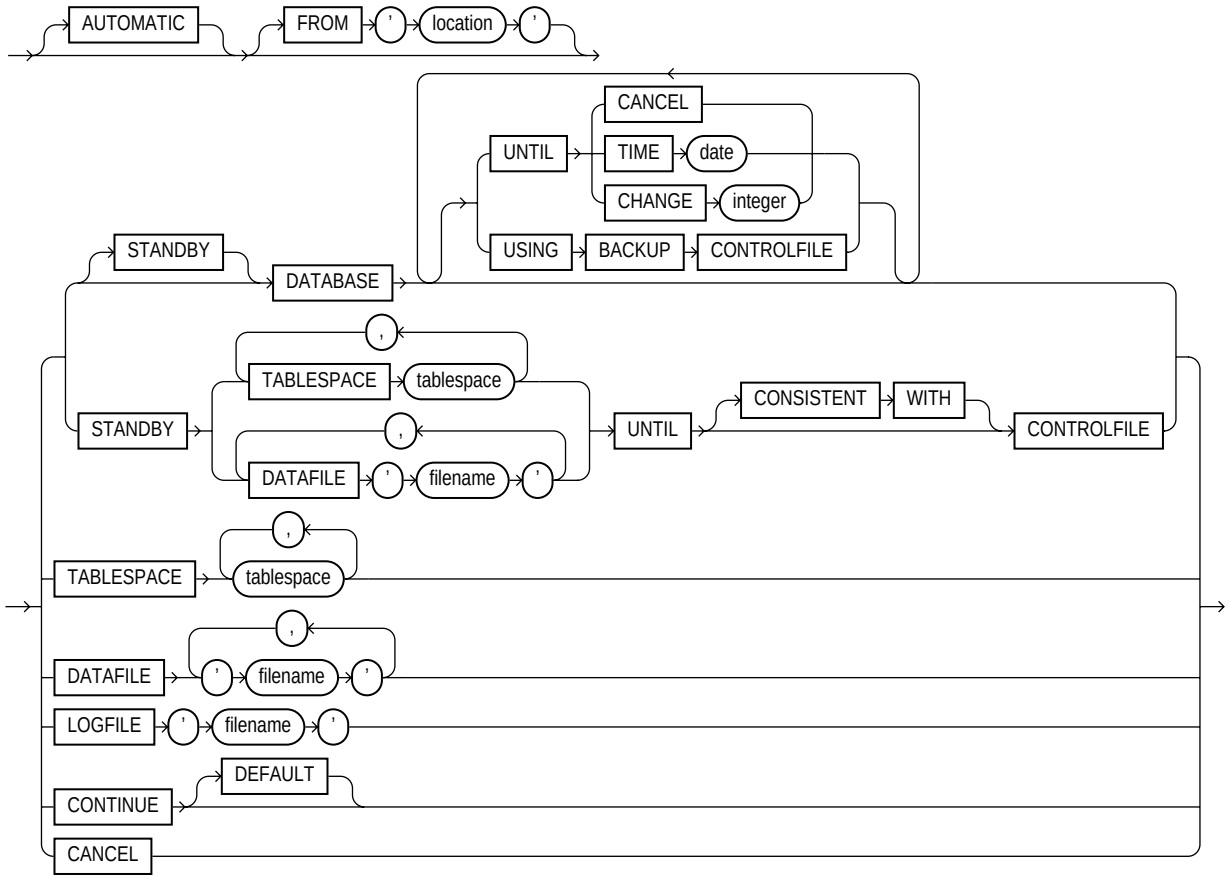
Syntax


ALTER
DATABASE
database
recover_clauses
datafile/tempfile_clauses
logfile_clauses
controlfile_clauses
STANDBY
CLONE
DATABASE
MOUNT
CONVERT
READ
WRITE
RESETLOGS
NORESETLOGS
READ
ONLY
OPEN
;
ACTIVATE
STANDBY
DATABASE
RENAME
GLOBAL_NAME
TO
database
.
domain
RENAME
FILE
'
filename
'
,
TO
'
filename
'
,
RESET
COMPATIBILITY
ENABLE
PUBLIC
THREAD
integer
DISABLE
THREAD
integer
CHARACTER
SET
character_set
NATIONAL
CHARACTER
SET
character_set

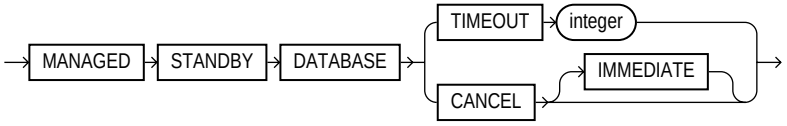
recover_clauses::=



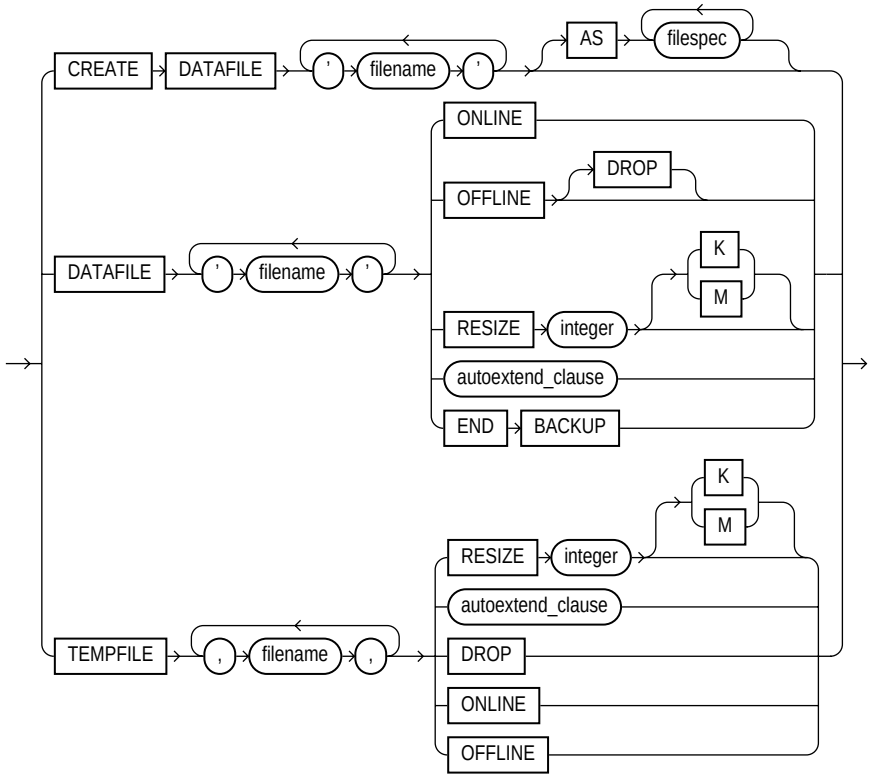
general_recovery_clause::=



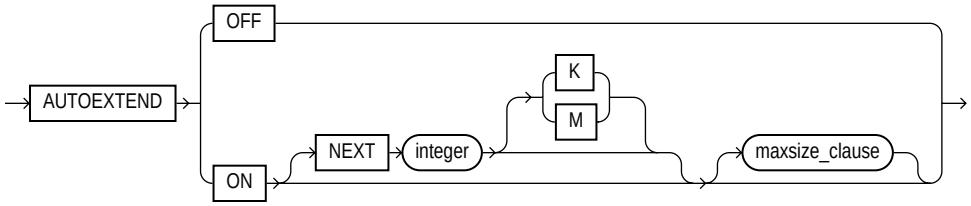
managed_recovery_clause::=



datafile_tempfile_clauses::=

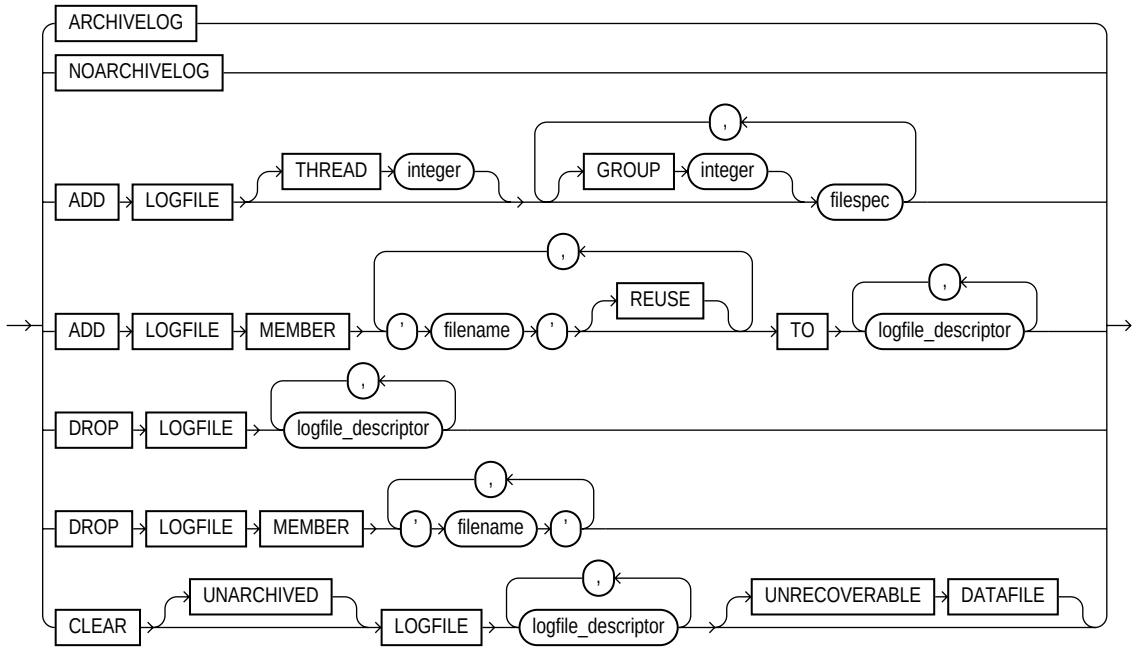


autoextend_clause::=

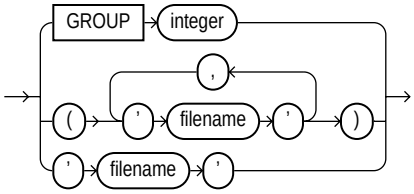


filespec: See filespec on page 11-27.

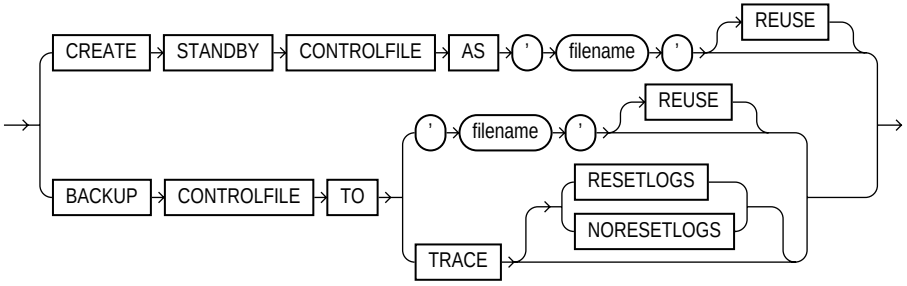
logfile_clauses::=



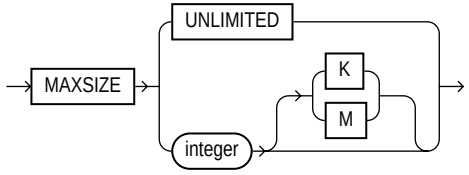
logfile_descriptor::=



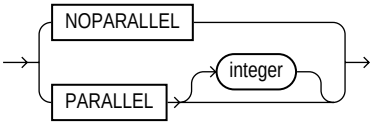
controlfile_clauses::=



maxsize_clause::=



parallel_clause::=



Keywords and Parameters

database

Specify the name of the database to be altered. The database name can contain only ASCII characters. If you omit database, Oracle alters the database identified by the value of the initialization parameter DB_NAME. You can alter only the database whose control files are specified by the initialization parameter CONTROL_FILES. The database identifier is not related to the Net8 database specification.

recover_clauses

You can use the following clauses when your instance has the database mounted, open or closed, and the files involved are not in use.

general_recovery_clause

The *general_recovery_clause* lets you design media recovery for the database or standby database, or for specified tablespaces or files.

Restrictions:

- You can recover the entire database only when the database is closed.
- Your instance must have the database mounted in exclusive mode.
- You can recover tablespaces or datafiles when the database is open or closed, provided that the tablespaces or datafiles to be recovered are offline.
- You cannot perform media recovery if you are connected to Oracle through the multi-threaded server architecture.

Note: If you do not have special media requirements, Oracle Corporation recommends that you use the SQL*Plus RECOVER statement.

See Also:

- *Oracle8i Backup and Recovery Guide* for more information on media recovery
- *SQL*Plus User's Guide and Reference*

AUTOMATIC — Specify AUTOMATIC if you want Oracle to automatically generate the name of the next archived redo log file needed to continue the recovery operation. If the LOG_ARCHIVE_DEST_*n* parameters are defined, Oracle scans those that are valid and enabled for the first local destination. It uses that destination in conjunction with LOG_ARCHIVE_FORMAT to generate the target redo log filename. If the LOG_ARCHIVE_DEST_*n* parameters are not defined, Oracle uses the value of the LOG_ARCHIVE_DEST parameter instead.

If the resulting file is found, Oracle applies the redo contained in that file. If the file is not found, Oracle prompts you for a filename, displaying the generated filename as a suggestion.

If you specify neither AUTOMATIC nor LOGFILE, Oracle prompts you for a filename, displaying the generated filename as a suggestion. You can then accept the generated filename or replace it with a fully qualified filename. If you know the archived filename differs from what Oracle would generate, you can save time by using the LOGFILE clause.

FROM '*location*' — Specify FROM *location* to indicate the location from which the archived redo log file group is read. The value of *location* must be a fully specified file location following the conventions of your operating system. If you omit this parameter, Oracle assumes the archived redo log file group is in the location specified by the initialization parameter LOG_ARCHIVE_DEST or LOG_ARCHIVE_DEST_1.

STANDBY DATABASE — Specify the STANDBY DATABASE clause to recover the standby database using the control file and archived redo log files copied from the primary database. The standby database must be mounted but not open.

DATABASE — Specify the DATABASE clause to recover the entire database. This is the default. You can use this clause only when the database is closed.

Note: This clause recovers only online datafiles.

- UNTIL: Use the UNTIL clause to specify the duration of the recovery operation.

- CANCEL indicates cancel-based recovery. This clause recovers the database until you issue the ALTER DATABASE RECOVER statement with the RECOVER CANCEL clause.

- TIME indicates time-based recovery. This parameter recovers the database to the time specified by the date. The date must be a character literal in the format 'YYYY-MM-DD:HH24:MI:SS'.

- CHANGE indicates change-based recovery. This parameter recovers the database to a transaction-consistent state immediately before the system change number (SCN) specified by integer.

- USING BACKUP CONTROLFILE: Specify this clause if you want to use a backup control file instead of the current control file.

TABLESPACE — Specify the TABLESPACE clause to recover only the specified tablespaces. You can use this clause if the database is open or closed, provided the tablespaces to be recovered are offline.

DATAFILE — Specify the DATAFILE clause to recover the specified datafiles. You can use this clause when the database is open or closed, provided the datafiles to be recovered are offline.

STANDBY TABLESPACE — Specify STANDBY TABLESPACE to reconstruct a lost or damaged tablespace in the standby database using archived redo log files copied from the primary database and a control file.

STANDBY | DATAFILE — Specify STANDBY DATAFILE to reconstruct a lost or damaged datafile in the standby database using archived redo log files copied from the primary database and a control file.

- UNTIL [CONSISTENT WITH] CONTROLFILE: Specify this clause if you want the recovery of an old standby datafile or tablespace to use the current standby database control file. However, any redo in advance of the standby controlfile will not be applied. The keywords CONSISTENT WITH are optional and are provided for semantic clarity.

LOGFILE — Specify the LOGFILE clause to continue media recovery by applying the specified redo log file.

CONTINUE — Specify CONTINUE to continue multi-instance recovery after it has been interrupted to disable a thread.

CONTINUE DEFAULT	Specify CONTINUE DEFAULT to continue recovery using the redo log file that Oracle would automatically generate if no other logfile were specified. This clause is equivalent to specifying AUTOMATIC, except that Oracle does not prompt for a filename.
CANCEL	Specify CANCEL to terminate cancel-based recovery.

managed_recovery_clause

The *managed_recovery_clause* specifies automated standby recovery mode. This mode assumes that the automated standby database is an active component of an overall standby database architecture. A primary database actively archives its redo log files to the standby site. As these archived redo logs arrive at the standby site, they become available for use by a managed standby recovery operation. Automated standby recovery is restricted to media recovery.

Restrictions: The same restrictions apply as are listed under *general_recovery_clause*.

> **See Also:** *Oracle8i Backup and Recovery Guide* for more information on the parameters of this clause.

TIMEOUT *integer*	Use the TIMEOUT clause to specify in minutes the wait period of the managed recovery operation. The recovery process waits for *integer* minutes for a requested archived log redo to be available for writing to the automated standby database. If the redo log file does not become available within that time, the recovery process terminates with an error message. You can then issue the statement again to return to automated standby recovery mode. If you do not specify this clause, the database remains in automated standby recovery mode until you reissue the statement with the RECOVER CANCEL clause or until instance shutdown or failure.
CANCEL	Use the CANCEL clause to terminate the managed recovery operation after applying all the redo in the current archived redo file.

CANCEL IMMEDIATE — Specify CANCEL IMMEDIATE to terminate the managed recovery operation after applying all the redo in the current archived redo file or after the next redo log file read, whichever comes first.

Restriction: This clause cannot be issued from the same session that issued the RECOVER MANAGED STANDBY DATABASE statement.

parallel_clause

Use the PARALLEL clause to specify whether the recovery of media will be parallelized.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL — Specify NOPARALLEL for serial execution. This is the default.

PARALLEL — Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

See Also: "Notes on the parallel_clause" for CREATE TABLE on page 10-41

datafile_tempfile_clauses

The datafile and tempfile clauses let you modify datafiles and tempfiles.

You can use any of the following clauses when your instance has the database mounted, open or closed, and the files involved are not in use:

CREATE DATAFILE

Use the CREATE DATAFILE clause to create a new empty datafile in place of an old one. You can use this clause to re-create a datafile that was lost with no backup. The '*filename*' must identify a file that is or was once part of the database. The *filespec* specifies the name and size of the new datafile. If you omit the AS clause, Oracle creates the new file with the name and size as the file specified by '*filename*'.

During recovery, all archived redo logs written to since the original datafile was created must be applied to the new, empty version of the lost datafile.

Oracle creates the new file in the same state as the old file when it was created. You must perform media recovery on the new file to return it to the state of the old file at the time it was lost.

Restriction: You cannot create a new file based on the first datafile of the SYSTEM tablespace.

DATAFILE *'filename'*

The DATAFILE clauses affect your database files as follows:

ONLINE

Specify ONLINE to bring the datafile online.

OFFLINE

Specify OFFLINE to take the datafile offline. If the database is open, you must perform media recovery on the datafile before bringing it back online, because a checkpoint is not performed on the datafile before it is taken offline.

DROP takes a datafile offline when the database is in NOARCHIVELOG mode.

RESIZE

Specify RESIZE if you want Oracle to attempt to increase or decrease the size of the datafile to the specified absolute size in bytes. Use K or M to specify this size in kilobytes or megabytes. There is no default, so you must specify a size.

If sufficient disk space is not available for the increased size, or if the file contains data beyond the specified decreased size, Oracle returns an error.

autoextend_clause — Use the *autoextend_clause* to enable or disable the automatic extension of a datafile. If you do not specify this clause, datafiles are not automatically extended.

OFF disables autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER DATABASE AUTOEXTEND statements.

- ON enables autoextend.
- NEXT specifies in bytes the size of the next increment of disk space to be automatically allocated to the datafile when more extents are required. Use K or M to specify this size in kilobytes or megabytes. The default is one data block.
- MAXSIZE specifies the maximum disk space allowed for automatic extension of the datafile.
- UNLIMITED sets no limit on allocating disk space to the datafile.

END BACKUP — Specify END BACKUP to avoid media recovery on database startup after an online tablespace backup was interrupted by a system failure or instance failure or SHUTDOWN ABORT.

Caution: Do not use ALTER TABLESPACE ... END BACKUP if you have restored any of the files affected from a backup. Media recovery is fully described in *Oracle8i Backup and Recovery Guide*.

TEMPFILE *'filename'* — Lets you resize your temporary datafile or specify the *autoextend_clause*, with the same effect as with a permanent datafile.

Restriction: You cannot specify TEMPFILE unless the database is open.

DROP — Specify DROP to drop *tempfile* from the database. The tablespace remains.

logfile_clauses

The logfile clauses let you add, drop, or modify log files.

ARCHIVELOG — Specify ARCHIVELOG if you want the contents of a redo log file group to be archived before the group can be reused. This mode prepares for the possibility of media recovery. Use this clause only after shutting down your instance normally or immediately with no errors and then restarting it, mounting the database in parallel server disabled mode.

NOARCHIVELOG — Specify NOARCHIVELOG if you do not want the contents of a redo log file group to be archived so that the group can be reused. This mode does not prepare for recovery after media failure.

Use the ARCHIVELOG clause and NOARCHIVELOG clause only if your instance has the database mounted in Oracle Parallel Server disabled mode, but not open.

ADD LOGFILE — Use the ADD LOGFILE clause to add one or more redo log file groups to the specified thread, making them available to the instance assigned the thread.

THREAD *integer* — The THREAD clause is applicable only if you are using Oracle with the Parallel Server option in parallel mode. *integer* is the thread number. The number of threads you can create is limited by the value of the MAXINSTANCES parameter specified in the CREATE DATABASE statement.

If you omit THREAD, the redo log file group is added to the thread assigned to your instance.

GROUP *integer* — The GROUP clause uniquely identifies the redo log file group among all groups in all threads and can range from 1 to the MAXLOGFILES value. You cannot add multiple redo log file groups having the same GROUP value. If you omit this parameter, Oracle generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance view V$LOG.

filespec — Each *filespec* specifies a redo log file group containing one or more members, or copies.

> **See Also:** the syntax description of *filespec* in *filespec* on page 11-27

ADD LOGFILE MEMBER — Use the ADD LOGFILE MEMBER clause to add new members to existing redo log file groups. Each new member is specified by '*filename*'. If the file already exists, it must be the same size as the other group members, and you must specify REUSE. If the file does not exist, Oracle creates a file of the correct size. You cannot add a member to a group if all of the group's members have been lost through media failure.

You can specify an existing redo log file group in one of these ways:

- GROUP *integer* — Specify the value of the GROUP parameter that identifies the redo log file group.
- *filename[s]* — List all members of the redo log file group. You must fully specify each filename according to the conventions of your operating system.

DROP LOGFILE — Use the DROP LOGFILE clause to drop all members of a redo log file group. Specify a redo log file group as indicated for the ADD LOGFILE MEMBER clause.

- To drop the current log file group, you must first issue an ALTER SYSTEM SWITCH LOGFILE statement.

  See Also: ALTER SYSTEM on page 7-127
- You cannot drop a redo log file group if it needs archiving.
- You cannot drop a redo log file group if doing so would cause the redo thread to contain less than two redo log file groups.

DROP LOGFILE MEMBER

Use the DROP LOGFILE MEMBER clause to drop one or more redo log file members. Each '*filename*' must fully specify a member using the conventions for filenames on your operating system.

- To drop a log file in the current log, you must first issue an ALTER SYSTEM SWITCH LOGFILE statement.

  See Also: ALTER SYSTEM on page 7-127

- You cannot use this clause to drop all members of a redo log file group that contains valid data. To perform this operation, use the DROP LOGFILE clause.

CLEAR LOGFILE

Use the CLEAR LOGFILE clause to reinitialize an online redo log, optionally without archiving the redo log. CLEAR LOGFILE is similar to adding and dropping a redo log, except that the statement may be issued even if there are only two logs for the thread and also may be issued for the current redo log of a closed thread.

UNARCHIVED

You must specify UNARCHIVED if you want to reuse a redo log that was not archived.

> **Caution:** Specifying UNARCHIVED makes backups unusable if the redo log is needed for recovery.

UNRECOVERABLE DATAFILE

You must specify UNRECOVERABLE DATAFILE if you have taken the datafile offline with the database in ARCHIVELOG mode (that is, you specified ALTER DATABSE ... DATAFILE OFFLINE without the DROP keyword), and if the unarchived log to be cleared is needed to recover the datafile before bringing it back online. In this case, you must drop the datafile and the entire tablespace once the CLEAR LOGFILE statement completes.

Do not use CLEAR LOGFILE to clear a log needed for media recovery. If it is necessary to clear a log containing redo after the database checkpoint, you must first perform incomplete media recovery. The current redo log of an open thread can be cleared. The current log of a closed thread can be cleared by switching logs in the closed thread.

If the CLEAR LOGFILE statement is interrupted by a system or instance failure, then the database may hang. If this occurs, reissue the statement after the database is restarted. If the failure occurred because of I/O errors accessing one member of a log group, then that member can be dropped and other members added.

controlfile_clauses

CREATE STANDBY CONTROLFILE Use the CREATE STANDBY CONTROLFILE clause to create a control file to be used to maintain a standby database. If the file already exists, you must specify REUSE.

> **See Also:** *Oracle8i Standby Database Concepts and Administration.*

BACKUP CONTROLFILE Use the BACKUP CONTROLFILE clause to back up the current control file.

- **TO '*filename*'** Specify the file to which the control file is backed up. You must fully specify the *filename* using the conventions for your operating system. If the specified file already exists, you must specify REUSE.
- **TO TRACE** Specify TO TRACE if you want Oracle to write SQL statements to the database's trace file rather than making a physical backup of the control file. The SQL statements can start up the database, re-create the control file, and recover and open the database appropriately, based on the created control file.

  You can copy the statements from the trace file into a script file, edit the statements as necessary, and use the database if all copies of the control file are lost (or to change the size of the control file).

- RESETLOGS indicates that the SQL statement written to the trace file for starting the database is ALTER DATABASE OPEN RESETLOGS.
- NORESETLOGS indicates that the SQL statement written to the trace file for starting the database is ALTER DATABASE OPEN NORESETLOGS.

MOUNT

Use the MOUNT clause to mount the database. Do not use this clause when the database is mounted.

STANDBY DATABASE — Specify STANDBY to mount the standby database.

See Also: *Oracle8i Standby Database Concepts and Administration*

CLONE DATABASE — Specify CLONE to mount the clone database.

See Also: *Oracle8i Backup and Recovery Guide*

CONVERT

Use the CONVERT clause to complete the conversion of the Oracle7 data dictionary. After you use this clause, the Oracle7 data dictionary no longer exists in the Oracle database.

Note: Use this clause only when you are migrating to Oracle8*i*, and do not use this clause when the database is mounted.

See Also: *Oracle8i Migration*

ACTIVATE STANDBY DATABASE

The ACTIVATE STANDBY DATABASE clause changes the state of a standby database to an active database. Do not use this clause when the database is mounted.

See Also: *Oracle8i Standby Database Concepts and Administration*

OPEN

Use the OPEN clause to make the database available for normal use. You must mount the database before you can open it. You must activate a standby database before you can open it.

READ ONLY — Specify READ ONLY to restrict users to read-only transactions, preventing them from generating redo logs. You can use this clause to make a standby database available for queries even while archive logs are being copied from the primary database site.

Restrictions:

- You cannot open a database READ ONLY if it is currently opened READ WRITE by another instance.
- You cannot open a database READ ONLY if it requires recovery.
- You cannot take tablespaces offline while the database is open READ ONLY. However, you can take datafiles offline and online, and you can recover offline datafiles and tablespaces while the database is open READ ONLY.

READ WRITE — Specify READ WRITE to open the database in read-write mode, allowing users to generate redo logs. This is the default.

RESETLOGS — Specify RESETLOGS to reset the current log sequence number to 1 and discards any redo information that was not applied during recovery, ensuring that it will never be applied. This effectively discards all changes that are in the redo log, but not in the database.

You must specify RESETLOGS to open the database after performing media recovery with an incomplete recovery using the RECOVER clause or with a backup control file. After opening the database with this clause, you should perform a complete database backup.

NORESETLOGS — Specify NORESETLOGS to retain the current stat of the log sequence number and redo log files.

Restriction: You can specify RESETLOGS and NORESETLOGS only after performing incomplete media recovery or complete media recovery with a backup control file. In any other case, Oracle uses the NORESETLOGS automatically.

RENAME GLOBAL_NAME

Specify RENAME GLOBAL_NAME to change the global name of the database. The *database* is the new database name and can be as long as eight bytes. The optional *domain* specifies where the database is effectively located in the network hierarchy. Do not use this clause when the database is mounted.

Note: Renaming your database does not change global references to your database from existing database links, synonyms, and stored procedures and functions on remote databases. Changing such references is the responsibility of the administrator of the remote databases.

See Also: *Oracle8i Distributed Database Systems* for more information on global names

RENAME FILE

Use the RENAME FILE clause to rename datafiles, tempfiles, or redo log file members. This clause renames only files in the control file. It does not actually rename them on your operating system. You must specify each filename using the conventions for filenames on your operating system before specifying this clause. Do not use this clause when the database is mounted.

RESET COMPATIBILITY

Specify RESET COMPATIBILITY to mark the database to be reset to an earlier version of Oracle when the database is next restarted. Do not use this clause when the database is mounted.

Note: RESET COMPATIBILITY works only if you have successfully disabled Oracle features that affect backward compatibility.

See Also: *Oracle8i Migration* for more information on downgrading to an earlier version of Oracle

ENABLE THREAD

In an Oracle Parallel Server environment, specify ENABLE THREAD to enable the specified thread of redo log file groups. The thread must have at least two redo log file groups before you can enable it. The database must be open.

PUBLIC — Specify PUBLIC to make the enabled thread available to any instance that does not explicitly request a specific thread with the initialization parameter THREAD. If you omit PUBLIC, the thread is available only to the instance that explicitly requests it with the initialization parameter THREAD.

See Also: *Oracle8i Parallel Server Admininstration and Tuning* for more information on enabling and disabling threads.

DISABLE THREAD

Specify DISABLE THREAD to disable the specified thread, making it unavailable to all instances. The database must be open, but you cannot disable a thread if an instance using it has the database mounted.

See Also: *Oracle8i Parallel Server Admininstration and Tuning* for more information on enabling and disabling threads.

CHARACTER SET, NATIONAL CHARACTER SET

CHARACTER SET changes the character set the database uses to store data. NATIONAL CHARACTER SET changes the national character set used to store data in columns specifically defined as NCHAR, NCLOB, or NVARCHAR2. Specify *character_set* without quotation marks. The database must be open.

Caution: You cannot roll back an ALTER DATABASE CHARACTER SET or ALTER DATABASE NATIONAL CHARACTER SET statement. Therefore, you should perform a full backup before issuing either of these statements.

Restrictions:

- You must have SYSDBA system privilege, and you must start up the database in restricted mode (for example, with the SQL*Plus STARTUP RESTRICT command).
- The current character set must be a strict subset of the character set to which you change. That is, each character represented by a codepoint value in the source character set must be represented by the same codepoint value in the target character set.

> **See Also:** *Oracle8i National Language Support Guide* for a list of valid character sets

Examples

READ ONLY / READ WRITE Example The first statement below opens the database in read-only mode. The second statement returns the database to read-write mode and clears the online redo logs:

```
ALTER DATABASE OPEN READ ONLY;

ALTER DATABASE OPEN READ WRITE RESETLOGS;
```

PARALLEL Example The following statement performs tablespace recovery using parallel recovery processes:

```
ALTER DATABASE
   RECOVER TABLESPACE binky
   PARALLEL;
```

Redo Log File Group Example The following statement adds a redo log file group with two members and identifies it with a GROUP parameter value of 3:

```
ALTER DATABASE stocks
  ADD LOGFILE GROUP 3
    ('diska:log3.log' ,
     'diskb:log3.log') SIZE 50K;
```

Redo Log File Group Member Example The following statement adds a member to the redo log file group added in the previous example:

```
ALTER DATABASE stocks
   ADD LOGFILE MEMBER 'diskc:log3.log'
   TO GROUP 3;
```

Dropping a Log File Member The following statement drops the redo log file member added in the previous example:

```
ALTER DATABASE stocks
    DROP LOGFILE MEMBER 'diskc:log3.log';
```

Renaming a Log File Member Example The following statement renames a redo log file member:

```
ALTER DATABASE stocks
    RENAME FILE 'diskb:log3.log' TO 'diskd:log3.log';
```

The above statement only changes the member of the redo log group from one file to another. The statement does not actually change the name of the file `'diskbk:log3.log'` to `'diskd:log3.log'`. You must perform this operation through your operating system.

Dropping All Log File Group Members Example The following statement drops all members of the redo log file group 3:

```
ALTER DATABASE stocks DROP LOGFILE GROUP 3;
```

Adding a Redo Log File Group Example The following statement adds a redo log file group containing three members to thread 5 (in an Oracle Parallel Server environment) and assigns it a GROUP parameter value of 4:

```
ALTER DATABASE stocks
    ADD LOGFILE THREAD 5 GROUP 4
        ('diska:log4.log',
        'diskb:log4:log',
        'diskc:log4.log' );
```

Disabling a Parallel Server Thread Example The following statement disables thread 5 in an Oracle Parallel Server environment:

```
ALTER DATABASE stocks
    DISABLE THREAD 5;
```

Enabling a Parallel Server Thread Example The following statement enables thread 5 in an Oracle Parallel Server, making it available to any Oracle instance that does not explicitly request a specific thread:

```
ALTER DATABASE stocks
    ENABLE PUBLIC THREAD 5;
```

Creating a New Datafile Example The following statement creates a new datafile 'disk2:db1.dat' based on the file 'disk1:db1.dat':

```
ALTER DATABASE
    CREATE DATAFILE 'disk1:db1.dat' AS 'disk2:db1.dat';
```

Changing the Global Database Name Example The following statement changes the global name of the database and includes both the database name and domain:

```
ALTER DATABASE
    RENAME GLOBAL_NAME TO sales.australia.acme.com;
```

CHARACTER SET Example The following statements change the database character set and national character set to the WE8ISO8859P1 character set:

```
ALTER DATABASE db1 CHARACTER SET WE8ISO8859P1;
ALTER DATABASE db1 NATIONAL CHARACTER SET WE8ISO8859P1;
```

The database name is optional, and the character set name is specified without quotation marks.

Resizing a Datafile Example The following statement attempts to change the size of datafile 'disk1:db1.dat':

```
ALTER DATABASE
    DATAFILE 'disk1:db1.dat' RESIZE 10 M;
```

Clearing a Log File The following statement clears a log file:

```
ALTER DATABASE
    CLEAR LOGFILE 'disk3:log.dbf';
```

Database Recovery Examples The following statement performs complete recovery of the entire database, letting Oracle generate the name of the next archived redo log file needed:

```
ALTER DATABASE
  RECOVER AUTOMATIC DATABASE;
```

The following statement explicitly names a redo log file for Oracle to apply:

```
ALTER DATABASE
    RECOVER LOGFILE 'diska:arch0006.arc';
```

The following statement performs time-based recovery of the database:

```
ALTER DATABASE
```

```
    RECOVER AUTOMATIC UNTIL TIME '1998-10-27:14:00:00';
```

Oracle recovers the database until 2:00 pm on October 27, 1998.

The following statement recovers the tablespace `user5`:

```
ALTER DATABASE
    RECOVER TABLESPACE user5;
```

The following statement recovers the standby datafile `/finance/stbs_21.f`, using the corresponding datafile in the original standby database, plus all relevant archived logs and the current standby database control file:

```
ALTER DATABASE
   RECOVER STANDBY DATAFILE '/finance/stbs_21.f'
   UNTIL CONTROLFILE;
```

Managed Standby Database Examples The following statement recovers the standby database in automated standby recovery mode:

```
ALTER DATABASE
   RECOVER MANAGED STANDBY DATABASE;
```

The following statement puts the database in automated standby recovery mode. The managed recovery process will wait up to 60 minutes for the next archive log:

```
ALTER DATABASE
   RECOVER MANAGED STANDBY DATABASE TIMEOUT 60;
```

If each subsequent log arrives within 60 minutes of the last log, recovery continues indefinitely or until manually terminated.

The following statement terminates the managed recovery operation:

```
ALTER DATABASE
   RECOVER MANAGED STANDBY DATABASE CANCEL IMMEDIATE;
```

The managed recovery operation terminates before the next group of redo is read from the current redo log file. Media recovery ends in the "middle" of applying redo from the current redo log file.

ALTER DIMENSION

Purpose

Use the ALTER DIMENSION statement to change the hierarchical relationships or dimension attributes of a dimension.

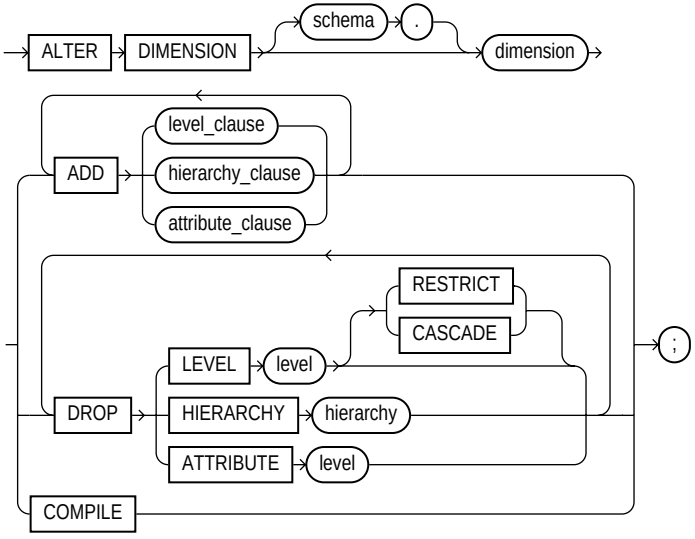
> **See Also:** CREATE DIMENSION on page 9-34 for more information on dimensions

Prerequisites

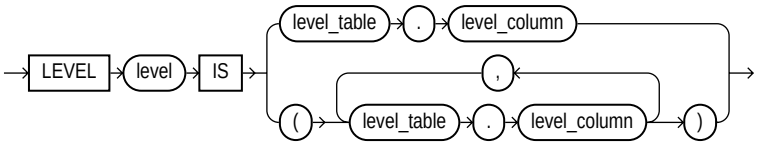
The dimension must be in your schema or you must have the ALTER ANY DIMENSION system privilege to use this statement.

A dimension is always altered under the rights of the owner.

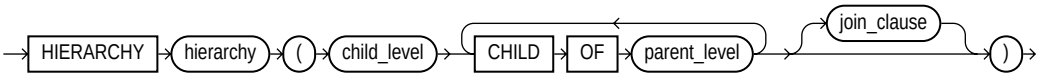
Syntax



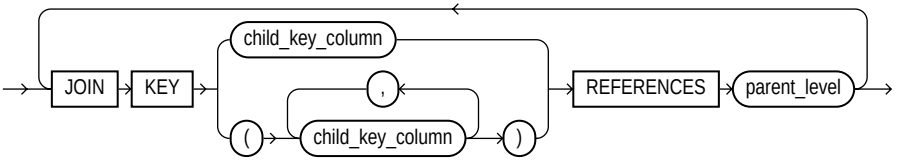
level_clause::=



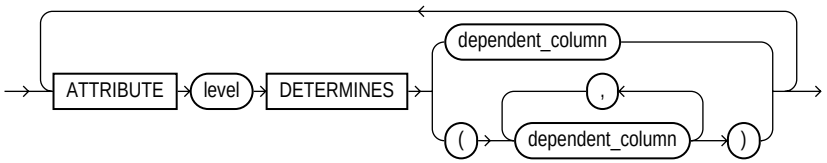
hierarchy_clause::=



join_clause::=



attribute_clause::=



Keywords and Parameters

The following keywords and parameters have meaning unique to ALTER DIMENSION. The remaining keywords and parameters have the same functionality that they have in the CREATE DIMENSION statement.

See Also: CREATE DIMENSION on page 9-34

schema

Specify the schema of the dimension you want to modify. If you do not specify *schema*, Oracle assumes the dimension is in your own schema.

dimension

Specify the name of the dimension. This dimension must already exist.

ADD

The ADD clauses let you add a level, hierarchy, or attribute to the dimension. Adding one of these elements does not invalidate any existing materialized view.

Oracle processes ADD LEVEL clauses prior to any other ADD clauses.

DROP

The DROP clauses let you drop a level, hierarchy, or attribute from the dimension. Any level, hierarchy, or attribute you specify must already exist.

Restriction: If any attributes or hierarchies reference a level, you cannot drop the level until you either drop all the referencing attributes and hierarchies or specify CASCADE.

CASCADE — Specify CASCADE if you want Oracle to drop any attributes or hierarchies that reference the level, along with the level itself.

RESTRICT — Specify RESTRICT if you want to prevent Oracle from dropping a level that is referenced by any attributes or hierarchies. This is the default.

COMPILE

Specify COMPILE to explicitly recompile an invalidated dimension. Oracle automatically compiles a dimension when you issue an ADD clause or DROP clause. However, if you alter an object referenced by the dimension (for example, if you drop and then re-create a table referenced in the dimension), the dimension will be invalidated, and you must recompile it explicitly.

Example

Modifying a Dimension Example This example modifies the time dimension:

```
ALTER DIMENSION time
   DROP HIERARCHY week_month;
```

```
ALTER DIMENSION time
   DROP ATTRIBUTE cur_date;
ALTER DIMENSION time
   ADD LEVEL day IS time_tab.t_day
   ADD ATTRIBUTE day DETERMINES t_holiday;
```

ALTER FUNCTION

Purpose

Use the ALTER FUNCTION statement to recompile an invalid standalone stored function. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

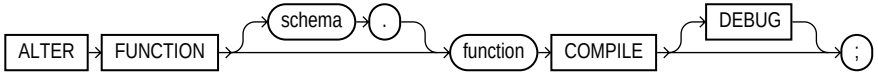
The ALTER FUNCTION statement is similar to ALTER PROCEDURE on page 7-88. For information on how Oracle recompiles functions and procedures, see *Oracle8i Concepts*.

> **Note:** This statement does not change the declaration or definition of an existing function. To redeclare or redefine a function, use the CREATE FUNCTION statement with the OR REPLACE clause; see CREATE FUNCTION on page 9-43.

Prerequisites

The function must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the function. If you omit *schema*, Oracle assumes the function is in your own schema.

function

Specify the name of the function to be recompiled.

COMPILE

Specify COMPILE to cause Oracle to recompile the function. The COMPILE keyword is required. If Oracle does not compile the function successfully, you can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

DEBUG

Specify DEBUG to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

Example

Recompiling a Function Example To explicitly recompile the function get_bal owned by the user merriweather, issue the following statement:

```
ALTER FUNCTION merriweather.get_bal
   COMPILE;
```

If Oracle encounters no compilation errors while recompiling get_bal, get_bal becomes valid. Oracle can subsequently execute it without recompiling it at run time. If recompiling get_bal results in compilation errors, Oracle returns an error, and get_bal remains invalid.

Oracle also invalidates all objects that depend upon get_bal. If you subsequently reference one of these objects without explicitly recompiling it first, Oracle recompiles it implicitly at run time.

ALTER INDEX

Purpose

Use the ALTER INDEX statement to change or rebuild an existing index.

> **See Also:** CREATE INDEX on page 9-52 for information on creating an index

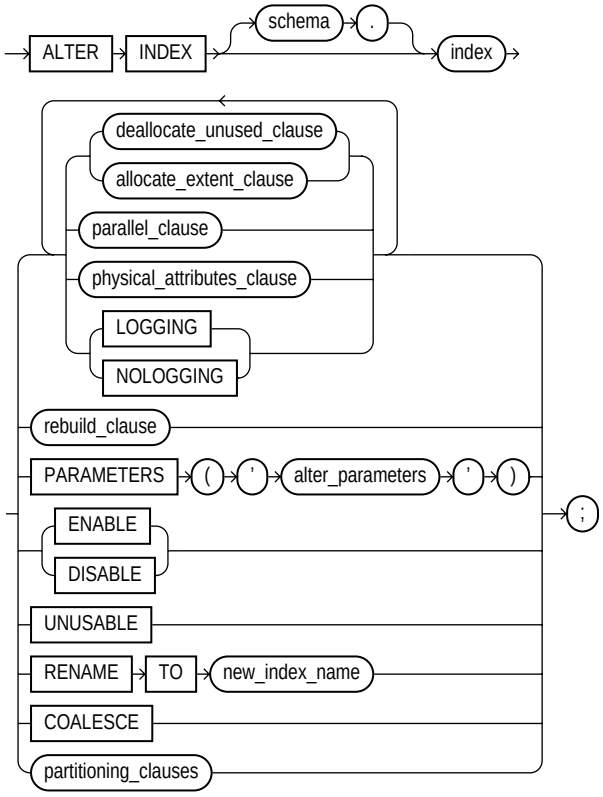
Prerequisites

The index must be in your own schema or you must have ALTER ANY INDEX system privilege.

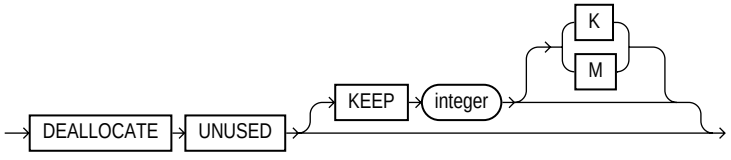
Schema object privileges are granted on the parent index, not on individual index partitions or subpartitions.

You must have tablespace quota to modify, rebuild, or split an index partition or to modify or rebuild an index subpartition.

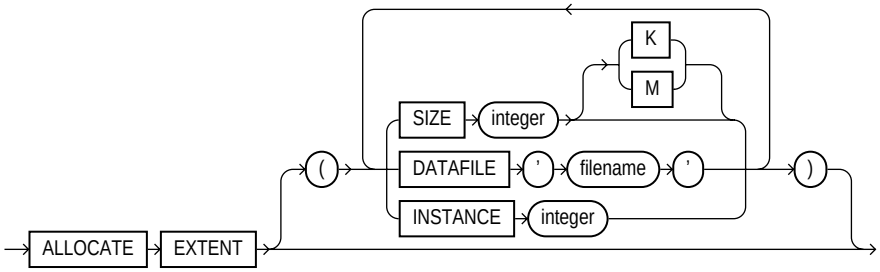
Syntax



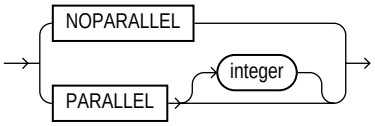
deallocate_unused_clause::=



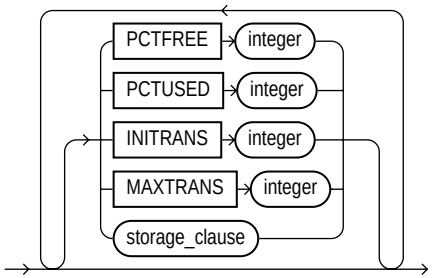
allocate_extent_clause::=



parallel_clause::=

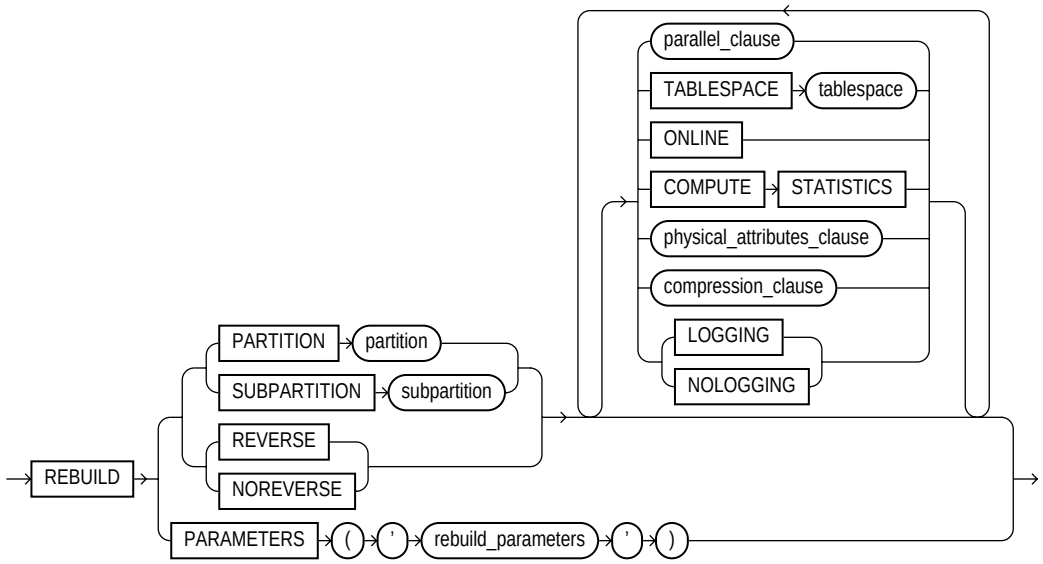


physical_attributes_clause::=

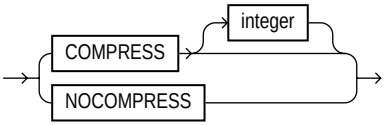


storage_clause: See storage_clause on page 11-129.

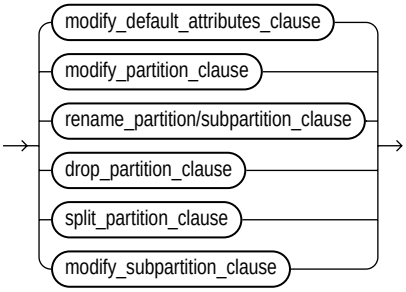
rebuild_clause::=



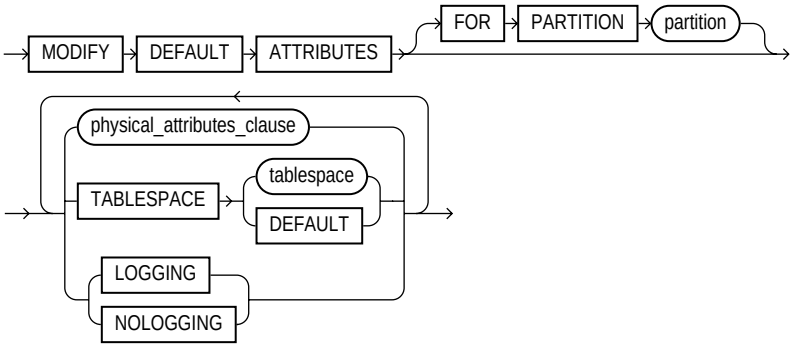
compression_clause::=



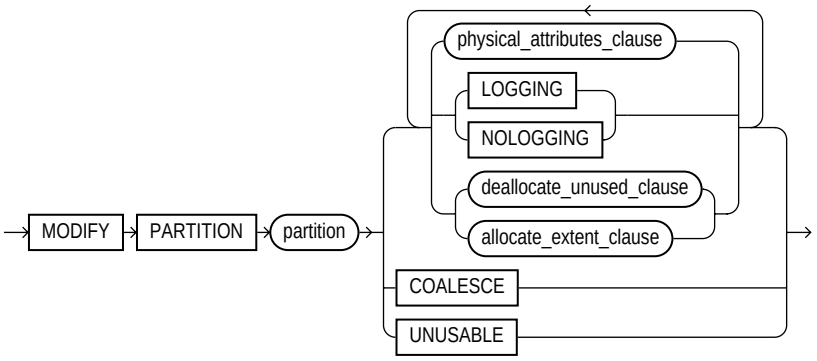
partitioning_clauses::=



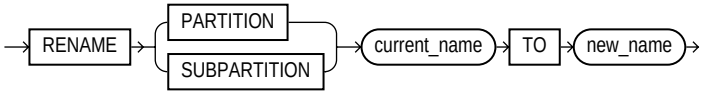
modify_default_attributes_clause::=



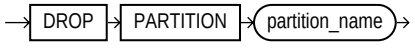
modify_partition_clause::=



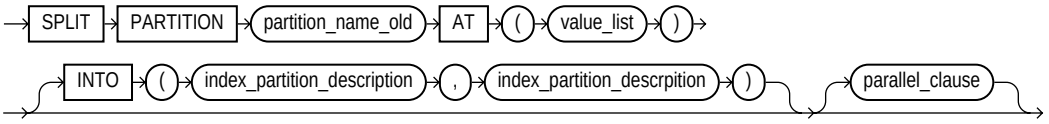
rename_partition / subpartition_clause::=



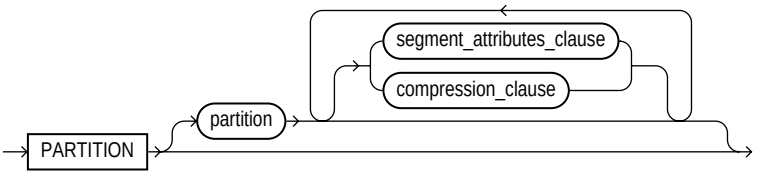
drop_partition_clause::=



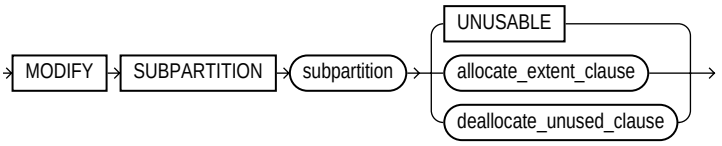
split_partition_clause::=



index_partition_description::=



modify_subpartition_clause::=



Keywords and Parameters

schema

Specify the schema containing the index. If you omit *schema*, Oracle assumes the index is in your own schema.

index

Specify the name of the index to be altered.

Restrictions:

- If *index* is a domain index, you can specify only the PARAMETERS clause, the RENAME clause, or the *rebuild_clause* (with or without the PARAMETERS clause). No other clauses are valid.

- You cannot alter or rename a domain index that is marked LOADING or FAILED. If an index is marked FAILED, the only clause you can specify is REBUILD.

  See Also: *Oracle8i Data Cartridge Developer's Guide* for information on the LOADING and FAILED states of domain indexes

deallocate_unused_clause

The *deallocate_unused_clause* lets you explicitly deallocate unused space at the end of the index and makes the freed space available for other segments in the tablespace. Only unused space above the high water mark can be freed.

If *index* is range-partitioned or hash-partitioned, Oracle deallocates unused space from each index partition. If *index* is a local index on a composite-partitioned table, Oracle deallocates unused space from each index subpartition.

Restrictions:

- You cannot specify this clause for an index on a temporary table.
- You cannot specify this clause and also specify the *rebuild_clause*.

  See Also: ALTER TABLE on page 8-2 for more information on this clause

KEEP *integer* The KEEP clause lets you specify the number of bytes above the high water mark that the index will have after deallocation. If the number of remaining extents are less than MINEXTENTS, then MINEXTENTS is set to the current number of extents. If the initial extent becomes smaller than INITIAL, then INITIAL is set to the value of the current initial extent. If you omit KEEP, all unused space is freed.

See Also: ALTER TABLE on page 8-2 for a complete description of this clause

allocate_extent_clause

The *allocate_extent_clause* lets you explicitly allocate a new extent for the index. For a local index on a hash-partitioned table, Oracle allocates a new extent for each partition of the index.

Restriction: You cannot specify this clause for an index on a temporary table or for a range-partitioned or composite-partitioned index.

SIZE *integer*	Specify the size of the extent in bytes. Use K or M to specify the extent size in kilobytes or megabytes. If you omit SIZE, Oracle determines the size based on the values of the index's storage parameters.
DATAFILE '*filename*'	Specify one of the datafiles in the index's tablespace to contain the new extent. If you omit DATAFILE, Oracle chooses the datafile.
INSTANCE *integer*	Use the INSTANCE clause to make the new extent available to the specified instance. An instance is identified by the value of its initialization parameter INSTANCE_NUMBER. If you omit this parameter, the extent is available to all instances. Use this parameter only if you are using Oracle with the Parallel Server option in parallel mode.

Explicitly allocating an extent with this clause does not change the values of the NEXT and PCTINCREASE storage parameters, so does not affect the size of the next extent to be allocated.

parallel_clause

Use the PARALLEL clause to change the default degree of parallelism for queries and DML on the index.

Restriction: You cannot specify this clause for an index on a temporary table.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL	Specify NOPARALLEL for serial execution. This is the default.
PARALLEL	Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 10-41

physical_attributes_clause

Use the *physical_attributes_clause* to change the values of parameters for a nonpartitioned index, all partitions and subpartitions of a partitioned index, a specified partition, or all subpartitions of a specified partition.

> **See Also:** the physical attributes parameters in CREATE TABLE on page 10-7

Restrictions:

- You cannot specify this clause for an index on a temporary table.
- You cannot specify the PCTUSED parameter when altering an index.
- You cannot change the value of the PCTFREE parameter for the index as a whole (ALTER INDEX) or for a partition (ALTER INDEX ... MODIFY PARTITION). You can specify PCTFREE in all other forms of the ALTER INDEX statement.

storage_clause — Use the *storage_clause* to change the storage parameters for a nonpartitioned index, index partition, or all partitions of a partitioned index, or default values of these parameters for a partitioned index.

> **See Also:** *storage_clause* on page 11-129

LOGGING | NOLOGGING — Use LOGGING or NOLOGGING to specify whether subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against a nonpartitioned index, a range or hash index partition, or all partitions or subpartitions of a composite-partitioned index will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file.

In NOLOGGING mode, data is modified with minimal logging (to mark new extents invalid and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not logged. Therefore, if you cannot afford to lose this index, you must take a backup after the operation in NOLOGGING mode.

If the database is run in ARCHIVELOG mode, media recovery from a backup taken before an operation in LOGGING mode will re-create the index. However, media recovery from a backup taken before an operation in NOLOGGING mode will not re-create the index.

An index segment can have logging attributes different from those of the base table and different from those of other index segments for the same base table.

Restriction: You cannot specify this clause for an index on a temporary table.

> **See Also:** *Oracle8i Concepts* and the *Oracle8i Parallel Server Concepts* for more information about LOGGING and parallel DML

RECOVERABLE | UNRECOVERABLE

These keywords are deprecated and have been replaced with LOGGING and NOLOGGING, respectively. Although RECOVERABLE and UNRECOVERABLE are supported for backward compatibility, Oracle Corporation strongly recommends that you use the LOGGING and NOLOGGING keywords.

RECOVERABLE is not a valid keyword for creating partitioned tables or LOB storage characteristics. UNRECOVERABLE is not a valid keyword for creating partitioned or index-organized tables. Also, it can be specified only with the AS subquery clause of CREATE INDEX.

rebuild_clause

Use the *rebuild_clause* to re-create an existing index or one of its partitions or subpartitions. For a function-based index, this clause also enables the index. If the function on which the index is based does not exist, the rebuild statement will fail.

Restrictions:

- You cannot rebuild an index on a temporary table.

- You cannot rebuild an entire partitioned index. You must rebuild each partition or subpartition, as described below.
- You cannot also specify the *deallocate_unused_clause* in this statement.
- You cannot change the value of the PCTFREE parameter for the index as a whole (ALTER INDEX) or for a partition (ALTER INDEX ... MODIFY PARTITION). You can specify PCTFREE in all other forms of the ALTER INDEX statement.

PARTITION *partition*
: Use the PARTITION clause to rebuild one partition of an index. You can also use this clause to move an index partition to another tablespace or to change a create-time physical attribute.

  Restriction: You cannot specify this clause for a local index on a composite-partitioned table. Instead, use the REBUILD SUBPARTITION clause.

  See Also: *Oracle8i Administrator's Guide* for more information about partition maintenance operations

SUBPARTITION *subpartition*
: Use the SUBPARTITION clause to rebuild one subpartition of an index. You can also use this clause to move an index subpartition to another tablespace. If you do not specify TABLESPACE, the subpartition is rebuilt in the same tablespace.

  Restrictions: The only parameters you can specify for a subpartition are TABLESPACE and the *parallel_clause*.

REVERSE | NOREVERSE
: Indicate whether the bytes of the index block are stored in reverse order:

  - REVERSE stores the bytes of the index block in reverse order and excludes the rowid when the index is rebuilt.
  - NOREVERSE stores the bytes of the index block without reversing the order when the index is rebuilt. Rebuilding a REVERSE index without the NOREVERSE keyword produces a rebuilt, reverse-keyed index.

  Restrictions:

  - You cannot reverse a bitmap index or an index-organized table.
  - You cannot specify REVERSE or NOREVERSE for a partition or subpartition.

TABLESPACE *tablespace*

Specify the tablespace where the rebuilt index, index partition, or index subpartition will be stored. The default is the default tablespace where the index or partition resided before you rebuilt it.

COMPRESS

Specify COMPRESS to enable key compression, which eliminates repeated occurrence of key column values. Use *integer* to specify the prefix length (number of prefix columns to compress).

- For unique indexes, the range of valid prefix length values is from 1 to the number of key columns minus 1. The default prefix length is the number of key columns minus 1.
- For nonunique indexes, the range of valid prefix length values is from 1 to the number of key columns. The default prefix length is number of key columns.

Oracle compresses only nonpartitioned indexes that are nonunique or unique indexes of at least two columns.

Restriction: You cannot specify COMPRESS for a bitmap index.

NOCOMPRESS

Specify NOCOMPRESS to disable key compression. This is the default.

ONLINE

Specify ONLINE to allow DML operations on the table or partition during rebuilding of the index.

Restrictions:

- You cannot specify ONLINE when rebuilding the secondary index of an index-organized table.
- Parallel DML is not supported during online index building. If you specify ONLINE and then issue parallel DML statements, Oracle returns an error.

COMPUTE STATISTICS

Specify COMPUTE STATISTICS if you want to collect statistics at relatively little cost during the rebuilding of an index. These statistics are stored in the data dictionary for ongoing use by the optimizer in choosing a plan of execution for SQL statements.

The types of statistics collected depend on the type of index you are rebuilding.

> **Note:** If you create an index using another index (instead of a table), the original index might not provide adequate statistical information. Therefore, Oracle generally uses the base table to compute the statistics, which will improve the statistics but may negatively affect performance.

Additional methods of collecting statistics are available in PL/SQL packages and procedures.

See Also: *Oracle8i Supplied PL/SQL Packages Reference*

`LOGGING | NOLOGGING` Specify whether the `ALTER INDEX ... REBUILD` operation will be logged.

PARAMETERS

The PARAMETERS clause applies only to **domain indexes**. This clause specifies the parameter string for altering the index (or, in the *`rebuild_clause`*, rebuilding the index). The maximum length of the parameter string is 1000 characters. This string is passed uninterpreted to the appropriate indextype routine.

Restrictions:

- You cannot specify this clause for any indexes other than domain indexes.
- The parameter string is passed to the appropriate routine only if *index* is not marked UNUSABLE.

See Also:

- *Oracle8i Data Cartridge Developer's Guide* for more information on indextype routines
- CREATE INDEX on page 9-52 for more information on domain indexes

ENABLE

ENABLE applies only to a **function-based index** that has been disabled because a user-defined function used by the index was dropped or replaced. This clause enables such an index if these conditions are true:

- The function is currently valid
- The signature of the current function matches the signature of the function when the index was created

- The function is currently marked as DETERMINISTIC

Restriction: You cannot specify any other clauses of ALTER INDEX in the same statement with ENABLE.

DISABLE

DISABLE applies only to a **function-based index**. This clause enables you to disable the use of a function-based index. You might want to do so, for example, while working on the body of the function. Afterward you can either rebuild the index or specify another ALTER INDEX statement with the ENABLE keyword.

UNUSABLE

Specify UNUSABLE to mark the index or index partition(s) or index subpartition(s) UNUSABLE. An unusable index must be rebuilt, or dropped and re-created, before it can be used. While one partition is marked UNUSABLE, the other partitions of the index are still valid. You can execute statements that require the index if the statements do not access the unusable partition. You can also split or rename the unusable partition before rebuilding it.

Restriction: You cannot specify this clause for an index on a temporary table.

RENAME TO

Use the RENAME clause to rename *index* to *new_index_name*. The *new_index_name* is a single identifier and does not include the schema name.

COALESCE

Specify COALESCE to instruct Oracle to merge the contents of index blocks where possible to free blocks for reuse.

Restriction: You cannot specify this clause for an index on a temporary table.

> **See Also:** *Oracle8i Administrator's Guide* for more information on space management and coalescing indexes

partitioning_clauses

The partitioning clauses of the ALTER INDEX statement are valid only for partitioned indexes.

Restrictions:

- You cannot specify any of these clauses for an index on a temporary table.

- You can combine several operations on the base index into one ALTER INDEX statement (except RENAME and REBUILD), but you cannot combine partition operations with other partition operations or with operations on the base index.

modify_default_attributes_clause

Specify new values for the default attributes of a partitioned index.

Restriction: The only attribute you can specify for an index on a hash-partitioned or composite-partitioned table is TABLESPACE.

TABLESPACE	Specify the default tablespace for new partitions of an index or subpartitions of an index partition.
LOGGING | NOLOGGING	Specify the default logging attribute of a partitioned index or an index partition.
FOR PARTITION *partition*	Use the FOR PARTITION clause to specify the default attributes for the subpartitions of a partition of a local index on a composite-partitioned table.

modify_partition_clause

Use the *modify_partition_clause* to modify the real physical attributes, logging attribute, or storage characteristics of index partition *partition* or its subpartitions.

Restriction: You cannot specify the *physical_attributes_clause* for an index on a hash-partitioned table.

Note: If the index is a local index on a composite-partitioned table, the changes you specify here will override any attributes specified earlier for the subpartitions of index, as well as establish default values of attributes for future subpartitions of that partition. To change the default attributes of the partition without overriding the attributes of subpartitions, use ALTER TABLE ... MODIFY DEFAULT ATTRIBUTES OF PARTITION.

---

rename_partition | subpartition_clause

Use the *rename_partition* or *rename_subpartition* to rename index partition or subpartition to *new_name*.

drop_partition_clause

Use the *drop_partition_clause* to remove a partition and the data in it from a partitioned global index. When you drop a partition of a global index, Oracle marks the index's next partition UNUSABLE. You cannot drop the highest partition of a global index.

split_partition_clause

Use the *split_partition_clause* to split a partition of a global partitioned index into two partitions, adding a new partition to the index.

Splitting a partition marked UNUSABLE results in two partitions, both marked UNUSABLE. You must rebuild the partitions before you can use them.

Splitting a usable partition results in two partitions populated with index data. Both new partitions are usable.

AT (*value_list*)	Specify the new noninclusive upper bound for *split_partition_1*. The *value_list* must evaluate to less than the presplit partition bound for *partition_name_old* and greater than the partition bound for the next lowest partition (if there is one).
INTO *index_partition_description*	Specify (optionally) the name and physical attributes of each of the two partitions resulting from the split.

modify_subpartition_clause

Use the *modify_subpartition_clause* to mark UNUSABLE or allocate or deallocate storage for a subpartition of a local index on a composite-partitioned table. All other attributes of such a subpartition are inherited from partition-level default attributes.

Examples

Modifying Real Attributes Example This statement alters Scott's `customer` index so that future data blocks within this index use 5 initial transaction entries and an incremental extent of 100 kilobytes:

```
ALTER INDEX scott.customer
    INITRANS 5
    STORAGE (NEXT 100K);
```

If the scott.customer index is partitioned, this statement also alters the default attributes of future partitions of the index. New partitions added in the future will use 5 initial transaction entries and an incremental extent of 100K.

Dropping an Index Partition Example The following statement drops index partition ix_antarctica:

```
ALTER INDEX sales_area_ix
  DROP PARTITION ix_antarctica;
```

Modifying Default Attributes Example This statement alters the default attributes of local partitioned index sales_ix3. New partitions added in the future will use 5 initial transaction entries and an incremental extent of 100K:

```
ALTER INDEX sales_ix3
  MODIFY DEFAULT ATTRIBUTES INITRANS 5 STORAGE ( NEXT 100K );
```

Marking an Index Unusable Example The following statement marks the odx_acctno index as UNUSABLE:

```
ALTER INDEX idx_acctno UNUSABLE;
```

Marking a Partition Unusable Example The following statement marks partition idx_feb96 of index idx_acctno as UNUSABLE:

```
ALTER INDEX idx_acctno MODIFY PARTITION idx_feb96 UNUSABLE;
```

Changing MAXEXTENTS Example The following statement changes the maximum number of extents for partition brix_ny and changes the logging attribute:

```
ALTER INDEX branch_ix MODIFY PARTITION brix_ny
  STORAGE( MAXEXTENTS 30 ) LOGGING;
```

Disabling Parallel Queries Example The following statement sets the parallel attributes for index artist_ix so that scans on the index will not be parallelized:

```
ALTER INDEX artist_ix NOPARALLEL;
```

Rebuilding a Partition Example The following statement rebuilds partition p063 in index artist_ix. The rebuilding of the index partition will not be logged:

```
ALTER INDEX artist_ix
  REBUILD PARTITION p063 NOLOGGING;
```

Renaming an Index Example The following statement renames an index:

```
ALTER INDEX emp_ix1 RENAME TO employee_ix1;
```

Renaming an Index Partition Example The following statement renames an index partition:

```
ALTER INDEX employee_ix1 RENAME PARTITION emp_ix1_p3
  TO employee_ix1_p3;
```

Splitting a Partition Example The following statement splits partition `partnum_ix_p6` in partitioned index `partnum_ix` into `partnum_ix_p5` and `partnum_ix_p6`:

```
ALTER INDEX partnum_ix
  SPLIT PARTITION partnum_ix_p6 AT ( 5001 )
  INTO ( PARTITION partnum_ix_p5 TABLESPACE ts017 LOGGING,
         PARTITION partnum_ix_p6 TABLESPACE ts004 );
```

The second partition retains the name of the old partition.

Storing Index Blocks in Reverse Order Example The following statement rebuilds index `emp_ix` so that the bytes of the index block are stored in REVERSE order:

```
ALTER INDEX emp_ix REBUILD REVERSE;
```

Collecting Index Statistics Example The following statement collects statistics on the nonpartitioned `emp_indx` index:

```
ALTER INDEX emp_indx REBUILD COMPUTE STATISTICS;
```

The type of statistics collected depends on the type of index you are rebuilding.

> **See Also:** *Oracle8i Concepts.*

PARALLEL Example The following statement causes the index to be rebuilt from the existing index by using parallel execution processes to scan the old and to build the new index:

```
ALTER INDEX emp_idx
   REBUILD
   PARALLEL;
```

ALTER JAVA

Purpose

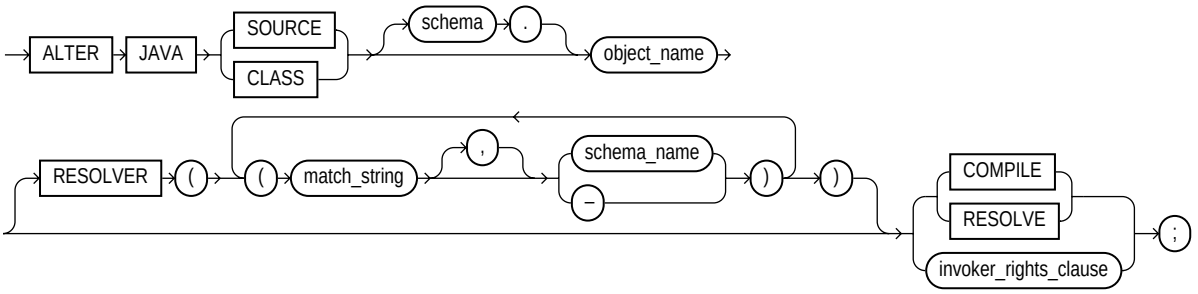
Use the ALTER JAVA statement to force the resolution of a Java class schema object or compilation of a Java source schema object. (You cannot call the methods of a Java class before all its external references to Java names are associated with other classes.)

> **See Also:** *Oracle8i Java Stored Procedures Developer's Guide* for more information on resolving Java classes and compiling Java sources

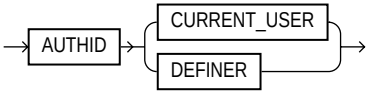
Prerequisites

The Java source or class must be in your own schema, or you must have the ALTER ANY PROCEDURE system privilege. You must also have the EXECUTE object privilege on Java classes.

Syntax



invoker_rights_clause::=



Keywords and Parameters

JAVA SOURCE

Use ALTER JAVA SOURCE to compile a Java source schema object.

JAVA CLASS

Use ALTER JAVA CLASS to resolve a Java class schema object.

object_name

Specify a previously created Java class or source schema object. Use double quotation marks to preserve lower- or mixed-case names.

RESOLVER

The RESOLVER clause lets you specify how schemas are searched for referenced fully specified Java names, using the mapping pairs specified when the Java class or source was created.

See Also: CREATE JAVA on page 9-79

RESOLVE | COMPILE

RESOLVE and COMPILE are synonymous keywords. They let you specify that Oracle should attempt to resolve the primary Java class schema object.

- When applied to a class, resolution of referenced names to other class schema objects occurs.
- When applied to a source, source compilation occurs.

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the methods of the class execute with the privileges and in the schema of the user who defined it or with the privileges and in the schema of CURRENT_USER.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

AUTHID CURRENT_USER	Specify CURRENT_USER if you want the methods of the class to execute with the privileges of CURRENT_USER. This clause is the default and creates an "invoker-rights class." This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the methods reside.
AUTHID DEFINER	Specify DEFINER if you want the methods of the class to execute with the privileges of the user who defined it. This clause also specifies that external names resolve in the schema where the methods reside.

See Also:

- *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined
- *Oracle8i Java Stored Procedures Developer's Guide*

Example

Resolving a Java Class Example The following statement forces the resolution of a Java class:

```
ALTER JAVA CLASS "Agent"
   RESOLVER (("/home/java/bin/*" scott)(* public))
   RESOLVE;
```

ALTER MATERIALIZED VIEW

Purpose

A materialized view is a database object that contains the results of a query of one or more tables. Use the ALTER MATERIALIZED VIEW statement to modify an existing materialized view in one or more of the following ways:

- To change its storage characteristics
- To change its refresh method, mode, or time
- To alter its structure so that it is a different type of materialized view
- To enable or disable query rewrite.

> **Note:** The keyword SNAPSHOT is supported in place of MATERIALIZED VIEW for backward compatibility.

The tables in the query are called **master tables** (a replication term) or **detail tables** (a data warehouse term). This reference uses "master tables" for consistency. The databases containing the master tables are called the **master databases**.

See Also:

- CREATE MATERIALIZED VIEW on page 9-88for more information on creating materialized views
- *Oracle8i Replication* for information on materialized views in a replication environment
- *Oracle8i Data Warehousing Guide* for information on materialized views in a data warehousing environment

Prerequisites

The privileges required to alter a materialized view should be granted directly, as follows:

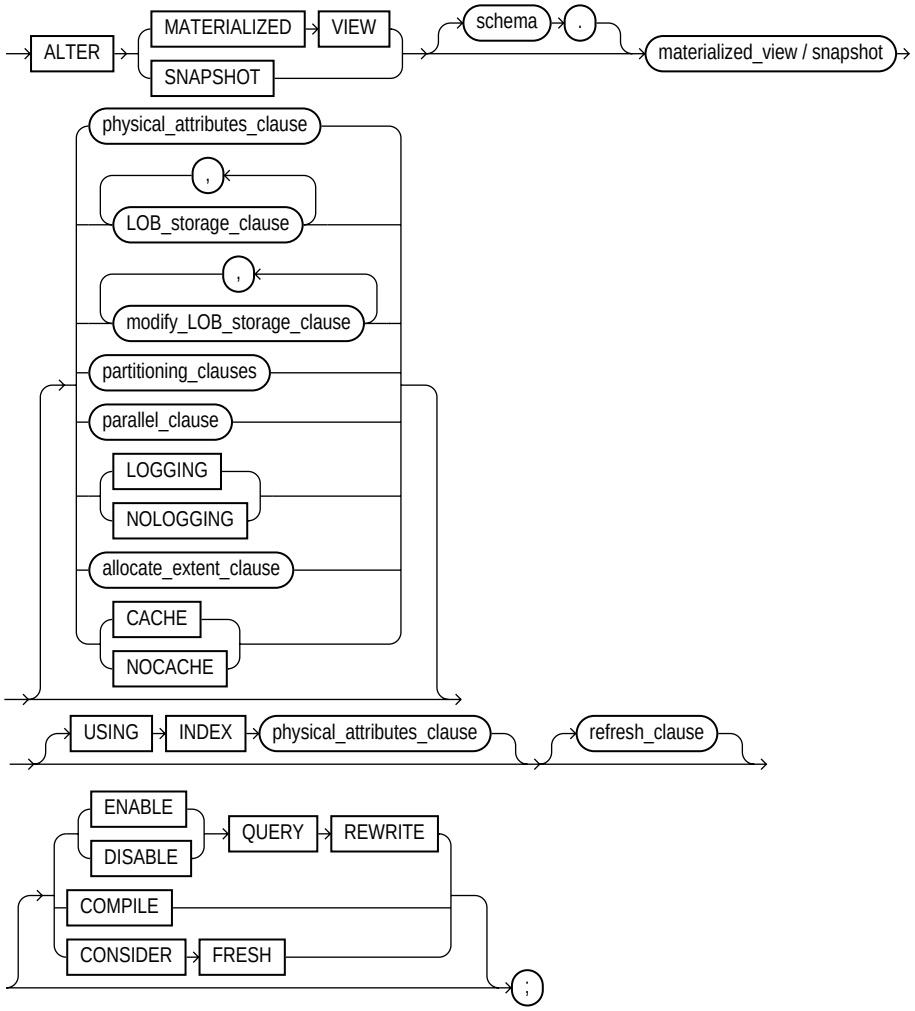
The materialized view must be in your own schema, or you must have the ALTER ANY MATERIALIZED VIEW system privilege.

To enable a materialized view for query rewrite:

- If all of the master tables in the materialized view are in your schema, you must have the QUERY REWRITE privilege.
- If any of the master tables are in another schema, you must have the GLOBAL QUERY REWRITE privilege.
- If the materialized view is in another user's schema, both you and the owner of that schema must have the appropriate QUERY REWRITE privilege, as described in the preceding two items. In addition, the owner of the materialized view must have SELECT access to any master tables that the materialized view owner does not own.

See Also: *Oracle8i Replication* and *Oracle8i Data Warehousing Guide*

Syntax

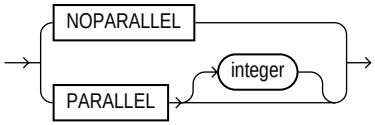

ALTER
MATERIALIZED
VIEW
SNAPSHOT
schema
.
materialized_view / snapshot
physical_attributes_clause
,
LOB_storage_clause
,
modify_LOB_storage_clause
partitioning_clauses
parallel_clause
LOGGING
NOLOGGING
allocate_extent_clause
CACHE
NOCACHE
USING
INDEX
physical_attributes_clause
refresh_clause
ENABLE
DISABLE
QUERY
REWRITE
COMPILE
CONSIDER
FRESH
;

LOB_storage_clause: See ALTER TABLE on page 8-2.

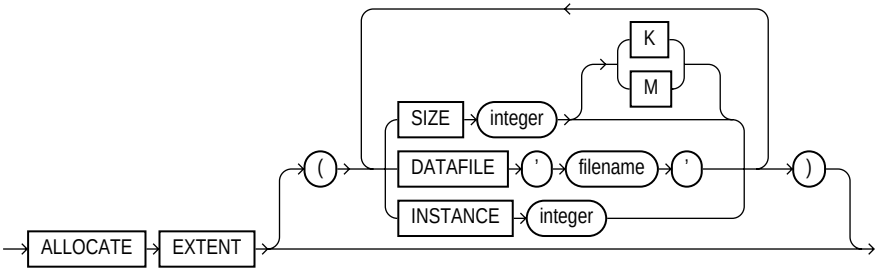
modify_LOB_storage_clause: See ALTER TABLE on page 8-2.

partitioning_clauses: See ALTER TABLE on page 8-2.

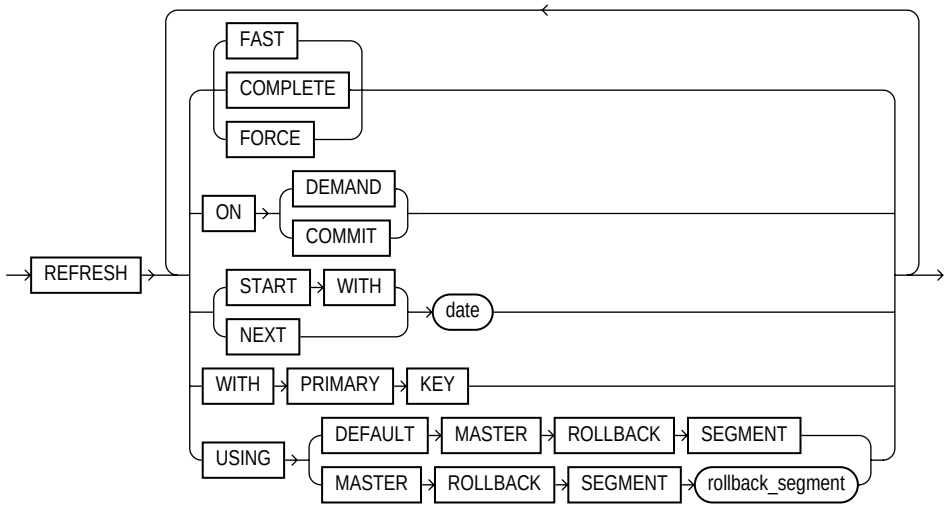
parallel_clause::=



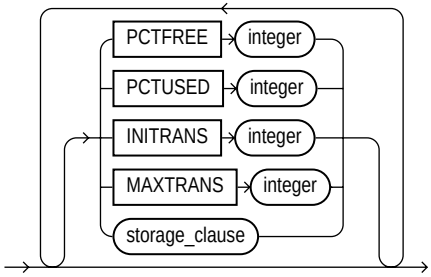
allocate_extent_clause::=



refresh_clause::=



physical_attributes_clause::=



storage_clause: See the storage_clause on page 11-129.

Keywords and Parameters

schema

Specify the schema containing the materialized view. If you omit *schema*, Oracle assumes the materialized view is in your own schema.

materialized_view

Specify the name of the materialized view to be altered.

physical_attributes_clause

Specify new values for the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters (or, when used in the USING INDEX clause, for the INITRANS and MAXTRANS parameters only) and the storage characteristics for the materialized view.

See Also:

- ALTER TABLE on page 8-2 for information on the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters
- *storage_clause* on page 11-129 for information about storage characteristics

LOB_storage_clause

The *LOB_storage_clause* lets you specify the LOB storage characteristics.

> **See Also:** ALTER TABLE on page 8-2 for information about specifying the parameters of this clause

modify_LOB_storage_clause

The *modify_LOB_storage_clause* lets you modify the physical attributes of the LOB attribute *lob_item* or LOB object attribute.

> **See Also:** ALTER TABLE on page 8-2 for information about specifying the parameters of this clause

partitioning_clauses

The syntax and general functioning of the partitioning clauses for materialized views is the same as for partitioned tables.

> **See Also:** ALTER TABLE on page 8-2

Restrictions:

- You cannot use the *LOB_storage_clause* or *modify_LOB_storage_clause* when modifying a materialized view.
- If you attempt to drop, truncate, or exchange a materialized view partition, Oracle raises an error.

> **Note:** If you wish to keep the contents of the materialized view synchronized with those of the master table, Oracle Corporation recommends that you manually perform a complete refresh of all materialized views dependent on the table after dropping or truncating a table partition.

`MODIFY PARTITION UNUSABLE LOCAL INDEXES`

Use this clause to mark `UNUSABLE` all the local index partitions associated with *partition*.

`MODIFY PARTITION REBUILD UNUSABLE LOCAL INDEXES`

Use this clause to rebuild the unusable local index partitions associated with *partition*.

parallel_clause

The *parallel_clause* lets you change the default degree of parallelism for the materialized view.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL	Specify NOPARALLEL for serial execution. This is the default.
PARALLEL	Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.
PARALLEL *integer*	Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

See Also: "Notes on the parallel_clause" for CREATE TABLE on page 10-41

LOGGING | NOLOGGING

Specify or change the logging characteristics of the materialized view.

See Also: ALTER TABLE on page 8-2 for information about logging characteristics

allocate_extent_clause

The *allocate_extent_clause* lets you explicitly allocate a new extent for the materialized view.

See Also: ALTER TABLE on page 8-2

CACHE | NOCACHE

For data that will be accessed frequently, CACHE specifies that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. NOCACHE specifies that the blocks are placed at the least recently used end of the LRU list.

> **See Also:** ALTER TABLE on page 8-2 for information about specifying CACHE or NOCACHE

USING INDEX

Use this clause to change the value of INITRANS, MAXTRANS, and STORAGE parameters for the index Oracle uses to maintain the materialized view's data.

Restriction: You cannot specify the PCTUSED or PCTFREE parameters in this clause.

refresh_clause

Use the *refresh_clause* to change the default method and mode and the default times for automatic refreshes. If the contents of a materialized view's master tables are modified, the data in the materialized view must be updated to make the materialized view accurately reflect the data currently in its master table(s). This clause lets you schedule the times and specify the method and mode for Oracle to refresh the materialized view.

Note: This clause only sets the default refresh options. For instructions on actually implementing the refresh, refer to *Oracle8i Replication* and *Oracle8i Data Warehousing Guide*.

FAST — Specify FAST for incremental refresh method, which performs the refresh according to the changes that have occurred to the master tables. The changes are stored either in the materialized view log associated with the master table (for conventional DML changes) or in the direct loader log (for direct-load INSERT operations).

For both conventional DML changes and for direct-path loads, other conditions may restrict the eligibility of a materialized view for fast refresh.

See Also:

- *Oracle8i Replication* for restrictions on fast refresh in replication environments

- *Oracle8i Data Warehousing Guide* for restrictions on fast refresh in data warehouse environments

Restrictions:

- When you specify FAST refresh at create time, Oracle verifies that the materialized view you are creating is eligible for fast refresh. When you change the refresh method to FAST in an ALTER MATERIALIZED VIEW statement, Oracle does not perform this verification. If the materialized view is not eligible for fast refresh, Oracle will return an error when you attempt to refresh this view.

- Materialized views are not eligible for fast refresh if the defining query contains an analytic function.

  See Also: "Analytic Functions" on page 4-8

COMPLETE — Specify COMPLETE for the complete refresh method, which is implemented by executing the materialized view's defining query. If you request a complete refresh, Oracle performs a complete refresh even if a fast refresh is possible.

FORCE — Specify FORCE if, when a refresh occurs, you want Oracle to perform a fast refresh if one is possible or a complete refresh otherwise.

ON COMMIT — Specify ON COMMIT if you want a fast refresh to occur whenever Oracle commits a transaction that operates on a master table of the materialized view.

Restriction: This clause is supported only for materialized join views and single-table materialized aggregate views.

See Also: *Oracle8i Replication* and *Oracle8i Data Warehousing Guide*

ON DEMAND — Specify ON DEMAND if you want the materialized view to be refreshed on demand by calling one of the three DBMS_MVIEW refresh procedures. If you omit both ON COMMIT and ON DEMAND, ON DEMAND is the default.

See Also:

- *Oracle8i Supplied PL/SQL Packages Reference* for information on these procedures

- *Oracle8i Data Warehousing Guide* on the types of materialized views you can create by specifying REFRESH ON DEMAND

Note: If you specify ON COMMIT or ON DEMAND, you cannot also specify START WITH or NEXT.

START WITH — Specify START WITH *date* to indicate a date for the first automatic refresh time.

NEXT — Specify NEXT to indicate a date expression for calculating the interval between automatic refreshes.

Both the START WITH and NEXT values must evaluate to a time in the future. If you omit the START WITH value, Oracle determines the first automatic refresh time by evaluating the NEXT expression with respect to the creation time of the materialized view. If you specify a START WITH value but omit the NEXT value, Oracle refreshes the materialized view only once. If you omit both the START WITH and NEXT values, or if you omit the *refresh_clause* entirely, Oracle does not automatically refresh the materialized view.

WITH PRIMARY KEY — Specify WITH PRIMARY KEY to change a rowid materialized view to a primary key materialized view. Primary key materialized views allow materialized view master tables to be reorganized without affecting the materialized view's ability to continue to fast refresh. The master table must contain an enabled primary key constraint.

See Also: *Oracle8i Replication* for detailed information about primary key materialized views

USING ROLLBACK SEGMENT — Specify USING ROLLBACK SEGMENT to change the remote rollback segment to be used during materialized view refresh, where *rollback_segment* is the name of the rollback segment to be used.

See Also: *Oracle8i Replication* for information on changing the local materialized view rollback segment using the DBMS_REFRESH package

DEFAULT — Specify DEFAULT if you want Oracle to choose the rollback segment to use. If you specify DEFAULT, you cannot specify *rollback_segment*.

MASTER ... *rollback_segment* — Specify the remote rollback segment to be used at the remote master for the individual materialized view. (To change the local materialized view rollback segment, use the DBMS_REFRESH package, described in *Oracle8i Replication*.)

The master rollback segment is stored on a per-materialized-view basis and is validated during materialized view creation and refresh. If the materialized view is complex, the master rollback segment, if specified, is ignored.

QUERY REWRITE

Use this clause to determine whether the materialized view is eligible to be used for query rewrite.

ENABLE — Specify ENABLE to enable the materialized view for query rewrite.

See Also: *Oracle8i Data Warehousing Guide* for more information on query rewrite.

Restrictions:

- If the materialized view is in an invalid or unusable state, it is not eligible for query rewrite in spite of the ENABLE mode.
- You cannot enable query rewrite if the materialized view was created totally or in part from a view.
- You can enable query rewrite only if all user-defined functions in the materialized view are DETERMINISTIC.

  See Also: CREATE FUNCTION on page 9-43

- You can enable query rewrite only if expressions in the statement are repeatable. For example, you cannot include CURRENT_TIME or USER.

  See Also: *Oracle8i Data Warehousing Guide*

DISABLE — Specify DISABLE if you do not want the materialized view to be eligible for use by query rewrite. (If a materialized view is in the invalid state, it is not eligible for use by query rewrite, whether or not it is disabled.) However, a disabled materialized view can be refreshed.

COMPILE

Specify COMPILE to explicitly revalidate a materialized view. If an object upon which the materialized view depends is dropped or altered, the materialized view remains accessible, but it is invalid for query rewrite. You can use this clause to explicitly revalidate the materialized view to make it eligible for query rewrite.

If the materialized view fails to revalidate, it cannot be refreshed or used for query rewrite.

CONSIDER FRESH

CONSIDER FRESH directs Oracle to consider the materialized view fresh and therefore eligible for query rewrite in the TRUSTED or STALE_TOLERATED modes. Because Oracle cannot guarantee the freshness of the materialized view, query rewrite in ENFORCED mode is not supported. This clause also sets the staleness state of the materialized view to UNKNOWN. The staleness state is displayed in the STALENESS column of the ALL_MVIEWS, DBA_MVIEWS, and USER_MVIEWS data dictionary views.

This clause is useful after performing partition maintenance operations against the master table. Such operations would otherwise render the materialized view

ineligible for fast refresh, and eligible for query rewrite only in STALE_TOLERATED mode.

> **Note:** A materialized view is stale if changes have been made to the contents of any of its master tables. This clause directs Oracle to assume that the materialized view is fresh and that no such changes have been made. Therefore, actual updates to those tables pending refresh are purged with respect to the materialized view.

> **See Also:** *Oracle8i Data Warehousing Guide* for more information on query rewrite and the implications of performing partition maintenance operations on master tables

Examples

Automatic Refresh Example The following statement changes the default refresh method for the hq_emp materialized view to FAST:

```
CREATE MATERIALIZED VIEW hq_emp
   REFRESH COMPLETE
   START WTIH SYSDATE NEXT SYSDATE +1/4096
   AS SELECT * FROM hq_emp;

ALTER MATERIALIZED VIEW hq_emp
   REFRESH FAST;
```

The next automatic refresh of the materialized view will be a fast refresh provided it is a simple materialized view and its master table has a materialized view log that was created before the materialized view was created or last refreshed.

Because the REFRESH clause does not specify START WITH or NEXT values, the refresh intervals established by the REFRESH clause when the hq_emp materialized view was created or last altered are still used.

NEXT Example The following statement stores a new interval between automatic refreshes for the branch_emp materialized view:

```
ALTER MATERIALIZED VIEW branch_emp
   REFRESH NEXT SYSDATE+7;
```

Because the REFRESH clause does not specify a START WITH value, the next automatic refresh occurs at the time established by the START WITH and NEXT

values specified when the branch_emp materialized view was created or last altered.

At the time of the next automatic refresh, Oracle refreshes the materialized view, evaluates the NEXT expression SYSDATE+7 to determine the next automatic refresh time, and continues to refresh the materialized view automatically once a week.

Because the REFRESH clause does not explicitly specify a refresh method, Oracle continues to use the refresh method specified by the REFRESH clause of the CREATE MATERIALIZED VIEW or most recent ALTER MATERIALIZED VIEW statement.

Complete Refresh Example The following statement specifies a new refresh method, a new next refresh time, and a new interval between automatic refreshes of the sf_emp materialized view:

```
ALTER MATERIALIZED VIEW sf_emp
   REFRESH COMPLETE
   START WITH TRUNC(SYSDATE+1) + 9/24
   NEXT SYSDATE+7;
```

The START WITH value establishes the next automatic refresh for the materialized view to be 9:00 a.m. tomorrow. At that point, Oracle performs a complete refresh of the materialized view, evaluates the NEXT expression, and subsequently refreshes the materialized view every week.

Enabling Query Rewrite Example The following statement enables query rewrite on the materialized view mv1 and implicitly revalidates it.

```
ALTER MATERIALIZED VIEW mv1
   ENABLE QUERY REWRITE;
```

Rollback Segment Examples The following statement changes the remote master rollback segment used during materialized view refresh to master_seg:

```
ALTER MATERIALIZED VIEW inventory
   REFRESH USING MASTER ROLLBACK SEGMENT master_seg;
```

The following statement changes the remote master rollback segment used during materialized view refresh to one chosen by Oracle:

```
ALTER MATERIALIZED VIEW sales
   REFRESH USING DEFAULT MASTER ROLLBACK SEGMENT;
```

Primary Key Example The following statement changes a rowid materialized view to a primary key materialized view:

```
ALTER MATERIALIZED VIEW emp_rs
   REFRESH WITH PRIMARY KEY;
```

COMPILE Example The following statement revalidates the materialized view `store_mv`:

```
ALTER MATERIALIZED VIEW store_mv COMPILE;
```

Modifying Refresh Method Example The following statement changes the refresh method of materialized view `store_mv` to FAST;

```
ALTER MATERIALIZED VIEW store_mv REFRESH FAST;
```

CONSIDER FRESH Example The following statement instructs Oracle that materialized view `mv1` should be considered fresh. This statement allows `mv1` to be eligible for query rewrite in TRUSTED mode even after you have performed partition maintenance operations on the master tables of `mv1`:

```
ALTER MATERIALIZED VIEW mv1 CONSIDER FRESH;
```

ALTER MATERIALIZED VIEW LOG

Purpose

Use the ALTER MATERIALIZED VIEW LOG statement to alter the storage characteristics, refresh mode or time, or type of an existing materialized view log. A **materialized view log** is a table associated with the master table of a materialized view.

Note: The keyword SNAPSHOT is supported in place of MATERIALIZED VIEW for backward compatibility.

See Also:

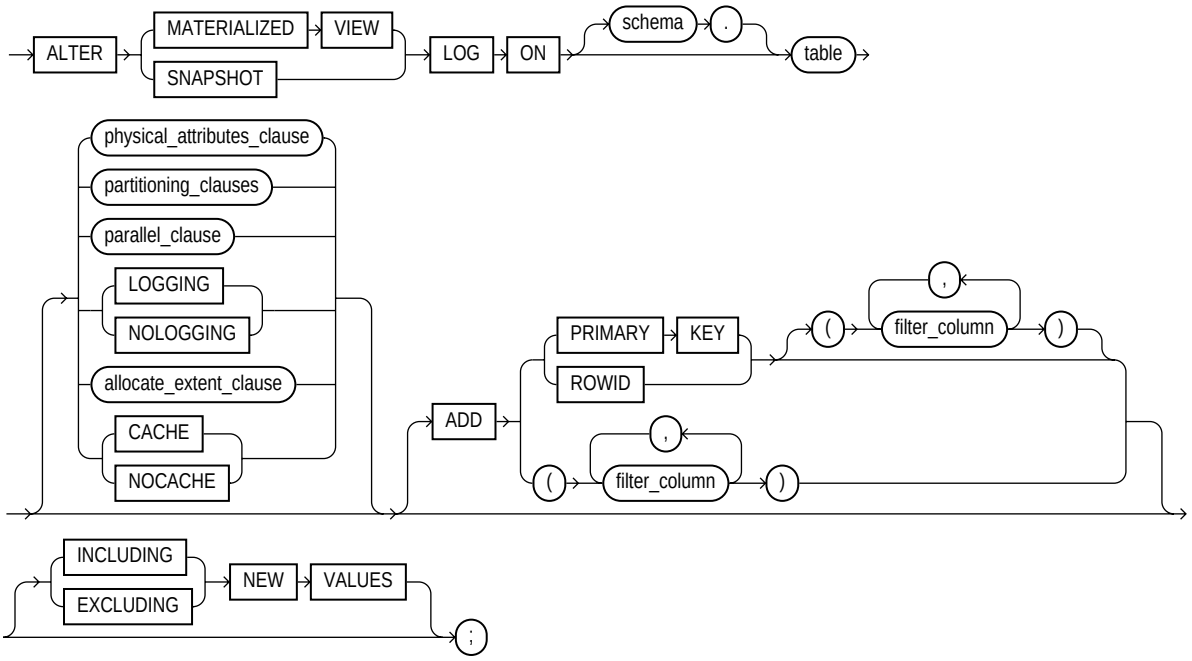
- ALTER MATERIALIZED VIEW on page 7-61 for more information on materialized views, including refreshing them
- CREATE MATERIALIZED VIEW on page 9-88 for a description of the various types of materialized views

Prerequisites

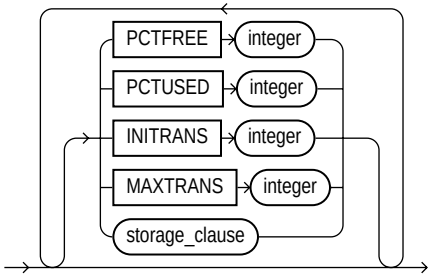
Only the owner of the master table or a user with the SELECT privilege for the master table can alter a materialized view log.

See Also: *Oracle8i Replication* for detailed information about the prerequisites for ALTER MATERIALIZED VIEW LOG

Syntax



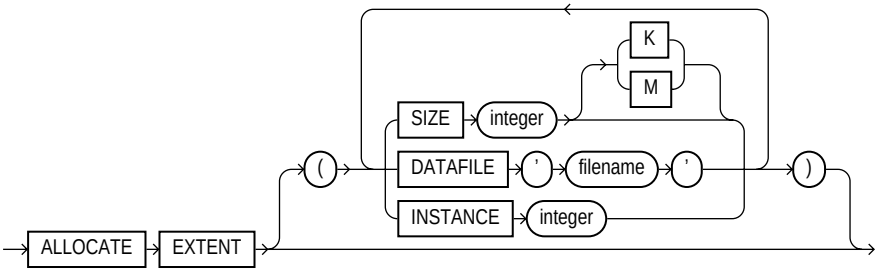
physical_attributes_clause::=



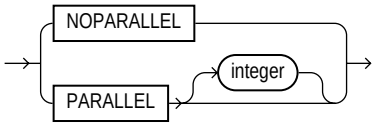
storage_clause: See storage_clause on page 11-129.

partitioning_clauses: See ALTER TABLE on page 8-2.

allocate_extent_clause::=



parallel_clause::=



Keywords and Parameters

schema

Specify the schema containing the master table. If you omit *schema*, Oracle assumes the materialized view log is in your own schema.

table

Specify the name of the master table associated with the materialized view log to be altered.

physical_attributes_clause

The *physical_attributes_clause* lets you change the value of PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters for the table, the partition, the overflow data segment, or the default characteristics of a partitioned table.

See Also: CREATE TABLE on page 10-7 and the "Materialized View Storage Example" on page 7-81 for a description of these parameters

partitioning_clauses

The syntax and general functioning of the partitioning clauses is the same as for the ALTER TABLE statement

Restrictions:

- You cannot use the *LOB_storage_clause* or *modify_LOB_storage_clause* when modifying a materialized view log.
- If you attempt to drop, truncate, or exchange a materialized view log partition, Oracle raises an error.

See Also: ALTER TABLE on page 8-2

parallel_clause

The *parallel_clause* lets you specify whether parallel operations will be supported for the materialized view log.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL	Specify NOPARALLEL for serial execution. This is the default.
PARALLEL	Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.
PARALLEL *integer*	Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 10-41

LOGGING | NOLOGGING

Specify the logging attribute of the materialized view log.

> **See Also:** ALTER TABLE on page 8-2 for information about specifying this attribute

allocate_extent_clause

The *allocate_extent_clause* lets you explicitly allocate a new extent for the materialized view log.

> **See Also:** ALTER TABLE on page 8-2

CACHE | NOCACHE

For data that will be accessed frequently, CACHE specifies that the blocks retrieved for this log are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. NOCACHE specifies that the blocks are placed at the least recently used end of the LRU list.

> **See Also:** ALTER TABLE on page 8-2 for information about specifying CACHE or NOCACHE

ADD

The ADD clause lets you augment the materialized view log so that it records the primary key values or rowid values when rows in the materialized view master table are updated. This clause can also be used to record additional filter columns.

To stop recording any of this information, you must first drop the materialized view log and then re-create it. Dropping the materialized view log and then re-creating it forces each of the existing materialized views that depend on the master table to complete refresh on its next refresh.

PRIMARY KEY	Specify PRIMARY KEY if you want the primary-key values of all rows that are updated to be recorded in the materialized view log.
ROWID	Specify ROWID if you want the rowid values of all rows that are updated to be recorded in the materialized view log.

filter_column(s) Specify the columns whose values you want to be recorded in the materialized view log for all rows that are updated. Filter columns are non-primary-key columns referenced by materialized views.

Restriction: You can specify only one PRIMARY KEY, one ROWID, and one filter column list per materialized view log. Therefore, if any of these three values were specified at create time (either implicitly or explicitly), you cannot specify those values in this ALTER statement.

See Also:

- CREATE MATERIALIZED VIEW on page 9-88 for information on explicit and implicit inclusion of materialized view log values
- *Oracle8i Replication* for more information about filter columns

NEW VALUES

The NEW VALUES clause lets you specify whether Oracle saves both old and new values in the materialized view log. The value you set in this clause applies to all columns in the log, not only to primary key, rowid, or filter columns you may have added in this statement.

INCLUDING Specify INCLUDING to save both new and old values in the log. If this log is for a table on which you have a single-table materialized aggregate view, and if you want the materialized view to be eligible for fast refresh, you must specify INCLUDING.

EXCLUDING Specify EXCLUDING to disable the recording of new values in the log. You can use this clause to avoid the overhead of recording new values. However, do not use this clause if you have a fast-refreshable single-table materialized aggregate view defined on this table.

Examples

Materialized View Storage Example The following statement changes the MAXEXTENTS value of a materialized view log:

```
ALTER MATERIALIZED VIEW LOG ON dept
   STORAGE MAXEXTENTS 50;
```

PRIMARY KEY Example The following statement alters an existing rowid materialized view log to also record primary key information:

```
ALTER MATERIALIZED VIEW LOG ON sales
   ADD PRIMARY KEY;
```

ALTER OUTLINE

Purpose

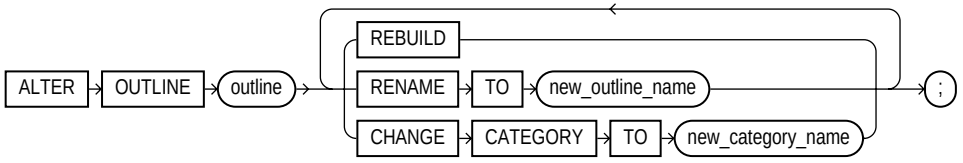
Use the ALTER OUTLINE statement to rename a stored outline, reassign it to a different category, or regenerate it by compiling the outline's SQL statement and replacing the old outline data with the outline created under current conditions.

> **See Also:** CREATE OUTLINE on page 9-119 and *Oracle8i Performance Guide and Reference* for more information on outlines

Prerequisites

To modify an outline, you must have the ALTER ANY OUTLINE system privilege.

Syntax



Keywords and Parameters

outline

Specify the name of the outline to be modified.

REBUILD

Specify REBUILD to regenerate the execution plan for *outline* using current conditions.

RENAME TO *new_outline_name*

Use the RENAME TO clause to specify an outline name to replace *outline*.

CHANGE CATEGORY TO *new_category_name*

Use the CHANGE CATEGORY TO clause to specify the name of the category into which the *outline* will be moved.

Example

ALTER OUTLINE Example The following statement regenerates a stored outline called `salaries` by compiling the outline's text and replacing the old outline data with the outline created under current conditions.

```
ALTER OUTLINE salaries REBUILD;
```

ALTER PACKAGE

Purpose

Use the ALTER PACKAGE statement to explicitly recompile a package specification, body, or both. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

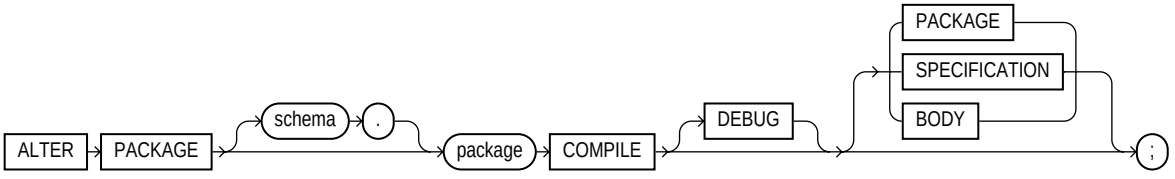
Because all objects in a package are stored as a unit, the ALTER PACKAGE statement recompiles all package objects together. You cannot use the ALTER PROCEDURE statement or ALTER FUNCTION statement to recompile individually a procedure or function that is part of a package.

> **Note:** This statement does not change the declaration or definition of an existing package. To redeclare or redefine a package, use the CREATE PACKAGE or the CREATE PACKAGE BODY on page 9-122 statement with the OR REPLACE clause.

Prerequisites

For you to modify a package, the package must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the package. If you omit *schema*, Oracle assumes the package is in your own schema.

package

Specify the name of the package to be recompiled.

COMPILE

You must specify COMPILE to recompile the package specification or body. The COMPILE keyword is required.

If recompiling the package results in compilation errors, Oracle returns an error and the body remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

SPECIFICATION

Specify SPECIFICATION to recompile only the package specification, regardless of whether it is invalid. You might want to recompile a package specification to check for compilation errors after modifying the specification.

When you recompile a package specification, Oracle invalidates any local objects that depend on the specification, such as procedures that call procedures or functions in the package. The body of a package also depends on its specification. If you subsequently reference one of these dependent objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.

BODY

Specify BODY to recompile only the package body regardless of whether it is invalid. You might want to recompile a package body after modifying it. Recompiling a package body does not invalidate objects that depend upon the package specification.

When you recompile a package body, Oracle first recompiles the objects on which the body depends, if any of those objects are invalid. If Oracle recompiles the body successfully, the body becomes valid.

PACKAGE

Specify PACKAGE to recompile both the package specification and the package body if one exists, regardless of whether they are invalid. This is the default. The recompilation of the package specification and body lead to the invalidation and recompilation as described above for SPECIFICATION and BODY.

> **See Also:** *Oracle8i Concepts* for information on how Oracle maintains dependencies among schema objects, including remote objects

DEBUG

Specify DEBUG to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

> **See Also:** *Oracle8i Supplied PL/SQL Packages Reference* for information on debugging packages

Examples

Recompiling a Package Examples This statement explicitly recompiles the specification and body of the accounting package in the schema blair:

```
ALTER PACKAGE blair.accounting
   COMPILE PACKAGE;
```

If Oracle encounters no compilation errors while recompiling the accounting specification and body, accounting becomes valid. Blair can subsequently call or reference all package objects declared in the specification of accounting without run-time recompilation. If recompiling accounting results in compilation errors, Oracle returns an error and accounting remains invalid.

Oracle also invalidates all objects that depend upon accounting. If you subsequently reference one of these objects without explicitly recompiling it first, Oracle recompiles it implicitly at run time.

To recompile the body of the accounting package in the schema blair, issue the following statement:

```
ALTER PACKAGE blair.accounting
   COMPILE BODY;
```

If Oracle encounters no compilation errors while recompiling the package body, the body becomes valid. Blair can subsequently call or reference all package objects declared in the specification of accounting without run-time recompilation. If recompiling the body results in compilation errors, Oracle returns an error message and the body remains invalid.

Because this statement recompiles the body and not the specification of accounting, Oracle does not invalidate dependent objects.

ALTER PROCEDURE

Purpose

Use the ALTER PROCEDURE statement to explicitly recompile a stand-alone stored procedure. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

To recompile a procedure that is part of a package, recompile the entire package using the ALTER PACKAGE statement (see ALTER PACKAGE on page 7-85).

> **Note:** This statement does not change the declaration or definition of an existing procedure. To redeclare or redefine a procedure, use the CREATE PROCEDURE statement with the OR REPLACE clause (see CREATE PROCEDURE on page 9-132)

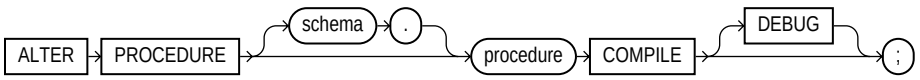
The ALTER PROCEDURE statement is quite similar to the ALTER FUNCTION statement.

> **See Also:** ALTER FUNCTION on page 7-38

Prerequisites

The procedure must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the procedure. If you omit *schema*, Oracle assumes the procedure is in your own schema.

procedure

Specify the name of the procedure to be recompiled.

COMPILE

Specify COMPILE to recompile the procedure. The COMPILE keyword is required. Oracle recompiles the procedure regardless of whether it is valid or invalid.

- Oracle first recompiles objects upon which the procedure depends, if any of those objects are invalid.
- Oracle also invalidates any local objects that depend upon the procedure, such as procedures that call the recompiled procedure or package bodies that define procedures that call the recompiled procedure.
- If Oracle recompiles the procedure successfully, the procedure becomes valid. If recompiling the procedure results in compilation errors, then Oracle returns an error and the procedure remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

> **See Also:** *Oracle8i Concepts* for information on how Oracle maintains dependencies among schema objects, including remote objects

DEBUG

Specify DEBUG to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* for information on debugging procedures

Example

Recompiling a Procedure Example To explicitly recompile the procedure close_acct owned by the user henry, issue the following statement:

```
ALTER PROCEDURE henry.close_acct
   COMPILE;
```

If Oracle encounters no compilation errors while recompiling close_acct, close_acct becomes valid. Oracle can subsequently execute it without recompiling it at run time. If recompiling close_acct results in compilation errors, Oracle returns an error and close_acct remains invalid.

Oracle also invalidates all dependent objects. These objects include any procedures, functions, and package bodies that call `close_acct`. If you subsequently reference one of these objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.

ALTER PROFILE

Purpose

Use the ALTER PROFILE statement to add, modify, or remove a resource limit or password management parameter in a profile.

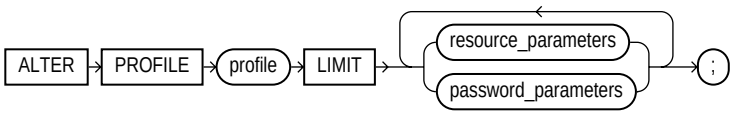
Changes made to a profile with an ALTER PROFILE statement affect users only in their subsequent sessions, not in their current sessions.

> **See Also:** CREATE PROFILE on page 9-139 for information on creating a profile

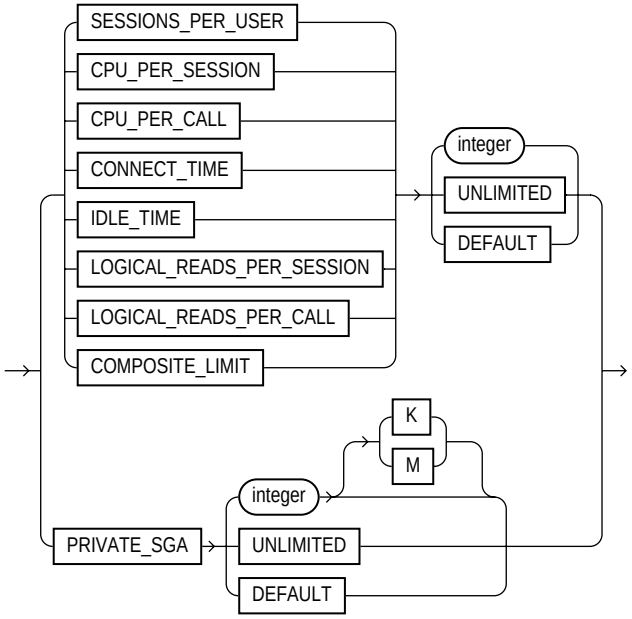
Prerequisites

You must have ALTER PROFILE system privilege to change profile resource limits. To modify password limits and protection, you must have ALTER PROFILE and ALTER USER system privileges.

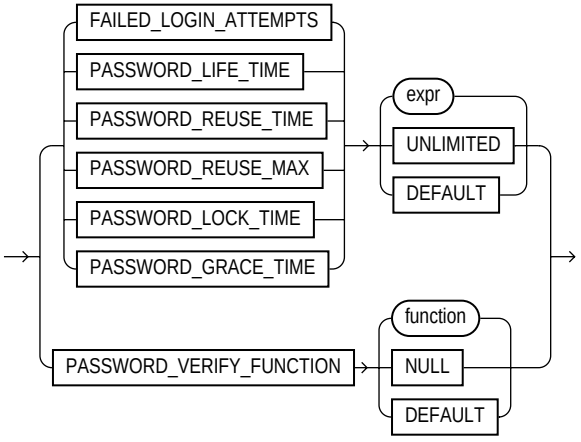
Syntax



resource_parameters::=



password_parameters::=



Keywords and Parameters

The keywords and parameters in the ALTER PROFILE statement all have the same meaning as in the CREATE PROFILE statement.

Note: You cannot remove a limit from the DEFAULT profile.

See Also: CREATE PROFILE on page 9-139

Examples

Making a Password Unavailable Example The following statement makes a password unavailable for reuse for 90 days:

```
ALTER PROFILE prof
   LIMIT PASSWORD_REUSE_TIME 90
   PASSWORD_REUSE_MAX UNLIMITED;
```

Setting Default Values Example The following statement defaults the PASSWORD_REUSE_TIME value to its defined value in the DEFAULT profile:

```
ALTER PROFILE prof
   LIMIT PASSWORD_REUSE_TIME DEFAULT
   PASSWORD_REUSE_MAX UNLIMITED;
```

Limiting Login Attempts and Password Lock Time Example The following statement alters profile prof with FAILED_LOGIN_ATTEMPTS set to 5 and PASSWORD_LOCK_TIME set to 1:

```
ALTER PROFILE prof LIMIT
   FAILED_LOGIN_ATTEMPTS 5
   PASSWORD_LOCK_TIME 1;
```

This statement causes prof's account to become locked for 1 day after 5 unsuccessful login attempts.

Changing Password Lifetime and Grace Period Example The following statement modifies profile prof's PASSWORD_LIFE_TIME to 60 days and PASSWORD_GRACE_TIME to 10 days:

```
ALTER PROFILE prof LIMIT
   PASSWORD_LIFE_TIME 60
   PASSWORD_GRACE_TIME 10;
```

Limiting Concurrent Sessions Example This statement defines a new limit of 5 concurrent sessions for the engineer profile:

```
ALTER PROFILE engineer LIMIT SESSIONS_PER_USER 5;
```

If the engineer profile does not currently define a limit for SESSIONS_PER_USER, the above statement adds the limit of 5 to the profile. If the profile already defines a limit, the above statement redefines it to 5. Any user assigned the engineer profile is subsequently limited to 5 concurrent sessions.

Removing Limits Example This statement removes the IDLE_TIME limit from the engineer profile:

```
ALTER PROFILE engineer LIMIT IDLE_TIME DEFAULT;
```

Any user assigned the engineer profile is subject in their subsequent sessions to the IDLE_TIME limit defined in the DEFAULT profile.

Limiting Idle Time Example This statement defines a limit of 2 minutes of idle time for the DEFAULT profile:

```
ALTER PROFILE default LIMIT IDLE_TIME  2;
```

This IDLE_TIME limit applies to these users:

- Users who are not explicitly assigned any profile
- Users who are explicitly assigned a profile that does not define an IDLE_TIME limit

This statement defines unlimited idle time for the engineer profile:

```
ALTER PROFILE engineer LIMIT IDLE_TIME UNLIMITED;
```

Any user assigned the engineer profile is subsequently permitted unlimited idle time.

ALTER RESOURCE COST

Purpose

Use the ALTER RESOURCE COST statement to specify or change the formula by which Oracle calculates the total resource cost used in a session. The weight that you assign to each resource determines how much the use of that resource contributes to the total resource cost. If you do not assign a weight to a resource, the weight defaults to 0, and use of the resource subsequently does not contribute to the cost. The weights you assign apply to all subsequent sessions in the database.

Oracle calculates the total resource cost by first multiplying the amount of each resource used in the session by the resource's weight, and then summing the products for all four resources. For any session, this cost is limited by the value of the COMPOSITE_LIMIT parameter in the user's profile. Both the products and the total cost are expressed in units called **service units**.

Although Oracle monitors the use of other resources, only the four resources shown in the syntax can contribute to the total resource cost for a session.

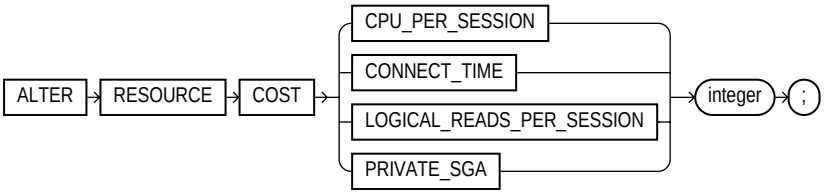
Once you have specified a formula for the total resource cost, you can limit this cost for a session with the COMPOSITE_LIMIT parameter of the CREATE PROFILE statement. If a session's cost exceeds the limit, Oracle aborts the session and returns an error. If you use the ALTER RESOURCE COST statement to change the weight assigned to each resource, Oracle uses these new weights to calculate the total resource cost for all current and subsequent sessions.

> **See Also:** CREATE PROFILE on page 9-139 for information on all resources and on establishing resource limits

Prerequisites

You must have ALTER RESOURCE COST system privilege.

Syntax



Keywords and Parameters

CPU_PER_SESSION

Specify the amount of CPU time that can be used by a session measured in hundredth of seconds.

CONNECT_TIME

Specify the elapsed time allowed for a session measured in minutes.

LOGICAL_READS_PER_SESSION

Specify the number of data blocks that can be read during a session, including blocks read from both memory and disk.

PRIVATE_SGA

Specify the number of bytes of private space in the system global area (SGA) that can be used by a session. This limit applies only if you are using the multi-threaded server architecture and allocating private space in the SGA for your session.

integer

Specify the weight of each resource.

Example

Altering Resource Costs Example The following statement assigns weights to the resources `CPU_PER_SESSION` and `CONNECT_TIME`:

```
ALTER RESOURCE COST
   CPU_PER_SESSION 100
   CONNECT_TIME      1;
```

The weights establish this cost formula for a session:

```
cost = (100 * CPU_PER_SESSION) + (1 * CONNECT_TIME)
```

where the values of CPU_PER_SESSION and CONNECT_TIME are either values in the DEFAULT profile or in the profile of the user of the session.

Because the above statement assigns no weight to the resources LOGICAL_READS_PER_SESSION and PRIVATE_SGA, these resources do not appear in the formula.

If a user is assigned a profile with a COMPOSITE_LIMIT value of 500, a session exceeds this limit whenever cost exceeds 500. For example, a session using 0.04 seconds of CPU time and 101 minutes of elapsed time exceeds the limit. A session 0.0301 seconds of CPU time and 200 minutes of elapsed time also exceeds the limit.

You can subsequently change the weights with another ALTER RESOURCE statement:

```
ALTER RESOURCE COST
   LOGICAL_READS_PER_SESSION 2
   CONNECT_TIME 0;
```

These new weights establish a new cost formula:

```
cost = (100 * CPU_PER_SESSION) + (2 * LOGICAL_READ_PER_SECOND)
```

where the values of CPU_PER_SESSION and LOGICAL_READS_PER_SECOND are either the values in the DEFAULT profile or in the profile of the user of this session.

This ALTER RESOURCE COST statement changes the formula in these ways:

- The statement omits a weight for the CPU_PER_SESSION resource and the resource was already assigned a weight, so the resource remains in the formula with its original weight.
- The statement assigns a weight to the LOGICAL_READS_PER_SESSION resource, so this resource now appears in the formula.
- The statement assigns a weight of 0 to the CONNECT_TIME resource, so this resource no longer appears in the formula.
- The statement omits a weight for the PRIVATE_SGA resource and the resource was not already assigned a weight, so the resource still does not appear in the formula.

ALTER ROLE

Purpose

Use the ALTER ROLE statement to change the authorization needed to enable a role.

See Also:

- CREATE ROLE on page 9-146 for information on creating a role
- SET ROLE on page 11-122 for information on enabling or disabling a role for your session

Prerequisites

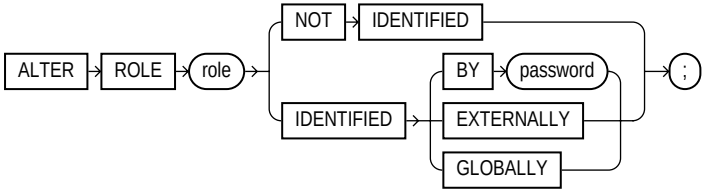
You must either have been granted the role with the ADMIN OPTION or have ALTER ANY ROLE system privilege.

Before you alter a role to IDENTIFIED GLOBALLY, you must:

- Revoke all grants of roles identified externally to the role and
- Revoke the grant of the role from all users, roles, and PUBLIC.

The one exception to this rule is that you should not revoke the role from the user who is currently altering the role.

Syntax



Keywords and Parameters

The keywords and parameters in the ALTER ROLE statement all have the same meaning as in the CREATE ROLE statement.

Note: If you have the ALTER ANY ROLE system privilege and you change a role that is IDENTIFIED GLOBALLY to IDENTIFIED BY *password*, IDENTIFIED EXTERNALLY, or NOT IDENTIFIED, then Oracle grants you the altered role with the ADMIN OPTION, as it would have if you had created the role identified nonglobally.

See Also: CREATE ROLE on page 9-146

Examples

The following statement changes the role `analyst` to `IDENTIFIED GLOBALLY`:

```
ALTER ROLE analyst IDENTIFIED GLOBALLY;
```

This statement changes the password on the `teller` role to `letter`:

```
ALTER ROLE teller
   IDENTIFIED BY letter;
```

Users granted the `teller` role must subsequently enter the new password "letter" to enable the role.

ALTER ROLLBACK SEGMENT

Purpose

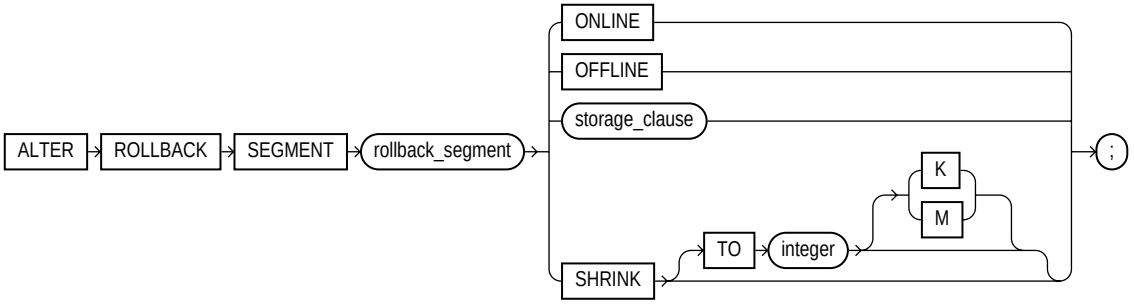
Use the ALTER ROLLBACK SEGMENT statement to bring a rollback segment online or offline, to change its storage characteristics, or to shrink it to an optimal or specified size.

> **See Also:** CREATE ROLLBACK SEGMENT on page 9-149 for information on creating a rollback segment

Prerequisites

You must have ALTER ROLLBACK SEGMENT system privilege.

Syntax



storage_clause: See storage_clause on page 11-129.

Keywords and Parameters

rollback_segment

Specify the name of an existing rollback segment.

ONLINE

Specify ONLINE to bring the rollback segment online. When you create a rollback segment, it is initially offline and not available for transactions. This clause brings the rollback segment online, making it available for transactions by your instance.

You can also bring a rollback segment online when you start your instance with the initialization parameter ROLLBACK_SEGMENTS.

OFFLINE

Specify OFFLINE to take the rollback segment offline.

- If the rollback segment does not contain any information needed to roll back an active transaction, Oracle takes it offline immediately.
- If the rollback segment does contain information for active transactions, Oracle makes the rollback segment unavailable for future transactions and takes it offline after all the active transactions are committed or rolled back.

Once the rollback segment is offline, it can be brought online by any instance.

To see whether a rollback segment is online or offline, query the data dictionary view DBA_ROLLBACK_SEGS. Online rollback segments have a STATUS value of IN_USE. Offline rollback segments have a STATUS value of AVAILABLE.

Restriction: You cannot take the SYSTEM rollback segment offline.

> **See Also:** *Oracle8i Administrator's Guide* for more information on making rollback segments available and unavailable

storage_clause

Use the *storage_clause* to change the rollback segment's storage characteristics.

Restriction: You cannot change the values of the INITIAL and MINEXTENTS for an existing rollback segment.

> **See Also:** *storage_clause* on page 11-129 for syntax and additional information

SHRINK

Specify SHRINK if you want Oracle to attempt to shrink the rollback segment to an optimal or specified size. The success and amount of shrinkage depend on the available free space in the rollback segment and how active transactions are holding space in the rollback segment.

The value of *integer* is in bytes, unless you specify K or M for kilobytes or megabytes.

If you do not specify TO *integer*, then the size defaults to the OPTIMAL value of the *storage_clause* of the CREATE ROLLBACK SEGMENT statement that created

the rollback segment. If OPTIMAL was not specified, then the size defaults to the MINEXTENTS value of the *storage_clause* of the CREATE ROLLBACK SEGMENT statement.

Regardless of whether you specify TO *integer*:

- The value to which Oracle shrinks the rollback segment is valid for the execution of the statement. Thereafter, the size reverts to the OPTIMAL value of the CREATE ROLLBACK SEGMENT statement.
- The rollback segment cannot shrink to less than two extents.

To determine the actual size of a rollback segment after attempting to shrink it, query the BYTES, BLOCKS, and EXTENTS columns of the DBA_SEGMENTS view.

Restriction: For Oracle Parallel Server, you can shrink only rollback segments that are online to your instance.

Examples

Bringing a Rollback Segment Online Example This statement brings the rollback segment RSONE online:

```
ALTER ROLLBACK SEGMENT rsone ONLINE;
```

Changing Rollback Segment Storage Example This statement changes the STORAGE parameters for RSONE:

```
ALTER ROLLBACK SEGMENT rsone
   STORAGE (NEXT 1000 MAXEXTENTS 20);
```

Resizing a Rollback Segment Example This statement attempts to resize a rollback segment to 100 megabytes:

```
ALTER ROLLBACK SEGMENT rsone
   SHRINK TO 100 M;
```

ALTER SEQUENCE

Purpose

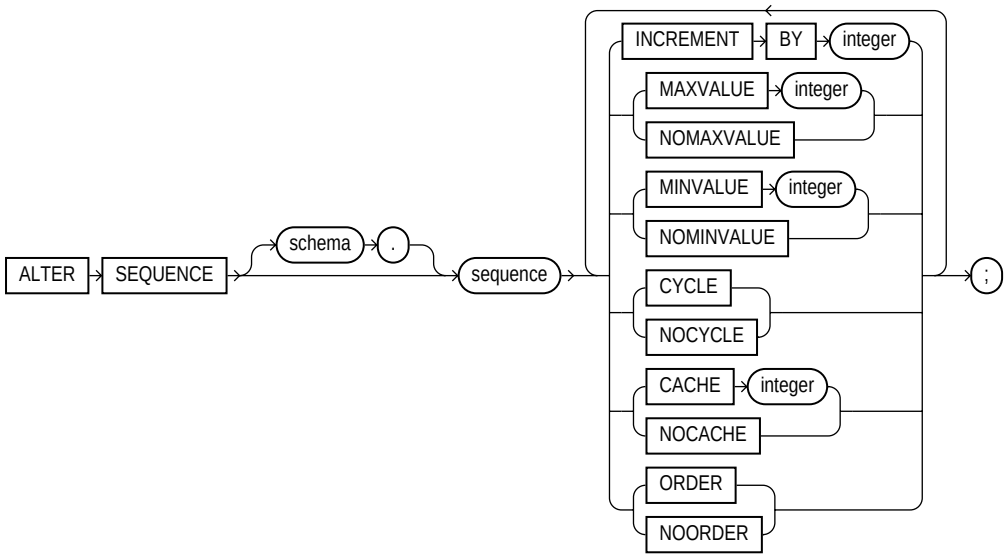
Use the ALTER SEQUENCE statement to change the increment, minimum and maximum values, cached numbers, and behavior of an existing sequence. This statement affects only future sequence numbers.

> **See Also:** CREATE SEQUENCE on page 9-155 for additional information on sequences

Prerequisites

The sequence must be in your own schema, or you must have the ALTER object privilege on the sequence, or you must have the ALTER ANY SEQUENCE system privilege.

Syntax



Keywords and Parameters

The keywords and parameters in this statement serve the same purposes they serve when you create a sequence.

- To restart the sequence at a different number, you must drop and re-create it.
- If you change the INCREMENT BY value before the first invocation of NEXTVAL, some sequence numbers will be skipped. Therefore, if you want to retain the original START WITH value, you must drop the sequence and re-create it with the original START WITH value and the new INCREMENT BY value.
- Oracle performs some validations. For example, a new MAXVALUE cannot be imposed that is less than the current sequence number.

See Also:

- CREATE SEQUENCE on page 9-155 for information on creating a sequence
- DROP SEQUENCE on page 11-3 for information on dropping and re-creating a sequence

Examples

Modifying a Sequence Examples This statement sets a new maximum value for the eseq sequence:

```
ALTER SEQUENCE eseq
   MAXVALUE 1500;
```

This statement turns on CYCLE and CACHE for the eseq sequence:

```
ALTER SEQUENCE eseq
   CYCLE
   CACHE 5;
```

ALTER SESSION

Purpose

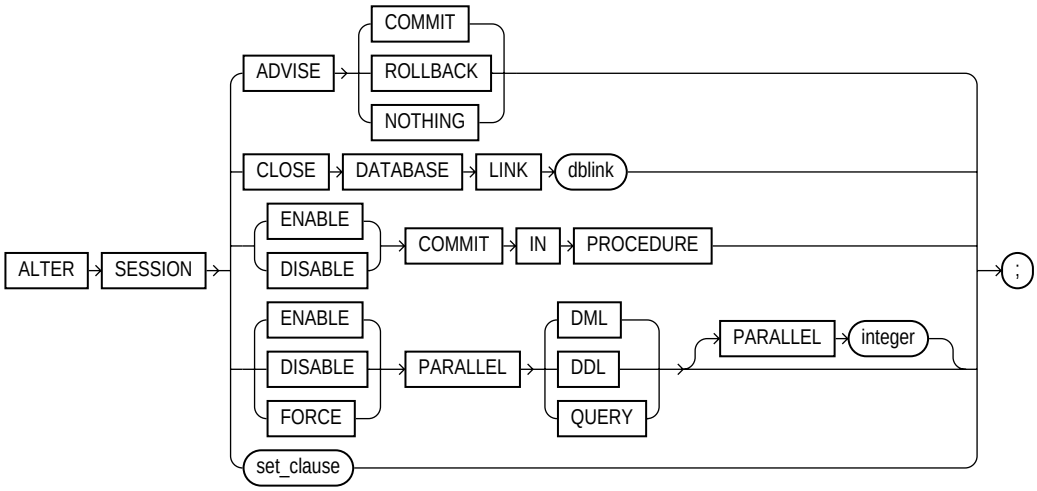
Use the `ALTER SESSION` statement to specify or modify any of the conditions or parameters that affect your connection to the database. The statement stays in effect until you disconnect from the database.

Prerequisites

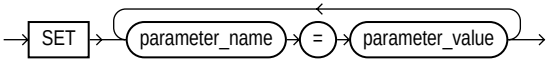
To enable and disable the SQL trace facility, you must have `ALTER SESSION` system privilege.

You do not need any privileges to perform the other operations of this statement unless otherwise indicated.

Syntax



set_clause::=



Keywords and Parameters

ADVISE

The ADVISE clause sends advice to a remote database to force a distributed transaction. The advice appears in the ADVICE column of the DBA_2PC_PENDING view on the remote database (the value 'C' for COMMIT, 'R' for ROLLBACK, and ' ' for NOTHING). If the transaction becomes in doubt, the administrator of that database can use this advice to decide whether to commit or roll back the transaction.

You can send different advice to different remote databases by issuing multiple ALTER SESSION statements with the ADVISE clause in a single transaction. Each such statement sends advice to the databases referenced in the following statements in the transaction until another such statement is issued.

> **See Also:** *Oracle8i Distributed Database Systems* for more information on distributed transactions and how to decide whether to commit or roll back in-doubt distributed transactions

CLOSE DATABASE LINK

Specify CLOSE DATABASE LINK to close the database link *dblink*. When you issue a statement that uses a database link, Oracle creates a session for you on the remote database using that link. The connection remains open until you end your local session or until the number of database links for your session exceeds the value of the initialization parameter OPEN_LINKS. If you want to reduce the network overhead associated with keeping the link open, use this clause to close the link explicitly if you do not plan to use it again in your session.

> **Note:** You must first close all cursors that use the link and then end your current transaction if it uses the link.

ENABLE | DISABLE COMMIT IN PROCEDURE

Procedures and stored functions written in PL/SQL can issue COMMIT and ROLLBACK statements. If your application would be disrupted by a COMMIT or ROLLBACK statement not issued directly by the application itself, use the DISABLE form of the COMMIT IN PROCEDURE clause to prevent procedures and stored functions called during your session from issuing these statements.

You can subsequently allow procedures and stored functions to issue COMMIT and ROLLBACK statements in your session by issuing the ENABLE form of this clause.

Some applications (such as SQL*Forms) automatically prohibit COMMIT and ROLLBACK statements in procedures and stored functions. Refer to your application documentation.

Note: This statement does not apply to database triggers. Triggers can never issue COMMIT or ROLLBACK statements.

PARALLEL DML | DDL | QUERY

The PARALLEL parameter determines whether all subsequent DML, DDL, or query statements in the session will be considered for parallel execution. This clause enables you to override the degree of parallelism of tables during the current session without changing the tables themselves. Uncommitted transactions must either be committed or rolled back prior to executing this clause for DML.

Note: You can execute this clause for DML only between committed transactions.

ENABLE — Specify ENABLE to execute subsequent statements in the session in parallel. This is the default for DDL and query statements.

- DML: The session's DML statements are executed in parallel mode if a parallel hint or a parallel clause is specified.
- DDL: The session's DDL statements are executed in parallel mode if a parallel clause is specified.
- QUERY: The session's queries are executed in parallel mode if a parallel hint or a parallel clause is specified

Restriction: You cannot specify the optional PARALLEL *integer* with ENABLE.

DISABLE — Specify DISABLE to execute subsequent statements serially. This is the default for DML statements.

- DML: The session's DML statements are executed serially.
- DDL: The session's DDL statements are executed serially.
- QUERY: The session's queries are executed serially.

Restriction: You cannot specify the optional PARALLEL *integer* with DISABLE.

FORCE

FORCE forces parallel execution of subsequent statements in the session. If no parallel clause or hint is specified, then a default degree of parallelism is used. This clause overrides any *parallel_clause* specified in subsequent statements in the session, but is overridden by a parallel hint.

- DML: Provided no parallel DML restrictions are violated, subsequent DML statements in the session are executed with the default degree of parallelism, unless a specific degree is specified in this clause.
- DDL: Subsequent DDL statements in the session are executed with the default degree of parallelism, unless a specific degree is specified in this clause. Resulting database objects will have associated with them the prevailing degree of parallelism.
- Using FORCE DDL automatically causes all tables created in this session to be created with a default level of parallelism. The effect is the same as if you had specified the *parallel_clause* (with default degree) with the CREATE TABLE statement.
- QUERY: Subsequent queries are executed with the default degree of parallelism, unless a specific degree is specified in this clause.
- PARALLEL *integer*: Specify an integer to explicitly specify a degree of parallelism

  - For force DDL, the degree overrides any parallel clause in subsequent DDL statements.

  - For force DML and QUERY, the degree overrides the degree currently stored for the table in the data dictionary.

  - A degree specified in a statement through a hint will override the degree being forced.

The following types of DML operations are not parallelized regardless of this clause:

- Operations on clustered tables
- Operations with embedded functions that either write or read database or package states
- Operations on tables with triggers that could fire
- Operations on tables or schema objects containing object types, or LONG or LOB datatypes.

> **See Also:** *Oracle8i Performance Guide and Reference* for a detailed description of parallel DML features and hints

set_clause

Use the *set_clause* to set the session parameters that follow (parameters that are dynamic in the scope of the ALTER SESSION statement). You can set values for multiple parameters in the same *set_clause*.

> **Caution:** Unless otherwise indicated, the parameters described here are initialization parameters, and the descriptions indicate only the general nature of the parameters. Before changing the values of initialization parameters, please refer to their full description in *Oracle8i Reference* or *Oracle8i National Language Support Guide*.

CONSTRAINT[S] = {immediate | deferred | default }

The CONSTRAINT[S] parameter determines when conditions specified by a deferrable constraint are enforced. CONSTRAINT[S] is a session parameter only, not an initialization parameter.

- immediate indicates that the conditions specified by the deferrable constraint are checked immediately after each DML statement. This setting is equivalent to issuing the SET CONSTRAINTS ALL IMMEDIATE statement at the beginning of each transaction in your session.

> **See Also:** the IMMEDIATE parameter of SET CONSTRAINT[S] on page 11-120

- `deferred` indicates that the conditions specified by the deferrable constraint are checked when the transaction is committed. This setting is equivalent to issuing the `SET CONSTRAINTS ALL DEFERRED` statement at the beginning of each transaction in your session.

  See Also: the `DEFERRED` parameter of SET CONSTRAINT[S] on page 11-120.

- `default` restores all constraints at the beginning of each transaction to their initial state of `DEFERRED` or `IMMEDIATE`.

`CREATE_STORED_OUTLINES = { true | false| 'category_name' }`

The `CREATE_STORED_OUTLINES` parameter determines whether Oracle should automatically create and store an outline for each query submitted during the session. `CREATE_STORED_OUTLINES` is not an initialization parameter.

- `true` enables automatic outline creation for subsequent queries in the same session. These outlines receive a unique system-generated name and are stored in the `DEFAULT` category. If a particular query already has an outline defined for it in the `DEFAULT` category, that outline will remain and a new outline will not be created.
- `false` disables automatic outline creation during the session. This is the default.
- *`category_name`* has the same behavior as `TRUE` except that any outline created during the session is stored in the *category_name* category.

`CURRENT_SCHEMA =` *`schema`*

The `CURRENT_SCHEMA` parameter changes the current schema of the session to the specified schema. Subsequent unqualified references to schema objects during the session will resolve to objects in the specified schema. The setting persists for the duration of the session or until you issue another `ALTER SESSION SET CURRENT_SCHEMA` statement. `CURRENT_SCHEMA` is a session parameter only, not an initialization parameter.

This setting offers a convenient way to perform operations on objects in a schema other than that of the current user without having to qualify the objects with the schema name. This setting changes the current schema, but it does not change the session user or the current user, nor does it give you any additional system or object privileges for the session.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* for more information on this parameter

CURSOR_SHARING = {force | exact}

The CURSOR_SHARING parameter determines what kind of SQL statements can share the same cursors.

- `exact` causes only identical SQL statements to share a cursor.
- `force` forces statements that may differ in some literals, but are otherwise identical, to share a cursor, unless the literals affect the meaning of the statement.

> **See Also:** *Oracle8i Performance Guide and Reference* for information on setting this parameter in these and other environments

DB_BLOCK_CHECKING = {true | false}

The DB_BLOCK_CHECKING parameter controls whether data block checking is done. The default is `false`.

DB_FILE_MULTIBLOCK_READ_COUNT = *integer*

The DB_FILE_MULTIBLOCK_READ_COUNT parameter specifies with *integer* the maximum number of blocks read in one I/O operation during a sequential scan. The default is 8.

FAST_START_IO_TARGET = *integer*

The FAST_START_IO_TARGET parameter specifies the target number of I/Os (reads and writes) to and from buffer cache that Oracle should perform upon crash or instance recovery. Oracle continuously calculates the actual number of I/Os that would be needed for recovery and compares that number against the target. If the actual number is greater than the target, Oracle attempts to write additional dirty buffers to advance the checkpoint, while minimizing the affect on performance.

> **See Also:** *Oracle8i Performance Guide and Reference* for information on how to tune this parameter

FLAGGER = { entry | intermediate | full | off }

The FLAGGER parameter specifies FIPS flagging, which causes an error message to be generated when a SQL statement issued is an extension of ANSI SQL92. FLAGGER is a session parameter only, not an initialization parameter.

In Oracle, there is currently no difference between Entry, Intermediate, or Full level flagging. Once flagging is set in a session, a subsequent ALTER SESSION SET FLAGGER statement will work, but generates the message, ORA-00097. This allows FIPS flagging to be altered without disconnecting the session. `off` turns off flagging.

GLOBAL_NAMES = { true | false }

When you start an instance, Oracle determines whether to enforce global name resolution for remote objects accessed in SQL statements based on the value of the initialization parameter GLOBAL_NAMES. This parameter enables or disables global name resolution for the duration of the session. `true` enables the enforcement of global names. `false` disables the enforcement of global names. You can also enable or disable global name resolution for your instance with the GLOBAL_NAMES parameter of the ALTER SYSTEM statement.

Oracle recommends that you enable global name resolution if you use or plan to use distributed processing.

> **See Also:** "Referring to Objects in Remote Databases" on page 2-90 and *Oracle8i Distributed Database Systems* for more information on global name resolution and how Oracle enforces it

HASH_AREA_SIZE = *integer*

The HASH_AREA_SIZE parameter specifies in bytes the amount of memory to use for hash join operations. The default is twice the value of the SORT_AREA_SIZE initialization parameter.

HASH_JOIN_ENABLED = {true | false}

The HASH_JOIN_ENABLED parameter enables or disables the use of the hash join operation in queries. The default is `true`, which enables hash joins.

HASH_MULTIBLOCK_IO_COUNT = *integer*

The HASH_MULTIBLOCK_IO_COUNT parameter specifies the number of data blocks to read and write during a hash join operation. The value multiplied by the DB_BLOCK_SIZE initialization parameter should not exceed 64 K. The default value for this parameter is 1. If the multi-threaded server is used, the value is always 1, and any value specified here is ignored.

INSTANCE = *integer*

The INSTANCE parameter in an Oracle Parallel Server environment accesses database files as if the session were connected to the instance specified by integer. INSTANCE is a session parameter only, not an initialization parameter. For optimum performance, each instance of Oracle Parallel Server uses its own private rollback segments, freelist groups, and so on. In an Oracle Parallel Server environment, you normally connect to a particular instance and access data that is partitioned primarily for your use. If you must connect to another instance, the data partitioning can be lost. Setting this parameter lets you access an instance as if you were connected to your own instance.

ISOLATION_LEVEL = { SERIALIZABLE | READ COMMITTED }

The ISOLATION_LEVEL parameter specifies how transactions containing database modifications are handled. ISOLATION_LEVEL is a session parameter only, not an initialization parameter.

- SERIALIZABLE indicates that transactions in the session use the serializable transaction isolation mode as specified in SQL92. That is, if a serializable transaction attempts to execute a DML statement that updates rows currently being updated by another uncommitted transaction at the start of the serializable transaction, then the DML statement fails. A serializable transaction can see its own updates.
- READ COMMITTED indicates that transactions in the session will use the default Oracle transaction behavior. Thus, if the transaction contains DML that requires row locks held by another transaction, then the DML statement will wait until the row locks are released.

LOG_ARCHIVE_DEST_*n*

```
LOG_ARCHIVE_DEST_n = {null_string
   | {LOCATION=local_pathname | SERVICE=tnsnames_service}
    [MANDATORY | OPTIONAL] [REOPEN[=integer]]}
```

The LOG_ARCHIVE_DEST_n parameter specifies up to five session-specific valid operating system pathnames or Oracle service names (plus other related options) as destinations for archive redo log file groups (n = integers 1 through 5).

Restrictions: If you set a value for this parameter, you cannot:

- Have definitions for the parameters LOG_ARCHIVE_DEST and LOG_ARCHIVE_DUPLEX_DEST in your initialization parameter file, nor can you set values for those parameters with the ALTER SYSTEM statement.

- Start archiving to a specific location using the ALTER SYSTEM ARCHIVE LOG TO *location* statement.

  See Also:

  - *Oracle8i Reference* for a description of the options
  - the LOG_ARCHIVE_DEST_*n* parameter in *Oracle8i Reference* for detailed information on specifying pathnames

LOG_ARCHIVE_DEST_STATE_*n* = {ENABLE | DEFER}

The LOG_ARCHIVE_DEST_STATE_*n* parameter specifies the session-specific state associated with the corresponding LOG_ARCHIVE_DEST_*n* parameter.

- ENABLE specifies that any associated valid destination can be used for archiving. This is the default.
- DEFER specifies that Oracle will not consider for archiving any destination associated with the corresponding LOG_ARCHIVE_DEST_*n* parameter.

LOG_ARCHIVE_MIN_SUCCEED_DEST *= integer*

The LOG_ARCHIVE_MIN_SUCCEED_DEST parameter specifies the session-specific minimum number of destinations that must succeed in order for the online log file to be available for reuse.

MAX_DUMP_FILE_SIZE = { *size* | UNLIMITED }

The MAX_DUMP_FILE_SIZE parameter specifies the upper limit of trace dump file size. Specify the maximum *size* as either a nonnegative integer that represents the number of blocks, or as UNLIMITED. If you specify UNLIMITED, no upper limit is imposed.

NLS Parameters

When you start an instance, Oracle establishes support based on the values of initialization parameters that begin with "NLS". You can query the dynamic performance table V$NLS_PARAMETERS to see the current NLS attributes for your session. For more information about NLS parameters, see *Oracle8i National Language Support Guide*.

NLS_CALENDAR = *'text'*

The NLS_CALENDAR parameter explicitly specifies a new calendar type.

NLS_COMP = *'text'*

The NLS_COMP parameter specifies that linguistic comparison is to be used according to the NLS_SORT parameter. This parameter obviates the need to specify NLS_SORT in SQL statements.

NLS_CURRENCY = *'text'*

The NLS_CURRENCY parameter explicitly specifies a new value for the L number format element (the local currency symbol). The symbol cannot exceed 10 characters.

NLS_DATE_FORMAT = *'fmt'*

The NLS_DATE_FORMAT parameter explicitly specifies a new default date format. The *fmt* value must be a valid date format model.

> **See Also:** "Date Format Models" on page 2-47 for information on valid date format models

NLS_DATE_LANGUAGE = *language*

The NLS_DATE_LANGUAGE parameter explicitly changes the language for names and abbreviations of days and months, and for spelled-out values of other date format elements.

NLS_DUAL_CURRENCY = *'text'*

The NLS_DUAL_CURRENCY parameter explicitly specifies a new "Euro" (or other) dual currency symbol. The value of *text* is returned by the number format element U, and *text* cannot exceed 10 characters.

> **See Also:** "Number Format Models" on page 2-43 for information on number format elements

NLS_ISO_CURRENCY = *territory*

The NLS_ISO_CURRENCY parameter explicitly specifies the territory whose ISO currency symbol should be used. That territory's currency symbol then becomes the value of the C number format element.

NLS_LANGUAGE = *language*

The NLS_LANGUAGE parameter changes the language in which Oracle returns errors and other messages. This parameter also implicitly specifies new values for these items:

- Language for day and month names and abbreviations and spelled values of other elements
- Linguistic sort sequences or binary sorts
- B.C. and A.D. indicators
- A.M. and P.M. meridian indicators

NLS_NUMERIC_CHARACTERS = *'text'*

The NLS_NUMERIC_CHARACTERS parameter explicitly specifies a new decimal character and group separator. The *text* value must have this form:

```
'dg'
```

where: *d* is the new decimal character, and *g* is the new group separator.

The decimal character and the group separator must be two different single-byte characters, and cannot be a numeric value or any of the following characters: plus sign ("+"), minus sign or hyphen ("–"), less-than sign ("<"), or greater-than sign (">").

If the decimal character is not a period (.), you must use single quotation marks to enclose all number values that appear in expressions in your SQL statements. When not using a period for the decimal point, use the TO_NUMBER function to ensure that a valid number is retrieved.

NLS_SORT = { *sort* | BINARY}

The NLS_SORT parameter changes the sequence into which Oracle sorts character values. *sort* specifies the name of a linguistic sort sequence. BINARY specifies a binary sort. The default is BINARY.

NLS_TERRITORY = *territory*

The NLS_TERRITORY parameter implicitly specifies new values for these items:

- Default date format
- Decimal character and group separators
- Local currency symbol
- ISO currency symbol
- First day of the week for D date format element

OBJECT_CACHE_MAX_SIZE_PERCENT = *integer*

The OBJECT_CACHE_MAX_SIZE_PERCENT parameter specifies the percentage of the optimal cache size that the session object cache can grow beyond the optimal size. The default is 10.

OBJECT_CACHE_OPTIMAL_SIZE = *integer*

The OBJECT_CACHE_OPTIMAL_SIZE parameter specifies (in kilobytes) the size to which the session object cache is reduced when it exceeds maximum size. The default is 100.

OPTIMIZER_INDEX_CACHING = *integer*

The OPTIMIZER_INDEX_CACHING parameter lets you tune the optimizer to favor nested loops joins and IN-list iterators. The value of *integer* indicates the percentage of the index blocks assumed to be in the cache.

OPTIMIZER_INDEX_COST_ADJ = *integer*

The OPTIMIZER_INDEX_COST_ADJ parameter lets you tune optimizer behavior for access path selection to make the optimizer more likely to select an index access path than a full table scan. The value of *integer* is a percentage indicating the importance the optimizer attaches to the index path compared with "normal". The default is 100 (indicating 100%), which makes the optimizer cost index access paths at the regular cost.

OPTIMIZER_MAX_PERMUTATIONS = *integer*

The OPTIMIZER_MAX_PERMUTATIONS parameter lets you limit the amount of work the optimizer expends on optimizing queries with large joins. The value of *integer* is the number of permutations of the tables the optimizer will consider with large joins.

OPTIMIZER_MODE = { all_rows | first_rows | rule | choose }

The OPTIMIZER_MODE parameter specifies the approach and mode of the optimizer for your session.

> **See Also:** *Oracle8i Concepts* and *Oracle8i Performance Guide and Reference* for information on how to choose a goal for the cost-based approach based on the characteristics of your application

- all_rows specifies the cost-based approach and optimizes for best throughput.

- `first_rows` specifies the cost-based approach and optimizes for best response time.
- `rule` specifies the rule-based approach. (The rule-based optimizer does not use function-based indexes.)
- `choose` causes the optimizer to choose an optimization approach based on the presence of statistics in the data dictionary.

`OPTIMIZER_PERCENT_PARALLEL = integer`

The `OPTIMIZER_PERCENT_PARALLEL` parameter specifies the amount of parallelism the optimizer uses in its cost functions. The default is 0 (no parallelism).

`PARALLEL_BROADCAST_ENABLED = { true | false }`

The `PARALLEL_BROADCAST_ENABLED` parameter lets you enhance performance during hash and merge joins.

`PARALLEL_INSTANCE_GROUP = ' text '`

The `PARALLEL_INSTANCE_GROUP` parameter identifies the parallel instance group to be used for spawning parallel query slaves. The default is all active instances.

Note: Set this parameter only if you are running Oracle Parallel Server in parallel mode.

`PARALLEL_MIN_PERCENT = integer`

The `PARALLEL_MIN_PERCENT` parameter specifies the minimum percent of threads required for parallel query. The default is 0 (no parallelism).

`PARTITION_VIEW_ENABLED = { true | false }`

The `PARTITION_VIEW_ENABLED` parameter, when set to `true`, causes the optimizer to skip unnecessary table accesses in a partition view.

Note: For important information on partition views, see "Partition Views" on page 10-106.

`PLSQL_V2_COMPATIBILITY = { true | false }`

The `PLSQL_V2_COMPATIBILITY` parameter, if `true`, modifies the compile-time behavior of PL/SQL programs to allow language constructs that are illegal in

Oracle8 and Oracle8i (PL/SQL V3), but were legal in Oracle7 (PL/SQL V2). `false` disallows illegal Oracle7 PL/SQL V2 constructs. This is the default.

> **See Also:** *PL/SQL User's Guide and Reference* and *Oracle8i Reference* for more information about this session parameter

`QUERY_REWRITE_ENABLED = { true | false }`

The `QUERY_REWRITE_ENABLED` parameter enables or disables query rewrite on all materialized views that have not been explicitly disabled. Query rewrite is disabled by default. It is also disabled by rule-based optimization (that is, if the `OPTIMIZER_MODE` parameter is set to `rule`).

This parameter has the following additional effect on the use of function-based indexes:

- `true`: Oracle will use function-based indexes to derive values of SQL expressions. If in addition the `QUERY_REWRITE_INTEGRITY` parameter is set to any value other than `enforced`, Oracle will derive such values even if the index is based on a user-defined (rather than SQL) function.
- `false`: Oracle will not use function-based indexes to derive values of SQL expressions, but it will use such indexes to obtain values of real columns in the index.

Enabling or disabling query rewrite does not affect descending indexes.

A setting of `true` has no effect on materialized views that cannot be created with the `ENABLE QUERY REWRITE` clause, such as materialized views created totally or in part from a view.

> **See Also:** *Oracle8i Data Warehousing Guide* for more information on query rewrite

`QUERY_REWRITE_INTEGRITY`

```
QUERY_REWRITE_INTEGRITY =
   { enforced | trusted | stale_tolerated }
```

The `QUERY_REWRITE_INTEGRITY` parameter sets the minimum consistency level for query rewrite. The following values are permitted:

- `enforced` is the safest level. It relies only on system-enforced relationships so that data integrity and correctness can be guaranteed. This level ensures that query rewrite will not use any function-based index or any materialized view that includes a call to a user-defined function.

In addition, this level ensures that query rewrite will not use any dimensional information or any constraints enabled with the RELY keyword.

- `trusted` specifies that materialized views created with the `ON PREBUILT TABLE` clause are supported, and trusted but unenforced join relationships are accepted. Query rewrite uses join information from dimensions and enables unenforced constraints with the RELY keyword.
- `stale_tolerated` specifies that any stale, usable materialized view may be used.

This parameter does not affect descending indexes.

See Also:

- *Oracle8i Data Warehousing Guide* for more information on query rewrite integrity level
- CREATE DIMENSION on page 9-34 for information on dimensions
- *constraint_clause* on page 8-136for information on constraints enabled with the RELY keyword

`REMOTE_DEPENDENCIES_MODE = { timestamp | signature }`

The `REMOTE_DEPENDENCIES_MODE` specifies how dependencies of remote stored procedures are handled by the session.

See Also: *Oracle8i Application Developer's Guide - Fundamentals*

`SESSION_CACHED_CURSORS = integer`

The `SESSION_CACHED_CURSORS` parameter specifies the number of frequently used cursors that can be retained in the cache. The cursors can be open or closed, which is particularly useful for Oracle tools that close all session cursors associated with a form when switching to another form. In such cases, frequently used cursors do not have to be reparsed. A least recently used algorithm ages out entries in the cache to make room for new entries when needed.

See Also: *Oracle8i Performance Guide and Reference* for more information on session cursor caching

SKIP_UNUSABLE_INDEXES = { true | false }

The SKIP_UNUSABLE_INDEXES parameter controls the use and reporting of tables with unusable indexes or index partitions. SKIP_UNUSABLE_INDEXES is a session parameter only, not an initialization parameter.

- true disables error reporting of indexes and index partitions marked UNUSABLE. Allows all operations (inserts, deletes, updates, and selects) to tables with unusable indexes or index partitions.

  Note: Statements that would normally use the unusable indexes or index partitions may be compiled with suboptimal optimizer plans, occasionally resulting in major degradation in response time and resource utilization.

- false enables error reporting of indexes marked UNUSABLE. Does not allow inserts, deletes, and updates to tables with unusable indexes or index partitions. This is the default.

SORT_AREA_RETAINED_SIZE = *integer*

The SORT_AREA_RETAINED_SIZE parameter specifies (in bytes) the maximum amount of memory that each sort operation will retain after the first fetch is done, until the cursor ends. If you do not explicitly set this parameter in the initialization parameter file or dynamically, Oracle uses the value of the SORT_AREA_SIZE parameter.

SORT_AREA_SIZE = *integer*

The SORT_AREA_SIZE parameter specifies (in bytes) the maximum amount of memory to use for each sort operation. The default is OS-dependent.

SORT_MULTIBLOCK_READ_COUNT = *integer*

The SORT_MULTIBLOCK_READ_COUNT parameter specifies the number of database blocks to read each time a sort performs a read from temporary segments. The default is 2.

SQL_TRACE = { true | false }

The SQL trace facility generates performance statistics for the processing of SQL statements. When you begin a session, Oracle enables or disables the SQL trace facility based on the value of this parameter. You can subsequently enable or disable

the SQL trace facility for your own session with the SQL_TRACE parameter of the ALTER SESSION statement.

- true enables the SQL trace facility.
- false disables the SQL trace facility.

SQL_TRACE is an initialization parameter. However, when you change its value with an ALTER SESSION statement, the results are not reflected in the V$PARAMETER view. Therefore, in this context it is considered a session parameter only.

> **See Also:** *Oracle8i Performance Guide and Reference* for more information on the SQL trace facility, including how to format and interpret its output

STAR_TRANSFORMATION_ENABLED = { true | false }

The STAR_TRANSFORMATION_ENABLED parameter determines whether a cost-based query transformation will be applied to star queries. The default is false.

TIMED_STATISTICS = {true | false }

The TIMED_STATISTICS parameter specifies whether the server requests the time from the operating system when generating time-related statistics. The default is false.

USE_STORED_OUTLINES = { true | false | *'category_name'* }

The USE_STORED_OUTLINES parameter determines whether the optimizer will use stored outlines to generate execution plans. USE_STORED_OUTLINES is not an initialization parameter.

- true causes the optimizer to use outlines stored in the DEFAULT category when compiling requests.
- false specifies that the optimizer should not use stored outlines. This is the default.
- *category_name* causes the optimizer to use outlines stored in the *category_name* category when compiling requests.

Note: If you want the execution plan to consider materialized views, you must specify them in the outline. If the outline does not use a materialized view, then the query that uses the outline will not use the materialized view either, even if you have enabled query rewrite.

Examples

Enabling Parallel DML Example Issue the following statement to enable parallel DML mode for the current session:

```
ALTER SESSION ENABLE PARALLEL DML;
```

Forcing a Distributed Transaction Example The following transaction inserts an employee record into the `emp` table on the database identified by the database link `site1` and deletes an employee record from the `emp` table on the database identified by `site2`:

```
ALTER SESSION
   ADVISE COMMIT;

INSERT INTO emp@site1
   VALUES (8002, 'FERNANDEZ', 'ANALYST', 7566,
   TO_DATE('04-OCT-1992', 'DD-MON-YYYY'), 3000, NULL, 20);

ALTER SESSION
   ADVISE ROLLBACK;

DELETE FROM emp@site2
   WHERE empno = 8002;

COMMIT;
```

This transaction has two `ALTER SESSION` statements with the `ADVISE` clause. If the transaction becomes in doubt, `site1` is sent the advice '`COMMIT`' by virtue of the first `ALTER SESSION` statement and `site2` is sent the advice '`ROLLBACK`' by virtue of the second.

Closing a Database Link Example This statement updates the employee table on the `sales` database using a database link, commits the transaction, and explicitly closes the database link:

```
UPDATE emp@sales
   SET sal = sal + 200
   WHERE empno = 9001;

COMMIT;

ALTER SESSION
   CLOSE DATABASE LINK sales;
```

Changing the Date Format Dynamically Example The following statement dynamically changes the default date format for your session to 'YYYY MM DD-HH24:MI:SS':

```
ALTER SESSION
   SET NLS_DATE_FORMAT = 'YYYY MM DD HH24:MI:SS';
```

Oracle uses the new default date format:

```
SELECT TO_CHAR(SYSDATE) Today
   FROM DUAL;

TODAY
-------------------
1997 08 12 14:25:56
```

Changing the Date Language Dynamically Example The following statement changes the language for date format elements to French:

```
ALTER SESSION
   SET NLS_DATE_LANGUAGE = French;

SELECT TO_CHAR(SYSDATE, 'Day DD Month YYYY') Today
   FROM DUAL;

TODAY
---------------------------
Mardi    28 Février   1997
```

Changing the ISO Currency Example The following statement dynamically changes the ISO currency symbol to the ISO currency symbol for the territory America:

```
ALTER SESSION
   SET NLS_ISO_CURRENCY = America;

SELECT TO_CHAR( SUM(sal), 'C999G999D99') Total
```

```
   FROM emp;

TOTAL
-------------
USD29,025.00
```

Changing the Decimal Character and Group Separator Example The following statement dynamically changes the decimal character to comma (,) and the group separator to period (.):

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS = ',.' ;
```

Oracle returns these new characters when you use their number format elements:

```
SELECT TO_CHAR( SUM(sal), 'L999G999D99') Total FROM emp ;

TOTAL
-------------
FF29.025,00
```

Changing the NLS Currency Example The following statement dynamically changes the local currency symbol to 'DM':

```
ALTER SESSION
   SET NLS_CURRENCY = 'DM';

SELECT TO_CHAR( SUM(sal), 'L999G999D99') Total
   FROM emp;

TOTAL
-------------
DM29.025,00
```

Changing the NLS Language Example The following statement dynamically changes to French the language in which error messages are displayed:

```
ALTER SESSION
   SET NLS_LANGUAGE = FRENCH;

SELECT * FROM DMP;

ORA-00942: Table ou vue inexistante
```

Changing the Linguistic Sort Sequence Example The following statement dynamically changes the linguistic sort sequence to Spanish:

```
ALTER SESSION
   SET NLS_SORT = XSpanish;
```

Oracle sorts character values based on their position in the Spanish linguistic sort sequence.

Enabling SQL Trace Example To enable the SQL trace facility for your session, issue the following statement:

```
ALTER SESSION
   SET SQL_TRACE = TRUE;
```

Enabling Query Rewrite Example This statement enables query rewrite in the current session for all materialized views that have not been explicitly disabled:

```
ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE;
```

ALTER SYSTEM

Purpose

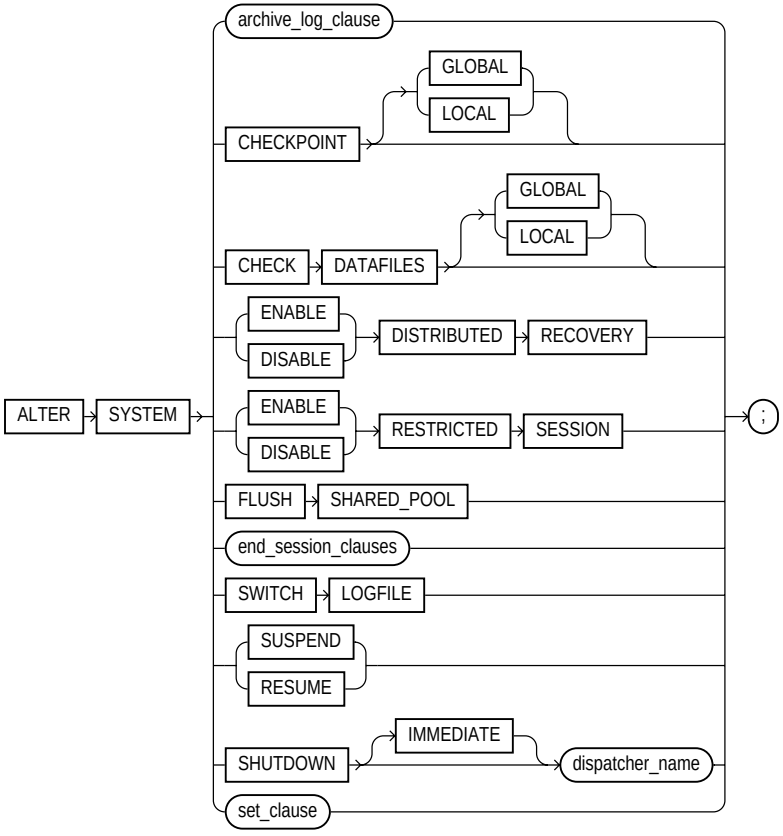
Use the ALTER SYSTEM statement to dynamically alter your Oracle instance. The settings stay in effect as long as the database is mounted.

Prerequisites

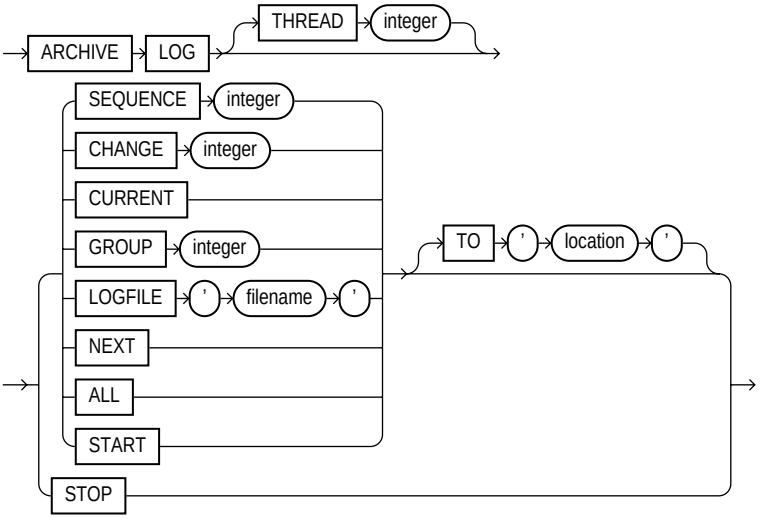
You must have ALTER SYSTEM system privilege.

To specify the *archive_log_clause*, you must have the OSDBA or OSOPER role enabled.

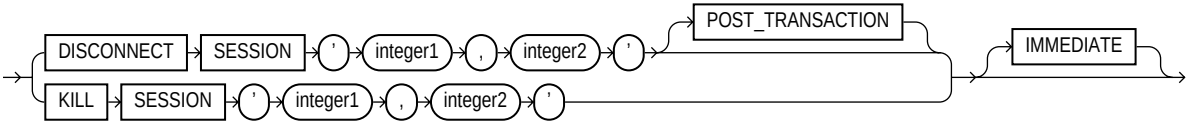
Syntax



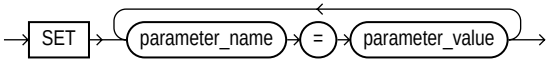
archive_log_clause::=



end_session_clauses::=



set_clause::=



Keywords and Parameters

archive_log_clause

The *archive_log_clause* manually archives redo log files or enables or disables automatic archiving. To use this clause, your instance must have the database mounted. The database can be either open or closed unless otherwise noted.

Notes:

- You can also manually archive redo log file groups with the ARCHIVE LOG SQL*Plus statement.
- You can also have Oracle archive redo log files groups automatically. You can always manually archive redo log file groups regardless of whether automatic archiving is enabled.

See Also:

- *Oracle8i Administrator's Guide* for information on automatic archiving
- *SQL*Plus User's Guide and Reference* for information on the ARCHIVE LOG statement

THREAD *integer*

Specify THREAD to indicate the thread containing the redo log file group to be archived.

Restriction: Set this parameter only if you are using Oracle with the Parallel Server option in parallel mode.

SEQUENCE *integer*

Specify SEQUENCE to manually archive the online redo log file group identified by the log sequence number *integer* in the specified thread. If you omit the THREAD parameter, Oracle archives the specified group from the thread assigned to your instance.

CHANGE *integer*

Specify CHANGE to manually archive the online redo log file group containing the redo log entry with the system change number (SCN) specified by *integer* in the specified thread. If the SCN is in the current redo log file group, Oracle performs a log switch. If you omit the THREAD parameter, Oracle archives the groups containing this SCN from all enabled threads.

Restriction: You can use this clause only when your instance has the database open.

CURRENT — Specify CURRENT to manually archive the current redo log file group of the specified thread, forcing a log switch. If you omit the THREAD parameter, Oracle archives all redo log file groups from all enabled threads, including logs previous to current logs.

Restriction: You can use this clause only when your instance has the database open.

Note: If you specify a redo log file group for archiving with the CHANGE or CURRENT clause, and earlier redo log file groups are not yet archived, Oracle archives all unarchived groups up to and including the specified group.

GROUP *integer* — Specify GROUP to manually archive the online redo log file group with the GROUP value specified by *integer*. You can determine the GROUP value for a redo log file group by examining the data dictionary view DBA_LOG_FILES. If you specify both the THREAD and GROUP parameters, the specified redo log file group must be in the specified thread.

LOGFILE *'filename'* — Specify LOGFILE to manually archive the online redo log file group containing the redo log file member identified by '*filename*'. If you specify both the THREAD and LOGFILE parameters, the specified redo log file group must be in the specified thread.

Restriction: You must archive redo log file groups in the order in which they are filled. If you specify a redo log file group for archiving with the LOGFILE parameter, and earlier redo log file groups are not yet archived, Oracle returns an error.

NEXT — Specify NEXT to manually archive the next online redo log file group from the specified thread that is full but has not yet been archived. If you omit the THREAD parameter, Oracle archives the earliest unarchived redo log file group from any enabled thread.

Note: The parameters SEQUENCE, CHANGE, CURRENT, GROUP, LOGFILE, and NEXT implicitly refer to one redo log file or group. However, Oracle maintains a "force system change number (SCN)." Whenever archiving occurs, Oracle archives all redo log files with SCNs lower than or equal to the force SCN. Therefore, when you specify any of these parameters, Oracle sometimes archives more than one redo log file or group.

ALL — Specify ALL to manually archive all online redo log file groups from the specified thread that are full but have not been archived. If you omit the THREAD parameter, Oracle archives all full unarchived redo log file groups from all enabled threads.

START — Specify START to enable automatic archiving of redo log file groups.

Restriction: You can enable automatic archiving only for the thread assigned to your instance.

TO '*location*' — Specify TO '*location*' to indicate the primary location to which the redo log file groups are archived. The value of this parameter must be a fully specified file location following the conventions of your operating system. If you omit this parameter, Oracle archives the redo log file group to the location specified by the initialization parameters LOG_ARCHIVE_DEST or LOG_ARCHIVE_DEST_*n*.

Note: You can enhance recovery reliability by setting the related archive parameters LOG_ARCHIVE_DEST_DUPLEX and LOG_ARCHIVE_MIN_SUCCEED_DEST.

STOP — Specify to disable automatic archiving of redo log file groups. You can disable automatic archiving only for the thread assigned to your instance.

CHECKPOINT

Specify CHECKPOINT to explicitly force Oracle to perform a checkpoint, ensuring that all changes made by committed transactions are written to datafiles on disk. You can specify this clause only when your instance has the database open. Oracle does not return control to you until the checkpoint is complete.

GLOBAL — In an Oracle Parallel Server environment, this setting causes Oracle to perform a checkpoint for all instances that have opened the database. This is the default.

LOCAL — In an Oracle Parallel Server environment, this setting causes Oracle to performs a checkpoint only for the thread of redo log file groups for your instance.

See Also: *Oracle8i Concepts* for more information on checkpoints

CHECK DATAFILES

In a distributed database system, such as an Oracle Parallel Server environment, this clause updates an instance's SGA from the database control file to reflect information on all online datafiles.

GLOBAL — Specify GLOBAL to perform this synchronization for all instances that have opened the database. This is the default.

LOCAL — Specify LOCAL to perform this synchronization only for the local instance.

Your instance should have the database open.

> **See Also:** *Oracle8i Parallel Server Installation, Configuration, and Administration*

end_session_clauses

DISCONNECT SESSION — Use the DISCONNECT SESSION clause to disconnect the current session by destroying the dedicated server process (or virtual circuit if the connection was made by way of a multi-threaded server). To use this clause, your instance must have the database open. You must identify the session with both of the following values from the V$SESSION view:

integer1 — The first integer is the value of the SID column.

integer2 — The second integer is the value of the SERIAL# column.

If system parameters are appropriately configured, application failover will take effect.

> **See Also:** *Oracle8i Parallel Server Installation, Configuration, and Administration* for more information about application failover

POST_TRANSACTION — The POST_TRANSACTION setting allows ongoing transactions to complete before the session is disconnected. If the session has no ongoing transactions, this clause has the same effect as KILL SESSION, described below.

`IMMEDIATE` The `IMMEDIATE` setting disconnects the session and recovers the entire session state immediately, without waiting for ongoing transactions to complete.

- If you also specify `POST_TRANSACTION` and the session has ongoing transactions, the `IMMEDIATE` keyword is ignored.
- If you do not specify `POST_TRANSACTION`, or you specify `POST_TRANSACTION` but the session has no ongoing transactions, this clause has the same effect as `KILL SESSION IMMEDIATE`, described below.

`KILL SESSION` The `KILL SESSION` clause lets you mark a session as dead, roll back ongoing transactions, release all session locks, and partially recover session resources. To use this clause, your instance must have the database open, and your session and the session to be killed must be on the same instance. You must identify the session with both of the following values from the `V$SESSION` view:

`integer1` The first integer is the value of the `SID` column.

`integer2` The second is the value of the `SERIAL#` column.

If the session is performing some activity that must be completed, such as waiting for a reply from a remote database or rolling back a transaction, Oracle waits for this activity to complete, marks the session as dead, and then returns control to you. If the waiting lasts a minute, Oracle marks the session to be killed and returns control to you with a message that the session is marked to be killed. The PMON background process then marks the session as dead when the activity is complete.

Whether or not the session has an ongoing transaction, Oracle does not recover the entire session state until the session user issues a request to the session and receives a message that the session has been killed.

`IMMEDIATE` Specify `IMMEDIATE` to roll back ongoing transactions, release all session locks, recover the entire session state, and return control to yourself immediately.

DISTRIBUTED RECOVERY

The DISTRIBUTED RECOVERY clause lets you enable or disable distributed recovery. To use this clause, your instance must have the database open.

ENABLE — Specify ENABLE to enable distributed recovery. In a single-process environment, you must use this clause to initiate distributed recovery.

You may need to issue the ENABLE DISTRIBUTED RECOVERY statement more than once to recover an in-doubt transaction if the remote node involved in the transaction is not accessible. In-doubt transactions appear in the data dictionary view DBA_2PC_PENDING.

> **See Also:** *Oracle8i Distributed Database Systems* for more information about distributed transactions and distributed recovery

DISABLE — Specify DISABLE to disable distributed recovery.

RESTRICTED SESSION

The RESTRICTED SESSION clause lets you restrict logon to Oracle.

You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

ENABLE — Specify ENABLE to allows only users with RESTRICTED SESSION system privilege to log on to Oracle. Existing sessions are not terminated.

DISABLE — Specify DISABLE to reverse the effect of the ENABLE RESTRICTED SESSION clause, allowing all users with CREATE SESSION system privilege to log on to Oracle. This is the default.

FLUSH SHARED_POOL

The FLUSH SHARED POOL clause lets you clear all data from the shared pool in the system global area (SGA). The shared pool stores

- Cached data dictionary information and
- Shared SQL and PL/SQL areas for SQL statements, stored procedures, function, packages, and triggers.

This statement does not clear shared SQL and PL/SQL areas for items that are currently being executed. You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

SWITCH LOGFILE

The SWITCH LOGFILE clause lets you explicitly force Oracle to begin writing to a new redo log file group, regardless of whether the files in the current redo log file group are full. When you force a log switch, Oracle begins to perform a checkpoint. Oracle returns control to you immediately rather than when the checkpoint is complete. To use this clause, your instance must have the database open.

SUSPEND | RESUME

SUSPEND — The SUSPEND clause lets you suspend all I/O (datafile, control file, and file header) as well as queries, in all instances, enabling you to make copies of the database without having to handle ongoing transactions.

Restrictions:

- Do not use this clause unless you have put the database tablespaces in hot backup mode.
- If you start a new instance while the system is suspended, that new instance will not be suspended.

RESUME — The RESUME clause lets you make the database available once again for queries and I/O.

See Also: *Oracle8i Backup and Recovery Guide* for more information on the SUSPEND clause and RESUME clause

SHUTDOWN

The SHUTDOWN clause is relevant only if your system is using Oracle's multi-threaded server architecture. It shuts down a dispatcher identified by *dispatcher_name*. The *dispatcher_name* must be a string of the form 'D*xxx*', where xxx indicates the number of the dispatcher. (For a listing of dispatcher names, query the NAME column of the V$DISPATCHER dynamic performance view.)

- If you specify IMMEDIATE, the dispatcher stops accepting new connections immediately and Oracle terminates all existing connections through that dispatcher. After all sessions are cleaned up, the dispatcher process literally shuts down.

- If you do not specify IMMEDIATE, the dispatcher stops accepting new connections immediately but waits for all its users to disconnect and for all its database links to terminate. Then it literally shuts down.

  See Also: *Oracle8i Administrator's Guide*, *Net8 Administrator's Guide*, and *Oracle8i Performance Guide and Reference* for more information on dispatchers and multi-threaded server architecture

set_clause

The *set_clause* lets you set the system parameters that follow. You can set values for multiple parameters in the same *set_clause*.

Note: The DEFERRED keyword sets or modifies the value of the parameter for future sessions that connect to the database.

Caution: Unless otherwise noted, these parameters are initialization parameters, and the descriptions provided here indicate only the general nature of the parameters. Before changing the values of initialization parameters, please refer to their full description in *Oracle8i Reference* and *Oracle8i National Language Support Guide*.

AQ_TM_PROCESSES = *integer*

AQ_TM_PROCESSES is an Advanced Queuing parameter that specifies whether a queue monitor process is created. Accepted values are 1 (creates one queue monitor process to monitor messages) and 0 (kills any existing queue monitor processes, whether they were created using an initialization parameter or another ALTER SYSTEM statement). You can create up to 10 queue monitor processes if you use this parameter in an initialization parameter file.

BACKGROUND_DUMP_DEST = *'text'*

The BACKGROUND_DUMP_DEST parameter specifies the pathname for a directory where debugging trace files for the background processes are written during Oracle operations.

BACKUP_TAPE_IO_SLAVES = {TRUE | FALSE} DEFERRED

The BACKUP_TAPE_IO_SLAVES parameter lets you specify whether I/O slaves are used by the Recovery Manager to back up, copy, or restore data to tape.

CONTROL_FILE_RECORD_KEEP_TIME = *integer*

The CONTROL_FILE_RECORD_KEEP_TIME parameter lets you specify the minimum of days before a reusable record in the control file can be reused.

CORE_DUMP_DEST = *'text'*

The CORE_DUMP_DEST parameter lets you specify the directory where Oracle dumps core files.

CREATE_STORED_OUTLINES = **{ true | false | 'category_name' } [nooverride]**

The CREATE_STORED_OUTLINES parameter determines whether Oracle should automatically create and store an outline for each query submitted on the system. CREATE_STORED_OUTLINES is not an initialization parameter.

- `true` enables automatic outline creation for subsequent queries in the system. These outlines receive a unique system-generated name and are stored in the DEFAULT category. If a particular query already has an outline defined for it in the DEFAULT category, that outline will remain and a new outline will not be created.
- `false` disables automatic outline creation for the system. This is the default.
- *`category_name`* has the same behavior as `true` except that any outline created in the system is stored in the *`category_name`* category.
- `nooverride` specifies that this system setting will not override the setting for any session in which this parameter was explicitly set. If you do not specify `nooverride`, this setting takes effect in all sessions.

CURSOR_SHARING = {force | exact}

The CURSOR_SHARING parameter determines what kind of SQL statements can share the same cursors.

- `exact` causes only identical SQL statements to share a cursor.
- `force` forces statements that may differ in some literals, but are otherwise identical, to share a cursor, unless the literals affect the meaning of the statement.

See Also: *Oracle8i Performance Guide and Reference* for information on setting this parameter in these and other environments

DB_BLOCK_CHECKING = {true | false} deferred

The DB_BLOCK_CHECKING parameter controls whether data block checking is done. The default is `false`, for compatibility with earlier releases where block checking is disabled as a default.

DB_BLOCK_CHECKSUM = {true | false}

The DB_BLOCK_CHECKSUM parameter determines whether the database writer background process and the direct loader will calculate a checksum and store it in the cache header of every data lock when writing to disk.

DB_BLOCK_MAX_DIRTY_TARGET = *integer*

The DB_BLOCK_MAX_DIRTY_TARGET parameter limits to *integer* the number of dirty buffers in the cache and reduces the number of buffers that will need to be read during crash or instance recovery. This parameter does **not** relate to media recovery. A value of 0 disables this parameter. The minimum accepted value to enable the parameter is 1000.

Note: Oracle Corporation recommends that Enterprise Edition users who were using incremental checkpointing in an earlier release now use fast-start checkpointing in Oracle8*i*. In fast-start checkpointing, the FAST_START_IO_TARGET parameter takes the place of DB_FILE_MAX_DIRTY_TARGET. See FAST_START_IO_TARGET below.

See Also:

- *Oracle8i Backup and Recovery Guide* for information on fast-start checkpointing
- *Oracle8i Reference* for information on the new parameters

DB_FILE_DIRECT_IO_COUNT = *integer* deferred

The DB_FILE_DIRECT_IO_COUNT parameter determines the number of blocks Oracle should use for I/O during backup, restore, or direct-path read and write operations.

DB_FILE_MULTIBLOCK_READ_COUNT = *integer*

The DB_FILE_MULTIBLOCK_READ_COUNT parameter determines the maximum number of blocks read in one I/O operation during a sequential scan.

FAST_START_IO_TARGET = *integer*

The FAST_START_IO_TARGET determines the target number of I/Os (reads and writes) to and from buffer cache that Oracle should perform upon crash or instance recovery. Oracle continuously calculates the actual number of I/Os that would be needed for recovery and compares that number against the target. If the actual number is greater than the target, Oracle attempts to write additional dirty buffers to advance the checkpoint, while minimizing the affect on performance.

> **See Also:** *Oracle8i Performance Guide and Reference* for information on how to tune this parameter

FAST_START_PARALLEL_ROLLBACK = { false | low | high}

The FAST_START_PARALLEL_ROLLBACK parameter determines the number of processes spawned to perform parallel recovery.

- false specifies no parallel recovery. SMON will serially recover dead transactions.
- low specifies that the number of recovery servers may not exceed twice the value of the CPU_COUNT parameter.
- high specifies that the number of recovery servers may not exceed four times the value of the CPU_COUNT parameter.

FIXED_DATE = { '*DD_MM_YY*' | '*YYYY_MI_DD_HH24_MI*-SS' }

The FIXED_DATE lets you specify a constant date for SYSDATE instead of the current date.

GC_DEFER_TIME = *integer*

The GC_DEFER_TIME parameter lets you specify the time (in hundredths of seconds) that Oracle waits before responding to forced-write requests from other instances.

GLOBAL_NAMES = {true | false}

When you start an instance, Oracle determines whether to enforce global name resolution for remote objects accessed in SQL statements based on the value of the initialization parameter GLOBAL_NAMES. This system parameter enables or disables

global name resolution while your instance is running. A setting of `true` enables the enforcement of global names. A setting of `false` disables the enforcement of global names. You can also enable or disable global name resolution for your session with the `GLOBAL_NAMES` parameter of the `ALTER SESSION` statement.

Oracle recommends that you enable global name resolution if you use or plan to use distributed processing.

> **See Also:** "Referring to Objects in Remote Databases" on page 2-90 and *Oracle8i Distributed Database Systems* for more information on global name resolution and how Oracle enforces it

`HASH_MULTIBLOCK_IO_COUNT = integer`

The `HASH_MULTIBLOCK_IO_COUNT` parameter determines the number of data blocks Oracle reads and writes during a hash join operation. The value multiplied by the `DB_BLOCK_SIZE` initialization parameter should not exceed 64K. The default value for this parameter is 1. If the multi-threaded server is used, the value is always 1, and any value given here is ignored.

`HS_AUTOREGISTER = {true | false}`

The `HS_AUTOREGISTER` lets you enable or disable automatic self-registration of non-Oracle system characteristics in the Oracle server's data dictionary by Heterogeneous Services agents.

> **See Also:** *Oracle8i Distributed Database Systems* for more information on accessing non-Oracle systems through Heterogeneous Services

`JOB_QUEUE_PROCESSES = integer`

The `JOB_QUEUE_PROCESSES` determines the number of job queue processes per instance (SNPn, where n is 0 to 9 followed by A to Z). Set this parameter to 1 or higher if you wish to have your snapshots updated automatically. One job queue process is usually sufficient unless you have many snapshots that refresh simultaneously.

Oracle also uses job queue processes to process requests created by the `DBMS_JOB` package.

> **See Also:** *Oracle8i Replication* for more information on managing table snapshots

LICENSE_MAX_SESSIONS = *integer*

The LICENSE_MAX_SESSIONS parameter lets you reset (for the current instance) the value of the initialization parameter LICENSE_MAX_SESSIONS, which establishes the concurrent usage licensing limit, or the limit for concurrent sessions. Once this limit is reached, only users with RESTRICTED SESSION system privilege can connect. A value of 0 disables the limit.

If you reduce the limit on sessions below the current number of sessions, Oracle does not end existing sessions to enforce the new limit. However, users without RESTRICTED SESSION system privilege can begin new sessions only when the number of sessions falls below the new limit.

> **Note:** Do not disable or raise session limits unless you have appropriately upgraded your Oracle license. For more information, contact your Oracle sales representative.

LICENSE_MAX_USERS = *integer*

The LICENSE_MAX_USERS parameter lets you reset (for the current instance) the value of the initialization parameter LICENSE_MAX_USERS, which establishes the limit for users connected to your database. Once this limit is reached, more users cannot connect. A value of 0 disables the limit.

Restriction: You cannot reduce the limit on users below the current number of users created for the database.

> **Note:** Do not disable or raise user limits unless you have appropriately upgraded your Oracle license. For more information, contact your Oracle sales representative.

LICENSE_SESSIONS_WARNING = *integer*

The LICENSE_SESSIONS_WARNING parameter lets you reset (for the current instance) the value of the initialization parameter LICENSE_SESSIONS_WARNING, which establishes a warning threshold for concurrent usage. Once this threshold is reached, Oracle writes warning messages to the database ALERT file for each subsequent session. Also, users with RESTICTED SESSION system privilege receive warning messages when they begin subsequent sessions. A value of 0 disables the warning threshold.

If you reduce the warning threshold for sessions below the current number of sessions, Oracle writes a message to the ALERT file for all subsequent sessions.

LOG_ARCHIVE_DEST = *string*

The LOG_ARCHIVE_DEST parameter lets you specify a valid operating system pathname as the primary destination for all archive redo log file groups.

Restrictions: If you set a value for this parameter:

- You cannot have a value for LOG_ARCHIVE_DEST_n in your initialization parameter file, nor can you set a value for that parameter using the ALTER SESSION or ALTER SYSTEM statement.
- You cannot set a value for the parameter LOG_ARCHIVE_MIN_SUCCEED_DEST using the ALTER SESSION statement.

> **See Also:** The LOG_ARCHIVE_DEST parameter in *Oracle8i Reference* for detailed information on specifying *string*

LOG_ARCHIVE_DEST_*n*

```
LOG_ARCHIVE_DEST_n = {null_string
   | {LOCATION=local_pathname | SERVICE=tnsnames_service}
     [MANDATORY | OPTIONAL] [REOPEN[=integer]]}
```

The LOG_ARCHIVE_DEST_*n* parameter lets you specify up to five valid operating system pathnames or Oracle service names (plus other related options) as destinations for archive redo log file groups (*n* = integers 1 through 5). For a description of the options, refer to *Oracle8i Reference.*

Restrictions: If you set a value for this parameter:

- You cannot have definitions for the parameters LOG_ARCHIVE_DEST or LOG_ARCHIVE_DUPLEX_DEST in your initialization parameter file, nor can you set values for those parameters using the ALTER SYSTEM statement.
- You cannot start archiving to a specific location using the ALTER SYSTEM ARCHIVE LOG TO *location* statement.

> **See Also:** The LOG_ARCHIVE_DEST_*n* parameter in *Oracle8i Reference* for detailed information on specifying pathnames

LOG_ARCHIVE_DEST_STATE_*n* = {enable | defer}

The LOG_ARCHIVE_DEST_STATE_*n* parameter lets you specify the state associated with the corresponding LOG_ARCHIVE_DEST_*n* parameter.

- enable specifies that any associated valid destination can be used for archiving. This is the default.

- `defer` specifies that Oracle will not consider for archiving any destination associated with the corresponding `LOG_ARCHIVE_DEST_`*`n`* parameter.

`LOG_ARCHIVE_DUPLEX_DEST = `*`string`*

The `LOG_ARCHIVE_DUPLEX_DEST` parameter lets you specify a valid operating system pathname as the secondary destination for all archive redo log file groups.

Restriction: If you set a value for this parameter:

- You must have a definition for `LOG_ARCHIVE_DEST`.
- You cannot have a value for the parameter `LOG_ARCHIVE_DEST_n` in your initialization parameter file, nor can you set a value for that parameter using the `ALTER SYSTEM` or `ALTER SESSION` statement.
- You cannot set a value for the parameter `LOG_ARCHIVE_MIN_SUCCEED_DEST` using the `ALTER SESSION` statement.

`LOG_ARCHIVE_MAX_PROCESSES = `*`integer`*

The `LOG_ARCHIVE_MAX_PROCESSES` lets you specify the number of archiver processes that are invoked. Permitted values are integers 1 through 10, inclusive. The default is 1.

`LOG_ARCHIVE_MIN_SUCCEED_DEST = `*`integer`*

The `LOG_ARCHIVE_MIN_SUCCEED_DEST` parameter lets you specify the minimum number of destinations that must succeed in order for the online log file to be available for reuse.

`LOG_ARCHIVE_TRACE` **`= `*`integer`***

The `LOG_ARCHIVE_TRACE` parameter controls the type of output information generated by archivelog processes.

See Also:

- *Oracle8i Backup and Recovery Guide* for more information on using this parameter
- *Oracle8i Reference* for a listing of valid values

`LOG_CHECKPOINT_INTERVAL = `*`integer`*

The `LOG_CHECKPOINT_INTERVAL` lets you limit to *`integer`* the number of redo blocks that can exist between an incremental checkpoint and the last block written to the redo log.

LOG_CHECKPOINT_TIMEOUT = *integer*

The LOG_CHECKPOINT_TIMEOUT parameter lets you limit the incremental checkpoint to be at the position where the last write to the redo log (sometimes called the "tail of the log") was *integer* seconds ago. This parameter signifies that no buffer will remain dirty (in the cache) for more than *integer* seconds. The default is 1800 seconds.

MAX_DUMP_FILE_SIZE = { *size* | 'unlimited'} [deferred]

The MAX_DUMP_FILE_SIZE lets you specify the trace dump file size upper limit for all user sessions. Specify the maximum *size* as either a nonnegative integer that represents the number of blocks, or as 'unlimited'. If you specify 'unlimited', no upper limit is imposed.

Multi-Threaded Server Parameters

When you start your instance, Oracle creates shared server processes and dispatcher processes for the multi-threaded server architecture based on the values of the MTS_SERVERS and MTS_DISPATCHERS initialization parameters. You can set the MTS_SERVERS and MTS_DISPATCHERS session parameters to perform one of the following operations while the instance is running:

- Create additional shared server processes by increasing the minimum number of shared server processes.
- Terminate existing shared server processes after their current calls finish processing.
- Create more dispatcher processes for a specific protocol, up to a maximum across all protocols specified by the initialization parameter MTS_MAX_DISPATCHERS.
- Terminate existing dispatcher processes for a specific protocol after their current user processes disconnect from the instance.

See Also:

- *Oracle8i Concepts*
- *Oracle8i Performance Guide and Reference*
- *Oracle8i Parallel Server Concepts.*

MTS_DISPATCHERS = '*dispatch_clause*'

dispatch_clause::=

```
(PROTOCOL = protocol) |
( ADDRESS = address) |
(DESCRIPTION = description )
[options_clause]
```

options_clause::=

```
(DISPATCHERS = integer |
 SESSIONS = integer |
 CONNECTIONS = integer |
 TICKS = seconds |
 POOL = { 1 | on | yes | true | both |
          ({in|out} = ticks) | 0 | off | no |
          false | ticks} |
 MULTIPLEX = {1 | on | yes | true | 0 | off | no |
              false | both | in | out} |
 LISTENER = tnsname |
 SERVICE = service |
 INDEX = integer)
```

The MTS_DISPATCHERS parameter lets you modify or create the configuration of dispatcher processes. A description of the parameters appears in *Oracle8i Reference*.

You can specify multiple MTS_DISPATCHERS parameters in a single statement for multiple network protocols.

> **See Also:** *Oracle8i Administrator's Guide* for more information on this parameter, see *Net8 Administrator's Guide*

MTS_SERVERS = *integer*

The MTS_SERVERS parameter lets you specify a new minimum number of shared server processes.

OBJECT_CACHE_MAX_SIZE_PERCENT = *integer* deferred

The OBJECT_CACHE_MAX_SIZE_PERCENT parameter lets you specify the percentage of the optimal cache size that the session object cache can grow past the optimal size.

OBJECT_CACHE_OPTIMAL_SIZE = *integer* deferred

The OBJECT_CACHE_OPTIMAL_SIZE parameter lets you specify (in kilobytes) the size to which the session object cache is reduced if it exceeds the maximum size.

OPTIMIZER_MAX_PERMUTATIONS = integer nooverride

The OPTIMIZER_MAX_PERMUTATIONS parameter lets you limit the amount of work the optimizer expends on optimizing queries with large joins. The value of *integer* is the number of permutations of the tables the optimizer will consider with large joins.

nooverride specifies that this system setting will not override the setting for any session in which this parameter was explicitly set.

PARALLEL_ADAPTIVE_MULTI_USER = {true | false}

The PARALLEL_ADAPTIVE_MULTI_USER parameter lets you specify that Oracle should vary the degree of parallelism based on the total perceived load on the system.

PARALLEL_INSTANCE_GROUP = *'text'*

The PARALLEL_INSTANCE_GROUP parameter lets you specify the name of the Oracle Parallel Server instance group to be used for spawning parallel query slaves.

PARALLEL_THREADS_PER_CPU = *integer*

Use the PARALLEL_THREADS_PER_CPU parameter to specify the degree of parallelism for parallel operations where the degree of parallelism is unset. The default is operating system dependent.

PLSQL_V2_COMPATIBILITY = {true | false} [deferred]

Use the PLSQL_V2_COMPATIBILITY parameter to modify the compile-time behavior of PL/SQL programs to allow language constructs that are illegal in Oracle8 and Oracle8*i* (PL/SQL V3), but were legal in Oracle7 (PL/SQL V2).

- Specify true to enable Oracle8*i* PL/SQL V3 programs to execute Oracle7 PL/SQL V2 constructs.
- Specify false to disallow illegal Oracle7 PL/SQL V2 constructs. This is the default.

> **See Also:** *PL/SQL User's Guide and Reference* and *Oracle8i Reference* for more information about this system parameter

QUERY_REWRITE_ENABLED = { true | false } [deferred | nooverride]

The QUERY_REWRITE_ENABLED parameter lets you enable or disable query rewrite on all materialized views that have not been explicitly disabled. By default, true

enables query rewrite for all sessions immediately. Query rewrite is superseded and disabled by rule-based optimization (that is, if the OPTIMIZER_MODE parameter is set to rule). Also enables or disables use of any function-based indexes defined on the materialized view.

> **See Also:** *Oracle8i Data Warehousing Guide* for more information on query rewrite

- deferred specifies that query rewrite is enabled or disabled only for future sessions.
- nooverride specifies that query rewrite is enabled or disabled for all sessions that have not explicitly set this parameter using ALTER SESSION.
- Enabling or disabling query rewrite does not affect queries that have already been compiled, even if they are reissued.
- Enabling or disabling query rewrite does not affect descending indexes.
- A true setting has no effect on materialized views that cannot be created with the ENABLE QUERY REWRITE clause, such as materialized views created totally or in part from a view.

QUERY_REWRITE_INTEGRITY

```
QUERY_REWRITE_INTEGRITY =
   { enforced | trusted | stale_tolerated }
```

The QUERY_REWRITE_INTEGRITY parameter lets you set the minimum consistency level for query rewrite for the duration of the instance. The following values are permitted:

- enforced is the safest level. It relies only on system-enforced relationships so that data integrity and correctness can be guaranteed. This level ensures that query rewrite will not use any function-based index or any materialized view that includes a call to a user-defined function.

  In addition, this level ensures that query rewrite will not use any dimensional information or any constraints enabled with the RELY keyword.
- trusted specifies that materialized views created with the ON PREBUILT TABLE clause are supported, and trusted but unenforced join relationships are accepted. Query rewrite uses join information from dimensions and enables unenforced constraints with the RELY keyword.

- `stale_tolerated` specifies that any stale, usable materialized view may be used.

This parameter does not affect descending indexes.

> **See Also:**
>
> - *Oracle8i Data Warehousing Guide* for more information on query rewrite integrity level
> - CREATE DIMENSION on page 9-34 for information on dimensions
> - *constraint_clause* on page 8-136 for information on constraints enabled with the RELY keyword

`REMOTE_DEPENDENCIES_MODE = {timestamp | signature}`

The `REMOTE_DEPENDENCIES_MODE` paraleter lets you specify how dependencies of remote stored procedures are handled by the server.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals*

`RESOURCE_LIMIT = {true | false}`

When you start an instance, Oracle enforces resource limits assigned to users based on the value of the `RESOURCE_LIMIT` initialization parameter. This system parameter enables or disables resource limits for subsequent sessions. `true` enables resource limits. `false` disables resource limits.

Enabling resource limits only causes Oracle to enforce the resource limits already assigned to users. To choose resource limit values for a user, you must create a profile and assign that profile to the user.

> **See Also:** CREATE PROFILE on page 9-139 and CREATE USER on page 10-99.

`RESOURCE_MANAGER_PLAN = plan_name`

The `RESOURCE_MANAGER_PLAN` parameter lets you specify the name of the resource plan Oracle should use to allocate system resources among resource consumer groups.

> **See Also:** *Oracle8i Administrator's Guide* for information on resource consumer groups and resource plans

SORT_AREA_RETAINED_SIZE = *integer* deferred

The SORT_AREA_RETAINED_SIZE parameter lets you specify (in bytes) the maximum amount of memory that each sort operation will retain after the first fetch is done, until the cursor ends. If you do not explicitly set this parameter in the initialization parameter file or dynamically, Oracle uses the value of the SORT_AREA_SIZE parameter.

SORT_AREA_SIZE = *integer* deferred

The SORT_AREA_SIZE parameter lets you specify (in bytes) the maximum amount of memory to use for each sort operation. The default is operating system dependent.

SORT_MULTIBLOCK_READ_COUNT = *integer* deferred

The SORT_MULTIBLOCK_READ_COUNT parameter lets you specify the number of database blocks to read each time a sort performs a read from temporary segments. The default is 2.

STANDBY_ARCHIVE_DEST = *string*

The STANDBY_ARCHIVE_DEST parameter lets you specify a valid operating system pathname as the standby database destination for the archive redo log files.

TIMED_STATISTICS = {true | false}

The TIMED_STATISTICS parameter lets you specify whether the server requests the time from the operating system when generating time-related statistics. The default is false.

TIMED_OS_STATISTICS = *integer*

The TIMED_OS_STATISTICS lets you specify that operating system statistics will be collected when a request is made from a client to the server or when a request completes.

TRANSACTION_AUDITING = {true | false} deferred

The TRANSACTION_AUDITING parameter lets you specify whether the transaction layer generates a special redo record containing session and user information.

USE_STORED_OUTLINES = { true | false | 'category_name' } [nooverride]

The USE_STORED_OUTLINES parameter determines whether the optimizer will use stored outlines to generate execution plans. USE_STORED_OUTLINES is not an initialization parameter.

- `true` causes the optimizer to use outlines stored in the `DEFAULT` category when compiling requests.
- `false` specifies that the optimizer should not use stored outlines. This is the default.
- *`category_name`* causes the optimizer to use outlines stored in the *`category_name`* category when compiling requests.
- `nooverride` specifies that this system setting will not override the setting for any session in which this parameter was explicitly set. If you do not specify `nooverride`, this setting takes effect in all sessions.

> **Note:** If you want the execution plan to consider materialized views, you must specify them in the outline. If the outline does not use a materialized view, then the query that uses the outline will not use the materialized view either, even if you have enabled query rewrite.

`USER_DUMP_DEST = '`*`directory_name`*`'`

The `USER_DUMP_DEST` parameter lets you specify the pathname where Oracle will write debugging trace files on behalf of a user process.

Examples

Archiving Redo Logs Manually Examples The following statement manually archives the redo log file group with the log sequence number 4 in thread number 3:

```
ALTER SYSTEM ARCHIVE LOG THREAD 3 SEQUENCE 4;
```

The following statement manually archives the redo log file group containing the redo log entry with the SCN 9356083:

```
ALTER SYSTEM ARCHIVE LOG CHANGE 9356083;
```

The following statement manually archives the redo log file group containing a member named '`diskl:log6.log`' to an archived redo log file in the location '`diska:[arch$]`':

```
ALTER SYSTEM ARCHIVE LOG
    LOGFILE 'diskl:log6.log'
    TO 'diska:[arch$]';
```

Enabling Query Rewrite Example This statement enables query rewrite in all sessions for all materialized views that have not been explicitly disabled:

```
ALTER SYSTEM SET QUERY_REWRITE_ENABLED = TRUE;
```

Restricting Session Logons Example You may want to restrict logons if you are performing application maintenance and you want only application developers with RESTRICTED SESSION system privilege to log on. To restrict logons, issue the following statement:

```
ALTER SYSTEM
   ENABLE RESTRICTED SESSION;
```

You can then terminate any existing sessions using the KILL SESSION clause of the ALTER SYSTEM statement.

After performing maintenance on your application, issue the following statement to allow any user with CREATE SESSION system privilege to log on:

```
ALTER SYSTEM
   DISABLE RESTRICTED SESSION;
```

Clearing the Shared Pool Example You might want to clear the shared pool before beginning performance analysis. To clear the shared pool, issue the following statement:

```
ALTER SYSTEM FLUSH SHARED_POOL;
```

Forcing a Checkpoint Example The following statement forces a checkpoint:

```
ALTER SYSTEM CHECKPOINT;
```

Enabling Resource Limits Example This ALTER SYSTEM statement dynamically enables resource limits:

```
ALTER SYSTEM SET RESOURCE_LIMIT = TRUE;
```

Multi-Threaded Server Examples The following statement changes the minimum number of shared server processes to 25:

```
ALTER SYSTEM SET MTS_SERVERS = 25;
```

If there are currently fewer than 25 shared server processes, Oracle creates more. If there are currently more than 25, Oracle terminates some of them when they are finished processing their current calls if the load could be managed by the remaining 25.

The following statement dynamically changes the number of dispatcher processes for the TCP/IP protocol to 5 and the number of dispatcher processes for the DECNet protocol to 10:

```
ALTER SYSTEM
   SET MTS_DISPATCHERS =
      '(INDEX=0)(PROTOCOL=TCP)(DISPATCHERS=5)',
      '(INDEX=1)(PROTOCOL=DECNet)(DISPATCHERS=10)';
```

If there are currently fewer than 5 dispatcher processes for TCP, Oracle creates new ones. If there are currently more than 5, Oracle terminates some of them after the connected users disconnect.

If there are currently fewer than 10 dispatcher processes for DECNet, Oracle creates new ones. If there are currently more than 10, Oracle terminates some of them after the connected users disconnect.

If there are currently existing dispatchers for another protocol, the above statement does not affect the number of dispatchers for that protocol.

Changing Licensing Parameters Examples The following statement dynamically changes the limit on sessions for your instance to 64 and the warning threshold for sessions on your instance to 54:

```
ALTER SYSTEM
   SET LICENSE_MAX_SESSIONS = 64
   LICENSE_SESSIONS_WARNING = 54;
```

If the number of sessions reaches 54, Oracle writes a warning message to the ALERT file for each subsequent session. Also, users with RESTRICTED SESSION system privilege receive warning messages when they begin subsequent sessions.

If the number of sessions reaches 64, only users with RESTRICTED SESSION system privilege can begin new sessions until the number of sessions falls below 64 again.

The following statement dynamically disables the limit for sessions on your instance. After you issue the above statement, Oracle no longer limits the number of sessions on your instance.

```
ALTER SYSTEM SET LICENSE_MAX_SESSIONS = 0;
```

The following statement dynamically changes the limit on the number of users in the database to 200. After you issue the above statement, Oracle prevents the number of users in the database from exceeding 200.

```
ALTER SYSTEM SET LICENSE_MAX_USERS = 200;
```

Forcing a Log Switch Example You may want to force a log switch to drop or rename the current redo log file group or one of its members, because you cannot drop or rename a file while Oracle is writing to it. The forced log switch affects only your instance's redo log thread. The following statement forces a log switch:

```
ALTER SYSTEM
   SWITCH LOGFILE;
```

Enabling Distributed Recovery Example The following statement enables distributed recovery:

```
ALTER SYSTEM ENABLE DISTRIBUTED RECOVERY;
```

You may want to disable distributed recovery for demonstration or testing purposes.You can disable distributed recovery in both single-process and multiprocess mode with the following statement:

```
ALTER SYSTEM DISABLE DISTRIBUTED RECOVERY;
```

When your demonstration or testing are complete, you can then enable distributed recovery again by issuing an ALTER SYSTEM statement with the ENABLE DISTRIBUTED RECOVERY clause.

Killing a Session Example You may want to kill the session of a user that is holding resources needed by other users. The user receives an error message indicating that the session has been killed. That user can no longer make calls to the database without beginning a new session. Consider this data from the V$SESSION dynamic performance table:

```
SELECT sid, serial#, username
FROM v$session

  SID   SERIAL# USERNAME
----- --------- ----------------
    1         1
    2         1
    3         1
    4         1
    5         1
    7         1
    8        28 OPS$BQUIGLEY
   10       211 OPS$SWIFT
   11        39 OPS$OBRIEN
   12        13 SYSTEM
   13         8 SCOTT
```

The following statement kills the session of the user scott using the SID and SERIAL# values from V$SESSION:

```
ALTER SYSTEM KILL SESSION '13, 8';
```

Disconnecting a Session Example The following statement disconnects user scott's session, using the SID and SERIAL# values from V$SESSION:

```
ALTER SYSTEM DISCONNECT SESSION '13, 8' POST_TRANSACTION;
```

> **See Also:** *Oracle8i Parallel Server Concepts* and *Oracle8i Performance Guide and Reference* for more information about application failover

8

SQL Statements: ALTER TABLE to *constraint_clause*

This chapter contains the following SQL statements:

- ALTER TABLE
- ALTER TABLESPACE
- ALTER TRIGGER
- ALTER TYPE
- ALTER USER
- ALTER VIEW
- ANALYZE
- ASSOCIATE STATISTICS
- AUDIT
- CALL
- COMMENT
- COMMIT
- constraint_clause

ALTER TABLE

Purpose

Use the ALTER TABLE statement to alter the definition of a nonpartitioned table, a partitioned table, a table partition, or a table subpartition.

■

Prerequisites

The table must be in your own schema, or you must have ALTER privilege on the table, or you must have ALTER ANY TABLE system privilege. For some operations you may also need the CREATE ANY INDEX privilege.

Additional Prerequisites for Partitioning Operations In addition, if you are not the owner of the table, you need the DROP ANY TABLE privilege in order to use the *drop_partition_clause* or *truncate_partition_clause*.

You must also have space quota in the tablespace in which space is to be acquired in order to use the *add_partition_clause*, *modify_partition_clause*, *move_partition_clause*, and *split_partition_clause*.

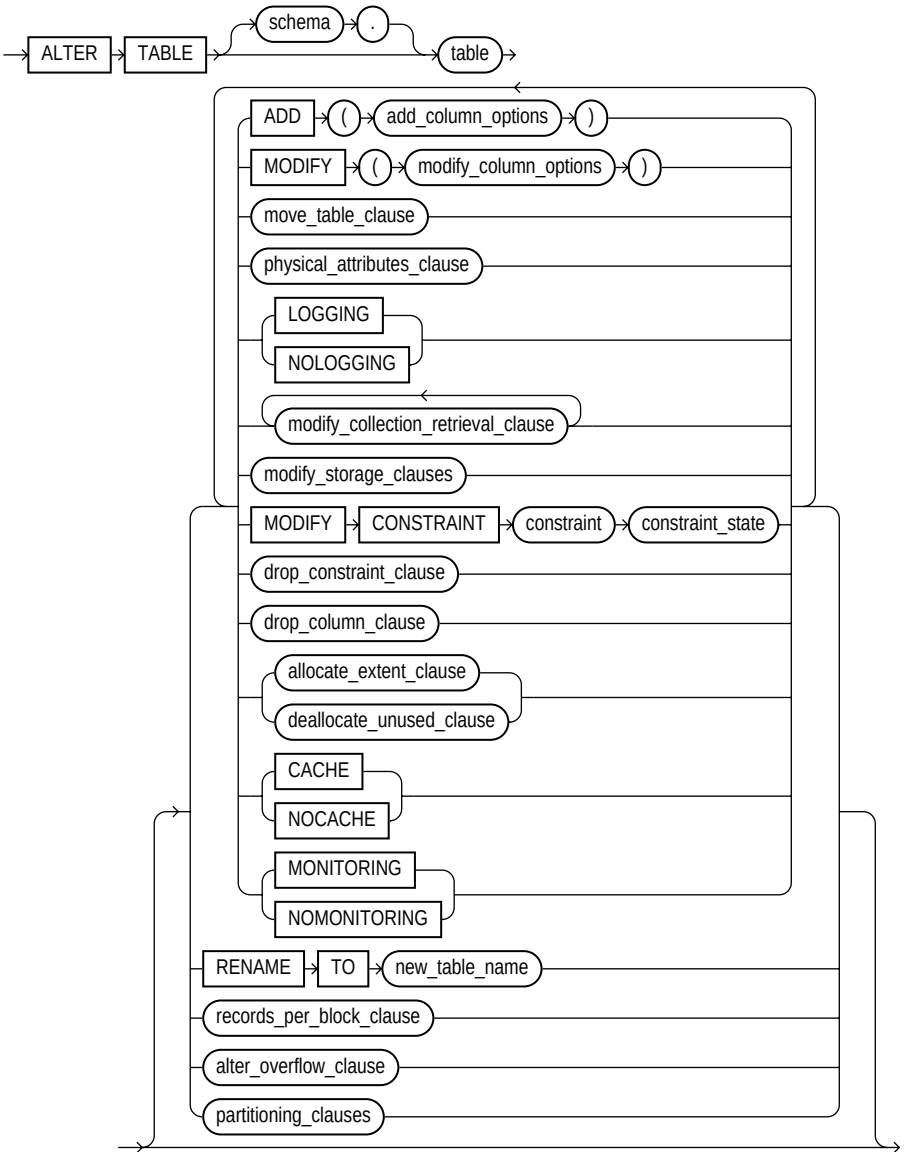
Additional Prerequisites for Constraints and Triggers To enable a UNIQUE or PRIMARY KEY constraint, you must have the privileges necessary to create an index on the table. You need these privileges because Oracle creates an index on the columns of the unique or primary key in the schema containing the table.

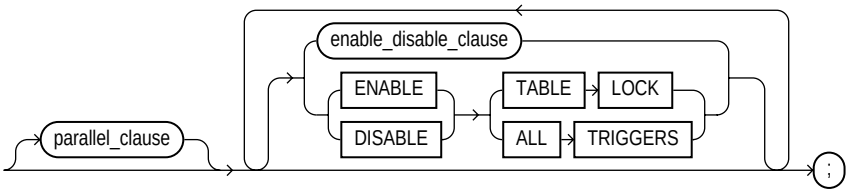
To enable or disable triggers, the triggers must be in your schema or you must have the ALTER ANY TRIGGER system privilege.

Additional Prerequisites When Using Object Types To use an object type in a column definition when modifying a table, either that object must belong to the same schema as the table being altered, or you must have either the EXECUTE ANY TYPE system privilege or the EXECUTE schema object privilege for the object type.

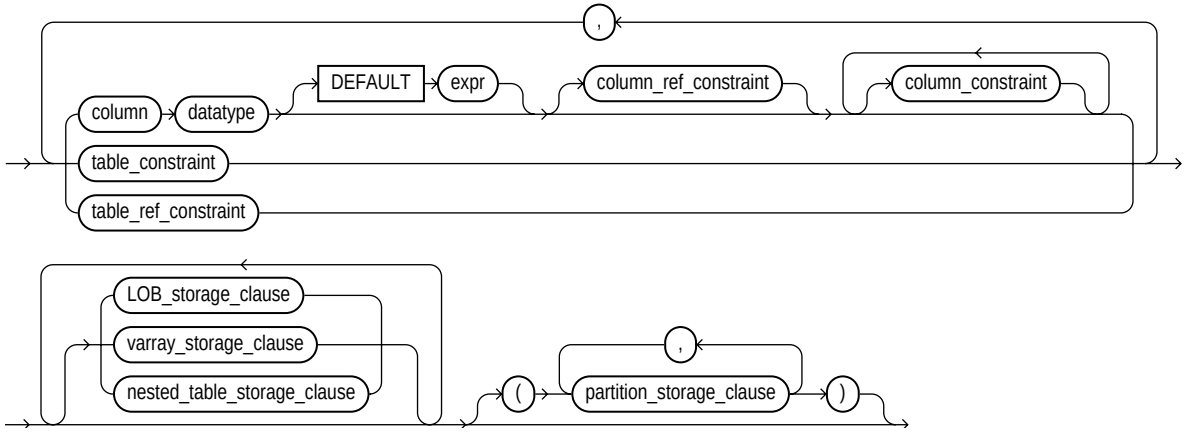
> **See Also:** CREATE INDEX on page 9-52 for information on the privileges needed to create indexes

Syntax


ALTER
TABLE
schema
.
table
ADD
(
add_column_options
)
MODIFY
(
modify_column_options
)
move_table_clause
physical_attributes_clause
LOGGING
NOLOGGING
modify_collection_retrieval_clause
modify_storage_clauses
MODIFY
CONSTRAINT
constraint
constraint_state
drop_constraint_clause
drop_column_clause
allocate_extent_clause
deallocate_unused_clause
CACHE
NOCACHE
MONITORING
NOMONITORING
RENAME
TO
new_table_name
records_per_block_clause
alter_overflow_clause
partitioning_clauses

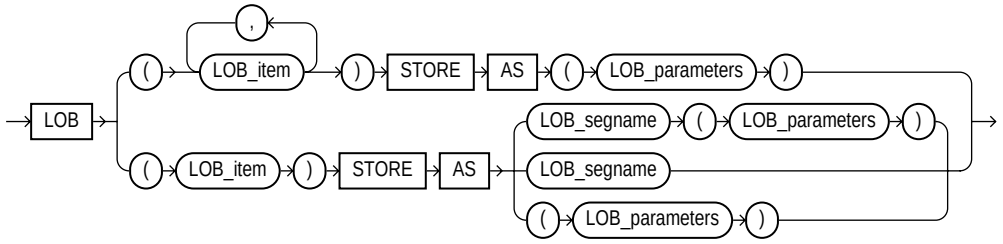


add_column_options::=

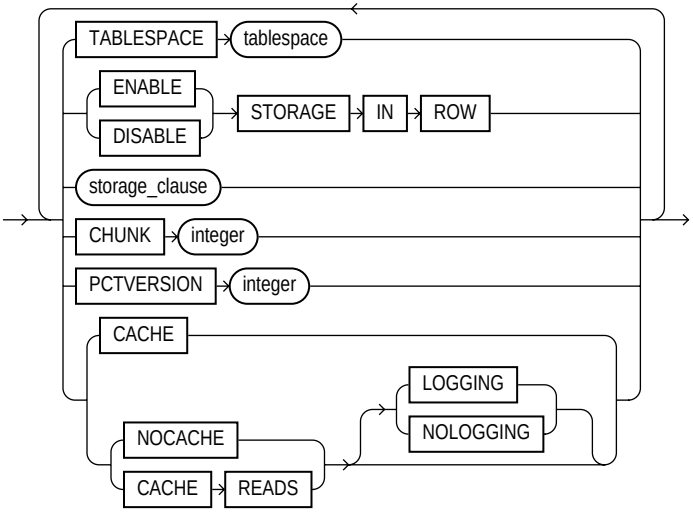


column_constraint, **table_constraint**, **column_ref_constraint**, **table_ref_constraint**, **constraint_state**: See the constraint_clause on page 8-136.

LOB_storage_clause::=

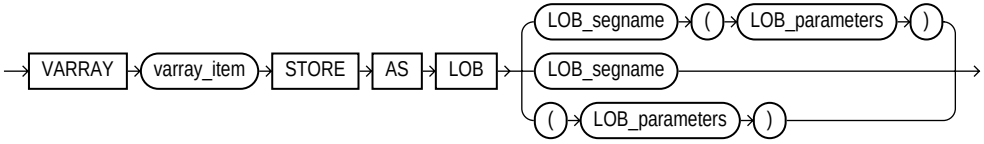


LOB_parameters::=

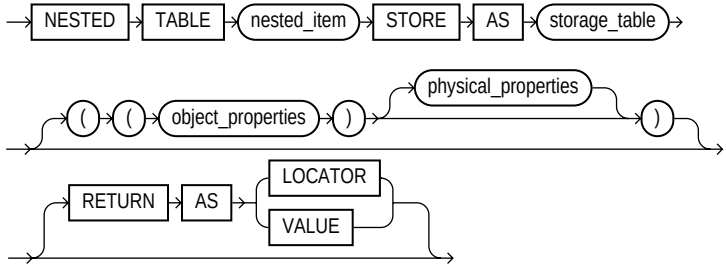


storage_clause: See storage_clause on page 11-129.

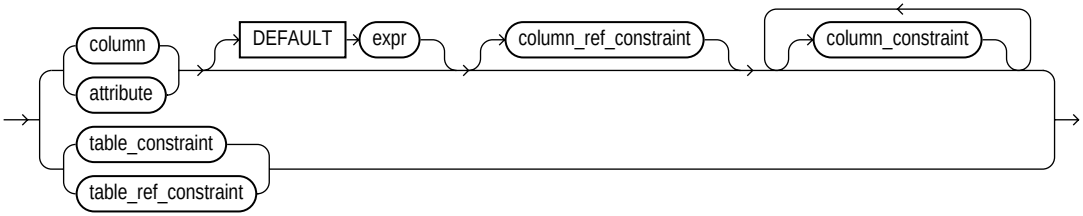
varray_storage_clause::=



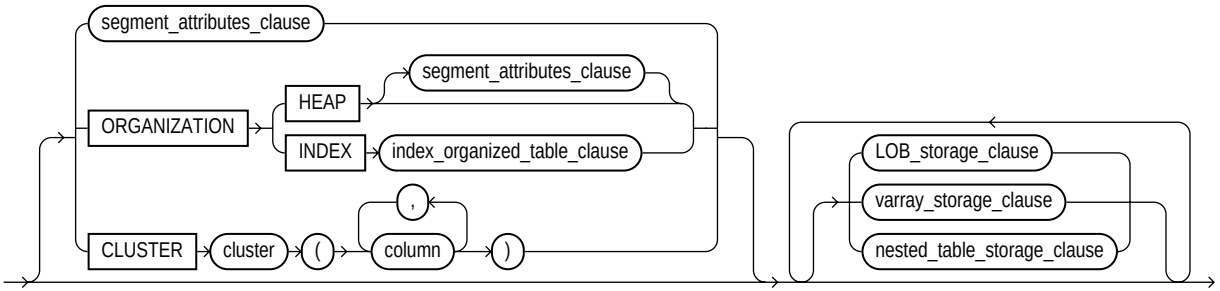
nested_table_storage_clause::=



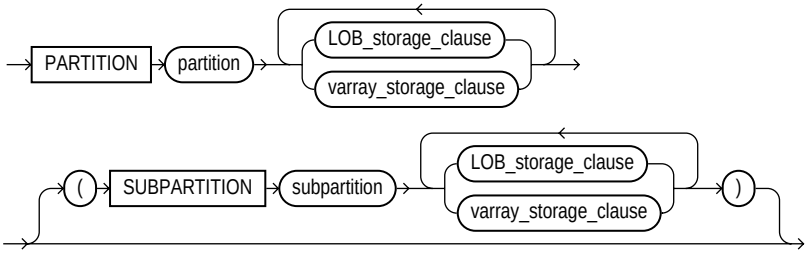
object_properties::=



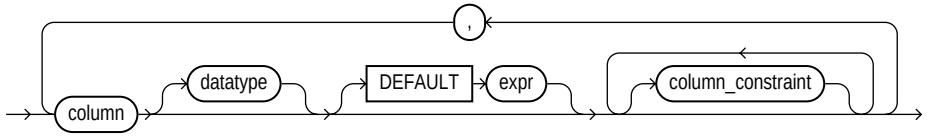
physical_properties::=



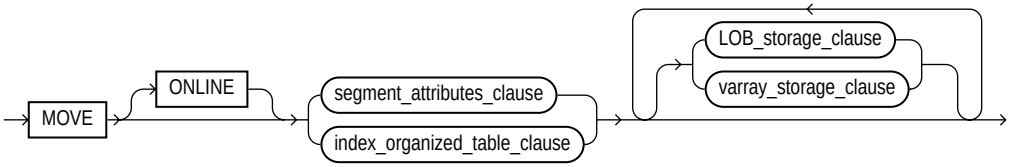
partition_storage_clause::=



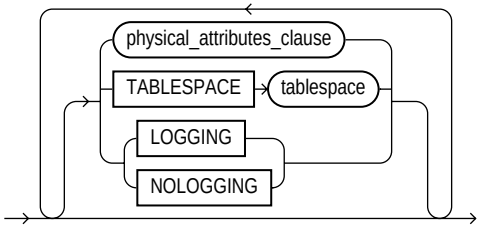
modify_column_options::=



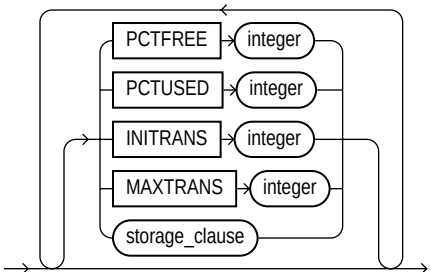
move_table_clause::=



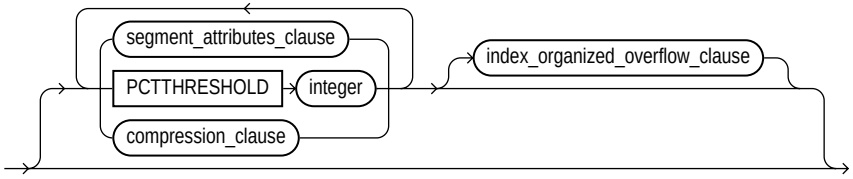
segment_attributes_clause::=



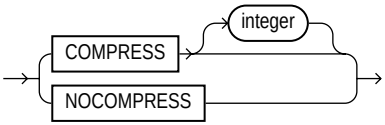
physical_attributes_clause::=



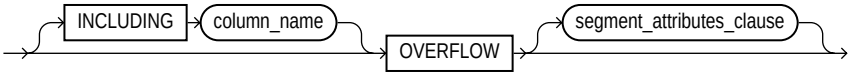
index_organized_table_clause::=



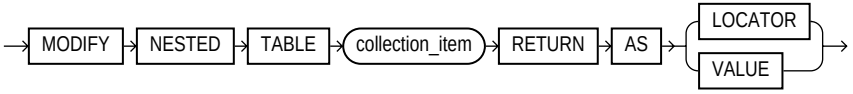
compression_clause::=



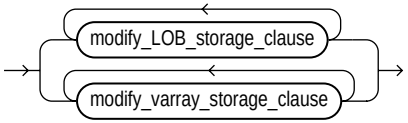
index_organized_overflow_clause::=



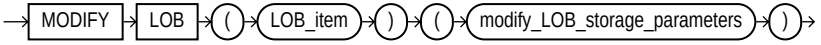
modify_collection_retrieval_clause::=



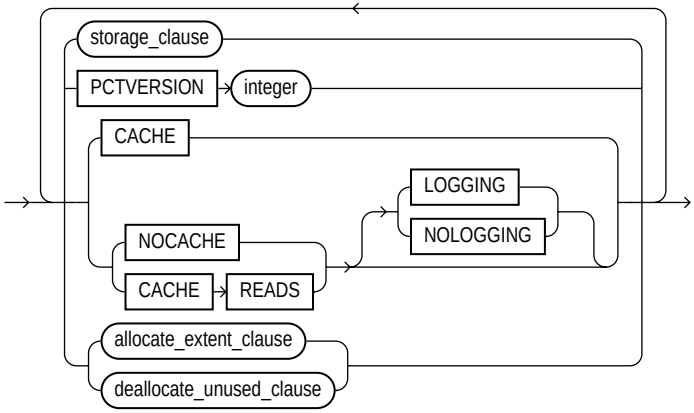
modify_storage_clauses::=



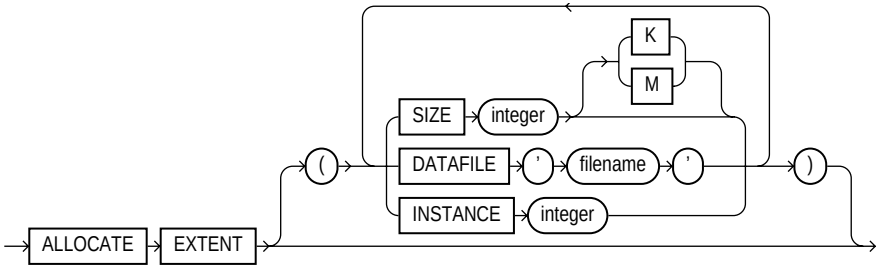
modify_LOB_storage_clause::=



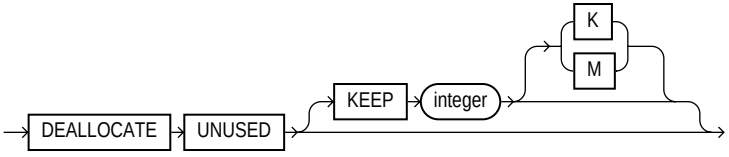
modify_LOB_storage_parameters::=



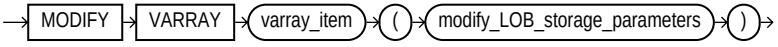
allocate_extent_clause::=



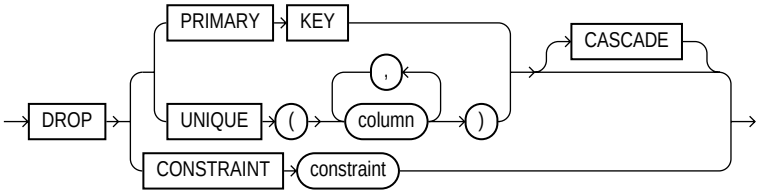
deallocate_unused_clause::=



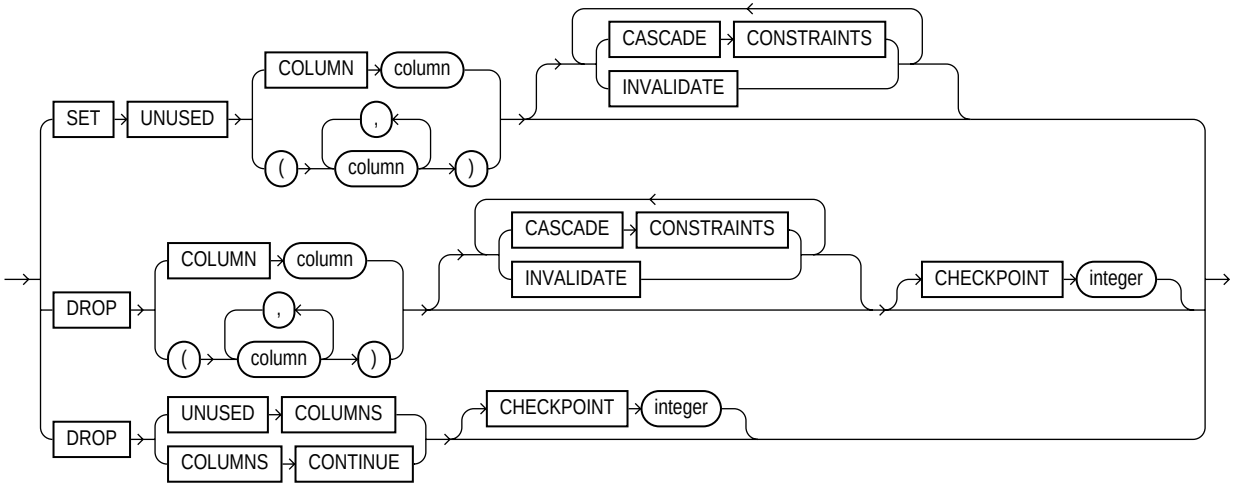
modify_varray_storage_clause::=



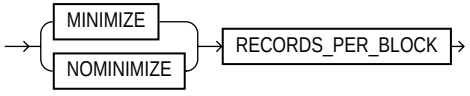
drop_constraint_clause::=



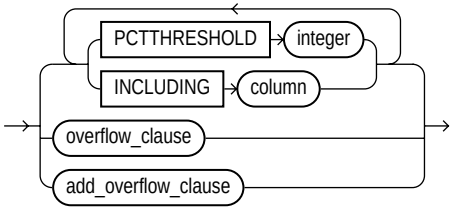
drop_column_clause::=



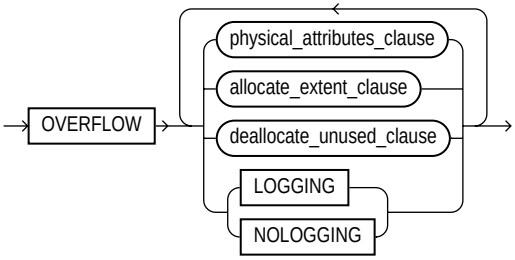
records_per_block_clause::=



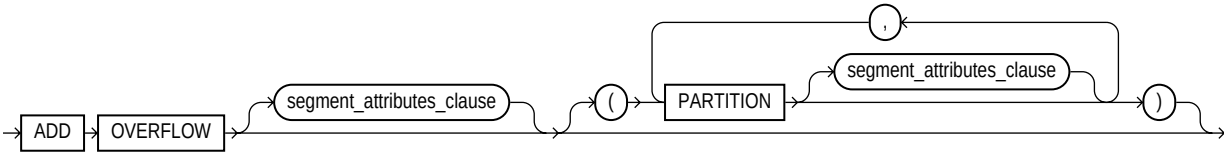
alter_overflow_clause::=



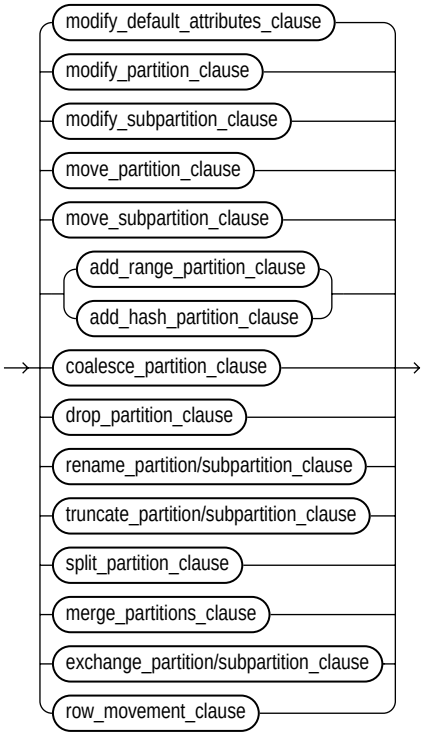
overflow_clause::=



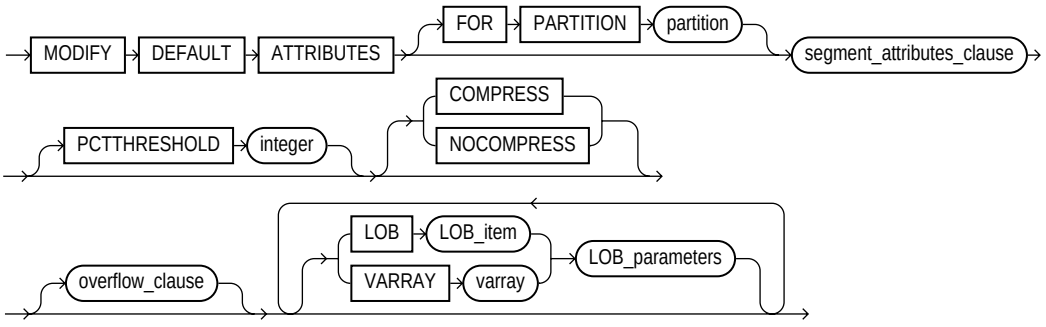
add_overflow_clause::=



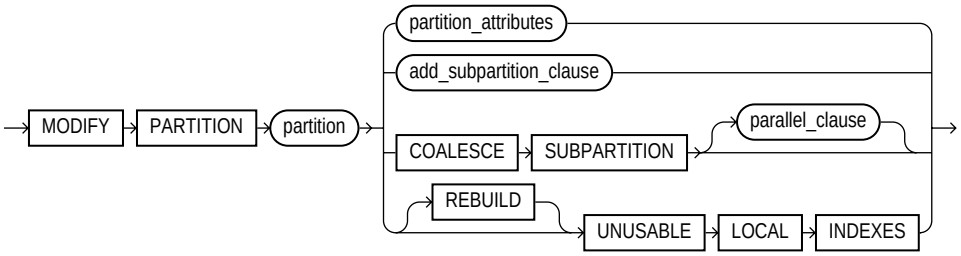
partitioning_clauses::=



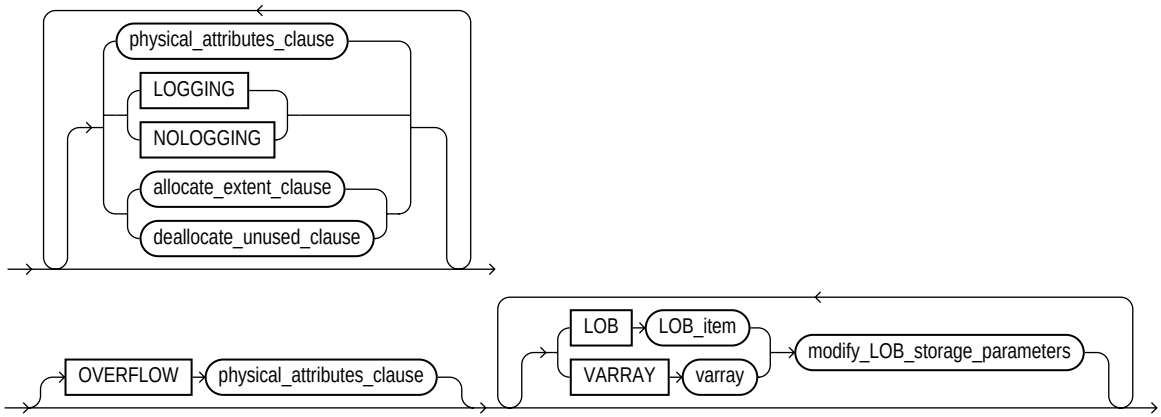
modify_default_attributes_clause::=



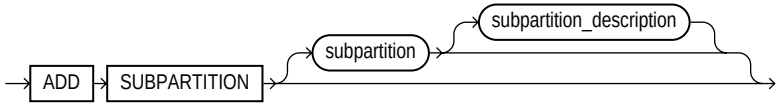
modify_partition_clause::=



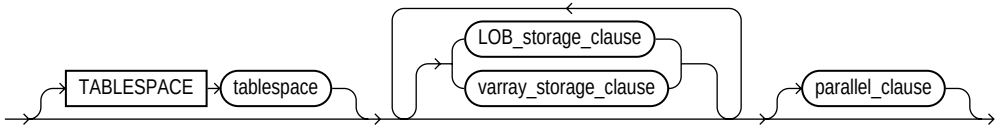
partition_attributes::=



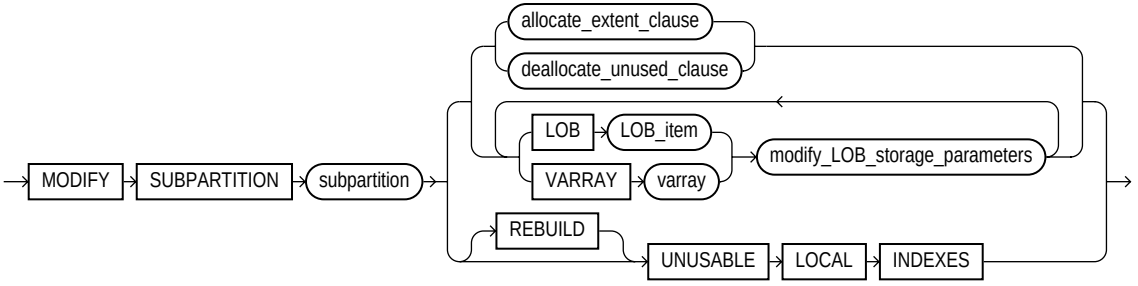
add_subpartition_clause::=



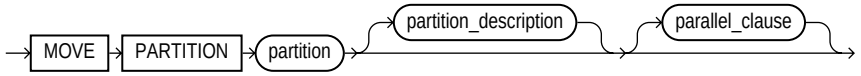
subpartition_description::=



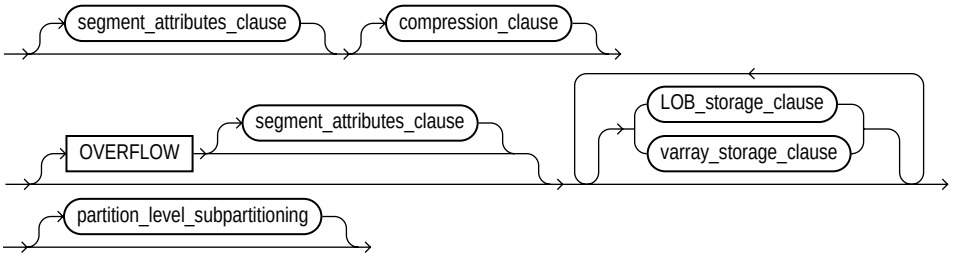
modify_subpartition_clause::=



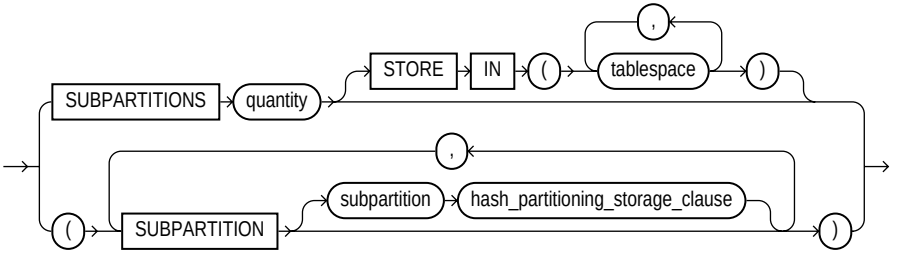
move_partition_clause::=



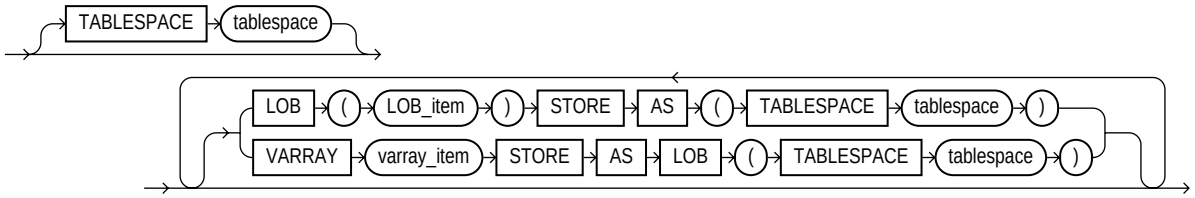
table_partition_description::=



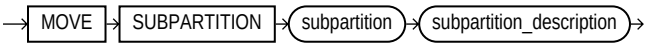
partition_level_subpartitioning::=



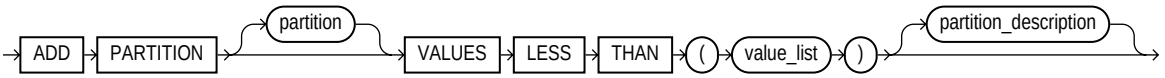
hash_partitioning_storage_clause::=



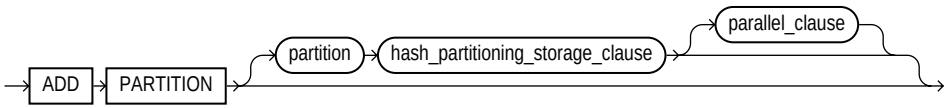
move_subpartition_clause::=



add_range_partition_clause::=



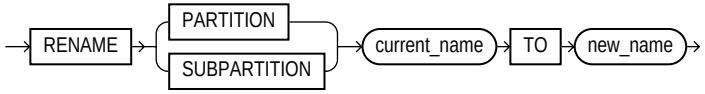
add_hash_partition_clause::=



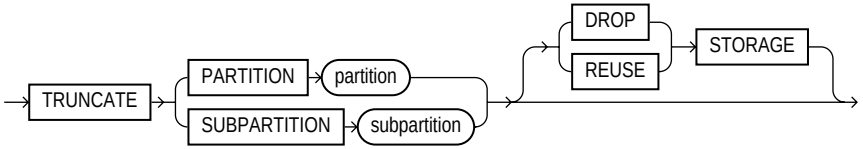
coalesce_partition_clause::=

drop_partition_clause::=

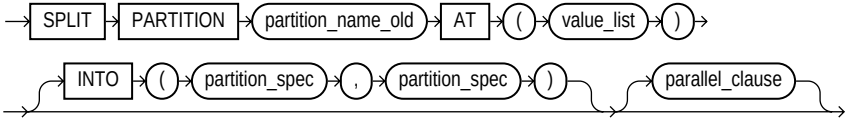
rename_partition/ subpartition_clause::=



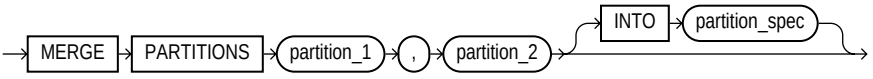
truncate_partition_clause and truncate_subpartition_clause::=



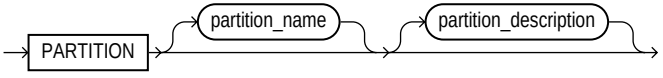
split_partition_clause::=



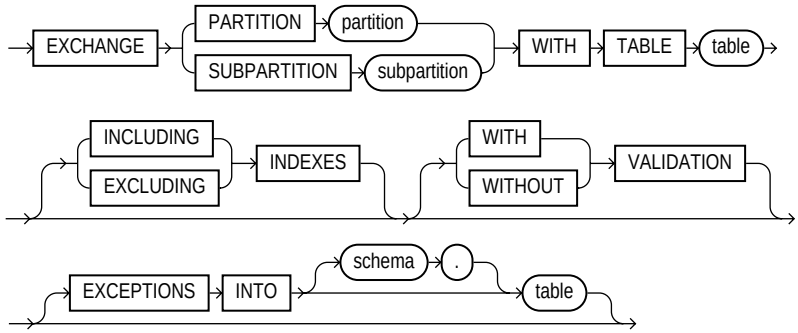
merge_partitions_clause::=



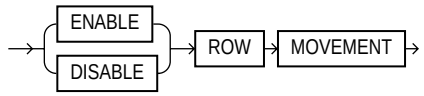
partition_spec::=



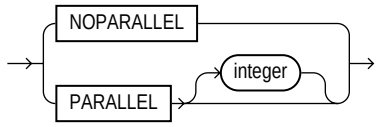
exchange_partition_clause and exchange_subpartition_clause::=



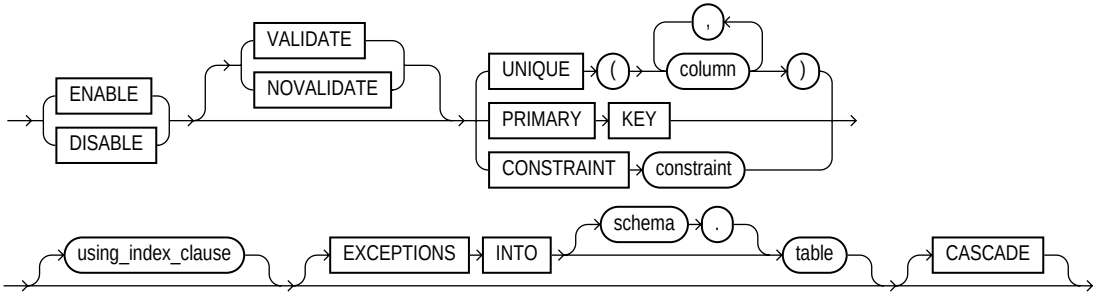
row_movement_clause::=



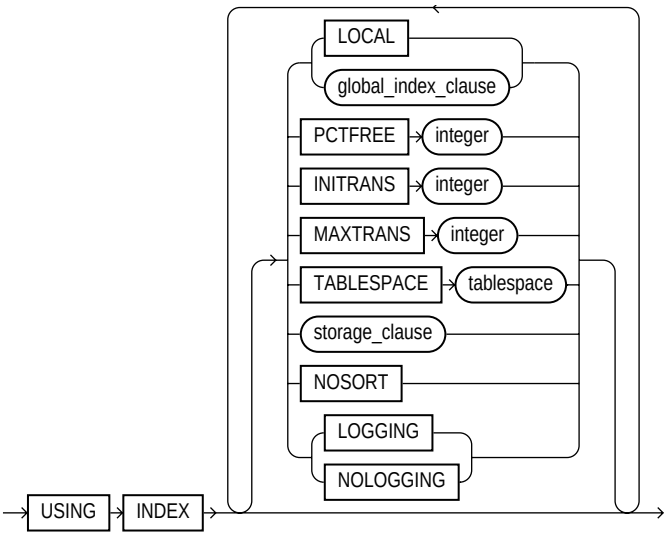
parallel_clause::=



enable_disable_clause::=



using_index_clause::=



Keywords and Parameters

The clauses described below have specialized meaning in the ALTER TABLE statement. For descriptions of the remaining keywords, see CREATE TABLE on page 10-7.

> **Note:** Operations performed by the ALTER TABLE statement can cause Oracle to invalidate procedures and stored functions that access the table. For information on how and when Oracle invalidates such objects, see *Oracle8i Concepts*.

schema

Specify the schema containing the table. If you omit *schema*, Oracle assumes the table is in your own schema.

table

Specify the name of the table to be altered.

You can modify, or drop columns from, or rename a temporary table. However, for a temporary table, you cannot:

- Add columns of nested-table or varray type. You can add columns of other types.
- Specify referential integrity (foreign key) constraints for an added or modified column
- Specify the following clauses of the *LOB_storage_clause* for an added or modified LOB column: TABLESPACE, *storage_clause*, LOGGING or NOLOGGING, or the *LOB_index_clause*.
- Specify the *physical_attribute_clause*, *nested_table_storage_clause*, *parallel_clause*, *allocate_extent_clause*, *deallocate_unused_clause*, or any of the *index_organized_table* clauses
- Exchange partitions between a partition and a temporary table
- Specify LOGGING or NOLOGGING
- Specify MOVE

Note: If you alter a table that is a master table for one or more materialized views, the materialized views are marked INVALID. Invalid materialized views cannot be used by query rewrite and cannot be refreshed. To revalidate a materialized view, see ALTER MATERIALIZED VIEW on page 7-61.

See Also: *Oracle8i Data Warehousing Guide* for more information on materialized views in general.

add_column_options

ADD *add_column_options* lets you add a column or integrity constraint.

See Also: CREATE TABLE on page 10-7 for a description of the keywords and parameters of this clause

If you add a column, the initial value of each row for the new column is null unless you specify the DEFAULT clause. In this case, Oracle updates each row in the new column with the value you specify for DEFAULT. This update operation, in turn, fires any AFTER UPDATE triggers defined on the table.

You can add an overflow data segment to each partition of a partitioned index-organized table.

You can add LOB columns to nonpartitioned and partitioned tables. You can specify LOB storage at the table and at the partition or subpartition level.

If you previously created a view with a query that used the "SELECT *" syntax to select all columns from table, and you now add a column to *table*, Oracle does not automatically add the new column to the view. To add the new column to the view, re-create the view using the CREATE VIEW statement with the OR REPLACE clause.

> **See Also:** CREATE VIEW on page 10-105

Restrictions:

- You cannot add a LOB column to a partitioned index-organized table. (This restriction does not apply to nonpartitioned index-organized tables.)
- You cannot add a column with a NOT NULL constraint if *table* has any rows unless you also specify the DEFAULT clause.
- If you specify this clause for an index-organized table, you cannot specify any other clauses in the same statement.

DEFAULT — Use the DEFAULT clause to specify a default for a new column or a new default for an existing column. Oracle assigns this value to the column if a subsequent INSERT statement omits a value for the column. If you are adding a new column to the table and specify the default value, Oracle inserts the default column value into all rows of the table.

The datatype of the default value must match the datatype specified for the column. The column must also be long enough to hold the default value. A DEFAULT expression cannot contain references to other columns, the pseudocolumns CURRVAL, NEXTVAL, LEVEL, and ROWNUM, or date constants that are not fully specified.

table_ref_constraint and *column_ref_constraint* — These clauses let you further describe a column of type REF. The only difference between these clauses is that you specify *table_ref* from the table level, so you must identify the REF column or attribute you are defining. You specify *column_ref* after you have already identified the REF column or attribute.

> **See Also:** *constraint_clause* on page 8-136 for syntax and description of these constraints, including restrictions

column_constraint — Use *column_constraint* to add or remove a NOT NULL constraint to or from an existing column. You cannot use this clause to modify any other type of constraint using ALTER TABLE.

See Also: *constraint_clause* on page 8-136

table_constraint — Use *table_constraint* to add or modify an integrity constraint on the table.

See Also: *constraint_clause* on page 8-136

LOB_storage_clause

Use the *LOB_storage_clause* to specify the LOB storage characteristics for the newly added LOB column. You cannot use this clause to modify an existing LOB column. Instead, you must use the *modify_LOB_storage_clause*.

Restrictions:

- The only parameter of *LOB_parameters* you can specify for a hash partition or hash subpartition is TABLESPACE.
- You cannot specify the *LOB_index_clause* if *table* is partitioned.

lob_item — Specify the LOB column name or LOB object attribute for which you are explicitly defining tablespace and storage characteristics that are different from those of the table.

lob_segname — Specify the name of the LOB data segment. You cannot use *lob_segname* if more than one *lob_item* is specified.

ENABLE | DISABLE STORAGE IN ROW — Specify whether the LOB value is to be stored in the row (inline) or outside of the row. (The LOB locator is always stored in the row regardless of where the LOB value is stored.)

- ENABLE specifies that the LOB value is stored inline if its length is less than approximately 4000 bytes minus system control information. This is the default.
- DISABLE specifies that the LOB value is stored outside of the row regardless of the length of the LOB value.

Restriction: You cannot change STORAGE IN ROW once it is set. Therefore, you cannot specify this clause as part of the *modify_column_options* clause. However, you can change this setting when adding a new column (*add_column_options*) or when moving the table (*move_table_clause*).

CHUNK *integer*

Specify the number of bytes to be allocated for LOB manipulation. If *integer* is not a multiple of the database block size, Oracle rounds up (in bytes) to the next multiple. For example, if the database block size is 2048 and *integer* is 2050, Oracle allocates 4096 bytes (2 blocks).The maximum value is 32768 (32 K), which is the largest Oracle block size allowed. The default CHUNK size is one Oracle database block.

You cannot change the value of CHUNK once it is set.

> **Note:** The value of CHUNK must be less than or equal to the value of NEXT (either the default value or that specified in the storage clause). If CHUNK exceeds the value of NEXT, Oracle returns an error.

PCTVERSION *integer*

Specify the maximum percentage of overall LOB storage space to be used for creating new versions of the LOB. The default value is 10, meaning that older versions of the LOB data are not overwritten until 10% of the overall LOB storage space is used.

LOB_index_clause

This clause is deprecated as of Oracle8*i*. Oracle generates an index for each LOB column. The LOB indexes are system named and system managed, and reside in the same tablespace as the LOB data segments.

It is still possible for you to specify this clause in some cases. However, Oracle Corporation strongly recommends that you no longer do so. In any event, do not put the LOB index in a different tablespace from the LOB data.

> **See Also:** *Oracle8i Migration* for information on how Oracle manages LOB indexes in tables migrated from earlier versions

varray_storage_clause

The *varray_storage_clause* lets you specify separate storage characteristics for the LOB in which a varray will be stored. In addition, if you specify this clause, Oracle will always store the varray in a LOB, even if it is small enough to be stored inline.

Restriction: You cannot specify the TABLESPACE clause of *LOB_parameters* as part of this clause. The LOB tablespace for a varray defaults to the containing table's tablespace.

nested_table_storage_clause

the *nested_table_storage_clause* lets you specify separate storage characteristics for a nested table, which in turn enables you to define the nested table as an index-organized table. You must include this clause when creating a table with columns or column attributes whose type is a nested table. (Clauses within this clause that function the same way they function for parent object tables are not repeated here.)

Restrictions:

- You cannot specify the *parallel_clause*.
- You cannot specify TABLESPACE (as part of the *segment_attributes_clause*) for a nested table. The tablespace is always that of the parent table.

nested_item	Specify the name of a column (or a top-level attribute of the table's object type) whose type is a nested table.
storage_table	Specify the name of the table where the rows of *nested_item* reside. The storage table is created in the same schema and the same tablespace as the parent table.

partition_storage_clause

The *partition_storage_clause* lets you specify a separate *LOB_storage_clause* or *varray_storage_clause* for each partition. You must specify the partitions in the order of partition position.

If you do not specify a *LOB_storage_clause* or *varray_storage_clause* for a particular partition, the storage characteristics are those specified for the LOB item at the table level. If you also did not specify any storage characteristics at the table level for the LOB item, Oracle stores the LOB data partition in the same tablespace as the table partition to which it corresponds.

Restriction: You can specify only one list of *partition_storage_clauses* per ALTER TABLE statement, and all *LOB_storage_clauses* and *varray_storage_clauses* must precede the list of *partition_storage_clauses*.

modify_column_options

Use MODIFY *modify_column_options* to modify the definition of an existing column. If you omit any of the optional parts of the column definition (datatype, default value, or column constraint), these parts remain unchanged.

- You can change a CHAR column to VARCHAR2 (or VARCHAR) and a VARCHAR2 (or VARCHAR) to CHAR only if the column contains nulls in all rows or if you do not attempt to change the column size.
- You can change any column's datatype or decrease any column's size if all rows for the column contain nulls.
- You can always increase the size of a character or raw column or the precision of a numeric column, whether or not all the columns contain nulls.

Restrictions:

- You cannot modify the datatype or length of a column that is part of a table or index partitioning or subpartitioning key.
- You cannot modify the definition of a column on which a domain index has been built.
- If you specify this clause for an index-organized table, you cannot specify any other clauses in the same statement.

column Specify the name of the column to be added or modified.

The only type of integrity constraint that you can add to an existing column using the MODIFY clause with the column constraint syntax is a NOT NULL constraint, and only if the column contains no nulls. To define other types of integrity constraints (UNIQUE, PRIMARY KEY, referential integrity, and CHECK constraints) on existing columns, using the ADD clause and the table constraint syntax.

datatype Specify a new datatype for an existing column.

You can omit the datatype only if the statement also designates the column as part of the foreign key of a referential integrity constraint. Oracle automatically assigns the column the same datatype as the corresponding column of the referenced key of the referential integrity constraint.

If you change the datatype of a column in a materialized view container table, the corresponding materialized view is invalidated.

> **See Also:** ALTER MATERIALIZED VIEW on page 7-61 for information on revalidating a materialized view

Restrictions:

- You cannot specify a column of datatype ROWID for an index-organized table, but you can specify a column of type UROWID.
- You cannot change a column's datatype to LOB or REF.

MODIFY CONSTRAINT *constraint*

MODIFY CONSTRAINT lets you change the state of an existing constraint named *constraint*.

See Also: *constraint_clause* on page 8-136 for a description of all the keywords and parameters of *constraint_state*

move_table_clause

For a **heap-organized table**, use the *segment_attributes_clause* of the syntax. The *move_table_clause* lets you relocate data of a nonpartitioned table into a new segment, optionally in a different tablespace, and optionally modify any of its storage attributes.

You can also move any LOB data segments associated with the table using the *LOB_storage_clause*. (LOB items not specified in this clause are not moved.)

For an **index-organized table**, use the *index_organized_table_clause* of the syntax. The *move_table_clause* rebuilds the index-organized table's primary key index B*-tree. The overflow data segment is not rebuilt unless the OVERFLOW keyword is explicitly stated, with two exceptions:

- If you alter the values of PCTTHRESHOLD or the INCLUDING column as part of this ALTER TABLE statement, the overflow data segment is rebuilt.
- If any of out-of-line columns (LOBs, varrays, nested table columns) in the index-organized table are moved explicitly, then the overflow data segment is also rebuilt.

The index and data segments of LOB columns are not rebuilt unless you specify the LOB columns explicitly as part of this ALTER TABLE statement.

`ONLINE` Specify `ONLINE` if you want DML operations on the index-organized table to be allowed during rebuilding of the table's primary key index B*-tree.

Restrictions:

- You can specify this clause only for a nonpartitioned index-organized table.
- Parallel DML is not supported during online `MOVE`. If you specify `ONLINE` and then issue parallel DML statements, Oracle returns an error.

`compression_clause` Use the *`compression_clause`* to enable and disable key compression in an index-organized table.

- `COMPRESS` enables key compression, which eliminates repeated occurrence of primary key column values in index-organized tables. Use *`integer`* to specify the prefix length (number of prefix columns to compress).

  The valid range of prefix length values is from 1 to the number of primary key columns minus 1. The default prefix length is the number of primary key columns minus 1.

  Restrictions:

  - You can specify this clause only for an index-organized table.

  - You can specify compression for a partition of an index-organized table only if compression has been specified at the table level.
- `NOCOMPRESS` disables key compression in index-organized tables. This is the default.

`TABLESPACE` *`tablespace`* Specify the tablespace into which the rebuilt index-organized table is stored.

Restrictions on the *move_table_clause*:

- If you specify `MOVE`, it must be the first clause. For an index-organized table, the only clauses outside this clause that are allowed are the *`physical_attribute_clause`* and the *`parallel_clause`*. For heap-organized tables, you can specify those two clauses and the *`LOB_storage_clauses`*.
- You cannot `MOVE` an entire partitioned table (either heap or index organized). You must move individual partitions or subpartitions.

See Also: move_partition_clause on page 8-44 and move_subpartition_clause on page 8-45

Notes regarding LOBs:

For any LOB columns you specify in a *move_table_clause*:

- Oracle drops the old LOB data segment and corresponding index segment and creates new segments, even if you do not specify a new tablespace.
- If the LOB index in *table* resided in a different tablespace from the LOB data, Oracle collocates the LOB index with the LOB data in the LOB data's tablespace after the move.

---

physical_attributes_clause

The *physical_attributes_clause* lets you change the value of PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and storage characteristics.

Restriction: You cannot specify the PCTUSED parameter for the index segment of an index-organized table.

See Also: The PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters of CREATE TABLE on page 10-7 and the *storage_clause* on page 11-129

Cautions:

- For a nonpartitioned table, the values you specify override any values specified for the table at create time.
- For a range- or hash-partitioned table, the values you specify are the default values for the table and the actual values for every existing partition, overriding any values already set for the partitions. To change default table attributes without overriding existing partition values, use the *modify_default_attributes_clause*.
- For a composite-partitioned table, the values you specify are the default values for the table and all partitions of the table and the actual values for all subpartitions of the table, overriding any values already set for the subpartitions. To change default partition attributes without overriding existing subpartition values, use the *modify_default_attributes_clause* with the FOR PARTITION clause.

modify_collection_retrieval_clause

Use the *modify_collection_retrieval_clause* to change what is returned when a collection item is retrieved from the database.

collection_item — Specify the name of a column-qualified attribute whose type is nested table or varray.

RETURN AS — Specify what Oracle should return as the result of a query.

- LOCATOR specifies that a unique locator for the nested table is returned.
- VALUE specifies that a copy of the nested table itself is returned.

modify_storage_clauses

modify_LOB_storage_clause

The *modify_LOB_storage_clause* lets you change the physical attributes of the LOB *lob_item*. You can specify only one *lob_item* for each *modify_LOB_storage_clause*.

Restrictions:

- You cannot modify the value of the INITIAL parameter in the *storage_clause* when modifying the LOB storage attributes.
- You cannot specify both the *allocate_extent_clause* and the *deallocate_unused_clause* in the same statement.

modify_varray_storage_clause

The *modify_varray_storage_clause* lets you change the storage characteristics of an existing LOB in which a varray is stored.

Restriction: You cannot specify the TABLESPACE clause of *LOB_parameters* as part of this clause. The LOB tablespace for a varray defaults to the containing table's tablespace.

drop_constraint_clause

The *drop_constraint_clause* lets you drop an integrity constraint from the database. Oracle stops enforcing the constraint and removes it from the data dictionary. You can specify only one constraint for each *drop_constraint_clause*, but you can specify multiple *drop_constraint_clauses* in one statement.

PRIMARY KEY — Specify PRIMARY KEY to drop the table's PRIMARY KEY constraint.

UNIQUE — Specify UNIQUE to drop the UNIQUE constraint on the specified columns.

CONSTRAINT *constraint* — Specify the integrity constraint you want dropped.

CASCADE — Specify CASCADE if you want all other integrity constraints that depend on the dropped integrity constraint to be dropped as well.

Restrictions on the *drop_constraint_clause*:

- You cannot drop a UNIQUE or PRIMARY KEY constraint that is part of a referential integrity constraint without also dropping the foreign key. To drop the referenced key and the foreign key together, use the CASCADE clause. If you

omit CASCADE, Oracle does not drop the PRIMARY KEY or UNIQUE constraint if any foreign key references it.

- You cannot drop a primary key constraint (even with the CASCADE clause) on a table that uses the primary key as its object identifier (OID).
- If you drop a referential integrity constraint on a REF column, the REF column remains scoped to the referenced table.
- You cannot drop the scope of the column.

drop_column_clause

The *drop_column_clause* lets you free space in the database by dropping columns you no longer need, or by marking them to be dropped at a future time when the demand on system resources is less.

- If you drop a nested table column, its storage table is removed.
- If you drop a LOB column, the LOB data and its corresponding LOB index segment are removed.
- If you drop a BFILE column, only the locators stored in that column are removed, not the files referenced by the locators.
- If you drop (or mark unused) a column defined as an INCLUDING column, the column stored immediately before this column will become the new INCLUDING column.

SET UNUSED

Use SET UNUSED to mark one or more columns as unused. Specifying this clause does not actually remove the target columns from each row in the table (that is, it does not restore the disk space used by these columns). Therefore, the response time is faster than it would be if you execute the DROP clause.

You can view all tables with columns marked as unused in the data dictionary views USER_UNUSED_COL_TABS, DBA_UNUSED_COL_TABS, and ALL_UNUSED_COL_TABS.

See Also: *Oracle8i Reference* for information on these views

Unused columns are treated as if they were dropped, even though their column data remains in the table's rows. After a column has been marked as unused, you have no access to that column. A "SELECT *" query will not retrieve data from unused columns. In addition, the names and types of columns marked unused will not be displayed during a DESCRIBE, and you can add to the table a new column with the same name as an unused column.

Note: Until you actually drop these columns, they continue to count toward the absolute limit of 1000 columns per table. However, as with all DDL statements, you cannot roll back the results of this clause. That is, you cannot issue SET USED counterpart to retrieve a column that you have SET UNUSED.

Also, if you mark a column of datatype LONG as UNUSED, you cannot add another LONG column to the table until you actually drop the unused LONG column.

See Also: CREATE TABLE on page 10-7 for more information on the 1000 column limit

DROP

Specify DROP to remove the column descriptor and the data associated with the target column from each row in the table. If you explicitly drop a particular column, all columns currently marked as unused in the target table are dropped at the same time.

When the column data is dropped:

- All indexes defined on any of the target columns are also dropped.
- All constraints that reference a target column are removed.
- If any statistics types are associated with the target columns, Oracle disassociates the statistics from the column with the FORCE option and drops any statistics collected using the statistics type.

  See Also: DISASSOCIATE STATISTICS on page 10-123 for more information on disassociating statistics types

> **Note:** If the target column is a parent key of a nontarget column, or if a check constraint references both the target and nontarget columns, Oracle returns an error and does not drop the column unless you have specified the CASCADE CONSTRAINTS clause. If you have specified that clause, Oracle removes all constraints that reference any of the target columns.

DROP UNUSED COLUMNS

Specify DROP UNUSED COLUMNS to remove from the table all columns currently marked as unused. Use this statement when you want to reclaim the extra disk space from unused columns in the table. If the table contains no unused columns, the statement returns with no errors.

column

Specify one or more columns to be set as unused or dropped. Use the COLUMN keyword only if you are specifying only one column. If you specify a column list, it cannot contain duplicates.

CASCADE CONSTRAINTS

Specify CASCADE CONSTRAINTS if you want to drop all referential integrity constraints that refer to the primary and unique keys defined on the dropped columns, and drop all multicolumn constraints defined on the dropped columns. If any constraint is referenced by columns from other tables or remaining columns in the target table, then you must specify CASCADE CONSTRAINTS. Otherwise, the statement aborts and an error is returned.

INVALIDATE

> **Note:** Currently, Oracle executes this clause regardless of whether you specify the keyword INVALIDATE.

Oracle invalidates all dependent objects, such as views, triggers, and stored program units. Object invalidation is a recursive process. Therefore, all directly dependent and indirectly dependent objects are invalidated. However, only local dependencies are invalidated, because Oracle manages remote dependencies differently from local dependencies.

An object invalidated by this statement is automatically revalidated when next referenced. You must then correct any errors that exist in that object before referencing it.

> **See Also:** *Oracle8i Concepts* for more information on dependencies

CHECKPOINT — Specify CHECKPOINT if you want Oracle to apply a checkpoint for the drop column operation after processing *integer* rows; *integer* is optional and must be greater than zero. If *integer* is greater than the number of rows in the table, Oracle applies a checkpoint after all the rows have been processed. If you do not specify *integer*, Oracle sets the default of 512.

Checkpointing cuts down the amount of undo logs accumulated during the drop column operation to avoid running out of rollback segment space. However, if this statement is interrupted after a checkpoint has been applied, the table remains in an unusable state. While the table is unusable, the only operations allowed on it are DROP TABLE, TRUNCATE TABLE, and ALTER TABLE DROP COLUMNS CONTINUE (described below).

You cannot use this clause with SET UNUSED, because that clause does not remove column data.

DROP COLUMNS CONTINUE — Specify DROP COLUMNS CONTINUE to continue the drop column operation from the point at which it was interrupted. Submitting this statement while the table is in a valid state results in an error.

Restrictions on the *drop_column_clause*:

- Each of the parts of this clause can be specified only once in the statement and cannot be mixed with any other ALTER TABLE clauses. For example, the following statements are not allowed:

```
ALTER TABLE t1 DROP COLUMN f1 DROP (f2);
ALTER TABLE t1 DROP COLUMN f1 SET UNUSED (f2);
ALTER TABLE t1 DROP (f1) ADD (f2 NUMBER);
ALTER TABLE t1 SET UNUSED (f3)
   ADD (CONSTRAINT ck1 CHECK (f2 > 0));
```

- You can drop an object type column only as an entity. Dropping an attribute from an object type column is not allowed.
- You can drop a column from an index-organized table only if it is not a primary key column. The primary key constraint of an index-organized table can never be dropped, so you cannot drop a primary key column even if you have specified CASCADE CONSTRAINTS.
- You can export tables with dropped or unused columns. However, you can import a table only if all the columns specified in the export files are present in the table (that is, none of those columns has been dropped or marked unused). Otherwise, Oracle returns an error.

- You cannot drop a column on which a domain index has been built.
- You cannot use this clause to drop:
  - A pseudocolumn, clustered column, or partitioning column. (You can drop nonpartitioning columns from a partitioned table if all the tablespaces where the partitions were created are online and in read-write mode.)
  - A column from a nested table, an object table, or a table owned by SYS

allocate_extent_clause

The *allocate_extent_clause* lets you explicitly allocates a new extent for the table, the partition or subpartition, the overflow data segment, the LOB data segment, or the LOB index.

Restriction: You cannot allocate an extent for a range- or composite-partitioned table.

> **Note:** Explicitly allocating an extent with this clause does affect the size for the next extent to be allocated as specified by the NEXT and PCTINCREASE storage parameters.

SIZE *integer*	Specify the size of the extent in bytes. Use *K* or *M* to specify the extent size in kilobytes or megabytes. If you omit this parameter, Oracle determines the size based on the values of the STORAGE parameters of the table's overflow data segment or of the LOB index.
DATAFILE *'filename'*	Specify one of the datafiles in the tablespace of the table, overflow data segment, LOB data tablespace, or LOB index to contain the new extent. If you omit this parameter, Oracle chooses the datafile.
INSTANCE *integer*	Specifying INSTANCE *integer* makes the new extent available to the freelist group associated with the specified instance. If the instance number exceeds the maximum number of freelist groups, the former is divided by the latter, and the remainder is used to identify the freelist group to be used. An instance is identified by the value of its initialization parameter INSTANCE_NUMBER. If you omit this parameter, the space is allocated to the table, but is not drawn from any particular freelist group. Rather, the master freelist is used, and space is allocated as needed.

> **Note:** Use this parameter only if you are using Oracle with the Parallel Server option in parallel mode.

See Also: *Oracle8i Concepts*

deallocate_unused_clause

Use the *deallocate_unused_clause* to explicitly deallocate unused space at the end of the table, partition or subpartition, overflow data segment, LOB data segment, or LOB index and makes the space available for other segments in the tablespace. You can free only unused space above the high water mark (that is, the point beyond which database blocks have not yet been formatted to receive data).

Oracle credits the amount of the released space to the user quota for the tablespace in which the deallocation occurs.

Oracle deallocates unused space from the end of the object toward the high water mark at the beginning of the object. If an extent is completely contained in the deallocation, then the whole extent is freed for reuse. If an extent is partially contained in the deallocation, then the used part up to the high water mark becomes the extent, and the remaining unused space is freed for reuse.

The exact amount of space freed depends on the values of the INITIAL, MINEXTENTS, and NEXT parameters.

See Also: *storage_clause* on page 11-129 for a description of these parameters

KEEP *integer* Specify the number of bytes above the high water mark that the table, overflow data segment, LOB data segment, or LOB index is to have after deallocation.

- If you omit KEEP and the high water mark is above the size of INITIAL and MINEXTENTS, then all unused space above the high water mark is freed. When the high water mark is less than the size of INITIAL or MINEXTENTS, then all unused space above MINEXTENTS is freed.

- If you specify KEEP, then the specified amount of space is kept and the remaining space is freed. When the remaining number of extents is less than MINEXTENTS, then MINEXTENTS is adjusted to the new number of extents. If the initial extent becomes smaller than INITIAL, then INITIAL is adjusted to the new size.
- In either case, NEXT is set to the size of the last extent that was deallocated.

CACHE | NOCACHE

CACHE — For data that is accessed frequently, this clause indicates that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables.

As a parameter in the *LOB_storage_clause*, CACHE specifies that Oracle places LOB data values in the buffer cache for faster access.

Restriction: You cannot specify CACHE for an index-organized table. However, index-organized tables implicitly provide CACHE behavior.

NOCACHE — For data that is not accessed frequently, this clause indicates that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed.

As a parameter in the *LOB_storage_clause*, NOCACHE specifies that the LOB value is either not brought into the buffer cache or brought into the buffer cache and placed at the least recently used end of the LRU list. (The latter is the default behavior.) NOCACHE is the default for LOB storage.

Restriction: You cannot specify NOCACHE for index-organized tables.

CACHE READS — CACHE READS applies only to LOB storage. It indicates that LOB values are brought into the buffer cache only during read operations, but not during write operations.

- When you add a new LOB column, you can specify the logging attribute with CACHE READS, as you can when defining a LOB column at create time.
- When you modify a LOB column from CACHE or NOCACHE to CACHE READS, or from CACHE READS to CACHE or NOCACHE, you can change the logging attribute. If you do not specify the LOGGING or NOLOGGING, this attribute defaults to the current logging attribute of the LOB column.

For existing LOBs, if you do not specify CACHE, NOCACHE, or CACHE READS, Oracle retains the existing values of the LOB attributes.

MONITORING | NOMONITORING

MONITORING — Specify MONITORING if you want Oracle to collect modification statistics on *table*. These statistics are estimates of the number of rows affected by DML statements over a particular period of time. They are available for use by the optimizer or for analysis by the user.

> **See Also:** *Oracle8i Performance Guide and Reference* for more information on using this clause

NOMONITORING — Specify NOMONITORING if you do not want Oracle to collect modification statistics on *table*.

Restriction: You cannot specify MONITORING or NOMONITORING for a temporary table.

LOGGING | NOLOGGING

LOGGING | NOLOGGING — Specify whether subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against a nonpartitioned table, table partition, all partitions of a partitioned table, or all subpartitions of a partition will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file.

When used with the *modify_default_attributes_clause*, this clause affects the logging attribute of a partitioned table.

LOGGING | NOLOGGING also specifies whether ALTER TABLE ... MOVE and ALTER TABLE ... SPLIT operations will be logged or not logged.

For a table or table partition, if you omit LOGGING | NOLOGGING, the logging attribute of the table or table partition defaults to the logging attribute of the tablespace in which it resides.

For LOBs, if you omit LOGGING | NOLOGGING,

- If you specify CACHE, then LOGGING is used (because you cannot have CACHE NOLOGGING).
- If you specify NOCACHE or CACHE READS, the logging attribute defaults to the logging attribute of the tablespace in which it resides.

NOLOGGING does not apply to LOBs that are stored inline with row data. That is, if you specify NOLOGGING for LOBs with values less than 4000 bytes and you have not disabled STORAGE IN ROW, Oracle ignores the NOLOGGING specification and treats the LOB data the same as other table data.

In NOLOGGING mode, data is modified with minimal logging (to mark new extents invalid and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not logged. Therefore, if you cannot afford to lose this table, it is important to take a backup after the NOLOGGING operation.

If the database is run in ARCHIVELOG mode, media recovery from a backup taken before the LOGGING operation will restore the table. However, media recovery from a backup taken before the NOLOGGING operation will not restore the table.

The logging attribute of the base table is independent of that of its indexes.

> **See Also:** *Oracle8i Parallel Server Concepts* for more information about the *logging_clause* and parallel DML

RENAME TO

RENAME TO — Use the RENAME clause to rename *table* to *new_table_name*.

Note: Using this clause will invalidate any dependent materialized views.

See Also: CREATE MATERIALIZED VIEW on page 9-88 and *Oracle8i Data Warehousing Guide* for more information on materialized views

records_per_block_clause

The *records_per_block_clause* lets you specify whether Oracle restricts the number of records that can be stored in a block. This clause ensures that any bitmap indexes subsequently created on the table will be as small (compressed) as possible.

Restrictions:

- You cannot specify either MINIMIZE or NOMINIMIZE if a bitmap index has already been defined on table. You must first drop the bitmap index.
- You cannot specify this clause for an index-organized table or nested table

MINIMIZE — Specify MINIMIZE to instruct Oracle to calculate the largest number of records in any block in the table, and limit future inserts so that no block can contain more than that number of records.

Restriction: You cannot specify MINIMIZE for an empty table.

NOMINIMIZE — Specify NOMINIMIZE to disable the MINIMIZE feature. This is the default.

alter_overflow_clause

The *alter_overflow_clause* lets you change the definition of an index-organized table. Index-organized tables keep data sorted on the primary key and are therefore best suited for primary-key-based access and manipulation.

Note: When you alter an index-organized table, Oracle evaluates the maximum size of each column to estimate the largest possible row. If an overflow segment is needed but you have not specified OVERFLOW, Oracle raises an error and does not execute the ALTER TABLE statement. This checking function guarantees that subsequent DML operations on the index-organized table will not fail because an overflow segment is lacking.

PCTTHRESHOLD *integer*

Specify the percentage of space reserved in the index block for an index-organized table row. All trailing columns of a row, starting with the column that causes the specified threshold to be exceeded, are stored in the overflow segment. PCTTHRESHOLD must be a value from 1 to 50. If you do not specify PCTTHRESHOLD, the default is 50.

Restrictions:

- PCTTHRESHOLD must be large enough to hold the primary key.
- You cannot specify PCTTHRESHOLD for individual partitions of an index-organized table.

See Also: the INCLUDING clause of the *index_organized_table_clause*

INCLUDING *column_name*

Specify a column at which to divide an index-organized table row into index and overflow portions. The primary key columns are always stored in the index. *column_name* can be either the last primary-key column or any non-primary-key column. All non-primary-key columns that follow *column_name* are stored in the overflow data segment.

Restriction: You cannot specify this clause for individual partitions of an index-organized table.

Note: If an attempt to divide a row at *column_name* causes the size of the index portion of the row to exceed the PCTTHRESHOLD value (either specified or default), Oracle breaks up the row based on the PCTTHRESHOLD value.

---

overflow_clause

The *overflow_clause* lets you specify the overflow data segment physical storage and logging attributes to be modified for the index-organized table. Parameters specified in this clause are applicable only to the overflow data segment.

Restriction: You cannot specify OVERFLOW for a partition of a partitioned index-organized table unless the table already has an overflow segment.

See Also: CREATE TABLE on page 10-7

add_overflow_clause

The *add_overflow_clause* lets you add an overflow data segment to the specified index-organized table.

For a partitioned index-organized table:

- If you do not specify PARTITION, Oracle automatically allocates an overflow segment for each partition. The physical attributes of these segments are inherited from the table level.
- If you wish to specify separate physical attributes for one or more partitions, you must specify such attributes for **every** partition in the table. You do not specify the name of the partitions, but you must specify their attributes in the order in which they were created.

You can find the order of the partitions by querying the PARTITION_NAME and PARTITION_POSITION columns of the USER_IND_PARTITIONS view.

If you do not specify TABLESPACE for a particular partition, Oracle uses the tablespace specified for the table. If you do not specify TABLESPACE at the table level, Oracle uses the tablespace of the partition's primary key index segment.

partitioning_clauses

The following clauses apply only to partitioned tables. You cannot combine partition operations with other partition operations or with operations on the base table in one ALTER TABLE statement.

Note: If you drop, exchange, truncate, move, modify, or split a partition on a table that is a master table for one or more materialized views, existing bulk load information about the table will be deleted. Therefore, be sure to refresh all dependent materialized views before performing any of these operations.

---

modify_default_attributes_clause

The *modify_default_attributes_clause* lets you specify new default values for the attributes of *table*. Partitions and LOB partitions you create subsequently will inherit these values unless you override them explicitly when creating the partition or LOB partition. Existing partitions and LOB partitions are not affected by this clause.

Only attributes named in the statement are affected, and the default values specified are overridden by any attributes specified at the individual partition level.

FOR PARTITION — FOR PARTITION applies only to composite-partitioned tables. This clause specifies new default values for the attributes of *partition*. Subpartitions and LOB subpartitions of *partition* that you create subsequently will inherit these values, unless you override them explicitly when creating the subpartition or LOB subpartition. Existing subpartitions are not affected by this clause.

Restrictions:

- The PCTTHRESHOLD, COMPRESS, *physical_attributes_clause*, and *overflow_clause* are valid only for partitioned index-organized tables.
- You cannot specify the PCTUSED parameter for the index segment of an index-organized table.
- You can specify COMPRESS only if compression is already specified at the table level.

modify_partition_clause

The *modify_partition_clause* lets you change the real physical attributes of the *partition* table partition. Optionally modifies the storage attributes of one or more LOB items for the partition. You can specify new values for any of the following physical attributes for the partition: the logging attribute; PCTFREE, PCTUSED, INITRANS, or MAXTRANS parameter; or storage parameters.

If *table* is composite-partitioned:

- If you specify the *allocate_extent_clause*, Oracle will allocate an extent for each subpartition of *partition*.
- If you specify *deallocate_unused_clause*, Oracle will deallocate unused storage from each subpartition of *partition*.
- Any other attributes changed in this clause will be changed in subpartitions of *partition* as well, overriding existing values. To avoid changing the attributes of existing subpartitions, use the FOR PARTITION clause of the *modify_default_attributes_clause*.

Restriction: If *table* is hash partitioned, you can specify only the *allocate_extent* and *deallocate_unused* clauses. All other attributes of the partition are inherited from the table-level defaults except TABLESPACE, which stays the same as it was at create time.

add_subpartition_clause

The *add_subpartition_clause* lets you add a hash subpartition to *partition*. Oracle populates the new subpartition with rows rehashed from the other subpartition(s) of *partition* as determined by the hash function.

Oracle marks UNUSABLE, and you must rebuild, the local index subpartitions corresponding to the added and to the rehashed subpartitions.

If you do not specify *subpartition*, Oracle assigns a name in the form SYS_SUBP*nnn*

If you do not specify TABLESPACE, the new subpartition will reside in the default tablespace of *partition*.

COALESCE SUBPARTITION

Specify COALESCE PARTITION if you want Oracle to select a hash subpartition, distribute its contents into one or more remaining subpartitions (determined by the hash function), and then drop the selected subpartition.

Local index subpartitions corresponding to the selected subpartition are also dropped. Oracle marks UNUSABLE, and you must rebuild, the index subpartitions corresponding to one or more absorbing subpartitions.

UNUSABLE LOCAL INDEXES

The next two clauses modify the attributes of local **index partitions** corresponding to *partition*.

- UNUSABLE LOCAL INDEXES marks UNUSABLE all the local index partitions associated with *partition*.
- REBUILD UNUSABLE LOCAL INDEXES rebuilds the unusable local index partitions associated with *partition*.

Restrictions:

- You cannot specify this clause with any other clauses of the *modify_partition_clause*.
- You cannot specify this clause for partitions that are subpartitioned.

modify_subpartition_clause

The *modify_subpartition_clause* lets you allocate or deallocate storage for an individual subpartition of *table*.

Restriction: The only *modify_LOB_storage_parameters* you can specify for subpartition are the *allocate_extent_clause* and *deallocate_unused_clause*.

- UNUSABLE LOCAL INDEXES marks UNUSABLE all the local index subpartitions associated with *subpartition*.
- REBUILD UNUSABLE LOCAL INDEXES rebuilds the unusable local index subpartitions associated with *subpartition*.

rename_partition/ subpartition_clause

Use the *rename_partition_clause* or *rename_subpartition_clause* to rename a table partition or subpartition *current_name* to *new_name*. For both partitions and subpartitions, *new_name* must be different from all existing partitions and subpartitions of the same table.

move_partition_clause

Use the *move_partition_clause* to move table partition *partition* to another segment. You can move partition data to another tablespace, recluster data to reduce fragmentation, or change create-time physical attributes.

If the table contains LOB columns, you can use the *LOB_storage_clause* to move the LOB data and LOB index segments associated with this partition. Only the LOBs named are affected. If you do not specify the *LOB_storage_clause* for a particular LOB column, its LOB data and LOB index segments are not moved.

If *partition* is not empty, MOVE PARTITION marks UNUSABLE all corresponding local index partitions and all global nonpartitioned indexes, and all the partitions of global partitioned indexes.

When you move a LOB data segment, Oracle drops the old data segment and corresponding index segment and creates new segments even if you do not specify a new tablespace.

The move operation obtains its parallel attribute from the *parallel_clause*, if specified. If not specified, the default parallel attributes of the table, if any, are used. If neither is specified, Oracle performs the move without using parallelism.

The *parallel_clause* on MOVE PARTITION does not change the default parallel attributes of *table*.

Note: For index-organized tables, Oracle uses the address of the primary key, as well as its value, to construct logical rowids. The logical rowids are stored in the secondary index of the table. If you move a partition of an index-organized table, the address portion of the rowids will change, which can hamper performance. To ensure optimal performance, rebuild the secondary index(es) on the moved partition to update the rowids.

See Also: *Oracle8i Concepts* for more information on logical rowids

Restrictions:

- If *partition* is a hash partition, the only attribute you can specify in this clause is TABLESPACE.
- You cannot move a partition of a composite-partitioned table. You must move each subpartition separately with the *move_subpartition_clause*.
- You cannot specify this clause for a partition containing subpartitions. However, you can move subpartitions using the *move_subpartition_clause*.

move_subpartition_clause

Use the *move_subpartition_clause* to move the table subpartition *subpartition* to another segment. If you do not specify TABLESPACE, the subpartition will remain in the same tablespace.

Unless the subpartition is empty, Oracle marks UNUSABLE all local index subpartitions corresponding to the subpartition being moved, as well as global nonpartitioned indexes and partitions of global indexes.

If the table contains LOB columns, you can use the *LOB_storage_clause* to move the LOB data and LOB index segments associated with this subpartition. Only the LOBs named are affected. If you do not specify the *LOB_storage_clause* for a particular LOB column, its LOB data and LOB index segments are not moved.

When you move a LOB data segment, Oracle drops the old data segment and corresponding index segment and creates new segments even if you do not specify a new tablespace.

add_range_partition_clause

The *add_range_partition_clause* lets you add a new range partition *partition* to the "high" end of a partitioned table (after the last existing partition). You can specify any create-time physical attributes for the new partition. If the table contains LOB columns, you can also specify partition-level attributes for one or more LOB items.

You can specify up to 64K-1 partitions.

> **See Also:** *Oracle8i Administrator's Guide* for a discussion of factors that might impose practical limits less than this number

Restrictions:

- If the first element of the partition bound of the high partition is MAXVALUE, you cannot add a partition to the table. Instead, use the *split_partition_clause* to add a partition at the beginning or the middle of the table.
- The *compression_clause*, *physical_attributes_clause*, and OVERFLOW are valid only for a partitioned index-organized table.
- You cannot specify the PCTUSED parameter for the index segment of an index-organized table.
- You can specify OVERFLOW only if the partitioned table already has an overflow segment.
- You can specify compression only if compression is enabled at the table level.

VALUES LESS THAN (*value_list*)	Specify the upper bound for the new partition. The *value_list* is a comma-separated, ordered list of literal values corresponding to *column_list*. The *value_list* must collate greater than the partition bound for the highest existing partition in the table.
partition_level_sub-partitioning	The *partition_level_subpartitioning* clause is permitted only for a composite-partitioned table. This clause lets you specify particular hash subpartitions for *partition*. You specify composite partitioning in one of two ways:

- You can specify individual subpartitions by name, and optionally the tablespace where each should be stored, or
- You can specify the number of subpartitions (and optionally one or more tablespaces where they are to be stored). In this case, Oracle assigns partition names of the form SYS_SUBP*nnn*. The number of tablespaces does not have to equal the number of subpartitions. If the number of subpartitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.

The subpartitions inherit all their attributes from any attributes specified for *new_partition*, except for TABLESPACE, which you can specify at the subpartition level. Any attributes not specified at the subpartition or partition level are inherited from table-level defaults.

This clause overrides any subpartitioning specified at the table level.

If you do not specify this clause but you specified default subpartitioning at the table level, *new_partition_name* will inherit the table-level default subpartitioning.

See Also: CREATE TABLE on page 10-7

add_hash_partition_clause

The *add_hash_partition_clause* lets you add a new hash partition to the "high" end of a partitioned table. Oracle will populate the new partition with rows rehashed from other partitions of *table* as determined by the hash function.

You can specify a name for the partition, and optionally a tablespace where it should be stored. If you do not specify *new_partition_name*, Oracle assigns a partition name of the form SYS_P*nnn*. If you do not specify TABLESPACE, the new partition is stored in the table's default tablespace. Other attributes are always inherited from table-level defaults.

See Also: CREATE TABLE on page 10-7 and *Oracle8i Concepts* for more information on hash partitioning

parallel_clause lets you specify whether to parallelize the creation of the new partition.

coalesce_partition_clause

COALESCE applies only to hash-partitioned tables. This clause specifies that Oracle should select a hash partition, distribute its contents into one or more remaining partitions (determined by the hash function), and then drop the selected partition. Local index partitions corresponding to the selected partition are also dropped. Oracle marks UNUSABLE, and you must rebuild, the local index partitions corresponding to one or more absorbing partitions.

drop_partition_clause

The *drop_partition_clause* applies only to tables partitioned using the range or composite method. This clause removes partition *partition*, and the data in that partition, from a partitioned table. If you want to drop a partition but keep its data in the table, you must merge the partition into one of the adjacent partitions.

> **See Also:** *merge_partitions_clause on page 8-50*

If the table has LOB columns, the LOB data and LOB index partitions (and their subpartitions, if any) corresponding to *partition* are also dropped.

- Oracle drops local index partitions and subpartitions corresponding to *partition*, even if they are marked UNUSABLE.
- Oracle marks UNUSABLE all global nonpartitioned indexes defined on the table and all partitions of global partitioned indexes, unless the partition being dropped or all of its subpartitions are empty.
- If you drop a partition and later insert a row that would have belonged to the dropped partition, Oracle stores the row in the next higher partition. However, if that partition is the highest partition, the insert will fail because the range of values represented by the dropped partition is no longer valid for the table.

Restriction: If *table* contains only one partition, you cannot drop the partition. You must drop the table.

truncate_partition_clause and *truncate_subpartition_clause*

TRUNCATE PARTITION removes all rows from *partition* or, if the table is composite-partitioned, all rows from *partition*'s subpartitions. TRUNCATE SUBPARTITION removes all rows from *subpartition*.

If the table contains any LOB columns, the LOB data and LOB index segments for this partition are also truncated. If the table is composite-partitioned, the LOB data and LOB index segments for this partition's subpartitions are truncated.

If the partition or subpartition to be truncated contains data, you must first disable any referential integrity constraints on the table. Alternatively, you can delete the rows and then truncate the partition.

For each partition or subpartition truncated, Oracle also truncates corresponding local index partitions and subpartitions. If those index partitions or subpartitions are marked UNUSABLE, Oracle truncates them and resets the UNUSABLE marker to VALID. In addition, if the truncated partition or subpartition, or any of the subpartitions of the truncated partition are not empty, Oracle marks as UNUSABLE **all** global nonpartitioned indexes and partitions of global indexes defined on the table.

DROP STORAGE	Specify DROP STORAGE to deallocate space from the deleted rows and make it available for use by other schema objects in the tablespace.
REUSE STORAGE	Specify REUSE STORAGE to keep space from the deleted rows allocated to the partition or subpartition. The space is subsequently available only for inserts and updates to the same partition or subpartition.

split_partition_clause

The *split_partition_clause* lets you create, from an original partition *partition_name_old*, two new partitions, each with a new segment and new physical attributes, and new initial extents. The segment associated with *partition_name_old* is discarded.

Restriction: You cannot specify this clause for a hash-partitioned table.

AT (*value_list*)	Specify the new noninclusive upper bound for *split_partition_1*. The *value_list* must compare less than the original partition bound for *partition_name_old* and greater than the partition bound for the next lowest partition (if there is one).
INTO *partition_spec, partition_spec*	The INTO clause lets you describe the two partitions resulting from the split. The keyword PARTITION is required. Specify optional names and physical attributes of the two partitions resulting from the split. If you do not specify new partition names, Oracle assigns names of the form SYS_P*n*. Any attributes you do not specify are inherited from *partition_name_old*.

Restriction:

- You can specify the *compression_clause*, *physical_attributes_clause*, and OVERFLOW only for a partitioned index-organized table.
- You cannot specify the PCTUSED parameter for the index segment of an index-organized table.

parallel_clause — The *parallel_clause* lets you parallelize the split operation, but does not change the default parallel attributes of the table.

If you specify subpartitioning for the new partitions, you can specify only TABLESPACE for the subpartitions. All other attributes will be inherited from the containing new partition.

If *partition_name_old* is subpartitioned, and you do not specify any subpartitioning for the new partitions, the new partitions will inherit the number and tablespaces of the subpartitions in *partition_name_old*.

Oracle also splits corresponding local index partitions, even if they are marked UNUSABLE. The resulting local index partitions inherit all their partition-level default attributes from the local index partition being split.

If *partition_name_old* was not empty, Oracle marks UNUSABLE **all** global nonpartitioned indexes and all partitions of global indexes on the table. (This action on global indexes does not apply to index-organized tables.) In addition, if any partitions or subpartitions resulting from the split are not empty, Oracle marks as UNUSABLE all corresponding local index partitions and subpartitions.

If *table* contains LOB columns, you can use the *LOB_storage_clause* to specify separate LOB storage attributes for the LOB data segments resulting from the split. Oracle drops the LOB data and LOB index segments of *partition_name_old* and creates new segments for each LOB column, for each partition, even if you do not specify a new tablespace.

merge_partitions_clause

The *merge_partitions_clause* lets you merge the contents of two adjacent partitions of *table* into one new partition, and then drops the original two partitions.

The new partition inherits the partition-bound of the higher of the two original partitions.

Any attributes not specified in the *segment_attributes_clause* are inherited from table-level defaults.

If you do not specify a new *partition_name*, Oracle assigns a name of the form SYS_P*nnn*. If the new partition has subpartitions, Oracle assigns subpartition names of the form SYS_SUBP*nnn*.

If either or both of the original partitions was not empty, Oracle marks UNUSABLE **all** global nonpartitioned global indexes and all partitions of global indexes on the table. In addition, if the partition or any of its subpartitions resulting from the merge is not empty, Oracle marks UNUSABLE all corresponding local index partitions and subpartitions.

Restriction: You cannot specify this clause for an index-organized table or for a table partitioned using the hash method.

partition_level_subpartitioning	The *partition_level_subpartitioning* clause lets you specify hash subpartitioning attributes for the new partition. Any attributes not specified in this clause are inherited from table-level defaults. If you do not specify this clause, the new merged partition inherits subpartitioning attributes from table-level defaults.
parallel_clause	The *parallel_clause* lets you parallelize the merging operation.

exchange_partition_clause and *exchange_subpartition_clause*

Use the EXCHANGE PARTITION or EXCHANGE SUBPARTITION clause to exchange the data and index segments of

- One nonpartitioned table with one hash or range partition (or subpartition)
- One hash-partitioned table with the hash subpartitions of a range partition of a composite-partitioned table

All of the segment attributes of the two objects (including tablespace) are also exchanged.

The default behavior is EXCLUDING INDEXES WITH VALIDATION. You must have ALTER TABLE privileges on **both** tables to perform this operation.

This clause facilitates high-speed data loading when used with transportable tablespaces.

> **See Also:** *Oracle8i Administrator's Guide* for information on transportable tablespaces

If *table* contains LOB columns, for each LOB column Oracle exchanges LOB data and LOB index partition or subpartition segments with corresponding LOB data and LOB index segments of *table*.

All statistics of the table and partition are exchanged, including table, column, index statistics, and histograms. The aggregate statistics of the table receiving the new partition are recalculated.

The logging attribute of the table and partition is also exchanged.

Restriction: Both tables involved in the exchange must have the same primary key, and no validated foreign keys can be referencing either of the tables unless the referenced table is empty.

WITH TABLE *table*	Specify the table with which the partition will be exchanged.
INCLUDING INDEXES	Specify INCLUDING INDEXES if you want local index partitions or subpartitions to be exchanged with the corresponding table index (for a nonpartitioned table) or local indexes (for a hash-partitioned table).
EXCLUDING INDEXES	Specify EXCLUDING NDEXES if you want all index partitions or subpartitions corresponding to the partition and all the regular indexes and index partitions on the exchanged table to be marked UNUSABLE.
WITH VALIDATION	Specify WITH VALIDATION if you want Oracle to return an error if any rows in the exchanged table do not map into partitions or subpartitions being exchanged.
WITHOUT VALIDATION	Specify WITHOUT VALIDATION if you do not want Oracle to check the proper mapping of rows in the exchanged table.
EXCEPTIONS INTO	Specify a table into which Oracle should place the rowids of all rows violating the constraint. If you omit *schema*, Oracle assumes the exceptions table is in your own schema. If you omit this clause altogether, Oracle assumes that the table is named EXCEPTIONS. The exceptions table must be on your local database. You can create the EXCEPTIONS table using one of these scripts: ■ UTLEXCPT.SQL uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.) ■ UTLEXPT1.SQL uses universal rowids, so it can accommodate rows from both conventional and index-organized tables.

If you create your own exceptions table, it must follow the format prescribed by one of these two scripts.

Note: If you are collecting exceptions from index-organized tables based on primary keys (rather than universal rowids), you must create a separate exceptions table for each index-organized table to accommodate its primary-key storage. You create multiple exceptions tables with different names by modifying and resubmitting the script.

See Also:

- The DBMS_IOT package in *Oracle8i Supplied PL/SQL Packages Reference* for information on the SQL scripts

- *Oracle8i Performance Guide and Reference* for information on eliminating migrated and chained rows

- *Oracle8i Migration* for compatibility issues related to the use of these scripts

Restrictions on EXCEPTIONS INTO clause:

- This clause is not valid with subpartitions.
- The partitioned table must have been defined with a UNIQUE constraint, and that constraint must be in DISABLE VALIDATE state.

If these conditions are not true, Oracle ignores this clause.

See Also: The *constraint_clause* on page 8-136 for more information on constraint checking

Restrictions on exchanging partitions:

When exchanging between a hash-partitioned table and the range partition of a composite-partitioned table, the following restrictions apply:

- The partitioning key of the hash-partitioned table must be identical to the subpartitioning key of the composite-partitioned table.
- The number of partitions in the hash-partitioned table must be identical to the number of subpartitions in the range partition of the composite-partitioned table.
- Oracle marks UNUSABLE all global indexes on both tables.

For partitioned index-organized tables, the following additional restrictions apply:

- The source and target table / partition must have their primary key set on the same columns, in the same order.
- If compression is enabled, it must be enabled for both the source and the target, and with the same prefix length.
- An index-organized table partition cannot be exchanged with a regular table or vice versa.
- Both the source and target must have overflow segments, or neither can have overflow segments.

row_movement_clause

The *row_movement_clause* determines whether a row can be moved to a different partition or subpartition because of a change to one or more of its key values.

Restriction: You can specify this clause only for partitioned tables.

ENABLE — Specify ENABLE to allow Oracle to move a row to a different partition or subpartition as the result of an update to the partitioning or subpartitioning key.

Restriction: You cannot specify this clause if a domain index has been built on any column of the table.

> **Caution:** Moving a row in the course of an UPDATE operation changes that row's rowid.

DISABLE — Specify DISABLE to have Oracle return an error if an update to a partitioning or subpartitioning key would result in a row moving to a different partition or subpartition. This is the default.

parallel_clause

The *parallel_clause* lets you change the default degree of parallelism for queries and DML on the table.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL — Specify NOPARALLEL for serial execution. This is the default.

PARALLEL — Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Restrictions:

- If *table* contains any columns of LOB or user-defined object type, subsequent INSERT, UPDATE, and DELETE operations on *table* are executed serially without notification. Subsequent queries, however, will be executed in parallel.
- If you specify the *parallel_clause* in conjunction with the *move_table_clause*, the parallelism applies only to the move, not to subsequent DML and query operations on the table.

See Also: "Notes on the parallel_clause" for CREATE TABLE on page 10-41

enable_disable_clause

The *enable_disable_clause* lets you specify whether Oracle should apply an integrity constraint.

See Also: The *enable_disable_clause* of CREATE TABLE on page 10-41 for a complete description of this clause, including notes and restrictions that relate to this statement

TABLE LOCK

Oracle permits DDL operations on a table only if the table can be locked during the operation. Such table locks are not required during DML operations.

Note: Table locks are not acquired on temporary tables.

ENABLE TABLE LOCK	Specify ENABLE TABLE LOCK to enable table locks, thereby allowing DDL operations on the table.
DISABLE TABLE LOCK	Specify DISABLE TABLE LOCK to disable table locks, thereby preventing DML operations on the table.

ALL TRIGGERS

ENABLE ALL TRIGGERS	Specify ENABLE ALL TRIGGERS to enable all triggers associated with the table. Oracle fires the triggers whenever their triggering condition is satisfied. See CREATE TRIGGER on page 10-66. To enable a single trigger, use the *enable_clause* of ALTER TRIGGER. **See Also:** ALTER TRIGGER on page 8-76
DISABLE ALL TRIGGERS	Specify DISABLE ALL TRIGGERS to disable all triggers associated with the table. Oracle will not fire a disabled trigger even if the triggering condition is satisfied.

Examples

Nested Table Example The following statement modifies the storage characteristics of a nested table column `projects` in table `emp` so that when queried it returns actual values instead of locators:

```
ALTER TABLE emp MODIFY NESTED TABLE projects RETURN AS VALUE;
```

PARALLEL Example The following statement specifies parallel processing for queries to the `emp` table:

```
ALTER TABLE emp
   PARALLEL;
```

ENABLE VALIDATE Example The following statement places in ENABLE VALIDATE state an integrity constraint named fk_deptno in the emp table:

```
ALTER TABLE emp
   ENABLE VALIDATE CONSTRAINT fk_deptno
   EXCEPTIONS INTO except_table;
```

Each row of the emp table must satisfy the constraint for Oracle to enable the constraint. If any row violates the constraint, the constraint remains disabled. Oracle lists any exceptions in the table except_table. You can also identify the exceptions in the EMP table with the following statement:

```
SELECT emp.*
   FROM emp e, except_table ex
   WHERE e.row_id = ex.row_id
      AND ex.table_name = 'EMP'
      AND ex.constraint = 'FK_DEPTNO';
```

ENABLE NOVALIDATE Example The following statement tries to place in ENABLE NOVALIDATE state two constraints on the emp table:

```
ALTER TABLE emp
   ENABLE NOVALIDATE UNIQUE (ename)
   ENABLE NOVALIDATE CONSTRAINT nn_ename;
```

This statement has two ENABLE clauses:

- The first places a unique constraint on the ename column in ENABLE NOVALIDATE state.
- The second places the constraint named nn_ename in ENABLE NOVALIDATE state.

In this case, Oracle enables the constraints only if both are satisfied by each row in the table. If any row violates either constraint, Oracle returns an error and both constraints remain disabled.

Disabling a Constraint Example Consider a referential integrity constraint involving a foreign key on the combination of the areaco and phoneno columns of the phone_calls table. The foreign key references a unique key on the combination of the areaco and phoneno columns of the customers table. The following statement disables the unique key on the combination of the areaco and phoneno columns of the customers table:

```
ALTER TABLE customers
   DISABLE UNIQUE (areaco, phoneno) CASCADE;
```

The unique key in the customers table is referenced by the foreign key in the phone_calls table, so you must use the CASCADE clause to disable the unique key. This clause disables the foreign key as well.

CHECK Constraint Example The following statement defines and disables a CHECK constraint on the emp table:

```
ALTER TABLE emp
   ADD (CONSTRAINT check_comp CHECK (sal + comm <= 5000) )
   DISABLE CONSTRAINT check_comp;
```

The constraint check_comp ensures that no employee's total compensation exceeds $5000. The constraint is disabled, so you can increase an employee's compensation above this limit.

Enabling Triggers Example The following statement enables all triggers associated with the emp table:

```
ALTER TABLE emp
   ENABLE ALL TRIGGERS;
```

DEALLOCATE UNUSED Example The following statement frees all unused space for reuse in table emp, where the high water mark is above MINEXTENTS:

```
ALTER TABLE emp
    DEALLOCATE UNUSED;
```

DROP COLUMN Example This statement illustrates the *drop_column_clause* with CASCADE CONSTRAINTS. Assume table t1 is created as follows:

```
CREATE TABLE t1 (
   pk NUMBER PRIMARY KEY,
   fk NUMBER,
   c1 NUMBER,
   c2 NUMBER,
   CONSTRAINT ri FOREIGN KEY (fk) REFERENCES t1,
   CONSTRAINT ck1 CHECK (pk > 0 and c1 > 0),
   CONSTRAINT ck2 CHECK (c2 > 0)
);
```

An error will be returned for the following statements:

```
ALTER TABLE t1 DROP (pk); -- pk is a parent key
ALTER TABLE t1 DROP (c1);  -- c1 is referenced by multicolumn
                           constraint ck1
```

Submitting the following statement drops column pk, the primary key constraint, the foreign key constraint, ri, and the check constraint, ck1:

```
ALTER TABLE t1 DROP (pk) CASCADE CONSTRAINTS;
```

If all columns referenced by the constraints defined on the dropped columns are also dropped, then CASCADE CONSTRAINTS is not required. For example, assuming that no other referential constraints from other tables refer to column pk, then it is valid to submit the following statement without the CASCADE CONSTRAINTS clause:

```
ALTER TABLE t1 DROP (pk, fk, c1);
```

Index-Organized Table Examples This statement modifies the INITRANS parameter for the index segment of index-organized table docindex:

```
ALTER TABLE docindex INITRANS 4;
```

The following statement adds an overflow data segment to index-organized table docindex:

```
ALTER TABLE docindex ADD OVERFLOW;
```

This statement modifies the INITRANS parameter for the overflow data segment of index-organized table docindex:

```
ALTER TABLE docindex OVERFLOW INITRANS 4;
```

ADD PARTITION Example The following statement adds a partition p3 and specifies storage characteristics for three of the table's LOB columns (b, c, and d):

```
ALTER TABLE pt ADD PARTITION p3 VALUES LESS THAN (30)
   LOB (b, d) STORE AS (TABLESPACE tsz)
   LOB (c) STORE AS mylobseg;
```

The LOB data and LOB index segments for columns b and d in partition p3 will reside in tablespace tsz. The remaining attributes for these LOB columns will be inherited first from the table-level defaults, and then from the tablespace defaults.

The LOB data segments for column c will reside in the mylobseg segment, and will inherit all other attributes from the table-level defaults and then from the tablespace defaults.

SPLIT PARTITION Example The following statement splits partition p3 into partitions p3_1 and p3_2:

```
ALTER TABLE pt SPLIT PARTITION p3 AT (25)
   INTO (PARTITION p3_1 TABLESPACE ts4
            LOB (b,d) STORE AS (TABLESPACE tsz),
         PARTITION p3_2 (TABLESPACE ts5)
            LOB (c) STORE AS (TABLESPACE ts5);
```

In partition p3_1, Oracle creates the LOB segments for columns b and d in tablespace tsz. In partition p3_2, Oracle creates the LOB segments for column c in tablespace ts5. The LOB segments for columns b and d in partition p3_2 and those for column c in partition p3_1 remain in original tablespace for the original partition p3. However, Oracle creates new segments for all the LOB data and LOB index segments, even though they are not moved to a new tablespace.

User-Defined Object Identifier Example The following statements create an object type, a corresponding object table with a primary-key-based object identifier, and a table having a user-defined REF column:

```
CREATE TYPE emp_t AS OBJECT (empno NUMBER, address CHAR(30));

CREATE TABLE emp OF emp_t (
   empno PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;

CREATE TABLE dept (dno NUMBER, mgr_ref REF emp_t SCOPE is emp);
```

The next statements add a constraint and a user-defined REF column, both of which reference table emp:

```
ALTER TABLE dept ADD CONSTRAINT mgr_cons FOREIGN KEY (mgr_ref)
   REFERENCES emp;
ALTER TABLE dept ADD sr_mgr REF emp_t REFERENCES emp;
```

Add Column Example The following statement adds a column named thriftplan of datatype NUMBER with a maximum of seven digits and two decimal places and a column named loancode of datatype CHAR with a size of one and a NOT NULL integrity constraint:

```
ALTER TABLE emp
   ADD (thriftplan NUMBER(7,2),
        loancode CHAR(1) NOT NULL);
```

Modify Column Examples The following statement increases the size of the thriftplan column to nine digits:

```
ALTER TABLE emp
```

```
   MODIFY (thriftplan NUMBER(9,2));
```

Because the MODIFY clause contains only one column definition, the parentheses around the definition are optional.

The following statement changes the values of the PCTFREE and PCTUSED parameters for the emp table to 30 and 60, respectively:

```
ALTER TABLE emp
    PCTFREE 30
    PCTUSED 60;
```

ALLOCATE EXTENT Example The following statement allocates an extent of 5 kilobytes for the emp table and makes it available to instance 4:

```
ALTER TABLE emp
  ALLOCATE EXTENT (SIZE 5K INSTANCE 4);
```

Because this statement omits the DATAFILE parameter, Oracle allocates the extent in one of the datafiles belonging to the tablespace containing the table.

Default Column Value Examples This statement modifies the bal column of the accounts table so that it has a default value of 0:

```
ALTER TABLE accounts
  MODIFY (bal DEFAULT 0);
```

If you subsequently add a new row to the accounts table and do not specify a value for the bal column, the value of the bal column is automatically 0:

```
INSERT INTO accounts(accno, accname)
  VALUES (accseq.nextval, 'LEWIS');

SELECT *
    FROM accounts
    WHERE accname = 'LEWIS';

ACCNO  ACCNAME BAL
------ ------- ---
815234  LEWIS     0
```

To discontinue previously specified default values, so that they are no longer automatically inserted into newly added rows, replace the values with nulls, as shown in this statement:

```
ALTER TABLE accounts
```

```
   MODIFY (bal DEFAULT NULL);
```

The MODIFY clause need only specify the column name and the modified part of the definition, rather than the entire column definition. This statement has no effect on any existing values in existing rows.

Drop Constraint Examples The following statement drops the primary key of the dept table:

```
ALTER TABLE dept
    DROP PRIMARY KEY CASCADE;
```

If you know that the name of the PRIMARY KEY constraint is pk_dept, you could also drop it with the following statement:

```
ALTER TABLE dept
    DROP CONSTRAINT pk_dept CASCADE;
```

The CASCADE clause drops any foreign keys that reference the primary key.

The following statement drops the unique key on the dname column of the dept table:

```
ALTER TABLE dept
    DROP UNIQUE (dname);
```

The DROP clause in this statement omits the CASCADE clause. Because of this omission, Oracle does not drop the unique key if any foreign key references it.

LOB Examples The following statement adds CLOB column resume to the employee table and specifies LOB storage characteristics for the new column:

```
ALTER TABLE employee ADD (resume CLOB)
  LOB (resume) STORE AS resume_seg (TABLESPACE resume_ts);
```

To modify the LOB column resume to use caching, enter the following statement:

```
ALTER TABLE employee MODIFY LOB (resume) (CACHE);
```

Nested Table Examples The following statement adds the nested table column skills to the employee table:

```
ALTER TABLE employee ADD (skills skill_table_type)
    NESTED TABLE skills STORE AS nested_skill_table;
```

You can also modify a nested table's storage characteristics. Use the name of the storage table specified in the *nested_table_storage_clause* to make the modification. You cannot query or perform DML statements on the storage table. Use the storage table only to modify the nested table column storage characteristics.

The following statement creates table vetservice with nested table column client and storage table client_tab. Nested table vetservice is modified to specify constraints:

```
CREATE TYPE pet_table AS OBJECT
   (pet_name VARCHAR2(10), pet_dob DATE);

CREATE TABLE vetservice (vet_name VARCHAR2(30),
                         client   pet_table)
  NESTED TABLE client STORE AS client_tab;

ALTER TABLE client_tab ADD UNIQUE (ssn);
```

The following statement adds a UNIQUE constraint to nested table nested_skill_table:

```
ALTER TABLE nested_skill_table ADD UNIQUE (a);
```

The following statement alters the storage table for a nested table of REF values to specify that the REF is scoped:

```
CREATE TYPE emp_t AS OBJECT (eno number, ename char(31));
CREATE TYPE emps_t AS TABLE OF REF emp_t;
CREATE TABLE emptab OF emp_t;
CREATE TABLE dept (dno NUMBER, employees emps_t)
   NESTED TABLE employees STORE AS deptemps;
ALTER TABLE deptemps ADD (SCOPE FOR (column_value) IS emptab);
```

Similarly, to specify storing the REF with rowid:

```
ALTER TABLE deptemps ADD (REF(column_value) WITH ROWID);
```

In order to execute these ALTER TABLE statements successfully, the storage table deptemps must be empty. Also, because the nested table is defined as a table of scalars (REFs), Oracle implicitly provides the column name COLUMN_VALUE for the storage table.

See Also:

- CREATE TABLE on page 10-7 for more information about nested table storage
- *Oracle8i Application Developer's Guide - Fundamentals* for more information about nested tables

REF Examples In the following statement an object type dept_t has been previously defined. Now, create table emp as follows:

```
CREATE TABLE emp
   (name VARCHAR(100),
    salary NUMBER,
    dept REF dept_t);
```

An object table DEPARTMENTS is created as:

```
CREATE TABLE departments OF dept_t;
```

The dept column can store references to objects of dept_t stored in any table. If you would like to restrict the references to point only to objects stored in the departments table, you could do so by adding a scope constraint on the dept column as follows:

```
ALTER TABLE emp
    ADD (SCOPE FOR (dept) IS departments);
```

The above ALTER TABLE statement will succeed only if the emp table is empty.

If you want the REF values in the dept column of emp to also store the rowids, issue the following statement:

```
ALTER TABLE emp
   ADD (REF(dept) WITH ROWID);
```

Add Partition Example The following statement adds partition jan99 to tablespace tsx:

```
ALTER TABLE sales
  ADD PARTITION jan99 VALUES LESS THAN( '970201' )
  TABLESPACE tsx;
```

Drop Partition Example The following statement drops partition dec98:

```
ALTER TABLE sales DROP PARTITION dec98;
```

Exchange Partition Example The following statement converts partition feb97 to table sales_feb97 without exchanging local index partitions with corresponding indexes on sales_feb97 and without verifying that data in sales_feb97 falls within the bounds of partition feb97:

```
ALTER TABLE sales
   EXCHANGE PARTITION feb97 WITH TABLE sales_feb97
   WITHOUT VALIDATION;
```

Modify Partition Examples The following statement marks all the local index partitions corresponding to the nov96 partition of the sales table UNUSABLE:

```
ALTER TABLE sales MODIFY PARTITION nov96
   UNUSABLE LOCAL INDEXES;
```

The following statement rebuilds all the local index partitions that were marked UNUSABLE:

```
ALTER TABLE sales MODIFY PARTITION jan97
   REBUILD UNUSABLE LOCAL INDEXES;
```

The following statement changes MAXEXTENTS and logging attribute for partition branch_ny:

```
ALTER TABLE branch MODIFY PARTITION branch_ny
   STORAGE (MAXEXTENTS 75) LOGGING;
```

Move Partition Example The following statement moves partition depot2 to tablespace ts094:

```
ALTER TABLE parts
  MOVE PARTITION depot2 TABLESPACE ts094 NOLOGGING;
```

Rename Partition Examples The following statement renames a table:

```
ALTER TABLE emp RENAME TO employee;
```

In the following statement, partition emp3 is renamed:

```
ALTER TABLE employee RENAME PARTITION emp3 TO employee3;
```

Split Partition Example The following statement splits the old partition depot4, creating two new partitions, naming one depot9 and reusing the name of the old partition for the other:

```
ALTER TABLE parts
   SPLIT PARTITION depot4 AT ( '40-001' )
```

```
   INTO ( PARTITION depot4 TABLESPACE ts009 STORAGE (MINEXTENTS 2),
         PARTITION depot9 TABLESPACE ts010 )
   PARALLEL (10);
```

Truncate Partition Example The following statement deletes all the data in the sys_p017 partition and deallocates the freed space:

```
ALTER TABLE deliveries
   TRUNCATE PARTITION sys_p017 DROP STORAGE;
```

Additional Examples For examples of defining integrity constraints with the ALTER TABLE statement, see the *constraint_clause* on page 8-136.

For examples of changing the value of a table's storage parameters, see the *storage_clause* on page 11-129.

ALTER TABLESPACE

Purpose

Use the ALTER TABLESPACE statement to alter an existing tablespace or one or more of its datafiles or tempfiles.

> **See Also:** CREATE TABLESPACE on page 10-56 for information on creating a tablespace

Prerequisites

If you have ALTER TABLESPACE system privilege, you can perform any of this statement's operations. If you have MANAGE TABLESPACE system privilege, you can only perform the following operations:

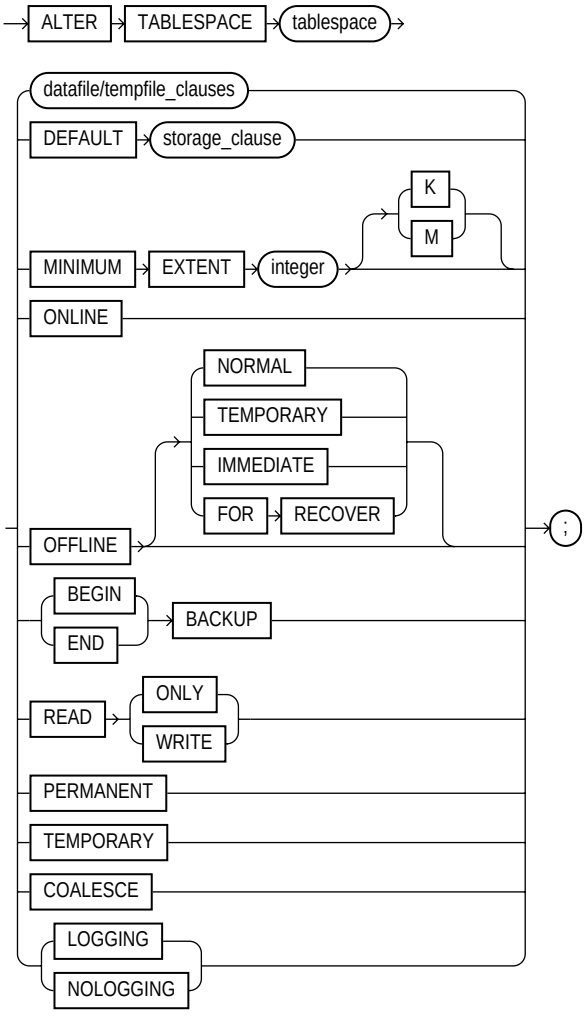
- Take the tablespace online or offline
- Begin or end a backup
- Make the tablespace read only or read write

Before you can make a tablespace read only, the following conditions must be met:

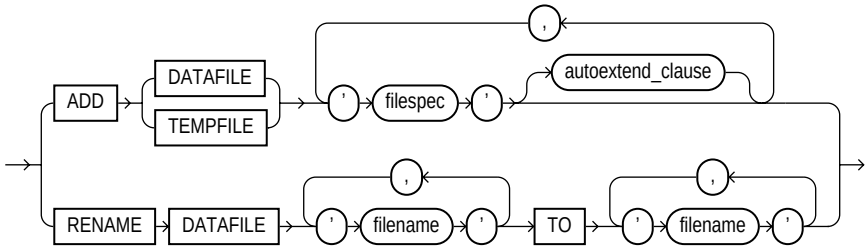
- The tablespace must be online.
- The tablespace must not contain any active rollback segments. For this reason, the SYSTEM tablespace can never be made read only, because it contains the SYSTEM rollback segment. Additionally, because the rollback segments of a read-only tablespace are not accessible, Oracle recommends that you drop the rollback segments before you make a tablespace read only.
- The tablespace must not be involved in an open backup, because the end of a backup updates the header file of all datafiles in the tablespace.

Performing this function in restricted mode may help you meet these restrictions, because only users with RESTRICTED SESSION system privilege can be logged on.

Syntax

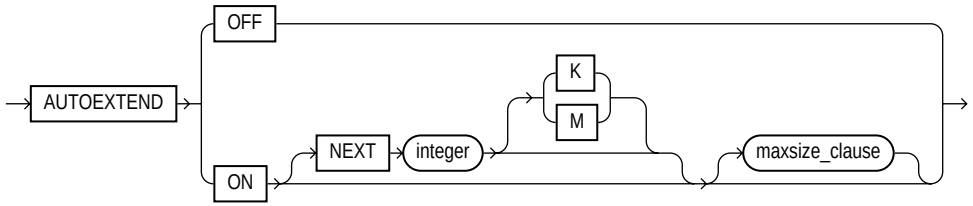

ALTER
TABLESPACE
tablespace
datafile/tempfile_clauses
DEFAULT
storage_clause
MINIMUM
EXTENT
integer
K
M
ONLINE
NORMAL
TEMPORARY
IMMEDIATE
FOR
RECOVER
OFFLINE
;
BEGIN
END
BACKUP
READ
ONLY
WRITE
PERMANENT
TEMPORARY
COALESCE
LOGGING
NOLOGGING

datafile / tempfile_clauses::=

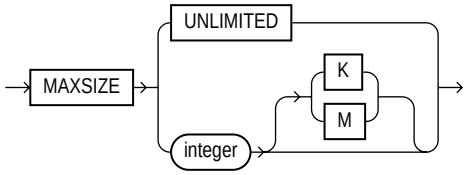


filespec: See filespec on page 11-27.

autoextend_clause::=



maxsize_clause::=



storage_clause: See storage_clause on page 11-129.

Keywords and Parameters

tablespace

Specify the name of the tablespace to be altered.

Note: For locally managed temporary tablespaces, the only clause you can specify in this statement in the ADD clause.

datafile / tempfile_clauses

The datafile and tempfile clauses let you add or modify a datafile or tempfile.

ADD DATAFILE | TEMPFILE Specify ADD to add to the tablespace a datafile or tempfile specified by *filespec*.

You can add a datafile or tempfile to a locally managed tablespace that is online or to a dictionary managed tablespace that is online or offline. Be sure the file is not in use by another database.

See Also: *filespec* on page 11-27

Note: For locally managed temporary tablespaces, this is the only clause you can specify at any time.

RENAME DATAFILE Specify RENAME DATAFILE to rename one or more of the tablespace's datafiles. Take the tablespace offline before renaming the datafile. Each '*filename*' must fully specify a datafile using the conventions for filenames on your operating system.

This clause merely associates the tablespace with the new file rather than the old one. This clause does not actually change the name of the operating system file. You must change the name of the file through your operating system.

autoextend_clause

The *autoextend_clause* lets you enable or disable the automatic extending of the size of the datafile in the tablespace.

OFF Specify OFF to disable autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER TABLESPACE AUTOEXTEND statements.

ON Specify ON to enable autoextend.

NEXT *integer* Specify the size in bytes of the next increment of disk space to be allocated automatically to the datafile when more extents are required. Use K or M to specify this size in kilobytes or megabytes. The default is one data block.

maxsize_clause The *maxsize_clause* lets you specify maximum disk space allowed for automatic extension of the datafile.

- UNLIMITED Specify UNLIMITED to set no limit on allocating disk space to the datafile.

DEFAULT *storage_clause*

DEFAULT *storage_clause* lets you specify the new default storage parameters for objects subsequently created in the tablespace. For a dictionary-managed temporary table, Oracle considers only the NEXT parameter of the *storage_clause*.

Restriction: You cannot specify this clause for a locally managed tablespace.

> **See Also:** *storage_clause* on page 11-129

MINIMUM EXTENT

The MINIMUM EXTENT clause lets you control free space fragmentation in the tablespace by ensuring that every used or free extent size in a tablespace is at least as large as, and is a multiple of, *integer*. This clause is not relevant for a dictionary-managed temporary tablespace.

Restriction: You cannot specify this clause for a locally managed tablespace.

> **See Also:** *Oracle8i Administrator's Guide* for more information about using MINIMUM EXTENT to control space fragmentation

ONLINE | OFFLINE

Specify ONLINE to bring the tablespace online.

Specify OFFLINE to take the tablespace offline and prevents further access to its segments.

Suggestion: Before taking a tablespace offline for a long time, you may want to alter the tablespace allocation of any users who have been assigned the tablespace as either a default or temporary tablespace. When the tablespace is offline, these users cannot allocate space for objects or sort areas in the tablespace.

See Also: ALTER USER on page 8-88

BEGIN BACKUP

Specify BEGIN BACKUP to indicate that an open backup is to be performed on the datafiles that make up this tablespace. This clause does not prevent users from accessing the tablespace. You must use this clause before beginning an open backup.

Restrictions: You cannot specify this clause for a read-only tablespace or for a temporary locally managed tablespace.

Note: While the backup is in progress, you cannot take the tablespace offline normally, shut down the instance, or begin another backup of the tablespace.

END BACKUP

Specify END BACKUP to indicate that an open backup of the tablespace is complete. Use this clause as soon as possible after completing an open backup. You cannot use this clause on a read-only tablespace.

If you forget to indicate the end of an online tablespace backup, and an instance failure or SHUTDOWN ABORT occurs, Oracle assumes that media recovery (possibly requiring archived redo log) is necessary at the next instance start up.

See Also: *Oracle8i Administrator's Guide* for information on restarting the database without media recovery

READ ONLY | READ WRITE

Specify READ ONLY to place the tablespace in **transition read-only mode**. In this state, existing transactions can complete (commit or roll back), but no further write operations (DML) are allowed to the tablespace except for rollback of existing transactions that previously modified blocks in the tablespace.

Once a tablespace is read only, you can copy its files to read-only media. You must then rename the datafiles in the control file to point to the new location by using the SQL statement ALTER DATABASE ... RENAME.

See Also:

- *Oracle8i Concepts* for more information on read-only tablespaces
- ALTER DATABASE on page 7-9

Specify READ WRITE to indicate that write operations are allowed on a previously read-only tablespace.

PERMANENT | TEMPORARY

Specify PERMANENT to indicate that the tablespace is to be converted from a temporary to a permanent one. A permanent tablespace is one in which permanent database objects can be stored. This is the default when a tablespace is created.

Specify TEMPORARY to indicate specifies that the tablespace is to be converted from a permanent to a temporary one. A temporary tablespace is one in which no permanent database objects can be stored. Objects in a temporary tablespace persist only for the duration of the session.

COALESCE

For each datafile in the tablespace, this clause coalesces all contiguous free extents into larger contiguous extents.

LOGGING | NOLOGGING

Specify LOGGING if you want logging of all tables, indexes, and partitions within the tablespace. The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

When an existing tablespace logging attribute is changed by an ALTER TABLESPACE statement, all tables, indexes, and partitions created *after* the statement will have the new default logging attribute (which you can still subsequently override). The logging attributes of existing objects are not changed.

Only the following operations support NOLOGGING mode:

- DML: direct-load INSERT (serial or parallel); Direct Loader (SQL*Loader)
- DDL: CREATE TABLE ... AS SELECT, CREATE INDEX, ALTER INDEX ... REBUILD, ALTER INDEX ... REBUILD PARTITION, ALTER INDEX ... SPLIT

PARTITION, ALTER TABLE ... SPLIT PARTITION, ALTER TABLE ... MOVE PARTITION.

In NOLOGGING mode, data is modified with minimal logging (to mark new extents invalid and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not logged. Therefore, if you cannot afford to lose the object, it is important to take a backup after the NOLOGGING operation.

Examples

Backup Examples The following statement signals to the database that a backup is about to begin:

```
ALTER TABLESPACE accounting
    BEGIN BACKUP;
```

The following statement signals to the database that the backup is finished:

```
ALTER TABLESPACE accounting
    END BACKUP;
```

Moving and Renaming Example This example moves and renames a datafile associated with the accounting tablespace from 'diska:pay1:dat' to 'diskb:receive1:dat':

1. Take the tablespace offline using an ALTER TABLESPACE statement with the OFFLINE clause:

   ```
   ALTER TABLESPACE accounting OFFLINE NORMAL;
   ```

2. Copy the file from 'diska:pay1.dat' to 'diskb:receive1.dat' using your operating system's commands.

3. Rename the datafile using the ALTER TABLESPACE statement with the RENAME DATAFILE clause:

   ```
   ALTER TABLESPACE accounting
     RENAME DATAFILE 'diska:pay1.dbf'
     TO     'diskb:receive1.dbf';
   ```

4. Bring the tablespace back online using an ALTER TABLESPACE statement with the ONLINE clause:

   ```
   ALTER TABLESPACE accounting ONLINE;
   ```

Adding a Datafile Example The following statement adds a datafile to the tablespace and changes the default logging attribute to NOLOGGING. When more space is needed, new extents of size 10 kilobytes will be added up to a maximum of 100 kilobytes:

```
ALTER TABLESPACE accounting NOLOGGING
    ADD DATAFILE 'disk3:pay3.dbf'
    SIZE 50K
    AUTOEXTEND ON
    NEXT 10K
    MAXSIZE 100K;
```

Altering a tablespace logging attribute has no affect on the logging attributes of the existing schema objects within the tablespace. The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

Changing Extent Allocation Example The following statement changes the allocation of every extent of tabspace_st to a multiple of 128K:

```
ALTER TABLESPACE tabspace_st MINIMUM EXTENT 128K;
```

ALTER TRIGGER

Purpose

Use the ALTER TRIGGER statement to enable, disable, or compile a database trigger.

Note: This statement does not change the declaration or definition of an existing trigger. To redeclare or redefine a trigger, use the CREATE TRIGGER statement with OR REPLACE.

See Also:

- CREATE TRIGGER on page 10-66 for information on creating a trigger
- DROP TRIGGER on page 11-13 for information on dropping a trigger

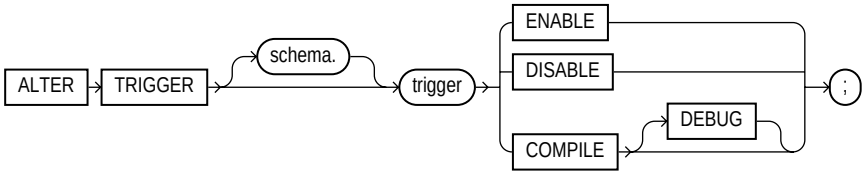
Prerequisites

The trigger must be in your own schema or you must have ALTER ANY TRIGGER system privilege.

In addition, to alter a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER system privilege.

See Also: CREATE TRIGGER on page 10-66 for more information on triggers based on DATABASE

Syntax



Keywords and Parameters

schema

Specify the schema containing the trigger. If you omit *schema*, Oracle assumes the trigger is in your own schema.

trigger

Specify the name of the trigger to be altered.

ENABLE

Specify ENABLE to enable the trigger. You can also use the ENABLE ALL TRIGGERS clause of ALTER TABLE to enable all triggers associated with a table.

See Also: ALTER TABLE on page 8-2

DISABLE

Specify DISABLE to disable the trigger. You can also use the DISABLE ALL TRIGGERS clause of ALTER TABLE to disable all triggers associated with a table.

See Also: ALTER TABLE on page 8-2

COMPILE

Specify COMPILE to explicitly compile the trigger, whether it is valid or invalid. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

Oracle first recompiles objects upon which the trigger depends, if any of these objects are invalid. If Oracle recompiles the trigger successfully, the trigger becomes valid.

If recompiling the trigger results in compilation errors, then Oracle returns an error and the trigger remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

DEBUG — Specify DEBUG to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. This clause can be used for normal triggers and for instead-of triggers.

See Also:

- *Oracle8i Application Developer's Guide - Fundamentals* for information on debugging procedures
- *Oracle8i Concepts f*or information on how Oracle maintains dependencies among schema objects, including remote objects

Examples

Disable Trigger Example Consider a trigger named `reorder` created on the `inventory` table. The trigger is fired whenever an `UPDATE` statement reduces the number of a particular part on hand below the part's reorder point. The trigger inserts into a table of pending orders a row that contains the part number, a reorder quantity, and the current date.

When this trigger is created, Oracle enables it automatically. You can subsequently disable the trigger with the following statement:

```
ALTER TRIGGER reorder DISABLE;
```

When the trigger is disabled, Oracle does not fire the trigger when an `UPDATE` statement causes the part's inventory to fall below its reorder point.

Enable Trigger Example After disabling the trigger, you can subsequently enable it with the following statement:

```
ALTER TRIGGER reorder ENABLE;
```

After you reenable the trigger, Oracle fires the trigger whenever a part's inventory falls below its reorder point as a result of an `UPDATE` statement. It is possible that a part's inventory falls below its reorder point while the trigger was disabled. In that case, when you reenable the trigger, Oracle does not automatically fire the trigger for this part until another transaction further reduces the inventory.

ALTER TYPE

Purpose

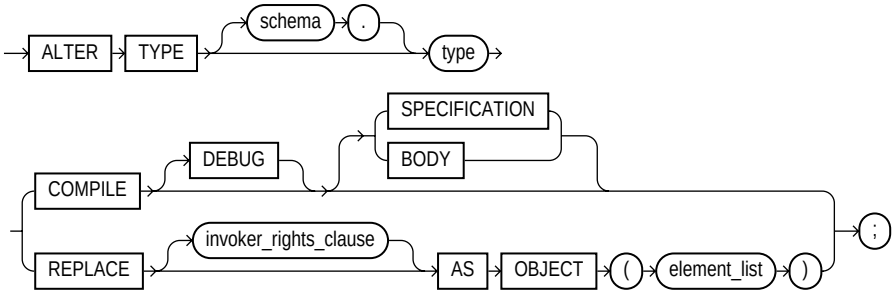
Use the ALTER TYPE statement to recompile the specification and / or body, or to change the specification of an object type by adding new object member subprogram specifications.

You cannot change the existing properties (attributes, member subprograms, map or order functions) of an object type, but you can add new member subprogram specifications.

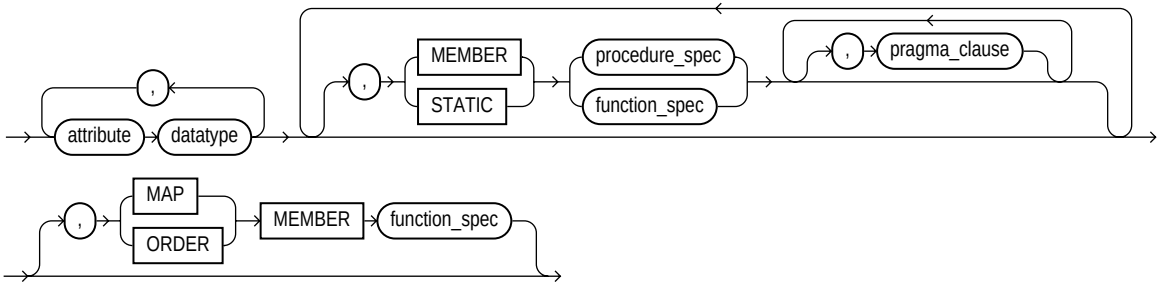
Prerequisites

The object type must be in your own schema and you must have CREATE TYPE or CREATE ANY TYPE system privilege, or you must have ALTER ANY TYPE system privileges.

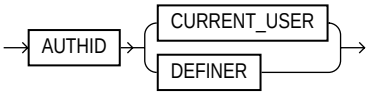
Syntax



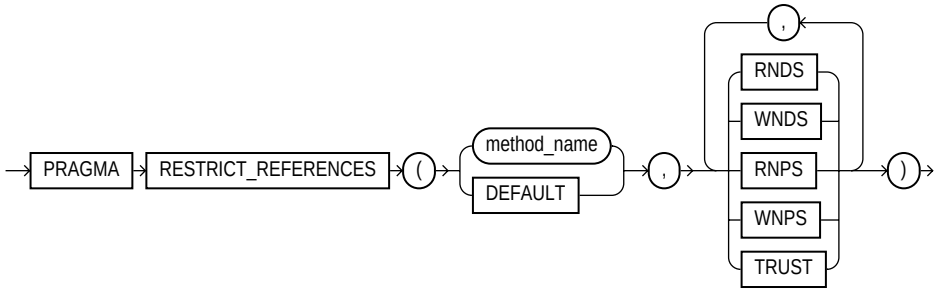
element_list::=



invoker_rights_clause::=



pragma_clause::=



Keywords and Parameters

schema

Specify the schema that contains the type. If you omit *schema*, Oracle assumes the type is in your current schema.

type

Specify the name of an object type, a nested table type, or a rowid type.

COMPILE

Specify COMPILE to compile the object type specification and body. This is the default if neither SPECIFICATION nor BODY is specified.

If recompiling the type results in compilation errors, then Oracle returns an error and the type remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

DEBUG — Specify DEBUG to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

`SPECIFICATION` Specify `SPECIFICATION` to compile only the object type specification.

`BODY` Specify `BODY` to compile only the object type body.

REPLACE AS OBJECT

The `REPLACE AS OBJECT` clause lets you add new member subprogram specifications. This clause is valid only for object types, not for nested table or varray types.

element_list

Specify the elements of the object.

attribute Specify an object attribute name. Attributes are data items with a name and a type specifier that form the structure of the object.

`MEMBER | STATIC` This clause lets you specify a function or procedure subprogram associated with the object type which is referenced as an attribute.

You must specify a corresponding method body in the object type body for each procedure or function specification.

See Also:

- CREATE TYPE on page 10-80 for a description of the difference between member and static methods, and for examples

- *PL/SQL User's Guide and Reference* for information about overloading subprogram names within a package

- CREATE TYPE BODY on page 10-93

procedure_spec Enter the specification of a procedure subprogram.

function_spec Enter the specification of a function subprogram.

pragma_clause The *pragma_clause* is a complier directive that denies member functions read / write access to database tables, packaged variables, or both, and thereby helps to avoid side effects.

See Also: *Oracle8i Application Developer's Guide - Fundamentals*

method	Specify the name of the MEMBER function or procedure to which the pragma is being applied.
DEFAULT	Specify DEFAULT if you want Oracle to apply the pragma to all methods in the type for which a pragma has not been explicitly specified.
WNDS	Specify WNDS to enforce the constraint **writes no database state** (does not modify database tables).
WNPS	Specify WNPS to enforce the constraint **writes no package state** (does not modify packaged variables).
RNDS	Specify RNDS to enforce the constraint **reads no database state** (does not query database tables).
RNPS	Specify WNPS to enforce the constraint **reads no package state** (does not reference package variables).
TRUST	Specify TRUST to indicate that the restrictions listed in the pragma are not actually to be enforced, but are simply trusted to be true.

MAP | ORDER MEMBER *function_spec*

You can declare either a MAP method or an ORDER method, but not both. If you declare either method, you can compare object instances in SQL.

If you do not declare either method, you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method needs to be specified to determine the equality of two object types.

> **See Also:** "Object Values" on page 2-29for more information about object value comparisons

MAP

Specify a member function (MAP method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined *scalar* type. Oracle uses the ordering for comparison operators and ORDER BY clauses.

If the argument to the map method is null, the map method returns null and the method is not invoked.

An object specification can contain only one map method, which must be a function. The result type must be a predefined SQL scalar type, and the map function can have no arguments other than the implicit SELF argument.

Note: If *type_name* will be referenced in queries involving sorts (through ORDER BY, GROUP BY, DISTINCT, or UNION clauses) or joins, and you want those queries to be parallelized, you must specify a MAP member function.

ORDER

Specify a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, zero, or positive indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

If either argument to the order method is null, the order method returns null and the method is not invoked.

When instances of the same object type definition are compared in an ORDER BY clause, the order method function is invoked.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the member functions and procedures of the object type execute with the privileges and in the schema of the user who owns the object type or with the privileges and in the schema of CURRENT_USER. This specification applies to the corresponding type body as well.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

Restriction: You can specify this clause only for an object type, not for a nested table or varray type.

AUTHID CURRENT_USER — Specify CURRENT_USER if you want the member functions and procedures of the object type to execute with the privileges of CURRENT_USER. This clause creates an **invoker-rights type**.

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the type resides.

> **Note:** You must specify this clause to maintain invoker-rights status for the type if you created it with this status. Otherwise the status will revert to definer rights.

AUTHID DEFINER — Specify DEFINER if you want the member functions and procedures of the object type to execute with the privileges of the owner of the schema in which the functions and procedures reside, and that external names resolve in the schema where the member functions and procedures reside. This is the default.

See Also:

- *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined
- *PL/SQL User's Guide and Reference*

Examples

Adding a Member Function In the following example, member function `qtr` is added to the type definition of `data_t`.

```
CREATE TYPE data_t AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION prod(invent NUMBER) RETURN NUMBER
   );

   CREATE TYPE BODY data_t IS
      MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
         BEGIN
             RETURN (year + invent);
         END;
      END;

   ALTER TYPE data_t REPLACE AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION  prod(invent NUMBER) RETURN NUMBER,
     MEMBER FUNCTION  qtr(der_qtr DATE) RETURN CHAR
   );

   CREATE OR REPLACE TYPE BODY data_t IS
      MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
      MEMBER FUNCTION qtr(der_qtr DATE) RETURN CHAR IS
         BEGIN
            RETURN (year + invent);
         END;
         BEGIN
            RETURN 'FIRST';
         END;
      END;
```

Recompiling a Type The following example creates and then recompiles type `loan_t`:

```
CREATE TYPE loan_t AS OBJECT
  ( loan_num        NUMBER,
    interest_rate   FLOAT,
    amount          FLOAT,
    start_date      DATE,
    end_date        DATE );

ALTER TYPE loan_t COMPILE;
```

Recompiling a Type Body The following example compiles the type body of `link2`.

```
CREATE TYPE link1 AS OBJECT
  (a NUMBER);

CREATE TYPE link2 AS OBJECT
  (a NUMBER,
   b link1,
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);

CREATE TYPE BODY link2 AS
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS t13 link1;
      BEGIN t13 := link1(13);
         dbms_output.put_line(t13.a);
         RETURN 5;
      END;
   END;

CREATE TYPE link3 AS OBJECT (a link2);
CREATE TYPE link4 AS OBJECT (a link3);
CREATE TYPE link5 AS OBJECT (a link4);
ALTER TYPE link2 COMPILE BODY;
```

Recompiling a Type Specification The following example compiles the type specification of `link2`.

```
CREATE TYPE link1 AS OBJECT
  (a NUMBER);

CREATE TYPE link2 AS OBJECT
  (a NUMBER,
   b link1,
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);

CREATE TYPE BODY link2 AS
```

```
    MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS t14 link1;
      BEGIN t14 := link1(14);
         dbms_output.put_line(t14.a);
         RETURN 5;
      END;
    END;

CREATE TYPE link3 AS OBJECT (a link2);
CREATE TYPE link4 AS OBJECT (a link3);
CREATE TYPE link5 AS OBJECT (a link4);
ALTER TYPE link2 COMPILE SPECIFICATION;
```

ALTER USER

Purpose

Use the ALTER USER statement to change the authentication or database resource characteristics of a database user.

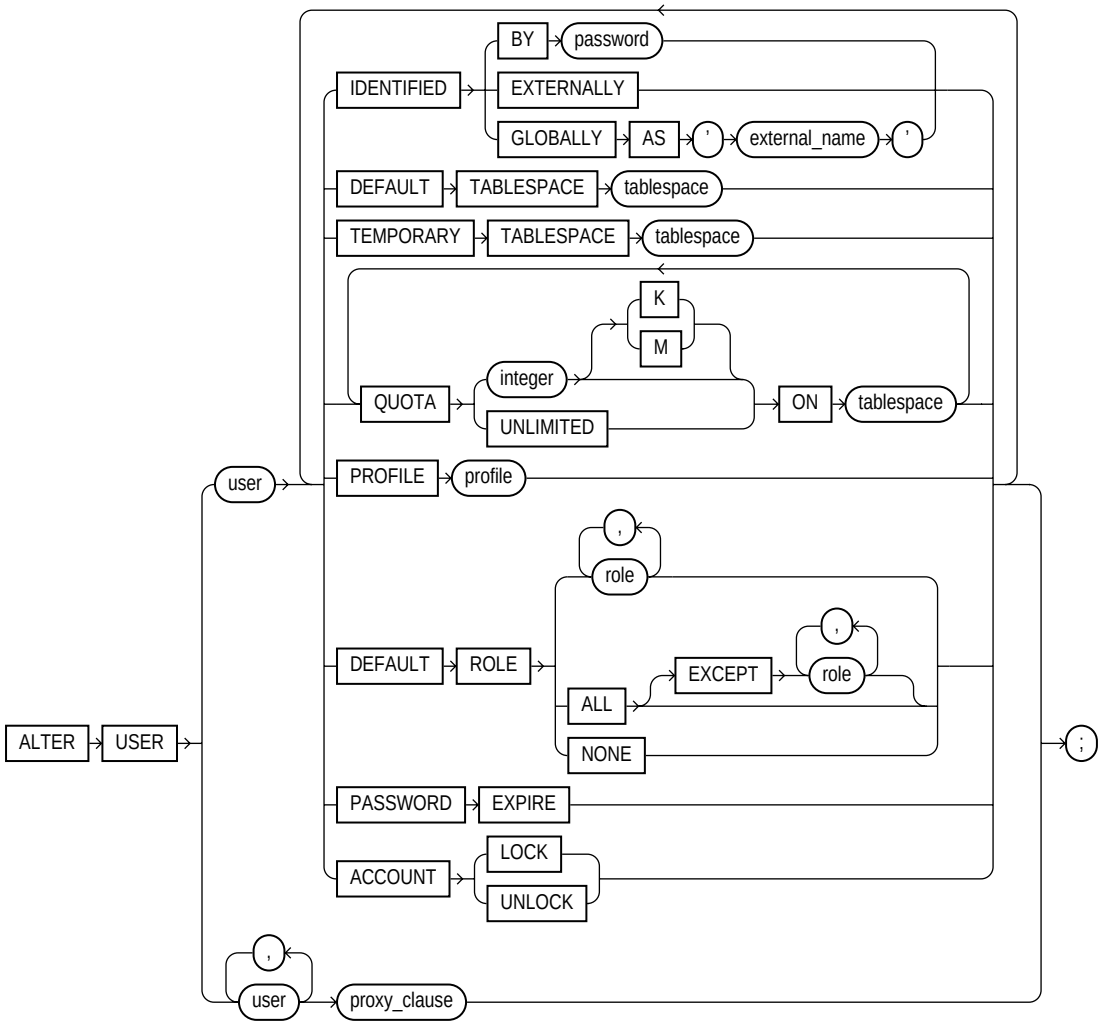
To permit a proxy server to connect as a client without authentication.

Note: ALTER USER syntax does not accept the old password. Therefore it neither authenticates using the old password nor checks the new password against the old before setting the new password. If these checks against the old password are important, use the `OCIPasswordChange()` call instead of ALTER USER. For more information, see *Oracle Call Interface Programmer's Guide*.

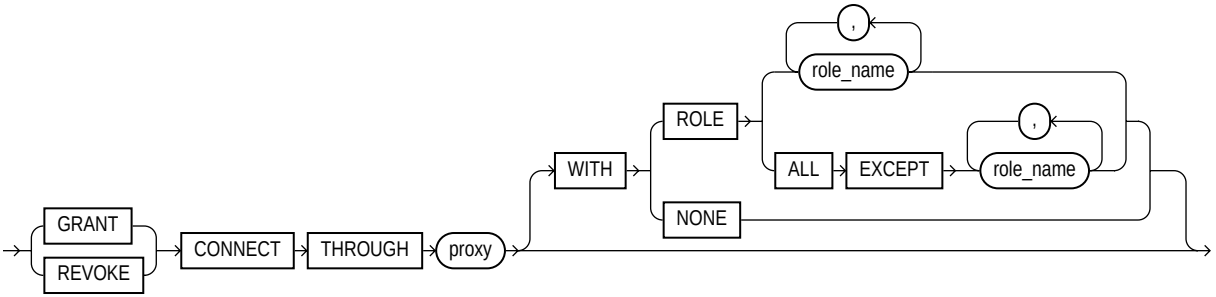
Prerequisites

You must have the ALTER USER system privilege. However, you can change your own password without this privilege.

Syntax


ALTER
USER
user
,
user
IDENTIFIED
BY
password
EXTERNALLY
GLOBALLY
AS
'
external_name
'
DEFAULT
TABLESPACE
tablespace
TEMPORARY
TABLESPACE
tablespace
QUOTA
integer
K
M
UNLIMITED
ON
tablespace
PROFILE
profile
DEFAULT
ROLE
,
role
ALL
EXCEPT
,
role
NONE
PASSWORD
EXPIRE
ACCOUNT
LOCK
UNLOCK
proxy_clause
;

proxy_clause::=



Keywords and Parameters

The keywords and parameters shown below are unique to ALTER USER or have different functionality than they have in CREATE USER. All the remaining keywords and parameters in the ALTER USER statement have the same meaning as in the CREATE USER statement.

See Also:

- CREATE USER on page 10-99 for information on the keywords and parameters
- CREATE PROFILE on page 9-139 for information on assigning limits on database resources to a user

IDENTIFIED

BY *password* Specify a password for the user.

> **Note:** Oracle expects a different timestamp for each resetting of a particular password. If you reset one password multiple times within one second (for example, by cycling through a set of passwords using a script), Oracle may return an error message that the password cannot be reused. For this reason, Oracle Corporation recommends that you avoid using scripts to reset passwords.

GLOBALLY AS — Specify '*external_name*' to indicate that the user must be authenticated by way of an LDAP V3 compliant directory service such as Oracle Internet Directory.

You can change a user's access verification method to IDENTIFIED GLOBALLY AS '*external_name*' only if all external roles granted directly to the user are revoked.

You can change a user created as IDENTIFIED GLOBALLY AS '*external_name*' to IDENTIFIED BY *password* or IDENTIFIED EXTERNALLY.

See Also: CREATE USER on page 10-99

DEFAULT ROLE

Specify the roles granted by default to the user at logon. This clause can contain only roles that have been granted directly to the user with a GRANT statement. You cannot use the DEFAULT ROLE clause to enable:

- Roles not granted to the user
- Roles granted through other roles
- Roles managed by an external service (such as the operating system), or by the Oracle Internet Directory

Oracle enables default roles at logon without requiring the user to specify their passwords.

See Also: CREATE ROLE on page 9-146

proxy_clause

The *proxy_clause* lets you control the ability of a **proxy** (an application or application server) to connect as the specified user and to activate all, some, or none of the user's roles.

See Also: *Oracle8i Concepts* for more information on proxies and their use of the database

GRANT — Specify GRANT to allow the connection.

REVOKE — Specify REVOKE to prohibit the connection.

proxy — Identify the proxy connecting to Oracle.

WITH Clause Specify the roles that the application is permitted to activate after it connects as the user. If you do not include this clause, Oracle activates all roles granted to the specified user automatically.

- ROLE *role_name* permits the proxy to connect as the specified user and to activate only the roles that are specified by *role_name*.
- ROLE ALL EXCEPT *role_name* permits the proxy to connect as the specified user and to activate all roles associated with that user except those specified by *role_name*.
- NONE permits the proxy to connect as the specified user, but prohibits the proxy from activating any of that user's roles after connecting.

Examples

ALTER USER Examples The following statement changes the user scott's password to lion and default tablespace to the tablespace tstest:

```
ALTER USER scott
    IDENTIFIED BY lion
    DEFAULT TABLESPACE tstest;
```

The following statement assigns the clerk profile to scott:

```
ALTER USER scott
    PROFILE clerk;
```

In subsequent sessions, scott restricted by limits in the clerk profile.

The following statement makes all roles granted directly to scott default roles, except the agent role:

```
ALTER USER scott
    DEFAULT ROLE ALL EXCEPT agent;
```

At the beginning of scott's next session, Oracle enables all roles granted directly to scott except the agent role.

User Authentication Examples The following statement changes user tom's authentication mechanism:

```
ALTER USER tom IDENTIFIED GLOBALLY AS 'CN=tom,O=oracle,C=US';
```

The following statement causes user fred's password to expire:

```
ALTER USER fred PASSWORD EXPIRE;
```

If you cause a database user's password to expire with PASSWORD EXPIRE, the user (or the DBA) must change the password before attempting to log in to the database following the expiration. However, tools such as SQL*Plus allow you to change the password on the first attempted login following the expiration.

Proxy User Examples The following statement permits the proxy user APPSERVER1 to connect as the user JANE. It also allows APPSERVER1 to activate the role INVENTORY:

```
ALTER USER jane GRANT CONNECT THROUGH appserver1 WITH ROLE
inventory;
```

The following statement takes away the right of proxy user appserver1to connect as the user jane:

```
ALTER USER jane REVOKE CONNECT THROUGH appserver1;
```

ALTER VIEW

Purpose

Use the ALTER VIEW statement to explicitly recompile a view that is invalid. Explicit recompilation allows you to locate recompilation errors before run time. You may want to recompile a view explicitly after altering one of its base tables to ensure that the alteration does not affect the view or other objects that depend on it.

When you issue an ALTER VIEW statement, Oracle recompiles the view regardless of whether it is valid or invalid. Oracle also invalidates any local objects that depend on the view.

Notes:

- This statement does not change the definition of an existing view. To redefine a view, you must use CREATE VIEW with OR REPLACE.
- If you alter a view that is referenced by one or more materialized views, those materialized views are invalidated. Invalid materialized views cannot be used by query rewrite and cannot be refreshed.

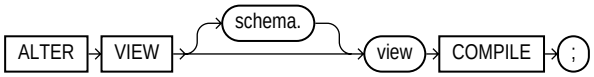
See Also:

- CREATE VIEW on page 10-105 for information on redefining a view
- ALTER MATERIALIZED VIEW on page 7-61 for information on revalidating an invalid materialized view
- *Oracle8i Data Warehousing Guide* for general information on data warehouses
- *Oracle8i Concepts* for more about dependencies among schema objects

Prerequisites

The view must be in your own schema or you must have ALTER ANY TABLE system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the view. If you omit *schema*, Oracle assumes the view is in your own schema.

view

Specify the name of the view to be recompiled.

COMPILE

The `COMPILE` keyword is required. It directs Oracle to recompile the view.

Example

ALTER VIEW example To recompile the view `customer_view`, issue the following statement:

```
ALTER VIEW customer_view
    COMPILE;
```

If Oracle encounters no compilation errors while recompiling `customer_view`, `customer_view` becomes valid. If recompiling results in compilation errors, Oracle returns an error and `customer_view` remains invalid.

Oracle also invalidates all dependent objects. These objects include any procedures, functions, package bodies, and views that reference `customer_view`. If you subsequently reference one of these objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.

ANALYZE

Purpose

Use the ANALYZE statement to:

- Collect or delete statistics about an index or index partition, table or table partition, index-organized table, cluster, or scalar object attribute.
- Validate the structure of an index or index partition, table or table partition, index-organized table, cluster, or object reference (REF).
- Identify migrated and chained rows of a table or cluster.

For most statistics collection purposes, Oracle Corporation recommends that you use the DBMS_STATS package. That package lets you collect statistics in parallel, collect global statistics for partitioned objects, and fine tune your statistics collection in other ways.

However, you can use this statement for any of the purposes described in this section, and you **must** use this statement (rather than the DBMS_STATS package) for the following purposes:

- To use the VALIDATE or LIST CHAINED ROWS clauses
- To sample a number (rather than a percentage) of rows
- To collect statistics not used by the optimizer (such as information on freelist blocks)

> **See Also:** *Oracle8i Supplied PL/SQL Packages Reference* for more information on this package

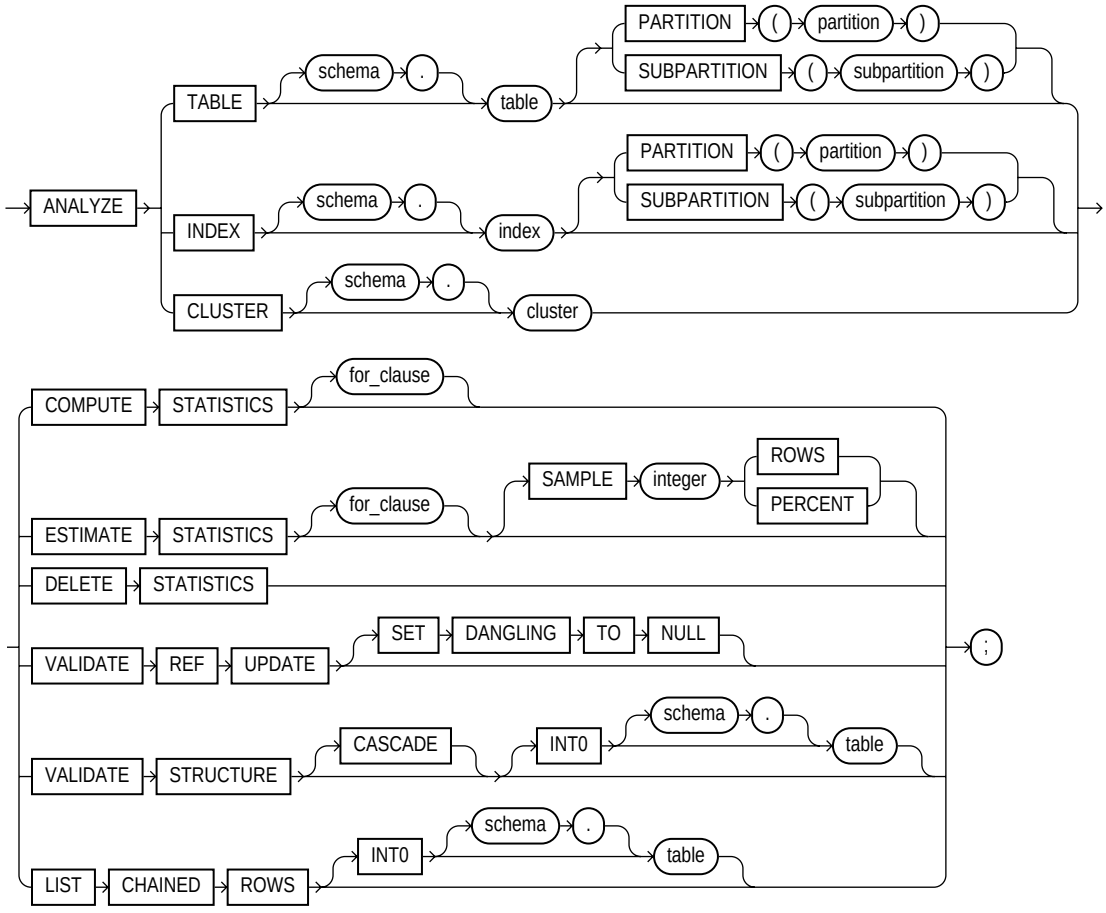
Prerequisites

The schema object to be analyzed must be local, and it must be in your own schema or you must have the ANALYZE ANY system privilege.

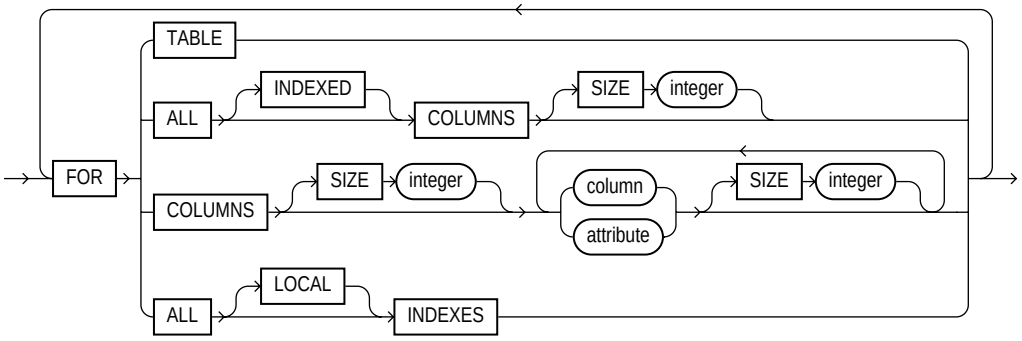
If you want to list chained rows of a table or cluster into a list table, the list table must be in your own schema, or you must have INSERT privilege on the list table, or you must have INSERT ANY TABLE system privilege.

If you want to validate a partitioned table, you must have INSERT privilege on the table into which you list analyzed rowids, or you must have INSERT ANY TABLE system privilege.

Syntax


ANALYZE
TABLE
schema
.
table
PARTITION
(
partition
)
SUBPARTITION
(
subpartition
)
INDEX
schema
.
index
PARTITION
(
partition
)
SUBPARTITION
(
subpartition
)
CLUSTER
schema
.
cluster
COMPUTE
STATISTICS
for_clause
ESTIMATE
STATISTICS
for_clause
SAMPLE
integer
ROWS
PERCENT
DELETE
STATISTICS
VALIDATE
REF
UPDATE
SET
DANGLING
TO
NULL
VALIDATE
STRUCTURE
CASCADE
INT0
schema
.
table
LIST
CHAINED
ROWS
INT0
schema
.
table
;

for_clause::=



Keywords and Parameters

schema

Specify the schema containing the index, table, or cluster. If you omit *schema*, Oracle assumes the index, table, or cluster is in your own schema.

INDEX *index*

Specify an index to be analyzed (if no *for_clause* is used).

Oracle collects the following statistics for an index. Statistics marked with an asterisk are always computed exactly. For **conventional indexes**, the statistics appear in the data dictionary views USER_INDEXES, ALL_INDEXES, and DBA_INDEXES in the columns in parentheses.

- Depth of the index from its root block to its leaf blocks* (BLEVEL)
- Number of leaf blocks (LEAF_BLOCKS)
- Number of distinct index values (DISTINCT_KEYS)
- Average number of leaf blocks per index value (AVG_LEAF_BLOCKS_PER_KEY)
- Average number of data blocks per index value (for an index on a table) (AVG_DATA_BLOCKS_PER_KEY)
- Clustering factor (how well ordered the rows are about the indexed values) (CLUSTERING_FACTOR)

For **domain indexes**, this statement invokes the user-defined statistics collection function specified in the statistics type associated with the index (see ASSOCIATE

STATISTICS on page 8-110). If no statistics type is associated with the domain index, the statistics type associated with its indextype is used. If no statistics type exists for either the index or its indextype, no user-defined statistics are collected. User-defined index statistics appear in the STATISTICS column of the data dictionary views USER_USTATS, ALL_USTATS, and DBA_USTATS.

Restriction: You cannot analyze a domain index that is marked LOADING or FAILED.

See Also:

- CREATE INDEX on page 9-52 for more information on domain indexes
- *Oracle8i Reference* for information on the data dictionary views

TABLE *table*

Specify a table to be analyzed. When you collect statistics for a table, Oracle also automatically collects the statistics for each of the table's indexes and domain indexes, provided that no *for_clauses* are used.

When you analyze a table, Oracle collects statistics about expressions occurring in any function-based indexes as well. Therefore, be sure to create function-based indexes on the table before analyzing the table.

See Also: CREATE INDEX on page 9-52 for more information about function-based indexes

When analyzing a table, Oracle skips all domain indexes marked LOADING or FAILED.

Oracle collects the following statistics for a table. Statistics marked with an asterisk are always computed exactly. Table statistics, including the status of domain indexes, appear in the data dictionary views USER_TABLES, ALL_TABLES, and DBA_TABLES in the columns shown in parentheses.

- Number of rows (NUM_ROWS)
- * Number of data blocks below the high water mark (that is, the number of data blocks that have been formatted to receive data, regardless whether they currently contain data or are empty) (BLOCKS)
- * Number of data blocks allocated to the table that have never been used (EMPTY_BLOCKS)

- Average available free space in each data block in bytes (AVG_SPACE)
- Number of chained rows (CHAIN_COUNT)
- Average row length, including the row's overhead, in bytes (AVG_ROW_LEN)

Restrictions:

- You cannot use ANALYZE to collect statistics on data dictionary tables.
- You cannot use ANALYZE to collect default statistics on a temporary table. However, if you have created an association between one or more columns of a temporary table and a user-defined statistics type, you can use ANALYZE to collect the user-defined statistics on the temporary table. (The association must already exist.)
- You cannot compute or estimate statistics for the following column types: REFs, varrays, nested tables, LOBs (LOBs are not analyzed, they are skipped), LONGs, or object types. However, if a statistics type is associated with such a column, user-defined statistics are collected.

See Also:

- ASSOCIATE STATISTICS on page 8-110
- *Oracle8i Reference* for information on the data dictionary views

PARTITION | SUBPARTITION

Specify the *partition* or *subpartition* on which you want statistics to be gathered. You cannot use this clause when analyzing clusters.

If you specify PARTITION and *table* is composite-partitioned, Oracle analyzes all the subpartitions within the specified partition.

CLUSTER *cluster*

Specify a cluster to be analyzed. When you collect statistics for a cluster, Oracle also automatically collects the statistics for all the cluster's tables and all their indexes, including the cluster index.

For both indexed and hash clusters, Oracle collects the average number of data blocks taken up by a single cluster key (AVG_BLOCKS_PER_KEY). These statistics appear in the data dictionary views ALL_CLUSTERS, USER_CLUSTERS and DBA_CLUSTERS.

See Also: *Oracle8i Reference* for information on the data dictionary views

COMPUTE STATISTICS

COMPUTE STATISTICS instructs Oracle to compute exact statistics about the analyzed object and store them in the data dictionary. When you analyze a table, both table and column statistics are collected.

Both computed and estimated statistics are used by the Oracle optimizer to choose the execution plan for SQL statements that access analyzed objects. These statistics may also be useful to application developers who write such statements.

> **See Also:** *Oracle8i Performance Guide and Reference* for information on how these statistics are used

for_clause

The *for_clause* lets you specify whether an entire table or index, or just particular columns, will be analyzed. The following clauses apply only to the ANALYZE TABLE version of this statement:

FOR TABLE	Specify FOR TABLE to restrict the statistics collected to only table statistics rather than table and column statistics.
FOR COLUMNS	Specify FOR COLUMNS to restrict the statistics collected to only column statistics for the specified columns and scalar object attributes, rather than for all columns and attributes; *attribute* specifies the qualified column name of an item in an object.
FOR ALL COLUMNS	Specify FOR ALL COLUMNS to collect column statistics for all columns and scalar object attributes.
FOR ALL INDEXED COLUMNS	Specify FOR ALL INDEXED COLUMNS to collect column statistics for all indexed columns in the table.

Column statistics can be based on the entire column or can use a histogram by specifying SIZE integer (see below).

Oracle collects the following column statistics:

- Number of distinct values in the column as a whole
- Maximum and minimum values in each band

> **See Also:** *Oracle8i Performance Guide and Reference* and "Histogram Examples" on page 8-107 for more information on histograms

Column statistics appear in the data dictionary views USER_TAB_COLUMNS, ALL_TAB_COLUMNS, and DBA_TAB_COLUMNS. Histograms appear in the data dictionary views USER_TAB_HISTOGRAMS, DBA_TAB_HISTOGRAMS, and ALL_TAB_HISTOGRAMS; USER_PART_HISTOGRAMS, DBA_PART_HISTOGRAMS, and ALL_PART_HISTOGRAMS; and USER_SUBPART_HISTOGRAMS, DBA_SUBPART_HISTOGRAMS, and ALL_SUBPART_HISTOGRAMS.

Note: The MAXVALUE and MINVALUE columns of USER_, DBA_, and ALL_TAB_COLUMNS have a length of 32 bytes. If you analyze columns with a length >32 bytes, and if the columns are padded with leading blanks, Oracle may take into account only the leading blanks and return unexpected statistics.

If a user-defined statistics type has been associated with any columns, the *for_clause* collects user-defined statistics using that statistics type. If no statistics type is associated with a column, Oracle checks to see if any statistics type has been associated with the type of the column, and uses that statistics type. If no statistics type has been associated with either the column or its user-defined type, no user-defined statistics are collected. User-defined column statistics appear in the STATISTICS column of the data dictionary views USER_USTATS, ALL_USTATS, and DBA_USTATS.

If you want to collect statistics on both the table as a whole and on one or more columns, be sure to generate the statistics for the table first, and then for the columns. Otherwise, the table-only ANALYZE will overwrite the histograms generated by the column ANALYZE. For example, issue the following statements:

```
ANALYZE TABLE emp ESTIMATE STATISTICS;
ANALYZE TABLE emp ESTIMATE STATISTICS
   FOR ALL COLUMNS;
```

FOR ALL INDEXES — Specify FOR ALL INDEXES if you want all indexes associated with the table to be analyzed.

FOR ALL LOCAL INDEXES	Specify FOR ALL LOCAL INDEXES if you want all local index partitions to be analyzed. You must specify the keyword LOCAL if the PARTITION clause and INDEX are specified.
SIZE *integer*	Specify the maximum number of buckets in the histogram. The default value is 75, minimum value is 1, and maximum value is 254.

> **Note:** Oracle does not create a histogram with more buckets than the number of rows in the sample. Also, if the sample contains any values that are very repetitious, Oracle creates the specified number of buckets, but the value indicated by the NUM_BUCKETS column of the ALL_, DBA_, and USER_TAB_COLUMNS views may be smaller because of an internal compression algorithm.

ESTIMATE STATISTICS

ESTIMATE STATISTICS instructs Oracle to estimate statistics about the analyzed object and stores them in the data dictionary.

Both computed and estimated statistics are used by the Oracle optimizer to choose the execution plan for SQL statements that access analyzed objects. These statistics may also be useful to application developers who write such statements.

> **See Also:** *Oracle8i Performance Guide and Reference* for information on how these statistics are used

for_clause	See the description under COMPUTE STATISTICS on page 8-101
SAMPLE *integer*	Specify the amount of data from the analyzed object Oracle should sample to estimate statistics. If you omit this parameter, Oracle samples 1064 rows. The default sample value is adequate for tables up to a few thousand rows. If your tables are larger, specify a higher value for SAMPLE. If you specify more than half of the data, Oracle reads all the data and computes the statistics.

- ROWS causes Oracle to sample *integer* rows of the table or cluster or *integer* entries from the index. The integer must be at least 1.
- PERCENT causes Oracle to sample *integer* percent of the rows from the table or cluster or *integer* percent of the index entries. The integer can range from 1 to 99.

DELETE STATISTICS

Specify DELETE STATISTICS to delete any statistics about the analyzed object that are currently stored in the data dictionary. Use this statement when you no longer want Oracle to use the statistics.

When you use this clause on a table, Oracle also automatically removes statistics for all the table's indexes. When you use this clause on a cluster, Oracle also automatically removes statistics for all the cluster's tables and all their indexes, including the cluster index.

If user-defined column or index statistics were collected for an object, Oracle also removes the user-defined statistics by invoking the statistics deletion function specified in the statistics type that was used to collect the statistics.

VALIDATE REF UPDATE

Specify VALIDATE REF UPDATE to validate the REFs in the specified table, checks the rowid portion in each REF, compares it with the true rowid, and corrects, if necessary. You can use this clause only when analyzing a table.

SET DANGLING TO NULL — SET DANGLING TO NULL sets to NULL any REFs (whether or not scoped) in the specified table that are found to point to an invalid or nonexistent object.

> **Note:** If the owner of the table does not have SELECT object privilege on the referenced objects, Oracle will consider them invalid and set them to NULL. Subsequently these REFs will not be available in a query, even if it is issued by a user with appropriate privileges on the objects.

VALIDATE STRUCTURE

Specify VALIDATE STRUCTURE to validate the structure of the analyzed object. The statistics collected by this clause are not used by the Oracle optimizer, as are

statistics collected by the COMPUTE STATISTICS and ESTIMATE STATISTICS clauses.

- For a table, Oracle verifies the integrity of each of the table's data blocks and rows.
- For a cluster, Oracle automatically validates the structure of the cluster's tables.
- For a partitioned table, Oracle also verifies that each row belongs to the correct partition. If a row does not collate correctly, its rowid is inserted into the INVALID_ROWS table.
- For a temporary table, Oracle validates the structure of the table and its indexes during the current session.
- For an index, Oracle verifies the integrity of each data block in the index and checks for block corruption. This clause does not confirm that each row in the table has an index entry or that each index entry points to a row in the table. You can perform these operations by validating the structure of the table with the CASCADE clause (described below).

Oracle stores statistics about the index in the data dictionary views INDEX_STATS and INDEX_HISTOGRAM.

See Also: *Oracle8i Reference* for information on these views

Validating the structure of an object prevents SELECT, INSERT, UPDATE, and DELETE statements from concurrently accessing the object. Therefore, do not use this clause on the tables, clusters, and indexes of your production applications during periods of high database activity.

If Oracle encounters corruption in the structure of the object, an error message is returned to you. In this case, drop and re-create the object.

INTO *table* Specify a table into which Oracle lists the rowids of the partitions whose rows do not collate correctly. If you omit *schema*, Oracle assumes the list is in your own schema. If you omit this clause altogether, Oracle assumes that the table is named INVALID_ROWS. The SQL script used to create this table is UTLVALID.SQL.

CASCADE — Specify CASCADE if you want Oracle to validate the structure of the indexes associated with the table or cluster. If you use this clause when validating a table, Oracle also validates the table's indexes. If you use this clause when validating a cluster, Oracle also validates all the clustered tables' indexes, including the cluster index.

If you use this clause to validate an enabled (but previously disabled) function-based index, validation errors may result. In this case, you must rebuild the index.

LIST CHAINED ROWS

LIST CHAINED ROWS lets you identify migrated and chained rows of the analyzed table or cluster. You cannot use this clause when analyzing an index.

INTO *table* — Specify a table into which Oracle lists the migrated and chained rows. If you omit *schema*, Oracle assumes the list table is in your own schema. If you omit this clause altogether, Oracle assumes that the table is named CHAINED_ROWS. The list table must be on your local database.

You can create the CHAINED_ROWS table using one of these scripts:

- UTLCHAIN.SQL uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.)
- UTLCHN1.SQL uses universal rowids, so it can accommodate rows from both conventional and index-organized tables.

If you create your own chained-rows table, it must follow the format prescribed by one of these two scripts.

See Also: *Oracle8i Migration* for compatibility issues related to the use of these scripts

Note: If you are analyzing index-organized tables based on primary keys (rather than universal rowids), you must create a separate chained-rows table for each index-organized table to accommodate its primary-key storage. Use the SQL scripts DBMSIOTC.SQL and PRVTIOTC.PLB to define the BUILD_CHAIN_ROWS_TABLE procedure, and then execute this procedure to create an IOT_CHAINED_ROWS table for each such index-organized table.

See Also:

- The DBMS_IOT package in *Oracle8i Supplied PL/SQL Packages Reference* for information on the SQL scripts
- *Oracle8i Performance Guide and Reference* for information on eliminating migrated and chained rows

Examples

Analyzing a Cluster Example The following statement estimates statistics for the cust_history table and all of its indexes:

```
ANALYZE TABLE cust_history
   ESTIMATE STATISTICS;
```

Deleting Statistics Example The following statement deletes statistics about the cust_history table and all its indexes from the data dictionary:

```
ANALYZE TABLE cust_history
   DELETE STATISTICS;
```

Histogram Examples The following statement creates a 10-band histogram on the SAL column of the EMP table:

```
ANALYZE TABLE emp
   COMPUTE STATISTICS FOR COLUMNS sal SIZE 10;
```

You can then query the USER_TAB_COLUMNS data dictionary view to retrieve statistics:

```
SELECT NUM_DISTINCT, NUM_BUCKETS, SAMPLE_SIZE
   FROM USER_TAB_COLUMNS
   WHERE TABLE_NAME = 'EMP' AND COLUMN_NAME = 'SAL';
```

```
NUM_DISTINCT NUM_BUCKETS SAMPLE_SIZE
------------ ---------- -----------
          12          7          14
```

Even though the ANALYZE statement specified 10 buckets, Oracle created only 7 in this example. For an explanation, see the note on SIZE integer on page 8-103.

You can also collect histograms for a single partition of a table. The following statement analyzes the emp table partition p1:

```
ANALYZE TABLE emp PARTITION (p1) COMPUTE STATISTICS;
```

Analyzing an Index Example The following statement validates the structure of the index parts_index:

```
ANALYZE INDEX parts_index VALIDATE STRUCTURE;
```

Analyzing a Table Examples The following statement analyzes the emp table and all of its indexes:

```
ANALYZE TABLE emp VALIDATE STRUCTURE CASCADE;
```

For a table, the VALIDATE REF UPDATE clause verifies the REFs in the specified table, checks the rowid portion of each REF, and then compares it with the true rowid. If the result is an incorrect rowid, the REF is updated so that the rowid portion is correct.

The following statement validates the REFs in the emp table:

```
ANALYZE TABLE emp
   VALIDATE REF UPDATE;
```

Analyzing a Cluster Example The following statement analyzes the order_ custs cluster, all of its tables, and all of their indexes, including the cluster index:

```
ANALYZE CLUSTER order_custs
    VALIDATE STRUCTURE CASCADE;
```

Listing Chained Rows Example The following statement collects information about all the chained rows of the table order_hist:

```
ANALYZE TABLE order_hist
    LIST CHAINED ROWS INTO cr;
```

The preceding statement places the information into the table cr. You can then examine the rows with this query:

```
SELECT *
    FROM cr;

OWNER_NAME  TABLE_NAME  CLUSTER_NAME  HEAD_ROWID         TIMESTAMP
----------  ----------  ------------  ------------------ ---------
SCOTT       ORDER_HIST                AAAAZzAABAAABrXAAA 15-MAR-96
```

COMPUTE STATISTICS Example The following statement calculates statistics for a scalar object attribute:

```
ANALYZE TABLE emp COMPUTE STATISTICS FOR COLUMNS addr.street;
```

ASSOCIATE STATISTICS

Purpose

Use the ASSOCIATE STATISTICS statement to associate a statistics type (or default statistics) containing functions relevant to statistics collection, selectivity, or cost with one or more columns, standalone functions, packages, types, domain indexes, or indextypes.

For a listing of all current statistics type associations, refer to the USER_ASSOCIATIONS table. If you analyze the object with which you are associating statistics, you can also view the associations in the USER_USTATS table.

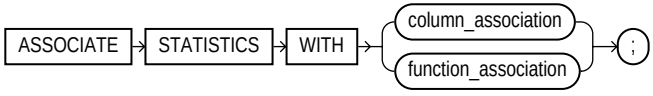
> **See Also:** ANALYZE on page 8-96 for information on the order of precedence with which ANALYZE uses associations

Prerequisites

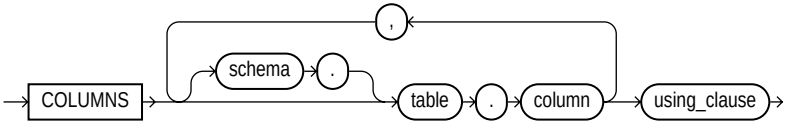
To issue this statement, you must have the appropriate privileges to alter the base object (table, function, package, type, domain index, or indextype). In addition, unless you are associating only default statistics, you must have execute privilege on the statistics type. The statistics type must already have been defined.

> **See Also:** CREATE TYPE on page 10-80 for information on defining types

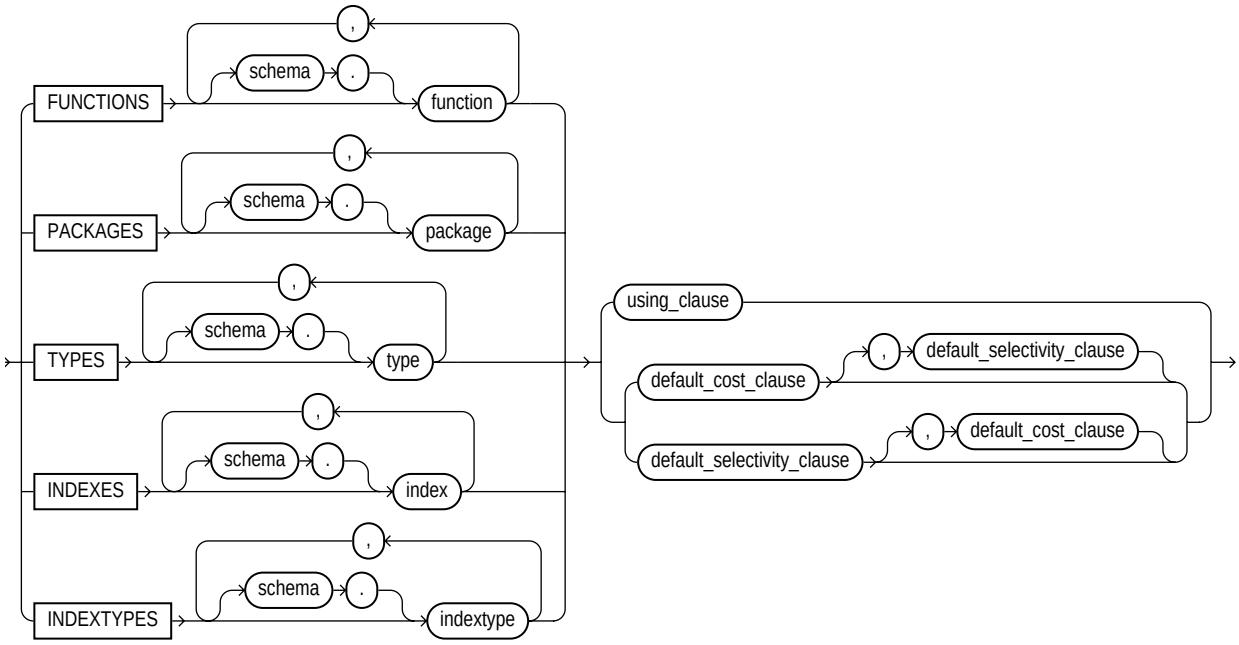
Syntax



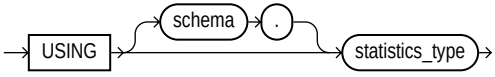
column_association::=



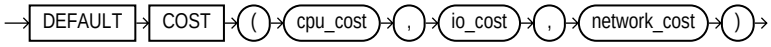
function_association::=



using_clause::=



default_cost_clause::=



default_selectivity_clause::=

Keywords and Parameters

column_association

Specify one or more table columns. If you do not specify *schema*, Oracle assumes the table is in your own schema.

function_association

Specify one or more standalone functions, packages, user-defined datatypes, domain indexes, or indextypes. If you do not specify *schema*, Oracle assumes the object is in your own schema.

- FUNCTIONS refers only to standalone functions, not to method types or to built-in functions.
- TYPES refers only to user-defined types, not to internal SQL datatypes.

Restriction: You cannot specify an object for which you have already defined an association. You must first disassociate the statistics from this object.

See Also: DISASSOCIATE STATISTICS on page 10-123

using_clause

Specify the statistics type being associated with columns, functions, packages, types, domain indexes, or indextypes. The *statistics_type* must already have been created.

default_cost_clause

Specify default costs for standalone functions, packages, types, domain indexes, or indextypes. If you specify this clause, you must include one number each for CPU cost, I/O cost, and network cost, in that order. Each cost is for a single execution of the function or method or for a single domain index access. Accepted values are integers of zero or greater.

default_selectivity_clause

Specify as a percent the default selectivity for predicates with standalone functions, types, packages, or user-defined operators. The *default_selectivity* must be a whole number between 0 and 100. Values outside this range are ignored.

Restriction: You cannot specify DEFAULT SELECTIVITY for domain indexes or indextypes.

Examples

Standalone Function Example This statement creates an association for a standalone function FN and causes the optimizer to call the appropriate cost function (if present) in the statistics type stat_fn.

```
ASSOCIATE STATISTICS WITH FUNCTIONS fn USING stat_fn;
```

Default Cost Example This statement specifies that using the domain index t_a to implement a given predicate always has a CPU cost of 100, I/O of 5, and network cost of 0.

```
ASSOCIATE STATISTICS WITH INDEXES t_a DEFAULT COST (100,5,0);
```

The optimizer will simply use these default costs instead of calling a cost function.

AUDIT

Purpose

Use the AUDIT statement to:

- Track the occurrence of SQL statements in subsequent user sessions. You can track the occurrence of a specific SQL statement or of all SQL statements authorized by a particular system privilege. Auditing operations on SQL statements apply only to subsequent sessions, not to current sessions.
- Track operations on a specific schema object. Auditing operations on schema objects apply to current sessions as well as to subsequent sessions.

> **See Also:** NOAUDIT on page 11-66 for information on disabling auditing of SQL statement

Prerequisites

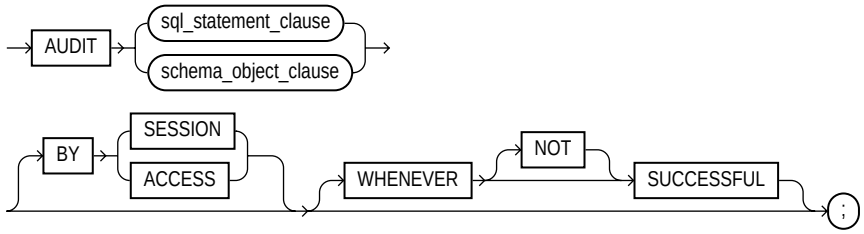
To audit occurrences of a SQL statement, you must have AUDIT SYSTEM system privilege.

To audit operations on a schema object, the object you choose for auditing must be in your own schema or you must have AUDIT ANY system privilege. In addition, if the object you choose for auditing is a directory object, even if you created it, you must have AUDIT ANY system privilege.

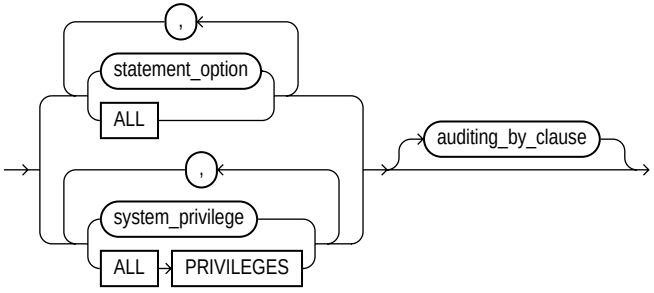
To collect auditing results, you must set the initialization parameter AUDIT_TRAIL to DB. You can specify auditing options regardless of whether auditing is enabled. However, Oracle does not generate audit records until you enable auditing.

> **See Also:** *Oracle8i Reference* for information on the AUDIT_TRAIL parameter

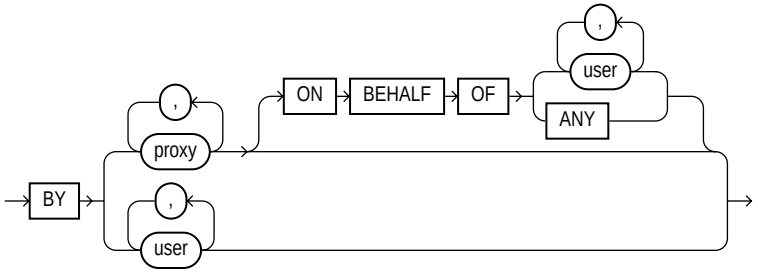
Syntax



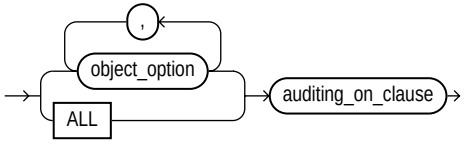
sql_statement_clause::=



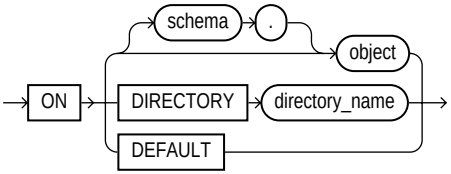
auditing_by_clause::=



schema_object_clause::=



auditing_on_clause::=



Keywords and Parameters

sql_statement_clause

statement_option Specify a statement option to audit specific SQL statements.

See Also: Table 8–1 on page 8-120 and Table 8–2 on page 8-122 for a list of these statement options and the SQL statements they audit

For each audited operation, Oracle produces an audit record containing this information:

- The user performing the operation
- The type of operation
- The object involved in the operation
- The date and time of the operation

Oracle writes audit records to the audit trail, which is a database table containing audit records. You can review database activity by examining the audit trail through data dictionary views.

See Also: *Oracle8i Reference* for information on these views

system_privilege Specify a system privilege to audit SQL statements that are authorized by the specified system privilege.

See Also: Table 11–1, " System Privileges" for a list of all system privileges and the SQL statements that they authorize

Rather than specifying many individual system privileges, you can specify the roles CONNECT, RESOURCE, and DBA. Doing so is equivalent to auditing all of the system privileges granted to those roles.

> **See Also:** GRANT on page 11-31 for more information on these roles

Oracle also provides two shortcuts for specifying groups of system privileges and statement options at once:

ALL	Specify ALL to audit all statements options shown in Table 8–1 but not the additional statement options shown in Table 8–2.
ALL PRIVILEGES	Specify ALL PRIVILEGES to audit system privileges.

> **Note:** Oracle Corporation recommends that you specify individual system privileges and statement options for auditing rather than roles or shortcuts. The specific system privileges and statement options encompassed by roles and shortcuts change from one release to the next and may not be supported in future versions of Oracle.

---

auditing_by_clause Specify the *auditing_by_clause* to audit only those SQL statements issued by particular users. If you omit this clause, Oracle audits all users' statements.

BY *user*	Use this clause to restrict auditing to only SQL statements issued by the specified users.
BY *proxy*	Use this clause to restrict auditing to only SQL statements issued by the specified proxies. **See Also:** *Oracle8i Concepts* for more information on proxies and their use of the database
ON BEHALF OF	■ *user* indicates auditing of statements executed on behalf of a particular user. ■ ANY indicates auditing of statements executed on behalf of any user.

schema_object_clause

object_option — Specify the particular operation for auditing. Table 8–3 on page 8-124 shows each object option and the types of objects to which it applies. The name of each object option specifies a SQL statement to be audited. For example, if you choose to audit a table with the ALTER option, Oracle audits all ALTER TABLE statements issued against the table. If you choose to audit a sequence with the SELECT option, Oracle audits all statements that use any of the sequence's values.

ALL — Specify ALL as a shortcut equivalent to specifying all object options applicable for the type of object.

auditing_on_clause — The *auditing_on_clause* lets you specify the particular schema object to be audited.

- *schema* — Specify the schema containing the object chosen for auditing. If you omit *schema*, Oracle assumes the object is in your own schema.
- *object* — Specify the name of the object to be audited. The object must be a table, view, sequence, stored procedure, function, package, materialized view, or library.

  You can also specify a synonym for a table, view, sequence, procedure, stored function, package, or materialized view.
- ON DEFAULT — Specify ON DEFAULT to establish the specified object options as default object options for subsequently created objects. Once you have established these default auditing options, any subsequently created object is automatically audited with those options. The default auditing options for a view are always the union of the auditing options for the view's base tables. You can see the current default auditing options by querying the ALL_DEF_AUDIT_OPTS data dictionary view.

If you change the default auditing options, the auditing options for previously created objects remain the same. You can change the auditing options for an existing object only by specifying the object in the ON clause of the AUDIT statement.

ON DIRECTORY *directory_name*

The ON DIRECTORY clause lets you specify the name of a directory chosen for auditing.

BY SESSION

Specify BY SESSION if you want Oracle to write a single record for all SQL statements of the same type issued and operations of the same type executed on the same schema objects in the same session.

BY ACCESS

Specify BY ACCESS if you want Oracle to write one record for each audited statement and operation.

If you specify statement options or system privileges that audit data definition language (DDL) statements, Oracle automatically audits by access regardless of whether you specify the BY SESSION clause or BY ACCESS clause.

For statement options and system privileges that audit SQL statements other than DDL, you can specify either BY SESSION or BY ACCESS. BY SESSION is the default.

WHENEVER [NOT] SUCCESSFUL

Specify WHENEVER SUCCESSFUL to audit only SQL statements and operations that succeed.

Specify WHENEVER NOT SUCCESSFUL to audit only statements and operations that fail or result in errors.

If you omit this clause, Oracle performs the audit regardless of success or failure.

Tables of Auditing Options

Table 8–1 Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
CLUSTER	CREATE CLUSTER
	AUDIT CLUSTER
	DROP CLUSTER
	TRUNCATE CLUSTER
CONTEXT	CREATE CONTEXT
	DROP CONTEXT
DATABASE LINK	CREATE DATABASE LINK
	DROP DATABASE LINK
DIMENSION	CREATE DIMENSION
	ALTER DIMENSION
	DROP DIMENSION
DIRECTORY	CREATE DIRECTORY
	DROP DIRECTORY
INDEX	CREATE INDEX
	ALTER INDEX
	DROP INDEX
NOT EXISTS	All SQL statements that fail because a specified object does not exist.
PROCEDURE[a]	CREATE FUNCTION
	CREATE LIBRARY
	CREATE PACKAGE
	CREATE PACKAGE BODY
	CREATE PROCEDURE
	DROP FUNCTION
	DROP LIBRARY
	DROP PACKAGE
	DROP PROCEDURE

Table 8–1 (Cont.) Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
PROFILE	CREATE PROFILE
	ALTER PROFILE
	DROP PROFILE
PUBLIC DATABASE LINK	CREATE PUBLIC DATABASE LINK
	DROP PUBLIC DATABASE LINK
PUBLIC SYNONYM	CREATE PUBLIC SYNONYM
	DROP PUBLIC SYNONYM
ROLE	CREATE ROLE
	ALTER ROLE
	DROP ROLE
	SET ROLE
ROLLBACK STATEMENT	CREATE ROLLBACK SEGMENT
	ALTER ROLLBACK SEGMENT
	DROP ROLLBACK SEGMENT
SEQUENCE	CREATE SEQUENCE
	DROP SEQUENCE
SESSION	Logons
SYNONYM	CREATE SYNONYM
	DROP SYNONYM
SYSTEM AUDIT	AUDIT *sql_statements*
	NOAUDIT *sql_statements*
SYSTEM GRANT	GRANT *system_privileges_and_roles*
	REVOKE *system_privileges_and_roles*
TABLE	CREATE TABLE
	DROP TABLE
	TRUNCATE TABLE
TABLESPACE	CREATE TABLESPACE
	ALTER TABLESPACE
	DROP TABLESPACE

Table 8–1 (Cont.) Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
TRIGGER	CREATE TRIGGER
	ALTER TRIGGER
	with ENABLE and DISABLE clauses
	DROP TRIGGER
	ALTER TABLE
	with ENABLE ALL TRIGGERS clause
	and DISABLE ALL TRIGGERS clause
TYPE	CREATE TYPE
	CREATE TYPE BODY
	ALTER TYPE
	DROP TYPE
	DROP TYPE BODY
USER	CREATE USER
	ALTER USER
	DROP USER
VIEW	CREATE VIEW
	DROP VIEW

[a]Java schema objects (sources, classes, and resources) are considered the same as procedures for purposes of auditing SQL statements.

Table 8–2 Additional Statement Auditing Options for SQL Statements

Statement Option	SQL Statements and Operations
ALTER SEQUENCE	ALTER SEQUENCE
ALTER TABLE	ALTER TABLE
COMMENT TABLE	COMMENT ON TABLE *table, view, materialized view*
	COMMENT ON COLUMN *table.column, view.column, materialized view.column*
DELETE TABLE	DELETE FROM *table, view*

Table 8–2 (Cont.) Additional Statement Auditing Options for SQL Statements

Statement Option	SQL Statements and Operations
EXECUTE PROCEDURE	CALL
	Execution of any procedure or function or access to any variable, library, or cursor inside a package.
GRANT DIRECTORY	GRANT privilege ON directory
	REVOKE privilege ON directory
GRANT PROCEDURE	GRANT privilege ON procedure, function, package
	REVOKE privilege ON procedure, function, package
GRANT SEQUENCE	GRANT privilege ON sequence
	REVOKE privilege ON sequence
GRANT TABLE	GRANT privilege ON table, view, materialized view.
	REVOKE privilege ON table, view, materialized view
GRANT TYPE	GRANT privilege ON TYPE
	REVOKE privilege ON TYPE
INSERT TABLE	INSERT INTO table, view
LOCK TABLE	LOCK TABLE table, view
SELECT SEQUENCE	Any statement containing sequence.CURRVAL or sequence.NEXTVAL
SELECT TABLE	SELECT FROM table, view, materialized view
UPDATE TABLE	UPDATE table, view

Table 8–3 Object Auditing Options

Object Option	Table	View	Sequence	Procedure Function Package[a]	Materialized View / Snapshot	Directory	Library	Object Type	Context
ALTER	X		X		X			X	
AUDIT	X	X	X	X	X	X		X	X
COMMENT	X	X			X				
DELETE	X	X			X				
EXECUTE				X			X		
GRANT	X	X	X	X		X	X	X	X
INDEX	X				X				
INSERT	X	X			X				
LOCK	X	X			X				
READ						X			
RENAME	X	X		X	X				
SELECT	X	X	X		X				
UPDATE	X	X			X				

[a] Java schema objects (sources, classes, and resources) are considered the same as procedures, functions, and packages for purposes of auditing options.

Examples

Audit SQL Statements Relating to Roles Example To choose auditing for every SQL statement that creates, alters, drops, or sets a role, regardless of whether the statement completes successfully, issue the following statement:

```
AUDIT ROLE;
```

To choose auditing for every statement that successfully creates, alters, drops, or sets a role, issue the following statement:

```
AUDIT ROLE
    WHENEVER SUCCESSFUL;
```

To choose auditing for every CREATE ROLE, ALTER ROLE, DROP ROLE, or SET ROLE statement that results in an Oracle error, issue the following statement:

```
AUDIT ROLE
    WHENEVER NOT SUCCESSFUL;
```

Audit Query and Update SQL Statements Example To choose auditing for any statement that queries or updates any table, issue the following statement:

```
AUDIT SELECT TABLE, UPDATE TABLE;
```

To choose auditing for statements issued by the users scott and blake that query or update a table or view, issue the following statement:

```
AUDIT SELECT TABLE, UPDATE TABLE
    BY scott, blake;
```

Audit Deletions Example To choose auditing for statements issued using the DELETE ANY TABLE system privilege, issue the following statement:

```
AUDIT DELETE ANY TABLE;
```

Audit Statements Relating to Directories Example To choose auditing for statements issued using the CREATE ANY DIRECTORY system privilege, issue the following statement:

```
AUDIT CREATE ANY DIRECTORY;
```

To choose auditing for CREATE DIRECTORY (and DROP DIRECTORY) statements that do not use the CREATE ANY DIRECTORY system privilege, issue the following statement:

```
AUDIT DIRECTORY;
```

Audit Queries on a Table Example To choose auditing for every SQL statement that queries the emp table in the schema scott, issue the following statement:

```
AUDIT SELECT
    ON scott.emp;
```

To choose auditing for every statement that successfully queries the emp table in the schema scott, issue the following statement:

```
AUDIT SELECT
    ON scott.emp
    WHENEVER SUCCESSFUL;
```

To choose auditing for every statement that queries the emp table in the schema scott and results in an Oracle error, issue the following statement:

```
AUDIT SELECT
    ON scott.emp
    WHENEVER NOT SUCCESSFUL;
```

Audit Inserts and Updates on a Table Example To choose auditing for every statement that inserts or updates a row in the dept table in the schema blake, issue the following statement:

```
AUDIT INSERT, UPDATE
    ON blake.dept;
```

Audit All Operations on a Sequence Example To choose auditing for every statement that performs any operation on the order sequence in the schema adams, issue the following statement:

```
AUDIT ALL
    ON adams.order;
```

The above statement uses the ALL shortcut to choose auditing for the following statements that operate on the sequence:

- ALTER SEQUENCE
- AUDIT
- GRANT
- any statement that accesses the sequence's values using the pseudocolumns CURRVAL or NEXTVAL

Audit Read Operations on a Directory Example To choose auditing for every statement that reads files from the bfile_dir1 directory, issue the following statement:

```
AUDIT READ ON DIRECTORY bfile_dir1;
```

Set Default Auditing Options Example The following statement specifies default auditing options for objects created in the future:

```
AUDIT ALTER, GRANT, INSERT, UPDATE, DELETE
    ON DEFAULT;
```

Any objects created later are automatically audited with the specified options that apply to them, provided that auditing has been enabled:

- If you create a table, Oracle automatically audits any ALTER, GRANT, INSERT, UPDATE, or DELETE statements issued against the table.
- If you create a view, Oracle automatically audits any GRANT, INSERT, UPDATE, or DELETE statements issued against the view.
- If you create a sequence, Oracle automatically audits any ALTER or GRANT statements issued against the sequence.
- If you create a procedure, package, or function, Oracle automatically audits any ALTER or GRANT statements issued against it.

CALL

Purpose

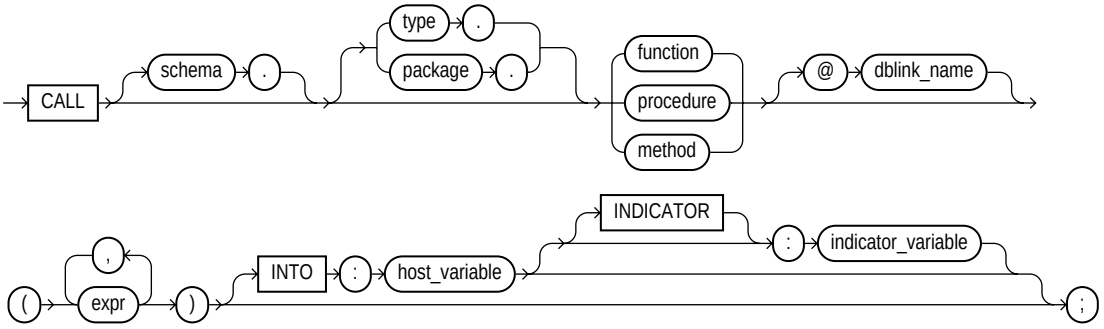
Use the CALL statement to execute a **routine** (a standalone procedure or function, or a procedure or function defined within a type or package) from within SQL.

> **See Also:** *PL/SQL User's Guide and Reference* for information on creating such routine

Prerequisites

You must have EXECUTE privilege on the standalone routine or on the type or package in which the routine is defined.

Syntax



Keywords and Parameters

schema

Specify the schema in which the standalone routine (or the package or type containing the routine) resides. If you do not specify *schema*, Oracle assumes the routine is in your own schema.

type or *package*

Specify the type or package in which the routine is defined.

function|*procedure*| *method*

Specify the name of the function or procedure being called, or a synonym that translates to a function or procedure.

When you call a type's member function or procedure, if the first argument (SELF) is a null IN OUT argument, Oracle returns an error. If SELF is a null IN argument, Oracle returns null. In both cases, the function or procedure is not invoked.

Restriction: If the routine is a function, the INTO clause is mandatory.

@dblink

In a distributed database system, specify the name of the database containing the standalone routine (or the package or function containing the routine). If you omit *dblink*, Oracle looks in your local database.

expr

Specify one or more arguments to the routine.

Restrictions:

- An *expr* cannot be a pseudocolumn or either of the object reference functions VALUE or REF.
- Any *expr* that is an IN OUT or OUT argument of the routine must correspond to a host variable expression.

INTO :*host_variable*

The INTO clause applies only to calls to functions. Specify which host variable will store the return value of the function.

:*indicator_variable*

Specify the value or condition of the host variable.

> **See Also:** *Pro*C/C++ Precompiler Programmer's Guide* for more information on host variables and indicator variables

Example

Calling a Procedure Example The following statement creates a procedure updatesalary, and then calls the procedure, which updates the specified employee ID with a new salary.

```
CREATE OR REPLACE PROCEDURE updateSalary
   (id NUMBER, newsalary NUMBER) IS
   BEGIN
      UPDATE emp SET sal=newsalary WHERE empno=id;
   END;

CALL updateSalary(1404, 50000);
```

COMMENT

Purpose

Use the COMMENT statement to add a comment about a table, view, snapshot, or column into the data dictionary.

You can view the comments on a particular table or column by querying the data dictionary views USER_TAB_COMMENTS, DBA_TAB_COMMENTS, or ALL_TAB_COMMENTS or USER_COL_COMMENTS, DBA_COL_COMMENTS, or ALL_COL_COMMENTS.

To drop a comment from the database, set it to the empty string ' '.

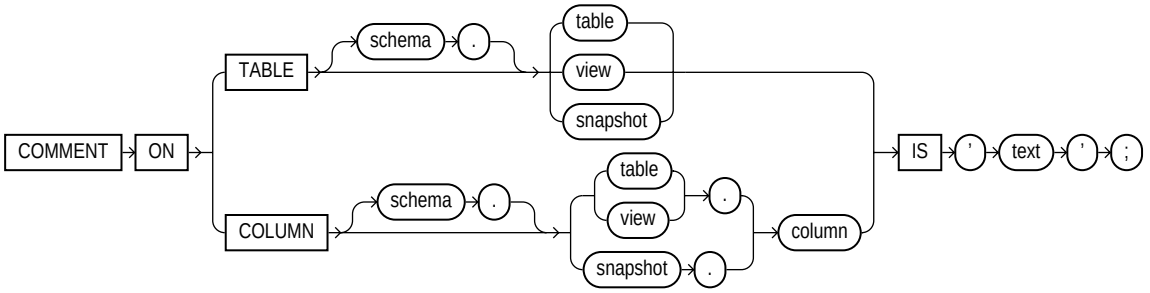
See Also:

- COMMENT on page 8-131
- *Oracle8i Reference* for information on the data dictionary views

Prerequisites

The table, view, or snapshot must be in your own schema or you must have COMMENT ANY TABLE system privilege.

Syntax



Keywords and Parameters

TABLE

Specify the schema and name of the table, view, or materialized view to be commented. If you omit *schema*, Oracle assumes the table, view, or snapshot is in your own schema.

COLUMN

Specify the name of the column of a table, view, or snapshot to be commented. If you omit *schema*, Oracle assumes the table, view, or snapshot is in your own schema.

IS '*text*'

Specify the text of the comment.

See Also: "Text" on page 2-33 for a syntax description of 'text'

Example

COMMENT Example To insert an explanatory remark on the `notes` column of the `shipping` table, you might issue the following statement:

```
COMMENT ON COLUMN shipping.notes
    IS 'Special packing or shipping instructions';
```

To drop this comment from the database, issue the following statement:

```
COMMENT ON COLUMN shipping.notes IS ' ';
```

COMMIT

Purpose

Use the COMMIT statement to end your current **transaction** and make permanent all changes performed in the transaction. A transaction is a sequence of SQL statements that Oracle treats as a single unit. This statement also erases all savepoints in the transaction and releases the transaction's locks.

Note: Oracle issues an implicit COMMIT before and after any data definition language (DDL) statement.

You can also use this statement to

- Commit an in-doubt distributed transaction manually
- Terminate a read-only transaction begun by a SET TRANSACTION statement.

Oracle Corporation recommends that you explicitly end every transaction in your application programs with a COMMIT or ROLLBACK statement, including the last transaction, before disconnecting from Oracle. If you do not explicitly commit the transaction and the program terminates abnormally, the last uncommitted transaction is automatically rolled back.

A normal exit from most Oracle utilities and tools causes the current transaction to be committed. A normal exit from an Oracle precompiler program does not commit the transaction and relies on Oracle to roll back the current transaction.

See Also:

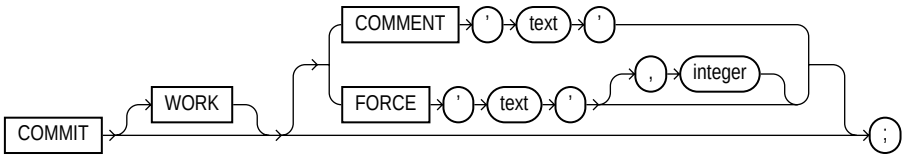
- *Oracle8i Concepts* for more information on transactions
- SET TRANSACTION on page 11-125 for more information on specifying characteristics of a transaction

Prerequisites

You need no privileges to commit your current transaction.

To manually commit a distributed in-doubt transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

Syntax



Keywords and Parameters

WORK

The WORK keyword is supported for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.

COMMENT '*text*'

Specify a comment to be associated with the current transaction. The '*text*' is a quoted literal of up to 50 characters that Oracle stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in-doubt.

> **See Also:** COMMENT on page 8-131 for more information on adding comments to SQL statements

FORCE '*text*'

In a distributed database system, the FORCE clause lets you manually commit an in-doubt distributed transaction. The transaction is identified by the '*text*' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can use *integer* to specifically assign the transaction a system change number (SCN). If you omit *integer*, the transaction is committed using the current SCN.

> **Note:** A COMMIT statement with a FORCE clause commits only the specified transaction. Such a statement does not affect your current transaction.

Restriction: COMMIT statements using the FORCE clause are not supported in PL/SQL.

See Also: *Oracle8i Distributed Database Systems* for more information on these topics

Examples

Committing an Insert Example This statement inserts a row into the dept table and commits this change:

```
INSERT INTO dept VALUES (50, 'MARKETING', 'TAMPA');
COMMIT WORK;
```

COMMIT and COMMENT Example The following statement commits the current transaction and associates a comment with it:

```
COMMIT
    COMMENT 'In-doubt transaction Code 36, Call (415) 555-2637';
```

If a network or machine failure prevents this distributed transaction from committing properly, Oracle stores the comment in the data dictionary along with the transaction ID. The comment indicates the part of the application in which the failure occurred and provides information for contacting the administrator of the database where the transaction was committed.

Forcing an In-Doubt Transaction Example The following statement manually commits an in-doubt distributed transaction:

```
COMMIT FORCE '22.57.53';
```

constraint_clause

Purpose

Use the *constraint_clause* in a CREATE TABLE or ALTER TABLE statement to define an integrity constraint. An **integrity constraint** is a rule that restricts the values for one or more columns in a table or an index-organized table.

> **Note:** Oracle does not support constraints on columns or attributes whose type is an object, nested table, varray, REF, or LOB. The only exception is that NOT NULL constraints are supported for columns or attributes whose type is object, VARRAY, REF, or LOB.

Prerequisites

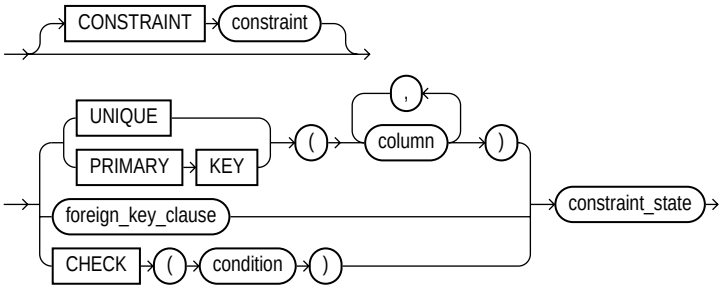
Constraint clauses can appear in either CREATE TABLE or ALTER TABLE statements. To define an integrity constraint, you must have the privileges necessary to issue one of these statements.

To create a referential integrity constraint, the parent table must be in your own schema, or you must have the REFERENCES privilege on the columns of the referenced key in the parent table.

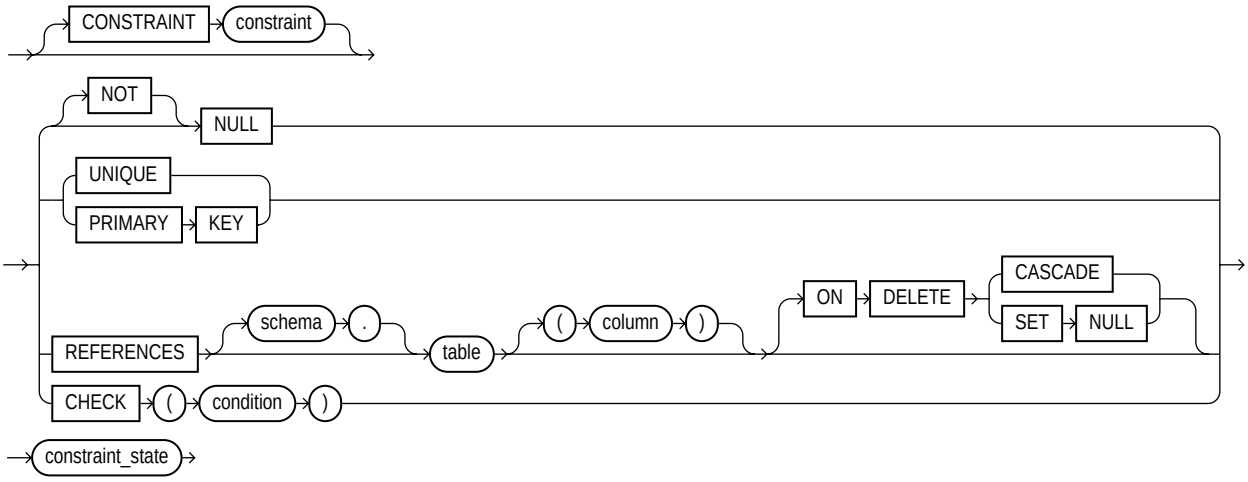
> **See Also:** CREATE TABLE on page 10-7 and ALTER TABLE on page 8-2

Syntax

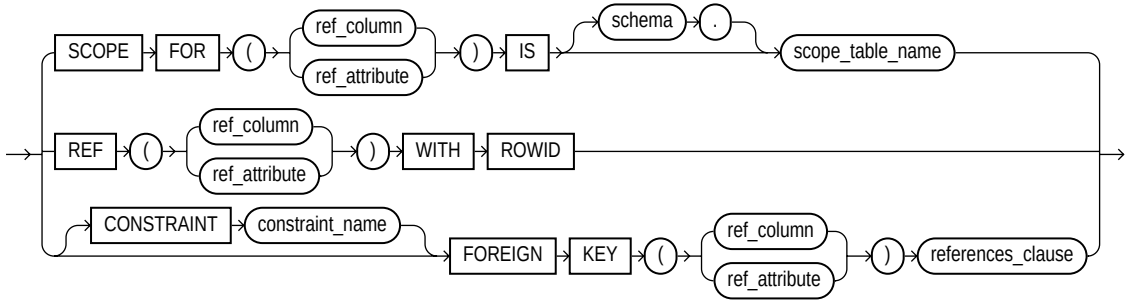
table_constraint::=



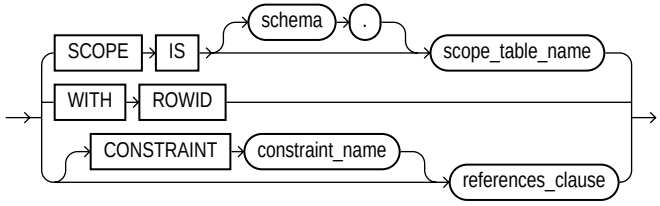
column_constraint::=



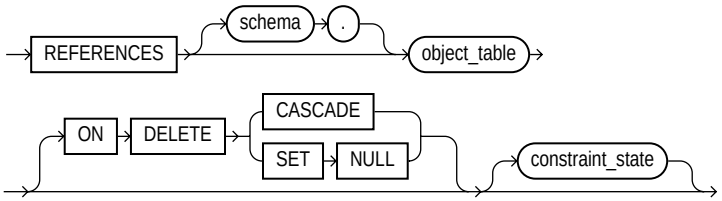
table_ref_constraint::=



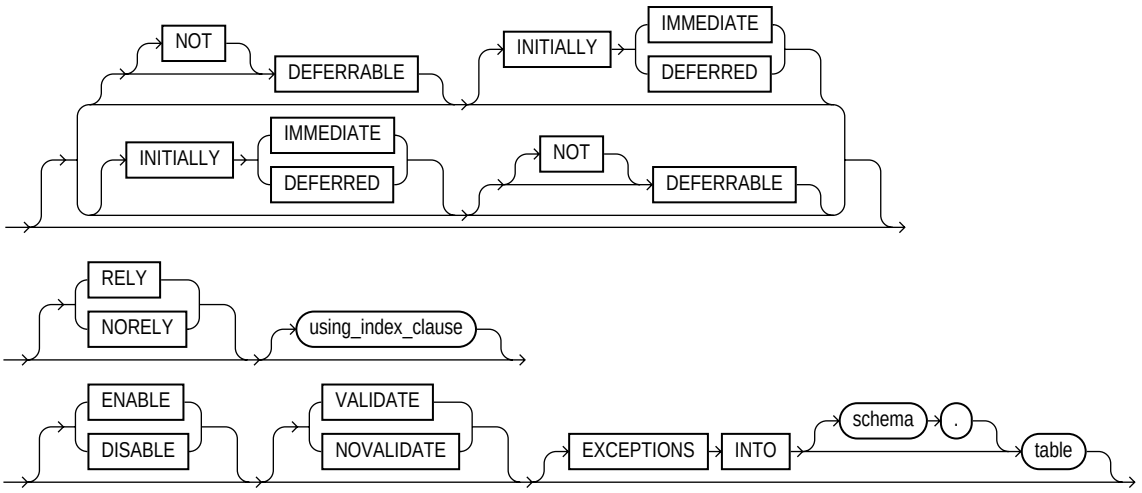
column_ref_constraint::=



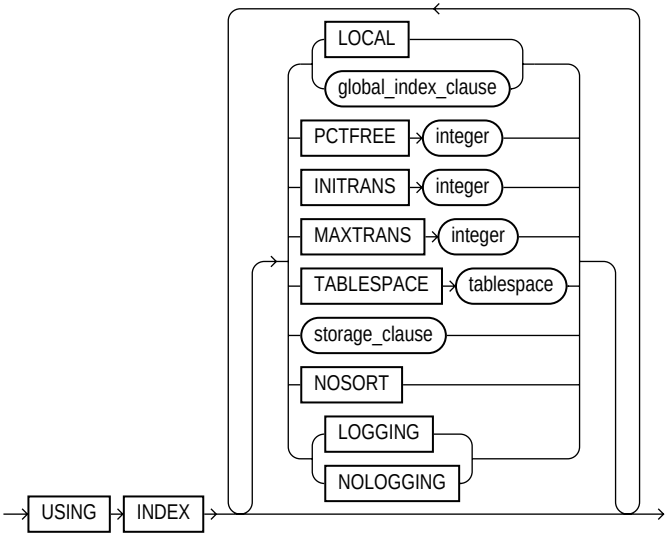
references_clause::=



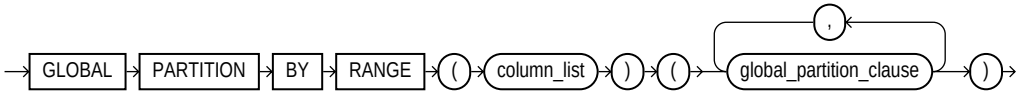
constraint_state::=



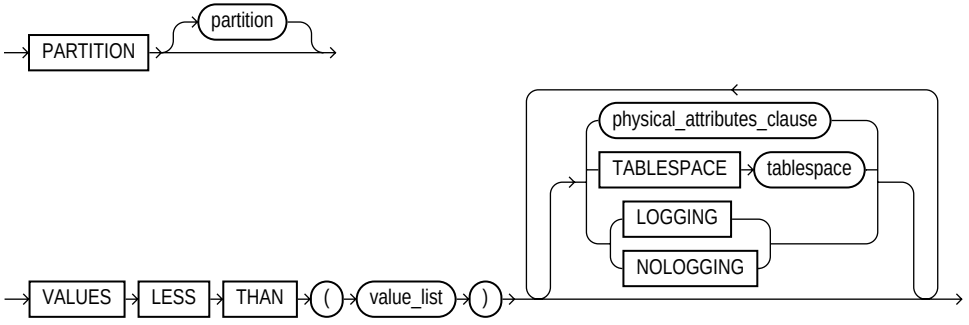
using_index_clause::=



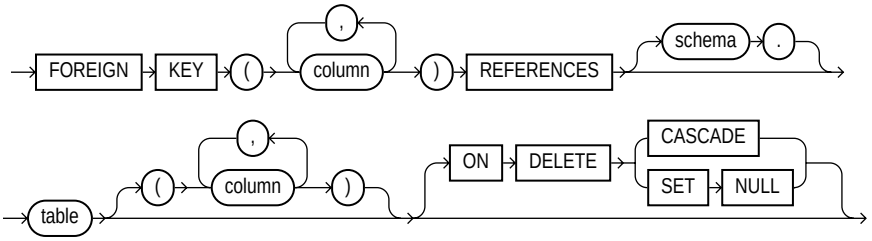
global_index_clause::=



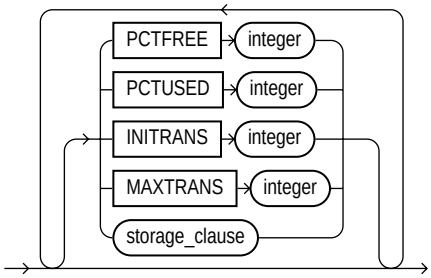
global_partition_clause::=



foreign_key_clause::=



physical_attributes_clause::=



storage_clause: See the storage_clause on page 11-129.

Keywords and Parameters

table_constraint

The *table_constraint* syntax is part of the table definition. An integrity constraint defined with this syntax can impose rules on any columns in the table.

The *table_constraint* syntax can appear in a CREATE TABLE or ALTER TABLE statement. This syntax can define any type of integrity constraint except a NOT NULL constraint.

column_constraint

The *column_constraint* syntax is part of a column definition. Usually, an integrity constraint defined with this syntax can impose rules only on the column in which it is defined.

- The *column_constraint* syntax that appears in a CREATE TABLE or ALTER TABLE ADD statement can define any type of integrity constraint.

- *Column_constraint* syntax that appears in an ALTER TABLE MODIFY *column_options* statement can only define or remove a NOT NULL constraint.

Restriction: The only column constraint allowed on a VARRAY column is NOT NULL. However, you can specify any type of column constraint on the scalar attributes of a NESTED TABLE column.

CONSTRAINT

Specify a name for the constraint. Oracle stores this name in the data dictionary along with the definition of the integrity constraint. If you omit this identifier, Oracle generates a name with the form SYS_C*n*.

If you do not specify NULL or NOT NULL in a column definition, NULL is the default.

Restriction: You cannot create a constraint on columns or attributes whose type is user-defined object, LOB, or REF, with the following exceptions:

- You can specify a NOT NULL constraint on columns or attributes of user-defined object type, varray, and LOB.
- You can specify NOT NULL **and** referential integrity constraints on a column of type REF.

UNIQUE

Specify UNIQUE to designate a column or combination of columns as a unique key. To satisfy a UNIQUE constraint, no two rows in the table can have the same value for the unique key. However, the unique key made up of a single column can contain nulls.

A **composite unique key** is made up of a combination of columns. To define a composite unique key, you must use *table_constraint* syntax rather than *column_constraint* syntax. Any row that contains nulls in all key columns automatically satisfies the constraint. However, two rows that contain nulls for one or more key columns and the same combination of values for the other key columns violate the constraint.

Restrictions:

- For a composite unique key, no two rows in the table can have the same combination of values in the key columns.
- A composite unique key cannot have more than 32 columns. The overall size of the key (in bytes) should not exceed approximately the width of all indexed columns plus the number of indexed columns.

- A unique key column cannot be of datatype LONG or LONG RAW.
- You cannot designate the same column or combination of columns as both a unique key and a primary key.

PRIMARY KEY

Specify PRIMARY KEY to designate a column or combination of columns as the table's primary key. A **composite primary key** is made up of a combination of columns. To define a composite primary key, you must use the *table_constraint* syntax rather than the *column_constraint* syntax.

Restrictions:

- A table can have only one primary key.
- None of the columns in the primary key can have datatype LONG, LONG RAW, VARRAY, NESTED TABLE, OBJECT, LOB, BFILE, or REF.
- No primary key value can appear in more than one row in the table.
- No column that is part of the primary key can contain a null.
- The size of the PRIMARY KEY of an index-organized table cannot exceed one-half of the database block size or 3800 bytes, whichever is less. (PRIMARY KEY is required for an index-organized table.)
- A composite primary key cannot have more than 32 columns. The overall size of the key (in bytes) should not exceed approximately the width of all indexed columns plus the number of indexed columns.
- You cannot designate the same column or combination of columns as both a primary key and a unique key.

NULL | NOT NULL

Indicate whether a column can contain nulls. You must specify NULL and NOT NULL with *column_constraint* syntax, not with *table_constraint* syntax.

NULL	Specify NULL if a column can contain null values. The NULL keyword does not actually define an integrity constraint. If you do not specify either NOT NULL or NULL, the column can contain nulls by default.
NOT NULL	Specify NOT NULL if a column cannot contain null values. To satisfy this constraint, every row in the table must contain a value for the column.

Restriction: You cannot specify NULL or NOT NULL for an attribute of an object. Instead, use a CHECK constraint with the IS [NOT] NULL condition.

See Also: "Attribute-Level Constraints Example" on page 8-154

Referential Integrity Constraints

Referential integrity constraints designate a column or combination of columns as the foreign key and establish a relationship between that foreign key and a specified primary or unique key, called the **referenced key**. The table containing the foreign key is called the **child table**, and the table containing the referenced key is called the **parent table**. The foreign key and the referenced key can be in the same table. In this case, the parent and child tables are the same.

- From the table level, specify referential integrity using the *foreign_key_clause* with the *table_constraint* syntax. This syntax allows you to specify a **composite foreign key**, which is made up of a combination of columns.
- From the column level, use the REFERENCES clause of the *column_constraint* syntax to specify a referential integrity constraint in which the foreign key is made up of a single column.

You can designate the same column or combination of columns as both a foreign key and a primary or unique key. You can also designate the same column or combination of columns as both a foreign key and a cluster key.

You can define multiple foreign keys in a table. Also, a single column can be part of more than one foreign key.

Restrictions on referential integrity constraints:

- A foreign key cannot be of type LONG or LONG RAW.
- The referenced UNIQUE or PRIMARY KEY constraint on the parent table must already be defined.
- The child and parent tables must be on the same database. To enable referential integrity constraints across nodes of a distributed database, you must use database triggers.
- You cannot define a referential integrity constraint in a CREATE TABLE statement that contains an AS *subquery* clause. Instead, you must create the table without the constraint and then add it later with an ALTER TABLE statement.

See Also: *Oracle8i Application Developer's Guide - Fundamentals*

foreign_key_clause

The *foreign_key_clause* lets you designate a column or combination of columns as the foreign key from the table level. You must use this syntax to define a composite foreign key.

To satisfy a referential integrity constraint involving composite keys, either the values of the foreign key columns must match the values of the referenced key columns in a row in the parent table, or the value of at least one of the columns of the foreign key must be null.

Restrictions:

- A composite foreign key cannot have more than 32 columns. The overall size of the key (in bytes) should not exceed approximately the width of all indexed columns plus the number of indexed columns.
- A composite foreign key must refer to a composite unique key or a composite primary key.

REFERENCES — The REFERENCES clause lets you designate the current column or attribute as the foreign key and identifies the parent table and the column or combination of columns that make up the referenced key. If you identify only the parent table and omit the column names, the foreign key automatically references the primary key of the parent table. The corresponding columns of the referenced key and the foreign key must match in number and datatypes.

ON DELETE — The ON DELETE clause lets you determine how Oracle automatically maintains referential integrity if you remove a referenced primary or unique key value. If you omit this clause, Oracle does not allow you to delete referenced key values in the parent table that have dependent rows in the child table.

- Specify CASCADE if you want Oracle to remove dependent foreign key values.
- Specify SET NULL if you want Oracle to convert dependent foreign key values to NULL.

CHECK Constraints

The CHECK clause lets you specify a condition that each row in the table must satisfy. To satisfy the constraint, each row in the table must make the condition

either TRUE or unknown (due to a null). When Oracle evaluates a CHECK constraint condition for a particular row, any column names in the condition refer to the column values in that row.

If you create multiple CHECK constraints for a column, design them carefully so their purposes do not conflict, and do not assume any particular order of evaluation of the conditions. Oracle does not verify that CHECK conditions are not mutually exclusive.

> **See Also:** "Conditions" on page 5-15 for additional information and syntax

Restrictions:

- The condition of a CHECK constraint can refer to any column in the table, but it cannot refer to columns of other tables.
- CHECK constraint conditions cannot contain the following constructs:
  - - Queries to refer to values in other rows
  - - Calls to the functions SYSDATE, UID, USER, or USERENV
  - - The pseudocolumns CURRVAL, NEXTVAL, LEVEL, or ROWNUM
  - - Date constants that are not fully specified

table_ref_constraint and *column_ref_constraint*

The *table_ref* and *column_ref* constraints let you further describe a column of type REF. The only difference between these clauses is that you specify *table_ref_constraint* from the table level, so you must identify the REF column or attribute you are defining. You specify *column_ref_constraint* after you have already identified the REF column or attribute. Both types of constraint let you specify a SCOPE constraint, a WITH ROWID constraint, or a referential integrity constraint.

As is the case for regular table and column constraints, you use FOREIGN KEY syntax for a referential integrity constraint at the table level, and REFERENCES syntax for a referential integrity constraint at the column level.

If the REF column's scope table or reference table has a primary-key-based object identifier, then it is a **user-defined REF column**.

> **See Also:** *Oracle8i Concepts* for more information on REFs and "Referential Integrity Constraints" on page 8-143

ref_column Specify the name of a REF column of an object or relational table.

ref_attribute Specify an embedded REF attribute within an object column of a relational table.

SCOPE In a table with a REF column, each REF value in the column can conceivably reference a row in a different object table. The SCOPE clause restricts the scope of references to a single table, *scope_table_name*. The values in the REF column or attribute point to objects in *scope_table_name*, in which object instances (of the same type as the REF column) are stored. You can only specify one scope table per REF column.

Restrictions:

- You can add a SCOPE constraint to an existing column only if the table is empty.
- You cannot specify SCOPE for the REF elements of a varray column.
- You must specify this clause if you specify AS *subquery* and the subquery returns user-defined REFs.
- The *scope_table_name* must be in your own schema or you must have SELECT privileges on *scope_table_name* or SELECT ANY TABLE system privileges.
- You cannot drop a SCOPE table constraint from a REF column.

WITH ROWID Specify WITH ROWID to store the rowid along with the REF value in *ref_column* or *ref_attribute*. Storing a REF value with a rowid can improve the performance of dereferencing operations, but will also use more space. Default storage of REF values is without rowids.

Restrictions:

- You cannot specify a WITH ROWID constraint for the REF elements of a varray column.
- You cannot drop a WITH ROWID constraint from a REF column.
- If the REF column or attribute is scoped, then this clause is ignored and the rowid is not stored with the REF value.

references_clause The *references_clause* lets you specify a referential integrity constraint on the REF column.This clause also implicitly restricts the scope of the REF column or attribute to the reference table.

If you do not specify CONSTRAINT, Oracle generates a system name for the constraint.

Restrictions:

- If you add a referential integrity constraint to an existing REF column that is scoped, then the referenced table must be the same as the scope table of the REF column.
- The system adds a scope constraint when you add a referential integrity constraint to an existing unscoped REF column. Therefore, all the restrictions that apply for the SCOPE constraint also apply in this case.
- If you later drop the referential integrity constraint, the REF column will remain scoped to the referenced table.

DEFERRABLE | NOT DEFERRABLE

Specify DEFERRABLE to indicate that constraint checking can be deferred until the end of the transaction by using the SET CONSTRAINT(S) statement.

See Also:

- SET CONSTRAINT[S] on page 11-120 for information on checking constraints after each DML statement
- *Oracle8i Administrator's Guide* and *Oracle8i Concepts* for more information about deferred constraints

Specify NOT DEFERRABLE to indicate that this constraint is checked at the end of each DML statement. If you do not specify either word, then NOT DEFERRABLE is the default.

INITIALLY IMMEDIATE Specify INITIALLY IMMEDIATE to indicate that at the start of every transaction, the default is to check this constraint at the end of every DML statement. If you do not specify INITIALLY, INITIALLY IMMEDIATE is the default.

INITIALLY DEFERRED	Specify INITIALLY DEFERRED to indicate that this constraint is DEFERRABLE and that, by default, the constraint is checked only at the end of each transaction.

Restrictions:

- You cannot defer a NOT DEFERRABLE constraint with the SET CONSTRAINT(S) statement.
- You cannot specify either DEFERRABLE or NOT DEFERRABLE if you are modifying an existing constraint directly (that is, by specifying the ALTER TABLE ... MODIFY *constraint* statement).
- You cannot alter a constraint's deferrability status. You must drop the constraint and re-create it.

RELY | NORELY

The RELY and NORELY parameters specify whether a constraint in NOVALIDATE mode is to be taken into account for query rewrite. Specify RELY to activate an existing constraint in NOVALIDATE mode for query rewrite in an unenforced query rewrite integrity mode. The constraint is in NOVALIDATE mode, so Oracle does not enforce it. The default is NORELY.

Unenforced constraints are generally useful only with materialized views and query rewrite. Depending on the QUERY_REWRITE_INTEGRITY mode (see ALTER SESSION on page 7-105), query rewrite can use only constraints that are in VALIDATE mode, or that are in NOVALIDATE mode with the RELY parameter set, to determine join information.

> **See Also:** *Oracle8i Data Warehousing Guide* for more information on materialized views and query rewrite

Restrictions:

- RELY and NORELY are relevant only if you are modifying an existing constraint (that is, you have issued the ALTER TABLE ... MODIFY constraint statement).
- You cannot set a NOT NULL constraint to RELY.

using_index_clause

The *using_index_clause* lets you specify parameters for the index Oracle uses to enable a UNIQUE or PRIMARY KEY constraint. The name of the index is the same as the name of the constraint.

You can choose the values of the INITRANS, MAXTRANS, TABLESPACE, STORAGE, and PCTFREE parameters for the index.

If table is partitioned, you can specify a locally or globally partitioned index for the unique or primary key constraint.

Restriction: Use this clause only when enabling UNIQUE and PRIMARY KEY constraints.

See Also:

- CREATE TABLE on page 10-7 for information on these parameters
- CREATE INDEX on page 9-52 for a description of LOCAL and *global_index_clause*, and for a description of NOSORT and LOGGING | NOLOGGING in relation to indexes

NOSORT

Specify NOSORT to indicate that the rows are stored in the database in ascending order and therefore Oracle does not have to sort the rows when creating the index.

ENABLE

Specify ENABLE if you want the constraint to be applied to all new data in the table. Before you can enable a referential integrity constraint, its referenced constraint must be enabled.

- ENABLE VALIDATE additionally indicates that all old data also complies with the constraint. An enabled validated constraint guarantees that all data is and will continue to be valid.

  If you place a primary key constraint in ENABLE VALIDATE mode, the validation process will verify that the primary key columns contain no nulls. To avoid this overhead, mark each column in the primary key NOT NULL before enabling the table's primary key constraint. (For optimal results, do this before inserting data into the column.)

- ENABLE NOVALIDATE ensures that all new DML operations on the constrained data comply with the constraint, but does not ensure that existing data in the table complies with the constraint.

  Enabling a primary key or unique key constraint automatically creates a unique index to enforce the constraint. This index is dropped if the constraint is subsequently disabled, causing Oracle to rebuild the index every time the

constraint is enabled. To avoid this behavior, create new primary key and unique key constraints initially disabled. Then create nonunique indexes or use existing nonunique indexes to enforce the constraints.

> **See Also:** the *enable_disable_clause* of CREATE TABLE on page 10-41 for additional notes and restrictions

DISABLE

Specify DISABLE to disable the integrity constraint. If you do not specify this clause when creating a constraint, Oracle automatically enables the constraint.

- DISABLE VALIDATE disables the constraint and drops the index on the constraint, but keeps the constraint valid. This feature is most useful in data warehousing situations, where the need arises to load into a range-partitioned table a quantity of data with a distinct range of values in the unique key. In such situations, the disable validate state enables you to save space by not having an index. You can then load data from a nonpartitioned table into a partitioned table using the *exchange_partition_clause* of the ALTER TABLE statement or using SQL*Loader. **All other modifications** to the table (inserts, updates, and deletes) by other SQL statements are disallowed.

  If the unique key coincides with the partitioning key of the partitioned table, disabling the constraint saves overhead and has no detrimental effects. If the unique key does not coincide with the partitioning key, Oracle performs automatic table scans during the exchange to validate the constraint, which might offset the benefit of loading without an index.

- DISABLE NOVALIDATE signifies that Oracle makes no effort to maintain the constraint (because it is disabled) and cannot guarantee that the constraint is true (because it is not being validated).

  You cannot drop a table whose primary key is being referenced by a foreign key even if the foreign key constraint is in DISABLE NOVALIDATE state. Further, the optimizer can use constraints in DISABLE NOVALIDATE state.

  > **See Also:** *Oracle8i Performance Guide and Reference* for information on when to use this setting

- If you specify neither VALIDATE nor NOVALIDATE, the default is NOVALIDATE.
- If you disable a unique or primary key constraint that is using a unique index, Oracle drops the unique index.

EXCEPTIONS INTO

The EXCEPTIONS INTO clause lets you specify a table into which Oracle places the rowids of all rows violating the constraint. If you omit schema, Oracle assumes the exceptions table is in your own schema. If you omit this clause altogether, Oracle assumes that the table is named EXCEPTIONS. The exceptions table must be on your local database.

The EXCEPTIONS INTO clause is valid only when validating a constraint.

You can create the EXCEPTIONS table using one of these scripts:

- UTLEXCPT.SQL uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.)
- UTLEXPT1.SQL uses universal rowids, so it can accommodate rows from both conventional and index-organized tables.

If you create your own exceptions table, it must follow the format prescribed by one of these two scripts.

Restriction: You cannot specify this clause in a CREATE TABLE statement, because no rowids exist until *after* the successful completion of the statement.

Note: If you are collecting exceptions from index-organized tables based on primary keys (rather than universal rowids), you must create a separate exceptions table for each index-organized table to accommodate its primary-key storage. You create multiple exceptions tables with different names by modifying and resubmitting the script.

See Also:

- *Oracle8i Migration* for compatibility issues related to the use of these scripts
- The DBMS_IOT package in *Oracle8i Supplied PL/SQL Packages Reference* for information on the SQL scripts
- *Oracle8i Performance Guide and Reference* for information on eliminating migrated and chained rows

Examples

Unique Key Example The following statement creates the dept table and defines and enables a unique key on the dname column:

```
CREATE TABLE dept
    (deptno  NUMBER(2),
     dname   VARCHAR2(9)  CONSTRAINT unq_dname UNIQUE,
      loc     VARCHAR2(10) );
```

The constraint unq_dname identifies the dname column as a unique key. This constraint ensures that no two departments in the table have the same name. However, the constraint does allow departments without names.

Alternatively, you can define and enable this constraint with the *table_constraint* syntax:

```
CREATE TABLE dept
    (deptno  NUMBER(2),
     dname   VARCHAR2(9),
     loc     VARCHAR2(10),
        CONSTRAINT unq_dname
        UNIQUE (dname)
    USING INDEX PCTFREE 20
        TABLESPACE user_x
        STORAGE (INITIAL 8K  NEXT 6K) );
```

The above statement also uses the USING INDEX clause to specify storage characteristics for the index that Oracle creates to enable the constraint.

Composite Unique Key Example The following statement defines and enables a composite unique key on the combination of the city and state columns of the census table:

```
ALTER TABLE census
    ADD CONSTRAINT unq_city_state
    UNIQUE (city, state)
    USING INDEX PCTFREE 5
        TABLESPACE user_y
    EXCEPTIONS INTO bad_keys_in_ship_cont;
```

The unq_city_state constraint ensures that the same combination of city and state values does not appear in the table more than once.

The ADD CONSTRAINT clause also specifies other properties of the constraint:

- The USING INDEX clause specifies storage characteristics for the index Oracle creates to enable the constraint.
- The EXCEPTIONS INTO clause causes Oracle to write information to the bad_keys_in_ship_cont table about any rows currently in the census table that violate the constraint.

Primary Key Example The following statement creates the dept table and defines and enables a primary key on the deptno column:

```
CREATE TABLE dept
    (deptno  NUMBER(2) CONSTRAINT pk_dept PRIMARY KEY,
     dname   VARCHAR2(9),
     loc     VARCHAR2(10) );
```

The pk_dept constraint identifies the deptno column as the primary key of the dept table. This constraint ensures that no two departments in the table have the same department number and that no department number is NULL.

Alternatively, you can define and enable this constraint with *table_constraint* syntax:

```
CREATE TABLE dept
    (deptno  NUMBER(2),
    dname   VARCHAR2(9),
    loc   VARCHAR2(10),
    CONSTRAINT pk_dept PRIMARY KEY (deptno) );
```

Composite Primary Key Example The following statement defines a composite primary key on the combination of the ship_nop and container_no columns of the ship_cont table:

```
ALTER TABLE ship_cont
    ADD PRIMARY KEY (ship_no, container_no) DISABLE;
```

This constraint identifies the combination of the ship_no and container_no columns as the primary key of the ship_cont table. The constraint ensures that no two rows in the table have the same values for both the ship_no column and the container_no column.

The CONSTRAINT clause also specifies the following properties of the constraint:

- The constraint definition does not include a constraint name, so Oracle generates a name for the constraint.
- The DISABLE clause causes Oracle to define the constraint but not enable it.

NOT NULL Example The following statement alters the emp table and defines and enables a NOT NULL constraint on the SAL column:

```
ALTER TABLE emp
   MODIFY (sal  NUMBER  CONSTRAINT nn_sal NOT NULL);
```

nn_sal ensures that no employee in the table has a null salary.

Attribute-Level Constraints Example The following example guarantees that a value exists for both the first_name and last_name attributes of the name column in the students table:

```
CREATE TYPE person_name AS OBJECT
   (first_name VARCHAR2(30), last_name VARCHAR2(30));

CREATE TABLE students (name person_name, age INTEGER,
   CHECK (name.first_name IS NOT NULL AND
          name.last_name IS NOT NULL));
```

Referential Integrity Constraint Example The following statement creates the emp table and defines and enables a foreign key on the deptno column that references the primary key on the deptno column of the dept table:

```
CREATE TABLE emp
   (empno      NUMBER(4),
    ename      VARCHAR2(10),
    job        VARCHAR2(9),
    mgr        NUMBER(4),
    hiredate   DATE,
    sal        NUMBER(7,2),
    comm       NUMBER(7,2),
    deptno     CONSTRAINT fk_deptno REFERENCES dept(deptno) );
```

The constraint fk_deptno ensures that all departments given for employees in the emp table are present in the dept table. However, employees can have null department numbers, meaning they are not assigned to any department. To ensure that all employees are assigned to a department, you could create a NOT NULL constraint on the deptno column in theemp table, in addition to the REFERENCES constraint.

Before you define and enable this constraint, you must define and enable a constraint that designates the deptno column of the dept table as a primary or unique key.

The referential integrity constraint definition does not use the FOREIGN KEY keyword to identify the columns that make up the foreign key. Because the constraint is defined with a column constraint clause on the deptno column, the foreign key is automatically on the deptno column.

The constraint definition identifies both the parent table and the columns of the referenced key. Because the referenced key is the parent table's primary key, the referenced key column names are optional.

The above statement omits the deptno column's datatype. Because this column is a foreign key, Oracle automatically assigns it the datatype of the dept.deptno column to which the foreign key refers.

Alternatively, you can define a referential integrity constraint with *table_constraint* syntax:

```
CREATE TABLE emp
  (empno     NUMBER(4),
   ename     VARCHAR2(10),
   job       VARCHAR2(9),
   mgr       NUMBER(4),
   hiredate  DATE,
   sal       NUMBER(7,2),
   comm      NUMBER(7,2),
   deptno,
   CONSTRAINT fk_deptno
      FOREIGN  KEY (deptno)
      REFERENCES  dept(deptno) );
```

The foreign key definitions in both statements of this statement omit the ON DELETE clause, causing Oracle to forbid the deletion of a department if any employee works in that department.

ON DELETE Example This statement creates the emp table, defines and enables two referential integrity constraints, and uses the ON DELETE clause:

```
CREATE TABLE emp
  (empno    NUMBER(4) PRIMARY KEY,
   ename    VARCHAR2(10),
   job      VARCHAR2(9),
   mgr      NUMBER(4) CONSTRAINT fk_mgr
            REFERENCES emp ON DELETE SET NULL,
   hiredate DATE,
   sal      NUMBER(7,2),
   comm     NUMBER(7,2),
```

```
   deptno   NUMBER(2)   CONSTRAINT fk_deptno
            REFERENCES dept(deptno)
            ON DELETE CASCADE );
```

Because of the first ON DELETE clause, if manager number 2332 is deleted from the emp table, Oracle sets to null the value of mgr for all employees in the emp table who previously had manager 2332.

Because of the second ON DELETE clause, Oracle cascades any deletion of a deptno value in the dept table to the deptno values of its dependent rows of the emp table. For example, if Department 20 is deleted from the dept table, Oracle deletes the department's employees from the emp table.

Composite Referential Integrity Constraint Example The following statement defines and enables a foreign key on the combination of the areaco and phoneno columns of the phone_calls table:

```
ALTER TABLE phone_calls
    ADD CONSTRAINT fk_areaco_phoneno
        FOREIGN KEY (areaco, phoneno)
        REFERENCES customers(areaco, phoneno)
        EXCEPTIONS INTO wrong_numbers;
```

The constraint fk_areaco_phoneno ensures that all the calls in the phone_calls table are made from phone numbers that are listed in the customers table. Before you define and enable this constraint, you must define and enable a constraint that designates the combination of the areaco and phoneno columns of the customers table as a primary or unique key.

The EXCEPTIONS INTO clause causes Oracle to write information to the wrong_numbers table about any rows in the phone_calls table that violate the constraint.

CHECK Constraint Examples The following statement creates the dept table and defines a check constraint in each of the table's columns:

```
CREATE TABLE dept
   (deptno NUMBER  CONSTRAINT check_deptno
          CHECK (deptno BETWEEN 10 AND 99)
          DISABLE,
    dname VARCHAR2(9)  CONSTRAINT check_dname
          CHECK (dname = UPPER(dname))
          DISABLE,
    loc VARCHAR2(10)  CONSTRAINT check_loc
          CHECK (loc IN ('DALLAS','BOSTON',
```

```
        'NEW YORK','CHICAGO'))
        DISABLE);
```

Each constraint restricts the values of the column in which it is defined:

- check_deptno ensures that no department numbers are less than 10 or greater than 99.
- check_dname ensures that all department names are in uppercase.
- check_loc restricts department locations to Dallas, Boston, New York, or Chicago.

Because each CONSTRAINT clause contains the DISABLE clause, Oracle only defines the constraints and does not enable them.

The following statement creates the emp table and uses a *table_constraint_clause* to define and enable a CHECK constraint:

```
CREATE TABLE emp
    (empno          NUMBER(4),
     ename          VARCHAR2(10),
     job            VARCHAR2(9),
     mgr            NUMBER(4),
     hiredate       DATE,
     sal            NUMBER(7,2),
     comm           NUMBER(7,2),
     deptno         NUMBER(2),
     CHECK (sal + comm <= 5000) );
```

This constraint uses an inequality condition to limit an employee's total compensation, the sum of salary and commission, to $5000:

- If an employee has non-null values for both salary and commission, the sum of these values must not exceed $5000 to satisfy the constraint.
- If an employee has a null salary or commission, the result of the condition is unknown and the employee automatically satisfies the constraint.

Because the CONSTRAINT clause in this example does not supply a constraint name, Oracle generates a name for the constraint.

The following statement defines and enables a PRIMARY KEY constraint, two referential integrity constraints, a NOT NULL constraint, and two CHECK constraints:

```
CREATE TABLE order_detail
  (CONSTRAINT pk_od PRIMARY KEY (order_id, part_no),
   order_id NUMBER
```

```
   CONSTRAINT fk_oid REFERENCES scott.order (order_id),
part_no            NUMBER
   CONSTRAINT fk_pno REFERENCES scott.part (part_no),
quantity            NUMBER
   CONSTRAINT nn_qty NOT NULL
   CONSTRAINT check_qty_low CHECK (quantity > 0),
cost             NUMBER
   CONSTRAINT check_cost CHECK (cost > 0) );
```

The constraints enable the following rules on table data:

- pk_od identifies the combination of the order_id and part_no columns as the primary key of the table. To satisfy this constraint, no two rows in the table can contain the same combination of values in the order_id and the part_no columns, and no row in the table can have a null in either the order_id column or the part_no column.
- fk_oid identifies the order_id column as a foreign key that references the order_id column in the order table in scott's schema. All new values added to the column order_detail.order_id must already appear in the column scott.order.order_id.
- fk_pno identifies the part_no column as a foreign key that references the part_no column in the part table owned by scott. All new values added to the column order_detail.part_no must already appear in the column scott.part.part_no.
- nn_qty forbids nulls in the quantity column.
- check_qty ensures that values in the quantity column are always greater than zero.
- check_cost ensures the values in the cost column are always greater than zero.

This example also illustrates the following points about constraint clauses and column definitions:

- *Table_constraint* syntax and column definitions can appear in any order. In this example, the *table_constraint* syntax that defines the pk_od constraint precedes the column definitions.
- A column definition can use *column_constraint* syntax multiple times. In this example, the definition of the quantity column contains the definitions of both the nn_qty and check_qty constraints.

- A table can have multiple CHECK constraints. Multiple CHECK constraints, each with a simple condition enforcing a single business rule, is better than a single CHECK constraint with a complicated condition enforcing multiple business rules. When a constraint is violated, Oracle returns an error identifying the constraint. Such an error more precisely identifies the violated business rule if the identified constraint enables a single business rule.

DEFERRABLE Constraint Examples The following statement creates table games with a NOT DEFERRABLE INITIALLY IMMEDIATE constraint check on the scores column:

```
CREATE TABLE games (scores NUMBER CHECK (scores >= 0));
```

To define a unique constraint on a column as INITIALLY DEFERRED DEFERRABLE, issue the following statement:

```
CREATE TABLE orders
  (ord_num NUMBER CONSTRAINT unq_num UNIQUE (ord_num)
   INITIALLY DEFERRED DEFERRABLE);
```

9

SQL Statements: CREATE CLUSTER to CREATE SEQUENCE

This chapter contains the following SQL statements:

- CREATE CLUSTER
- CREATE CONTEXT
- CREATE CONTROLFILE
- CREATE DATABASE
- CREATE DATABASE LINK
- CREATE DIMENSION
- CREATE DIRECTORY
- CREATE FUNCTION
- CREATE INDEX
- CREATE INDEXTYPE
- CREATE JAVA
- CREATE LIBRARY
- CREATE MATERIALIZED VIEW
- CREATE MATERIALIZED VIEW LOG
- CREATE OPERATOR
- CREATE OUTLINE
- CREATE PACKAGE

- CREATE PACKAGE BODY
- CREATE PROCEDURE
- CREATE PROFILE
- CREATE ROLE
- CREATE ROLLBACK SEGMENT
- CREATE SCHEMA
- CREATE SEQUENCE

CREATE CLUSTER

Purpose

Use the CREATE CLUSTER statement to create a cluster. A **cluster** is a schema object that contains data from one or more tables, all of which have one or more columns in common. Oracle stores together all the rows (from all the tables) that share the same cluster key.

For information on existing clusters, query the USER_CLUSTERS, ALL_CLUSTERS, and DBA_CLUSTERS data dictionary views.

See Also:

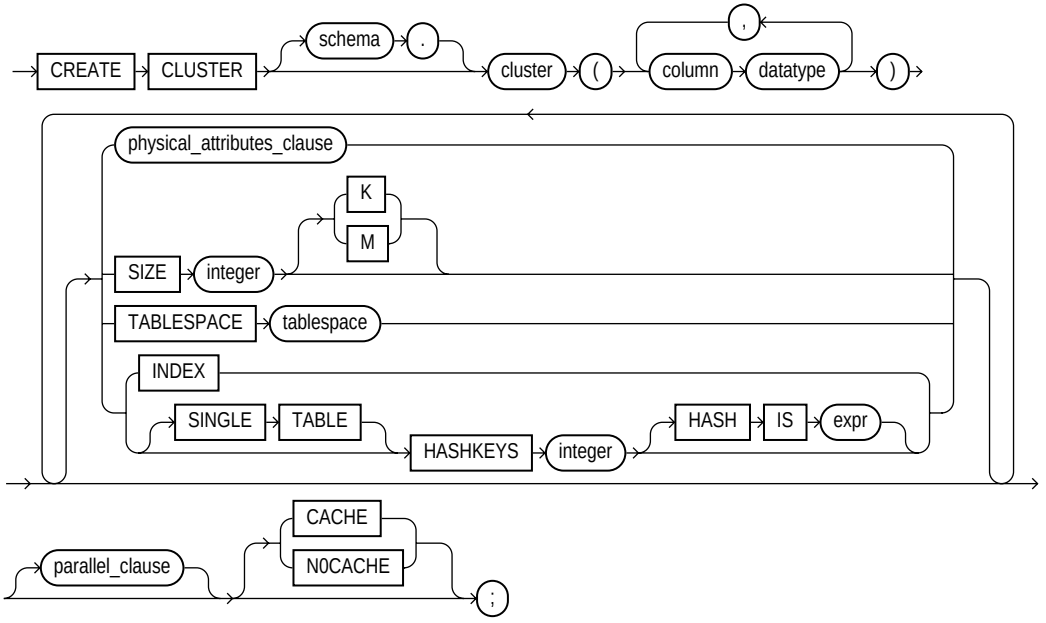
- *Oracle8i Concepts* for general information on clusters
- *Oracle8i Application Developer's Guide - Fundamentals* for information on performance considerations of clusters
- *Oracle8i Performance Guide and Reference* for suggestions on when to use clusters
- *Oracle8i Reference* for information on the data dictionary views

Prerequisites

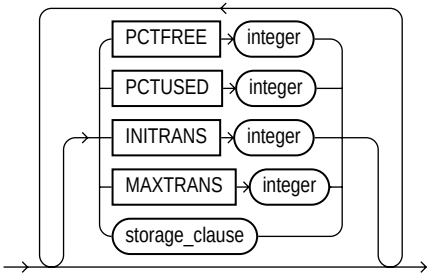
To create a cluster in your own schema, you must have CREATE CLUSTER system privilege. To create a cluster in another user's schema, you must have CREATE ANY CLUSTER system privilege. Also, the owner of the schema to contain the cluster must have either space quota on the tablespace containing the cluster or UNLIMITED TABLESPACE system privilege.

Oracle does not automatically create an index for a cluster when the cluster is initially created. Data manipulation language (DML) statements cannot be issued against clustered tables until a cluster index has been created.

Syntax

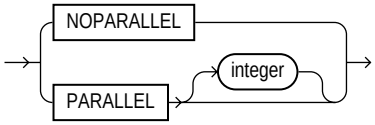


physical_attributes_clause::=



storage_clause: See the storage_clause on page 11-129.

parallel_clause::=



Keywords and Parameters

schema

Specify the schema to contain the cluster. If you omit *schema*, Oracle creates the cluster in your current schema.

cluster

Specify is the name of the cluster to be created.

After you create a cluster, you add tables to it. A cluster can contain a maximum of 32 tables. After you create a cluster and add tables to it, the cluster is transparent. You can access clustered tables with SQL statements just as you can nonclustered tables.

> **See Also:** CREATE TABLE on page 10-7 for information on adding tables to a cluster

column

Specify one or more names of columns in the cluster key. You can specify up to 16 cluster key columns. These columns must correspond in both datatype and size to columns in each of the clustered tables, although they need not correspond in name.

You cannot specify integrity constraints as part of the definition of a cluster key column. Instead, you can associate integrity constraints with the tables that belong to the cluster.

datatype

Specify the datatype of each cluster key column.

Restrictions:

- You cannot specify a cluster key column of datatype LONG, LONG RAW, REF, nested table, varray, BLOB, CLOB, BFILE, or user-defined object type.

- You cannot use the HASH IS clause if any column datatype is not INTEGER or NUMBER with scale 0.
- You can specify a column of type ROWID, but Oracle does not guarantee that the values in such columns are valid rowids.

See Also: "Datatypes" on page 2-2 for information on datatypes

physical_attributes_clause

The *physical_attributes_clause* lets you specify the storage characteristics of the cluster. Each table in the cluster uses these storage characteristics as well.

PCTUSED — Specify the limit that Oracle should use to determine when additional rows can be added to a cluster's data block. The value of this parameter is expressed as a whole number and interpreted as a percentage.

PCTFREE — Specify the space to be reserved in each of the cluster's data blocks for future expansion. The value of the parameter is expressed as a whole number and interpreted as a percentage.

INITRANS — Specify the initial number of concurrent update transactions allocated for data blocks of the cluster. The value of this parameter for a cluster cannot be less than 2 or more than the value of the MAXTRANS parameter. The default value is 2 or the INITRANS value for the cluster's tablespace, whichever is greater.

MAXTRANS — Specify the maximum number of concurrent update transactions for any given data block belonging to the cluster. The value of this parameter cannot be less than the value of the INITRANS parameter. The maximum value of this parameter is 255. The default value is the MAXTRANS value for the tablespace to contain the cluster.

See Also: CREATE TABLE on page 10-7 for a complete description of the PCTUSED, PCTFREE, INITRANS, and MAXTRANS parameters

storage_clause — The *storage_clause* lets you specify how data blocks are allocated to the cluster.

See Also: *storage_clause* on page 11-129

SIZE

Specify the amount of space in bytes to store all rows with the same cluster key value or the same hash value. Use K or M to specify this space in kilobytes or megabytes. This space determines the maximum number of cluster or hash values stored in a data block. If SIZE is not a divisor of the data block size, Oracle uses the next largest divisor. If SIZE is larger than the data block size, Oracle uses the operating system block size, reserving at least one data block per cluster or hash value.

Oracle also considers the length of the cluster key when determining how much space to reserve for the rows having a cluster key value. Larger cluster keys require larger sizes. To see the actual size, query the KEY_SIZE column of the USER_CLUSTERS data dictionary view. (This does not apply to hash clusters, because hash values are not actually stored in the cluster.)

If you omit this parameter, Oracle reserves one data block for each cluster key value or hash value.

TABLESPACE

Specify the tablespace in which the cluster is created.

INDEX | HASH

INDEX — Specify INDEX to create an **indexed cluster**. In an indexed cluster, Oracle stores together rows having the same cluster key value. Each distinct cluster key value is stored only once in each data block, regardless of the number of tables and rows in which it occurs.

After you create an indexed cluster, you must create an index on the cluster key before you can issue any data manipulation language (DML) statements against a table in the cluster. This index is called the **cluster index**.

Note: You cannot create a cluster index for a hash cluster, and you need not create an index on a hash cluster key. If you specify neither INDEX nor HASHKEYS, Oracle creates an indexed cluster by default.

See Also: CREATE INDEX on page 9-52 for information on creating a cluster index and *Oracle8i Concepts* for general information in indexed clusters

HASHKEYS

Specify the HASHKEYS clause to create a **hash cluster** and specifies the number of hash values for a hash cluster. In a hash cluster, Oracle stores together rows that have the same hash key value. The hash value for a row is the value returned by the cluster's hash function.

Oracle rounds up the HASHKEYS value to the nearest prime number to obtain the actual number of hash values. The minimum value for this parameter is 2. If you omit both the INDEX clause and the HASHKEYS parameter, Oracle creates an indexed cluster by default.

When you create a hash cluster, Oracle immediately allocates space for the cluster based on the values of the SIZE and HASHKEYS parameters.

See Also: *Oracle8i Concepts* for more information on how Oracle allocates space for clusters

SINGLE TABLE

SINGLE TABLE indicates that the cluster is a type of hash cluster containing only one table. This clause can provide faster access to rows than would result if the table were not part of a cluster.

Restriction: Only one table can be present in the cluster at a time. However, you can drop the table and create a different table in the same cluster.

HASH IS *expr*

Specify an expression to be used as the hash function for the hash cluster. The expression:

- Must evaluate to a positive value
- Must contain at least one column with referenced columns of any datatype as long as the entire expression evaluates to a number of scale 0. For example: NUM_COLUMN * length(VARCHAR2_COLUMN)
- Cannot reference user-defined PL/SQL functions
- Cannot reference SYSDATE, USERENV, TO_DATE, UID, USER, LEVEL, or ROWNUM
- Cannot evaluate to a constant
- Cannot contain a subquery
- Cannot contain columns qualified with a schema or object name (other than the cluster name)

If you omit the HASH IS clause, Oracle uses an internal hash function for the hash cluster.

For information on existing hash functions, query the USER_, ALL_, and DBA_CLUSTER_HASH_EXPRESSIONS data dictionary tables.

See Also: *Oracle8i Reference* for information on the data dictionary views

The cluster key of a hash column can have one or more columns of any datatype. Hash clusters with composite cluster keys or cluster keys made up of noninteger columns must use the internal hash function.

parallel_clause

The *parallel_clause* lets you parallelize the creation of the cluster.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL — Specify NOPARALLEL for serial execution. This is the default.

PARALLEL — Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Restriction: If the tables in *cluster* contain any columns of LOB or user-defined object type, this statement as well as subsequent INSERT, UPDATE, or DELETE operations on *cluster* are executed serially without notification.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 10-41

CACHE | NOCACHE

CACHE — Specify CACHE if you want the blocks retrieved for this table to be placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This clause is useful for small lookup tables.

NOCACHE — Specify NOCACHE if you want the blocks retrieved for this table to be placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the default behavior.

> **Note:** NOCACHE has no effect on clusters for which you specify KEEP in the *storage_clause*.

Examples

Creating a Cluster Example The following statement creates an indexed cluster named personnel with the cluster key column department_number, a cluster size of 512 bytes, and storage parameter values:

```
CREATE CLUSTER personnel
    ( department_number  NUMBER(2) )
    SIZE 512
    STORAGE (INITIAL 100K NEXT 50K);
```

Adding Tables to a Cluster Example The following statements add the `emp` and `dept` tables to the cluster:

```
CREATE TABLE emp
    (empno     NUMBER        PRIMARY KEY,
     ename     VARCHAR2(10)  NOT NULL
                             CHECK (ename = UPPER(ename)),
     job       VARCHAR2(9),
     mgr       NUMBER        REFERENCES scott.emp(empno),
     hiredate  DATE
        CHECK (hiredate < TO_DATE ('08-14-1998', 'MM-DD-YYYY')),
     sal       NUMBER(10,2)  CHECK (sal > 500),
     comm      NUMBER(9,0)   DEFAULT NULL,
     deptno    NUMBER(2)      NOT NULL )
     CLUSTER personnel (deptno);

CREATE TABLE dept
    (deptno  NUMBER(2),
     dname   VARCHAR2(9),
     loc     VARCHAR2(9))
     CLUSTER personnel (deptno);
```

Cluster Key Example The following statement creates the cluster index on the cluster key of `personnel`:

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

After creating the cluster index, you can insert rows into either the `emp` or `dept` tables.

Hash Cluster Examples The following statement creates a hash cluster named `personnel` with the cluster key column `department_number`, a maximum of 503 hash key values, each of which is allocated 512 bytes, and storage parameter values:

```
CREATE CLUSTER personnel
( department_number  NUMBER )
    SIZE 512  HASHKEYS 500
    STORAGE (INITIAL 100K  NEXT 50K);
```

Because the above statement omits the HASH IS clause, Oracle uses the internal hash function for the cluster.

The following statement creates a hash cluster named personnel with the cluster key made up of the columns home_area_code and home_prefix, and uses a SQL expression containing these columns for the hash function:

```
CREATE CLUSTER personnel
    ( home_area_code  NUMBER,
      home_prefix     NUMBER )
    HASHKEYS 20
    HASH IS MOD(home_area_code + home_prefix, 101);
```

Single-Table Hash Cluster Example The following statement creates a single-table hash cluster named personnel with the cluster key deptno and a maximum of 503 hash key values, each of which is allocated 512 bytes:

```
CREATE CLUSTER personnel
   (deptno NUMBER)
   SIZE 512 SINGLE TABLE HASHKEYS 500;
```

CREATE CONTEXT

Purpose

Use the CREATE CONTEXT statement to create a namespace for a **context** (a set of application-defined attributes that validates and secures an application) and to associate the namespace with the externally created package that sets the context. You can use the DBMS_SESSION.set_context procedure in your designated package to set or reset the attributes of the context.

See Also:

- *Oracle8i Concepts* for a definition and discussion of contexts
- *Oracle8i Supplied PL/SQL Packages Reference* for information on the DBMS_SESSION.set_context procedure

Prerequisites

To create a context namespace, you must have CREATE ANY CONTEXT system privilege.

Syntax

Keywords and Parameters

OR REPLACE

Specify OR REPLACE to redefine an existing context namespace using a different package.

namespace

Specify the name of the context namespace to create or modify. Context namespaces are always stored in the schema SYS.

schema

Specify the schema owning *package*. If you omit *schema*, Oracle uses the current schema.

package

Specify the PL/SQL package that sets or resets the context attributes under the namespace for a user session.

Note: To provide some design flexibility, Oracle does not verify the existence of the schema or the validity of the package at the time you create the context.

See Also: *Oracle8i Supplied PL/SQL Packages Reference* for more information on setting the package

Examples

CREATE CONTEXT Example Suppose you have a human resources (hr) application and a PL/SQL package (hr_secure_context), which validates and secures the hr application. The following statement creates the context namespace hr_context and associates it with the package hr_secure_context:

```
CREATE CONTEXT hr_context USING hr_secure_context;
```

You can control data access based on this context using the SYS_CONTEXT function. For example, suppose your hr_secure_context package has defined an attribute org_id as a particular organization identifier. You can secure a base table hr_org_unit by creating a view that restricts access based on the value of org_id, as follows:

```
CREATE VIEW hr_org_secure_view AS
     SELECT * FROM hr_org_unit
     WHERE organization_id = SYS_CONTEXT('hr_context', 'org_id');
```

See Also: SYS_CONTEXT on page 4-101 for more information on the SYS_CONTEXT function

CREATE CONTROLFILE

> **Caution:** Oracle recommends that you perform a full backup of all files in the database before using this statement. For more information, see *Oracle8i Backup and Recovery Guide*.

Purpose

Use the CREATE CONTROLFILE statement to re-create a control file in one of the following cases:

- All copies of your existing control files have been lost through media failure.
- You want to change the name of the database.
- You want to change the maximum number of redo log file groups, redo log file members, archived redo log files, datafiles, or instances that can concurrently have the database mounted and open.

When you issue a CREATE CONTROLFILE statement, Oracle creates a new control file based on the information you specify in the statement. If you omit any clauses, Oracle uses the default values rather than the values for the previous control file. After successfully creating the control file, Oracle mounts the database in the mode specified by the initialization parameter PARALLEL_SERVER. You then must perform media recovery before opening the database. It is recommended that you then shut down the instance and take a full backup of all files in the database.

> **See Also:** *Oracle8i Backup and Recovery Guide*

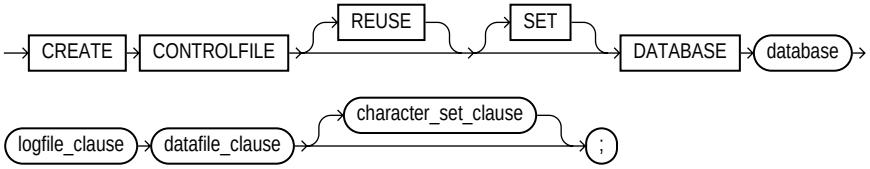
Prerequisites

You must have the OSDBA role enabled. The database must not be mounted by any instance.

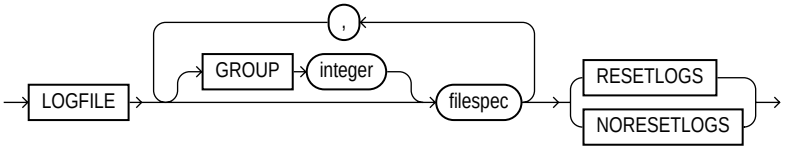
If the REMOTE_LOGIN_PASSWORDFILE initialization parameter is set to exclusive, Oracle returns an error when you attempt to re-create the control file. To avoid this message, either set the parameter to shared, or re-create your password file before re-creating the control file.

> **See Also:** *Oracle8i Reference* for more information about the REMOTE_LOGIN_PASSWORDFILE parameter

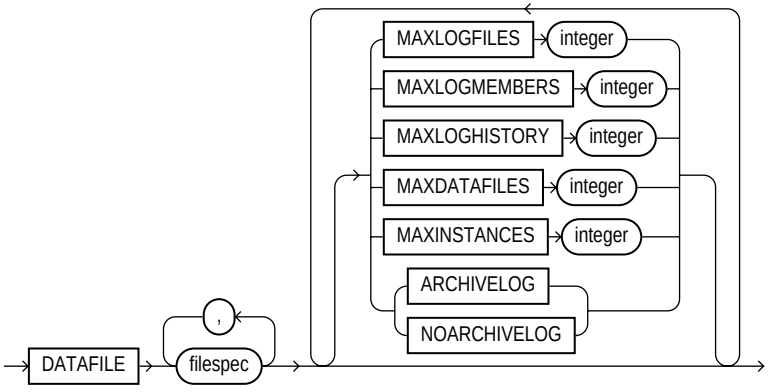
Syntax



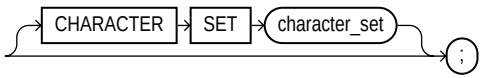
logfile_clause::=



datafile_clause::=



character_set_clause::=



filespec: See filespec on page 11-27.

Keywords and Parameters

REUSE

Specify REUSE to indicate that existing control files identified by the initialization parameter CONTROL_FILES can be reused, thus ignoring and overwriting any information they may currently contain. If you omit this clause and any of these control files already exists, Oracle returns an error.

DATABASE *database*

Specify the name of the database. The value of this parameter must be the existing database name established by the previous CREATE DATABASE statement or CREATE CONTROLFILE statement.

SET DATABASE *database*

Use SET DATABASE to change the name of the database. The name of a database can be as long as eight bytes.

logfile_clause

LOGFILE *filespec*
: Specify the redo log files for your database. You must list all members of all redo log file groups.

  See Also: *filespec* on page 11-27 for the syntax of *filespec*

  GROUP *integer*
  : Specify the logfile group number. If you specify GROUP values, Oracle verifies these values with the GROUP values when the database was last open.

RESETLOGS
: Specify RESETLOGS if you want Oracle to ignore the contents of the files listed in the LOGFILE clause. These files do not have to exist. Each *filespec* in the LOGFILE clause must specify the SIZE parameter. Oracle assigns all online redo log file groups to thread 1 and enables this thread for public use by any instance. After using this clause, you must open the database using the RESETLOGS clause of the ALTER DATABASE statement.

NORESETLOGS — Specify NORESETLOGS if you want Oracle to use all files in the LOGFILE clause as they were when the database was last open. These files must exist and must be the current online redo log files rather than restored backups. Oracle reassigns the redo log file groups to the threads to which they were previously assigned and reenables the threads as they were previously enabled.

datafile_clause

DATAFILE *filespec* — Specify the datafiles of the database. You must list all datafiles. These files must all exist, although they may be restored backups that require media recovery. See the syntax description of *filespec* in *filespec* on page 11-27.

MAXLOGFILES *integer* — Specify the maximum number of online redo log file groups that can ever be created for the database. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The default and maximum values depend on your operating system. The value that you specify should not be less than the greatest GROUP value for any redo log file group.

MAX-LOGMEMBERS *integer* — Specify the maximum number of members, or identical copies, for a redo log file group. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.

MAX-LOGHISTORY *integer* — Specify the maximum number of archived redo log file groups for automatic media recovery of the Oracle Parallel Server. Oracle uses this value to determine how much space in the control file to allocate for the names of archived redo log files. The minimum value is 0. The default value is a multiple of the MAXINSTANCES value and depends on your operating system. The maximum value is limited only by the maximum size of the control file. This parameter is useful only if you are using Oracle with the Parallel Server option in both parallel mode and archivelog mode.

MAX-DATAFILES *integer*	Specify the initial sizing of the datafiles section of the control file at CREATE DATABASE or CREATE CONTROLFILE time. An attempt to add a file whose number is greater than MAXDATAFILES, but less than or equal to DB_FILES, causes the Oracle control file to expand automatically so that the datafiles section can accommodate more files. The number of datafiles accessible to your instance is also limited by the initialization parameter DB_FILES.
MAX-INSTANCES *integer*	Specify the maximum number of instances that can simultaneously have the database mounted and open. This value takes precedence over the value of the initialization parameter INSTANCES. The minimum value is 1. The maximum and default values depend on your operating system.
ARCHIVELOG	Specify ARCHIVELOG to archive the contents of redo log files before reusing them. This clause prepares for the possibility of media recovery as well as instance or crash recovery.
NOARCHIVELOG	If you omit both the ARCHIVELOG clause and NOARCHIVELOG clause, Oracle chooses NOARCHIVELOG mode by default. After creating the control file, you can change between ARCHIVELOG mode and NOARCHIVELOG mode with the ALTER DATABASE statement.

character_set_clause

If you specify a character set, Oracle reconstructs character set information in the control file. In case media recovery of the database is required, this information will be available before the database is open, so that tablespace names can be correctly interpreted during recovery. This clause is useful only if you are using a character set other than the default US7ASCII.

If you are re-creating your control file and you are using Recovery Manager for tablespace recovery, and if you specify a different character set from the one stored in the data dictionary, then tablespace recovery will not succeed. (However, at database open, the control file character set will be updated with the correct character set from the data dictionary.)

Note: You cannot modify the character set of the database with this clause.

See Also: *Oracle8i Recovery Manager User's Guide and Reference* for more information on tablespace recovery

Example

CREATE CONTROLFILE Example This statement re-creates a control file. In this statement, database `orders_2` was created with the F7DEC character set.

```
CREATE CONTROLFILE REUSE
   DATABASE orders_2
   LOGFILE GROUP 1 ('diskb:log1.log', 'diskc:log1.log') SIZE 50K,
           GROUP 2 ('diskb:log2.log', 'diskc:log2.log') SIZE 50K
   NORESETLOGS
   DATAFILE 'diska:dbone.dat' SIZE 2M
                              MAXLOGFILES 5
                              MAXLOGHISTORY 100
                              MAXDATAFILES 10
                              MAXINSTANCES 2
                              ARCHIVELOG
   CHARACTER SET F7DEC;
```

CREATE DATABASE

Caution: This statement prepares a database for initial use and erases any data currently in the specified files. Use this statement only when you understand its ramifications.

Purpose

Use the CREATE DATABASE statement to create a database, making it available for general use.

This statement erases all data in any specified datafiles that already exist in order to prepare them for initial database use. If you use the statement on an existing database, all data in the datafiles is lost.

After creating the database, this statement mounts it in either exclusive or parallel mode (depending on the value of the PARALLEL_SERVER initialization parameter) and opens it, making it available for normal use. You can then create tablespaces and rollback segments for the database.

See Also:

- ALTER DATABASE on page 7-9 for information on modifying a database
- *Oracle8i Java Developer's Guide* for information on creating an Oracle8i Java virtual machine
- CREATE ROLLBACK SEGMENT on page 9-149 and CREATE TABLESPACE on page 10-56 for information on creating rollback segments and tablespaces

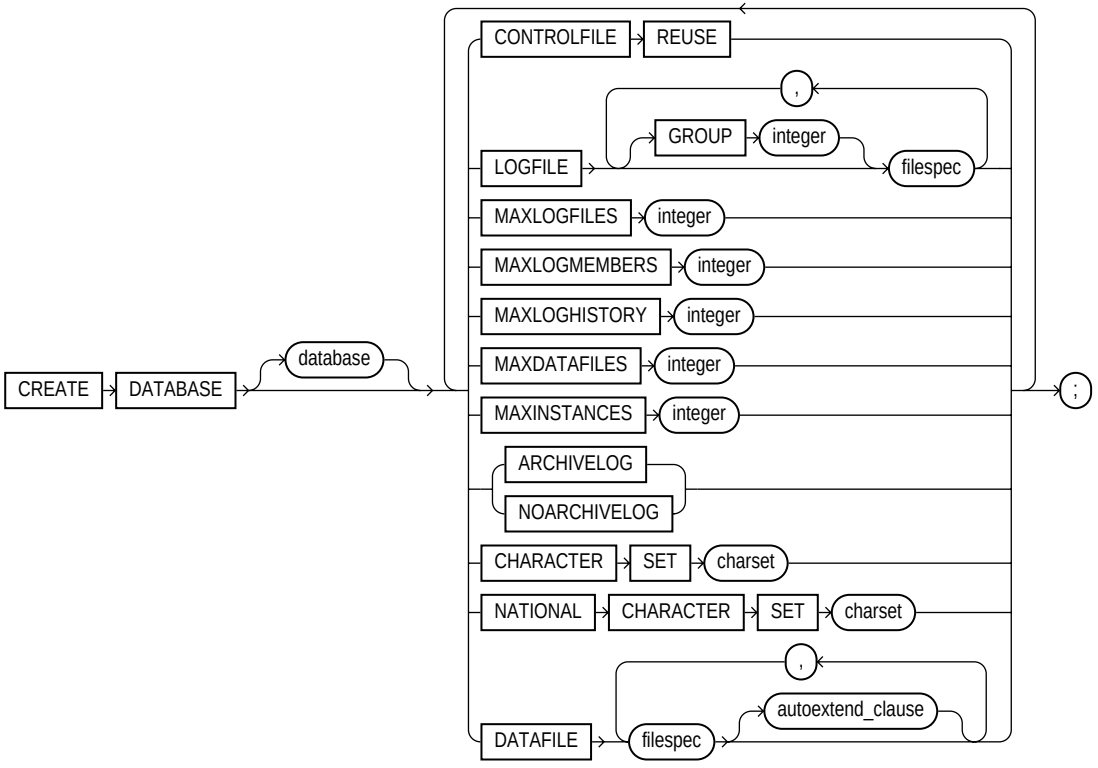
Prerequisites

You must have the OSDBA role enabled.

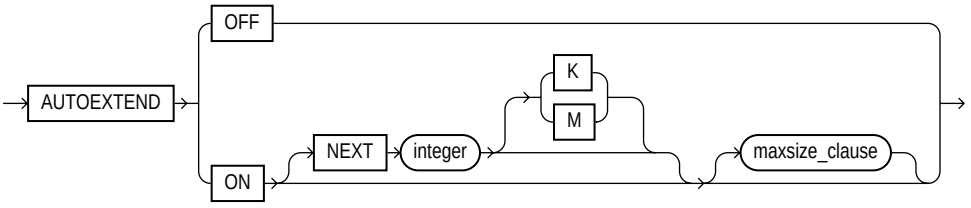
If the REMOTE_LOGIN_PASSWORDFILE initialization parameter is set to exclusive, Oracle returns an error when you attempt to re-create the database. To avoid this message, either set the parameter to shared, or re-create your password file before re-creating the database.

See Also: *Oracle8i Reference* for more information about the `REMOTE_LOGIN_PASSWORDFILE` parameter

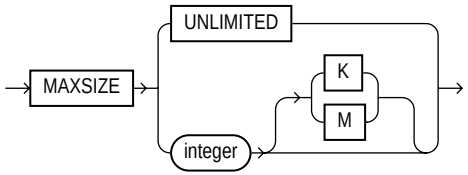
Syntax



`autoextend_clause`::=



maxsize_clause::=



filespec: See filespec on page 11-27.

Keyword and Parameters

database

Specify the name of the database to be created and can be up to 8 bytes long. The database name can contain only ASCII characters. Oracle writes this name into the control file. If you subsequently issue an ALTER DATABASE statement that explicitly specifies a database name, Oracle verifies that name with the name in the control file.

> **Note:** You cannot use special characters from European or Asian character sets in a database name. For example, characters with umlauts are not allowed.

If you omit the database name from a CREATE DATABASE statement, Oracle uses the name specified by the initialization parameter DB_NAME. If the DB_NAME initialization parameter has been set, and you specify a different name from the value of that parameter, Oracle returns an error.

> **See Also:** "Schema Object Naming Guidelines" on page 2-87 for additional rules to which database names should adhere

CONTROLFILE REUSE

Specify CONTROLFILE REUSE to reuse existing control files identified by the initialization parameter CONTROL_FILES, thus ignoring and overwriting any information they currently contain. Normally you use this clause only when you are re-creating a database, rather than creating one for the first time. You cannot use this clause if you also specify a parameter value that requires that the control file be

larger than the existing files. These parameters are MAXLOGFILES, MAXLOGMEMBERS, MAXLOGHISTORY, MAXDATAFILES, and MAXINSTANCES.

If you omit this clause and any of the files specified by CONTROL_FILES already exist, Oracle returns an error.

LOGFILE *filespec*

Specify one or more files to be used as redo log files. Each *filespec* specifies a redo log file group containing one or more redo log file members (copies). All redo log files specified in a CREATE DATABASE statement are added to redo log thread number 1.

See Also: *filespec* on page 11-27 for the syntax of *filespec*

GROUP *integer* Specify the number that identifies the redo log file group. The value of *integer* can range from 1 to the value of the MAXLOGFILES parameter. A database must have at least two redo log file groups. You cannot specify multiple redo log file groups having the same GROUP value. If you omit this parameter, Oracle generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance table V$LOG.

If you omit the LOGFILE clause, Oracle creates two redo log file groups by default. The names and sizes of the default files depend on your operating system.

MAXLOGFILES *integer*

Specify the maximum number of redo log file groups that can ever be created for the database. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The default, minimum, and maximum values depend on your operating system.

MAXLOGMEMBERS *integer*

Specify the maximum number of members, or copies, for a redo log file group. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.

MAXLOGHISTORY *integer*

Specify the maximum number of archived redo log files for automatic media recovery with Oracle Parallel Server. Oracle uses this value to determine how much space in the control file to allocate for the names of archived redo log files. The minimum value is 0. The default value is a multiple of the MAXINSTANCES value and depends on your operating system. The maximum value is limited only by the maximum size of the control file.

Note: This parameter is useful only if you are using Oracle with the Parallel Server option in parallel mode, and archivelog mode enabled.

MAXDATAFILES *integer*

Specify the initial sizing of the datafiles section of the control file at CREATE DATABASE or CREATE CONTROLFILE time. An attempt to add a file whose number is greater than MAXDATAFILES, but less than or equal to DB_FILES, causes the Oracle control file to expand automatically so that the datafiles section can accommodate more files.

The number of datafiles accessible to your instance is also limited by the initialization parameter DB_FILES.

MAXINSTANCES *integer*

Specify the maximum number of instances that can simultaneously have this database mounted and open. This value takes precedence over the value of initialization parameter INSTANCES. The minimum value is 1. The maximum and default values depend on your operating system.

ARCHIVELOG | NOARCHIVELOG

ARCHIVELOG — Specify ARCHIVELOG if you want the contents of a redo log file group to be archived before the group can be reused. This clause prepares for the possibility of media recovery.

NOARCHIVELOG — Specify NOARCHIVELOG if the contents of a redo log file group need not be archived before the group can be reused. This clause does not allow for the possibility of media recovery.

The default is NOARCHIVELOG mode. After creating the database, you can change between ARCHIVELOG mode and NOARCHIVELOG mode with the ALTER DATABASE statement.

CHARACTER SET *character_set*

Specify the character set the database uses to store data. The supported character sets and default value of this parameter depend on your operating system.

Restriction: You cannot specify any fixed-width multibyte character sets as the database character set.

> **See Also:** *Oracle8i National Language Support Guide* for more information about character sets

NATIONAL CHARACTER SET *character_set*

Specify the national character set used to store data in columns specifically defined as NCHAR, NCLOB, or NVARCHAR2. If not specified, the national character set defaults to the database character set.

> **See Also:** *Oracle8i National Language Support Guide* for valid character set names

DATAFILE *filespec*

Specify one or more files to be used as datafiles. All these files become part of the SYSTEM tablespace. If you omit this clause, Oracle creates one datafile by default. The name and size of this default file depend on your operating system.

Note: Oracle recommends that the total initial space allocated for the SYSTEM tablespace be a minimum of 5 megabytes.

> **See Also:** *filespec* on page 11-27 for syntax

autoextend_clause

The *autoextend_clause* lets you enable or disable the automatic extension of a datafile. If you do not specify this clause, datafiles are not automatically extended.

OFF	Specify OFF to disable autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in ALTER DATABASE AUTOEXTEND or ALTER TABLESPACE AUTOEXTEND statements.
ON	Specify ON to enable autoextend.
NEXT *integer*	Specify the size in bytes of the next increment of disk space to be allocated to the datafile automatically when more extents are required. Use K or M to specify this size in kilobytes or megabytes. The default is the size of one data block.
MAXSIZE	Specify the maximum disk space allowed for automatic extension of the datafile: ■ *integer* indicates the maximum disk space in bytes. Use K or M to specify this size in kilobytes or megabytes. ■ UNLIMITED indicates that there is no limit on the allocation of disk space to the datafile.

Examples

CREATE DATABASE Example The following statement creates a small database using defaults for all arguments:

```
CREATE DATABASE;
```

The following statement creates a database and fully specifies each argument:

```
CREATE DATABASE newtest
   CONTROLFILE REUSE
   LOGFILE
      GROUP 1 ('diskb:log1.log', 'diskc:log1.log') SIZE 50K,
      GROUP 2 ('diskb:log2.log', 'diskc:log2.log') SIZE 50K
   MAXLOGFILES 5
   MAXLOGHISTORY 100
   DATAFILE 'diska:dbone.dat' SIZE 2M
   MAXDATAFILES 10
   MAXINSTANCES 2
   ARCHIVELOG
   CHARACTER SET US7ASCII
   NATIONAL CHARACTER SET JA16SJISFIXED
   DATAFILE
   'disk1:df1.dbf' AUTOEXTEND ON
   'disk2:df2.dbf' AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED;
```

CREATE DATABASE LINK

Purpose

Use the CREATE DATABASE LINK statement to create a database link. A **database link** is a schema object in the local database that enables you to access objects on a remote database. The remote database need not be an Oracle system.

Once you have created a database link, you can use it to refer to tables and views on the remote database. You can refer to a remote table or view in a SQL statement by appending *@dblink* to the table or view name. You can query a remote table or view with the SELECT statement. If you are using Oracle with the distributed option, you can also access remote tables and views using any INSERT, UPDATE, DELETE, or LOCK TABLE statement.

See Also:

- *Oracle8i Application Developer's Guide - Fundamentals* for information about accessing remote tables or views with PL/SQL functions, procedures, packages, and datatypes
- *Oracle8i Distributed Database Systems* for information on distributed database systems
- *Oracle8i Reference* for descriptions of existing database links in the ALL_DB_LINKS, DBA_DB_LINKS, and USER_DB_LINKS data dictionary views and to monitor the performance of existing links through the V$DBLINK dynamic performance view
- DROP DATABASE LINK on page 10-129 for information on dropping existing database links
- INSERT on page 11-51, UPDATE on page 11-141, DELETE on page 10-115, and LOCK TABLE on page 11-62 for using links in DML operations

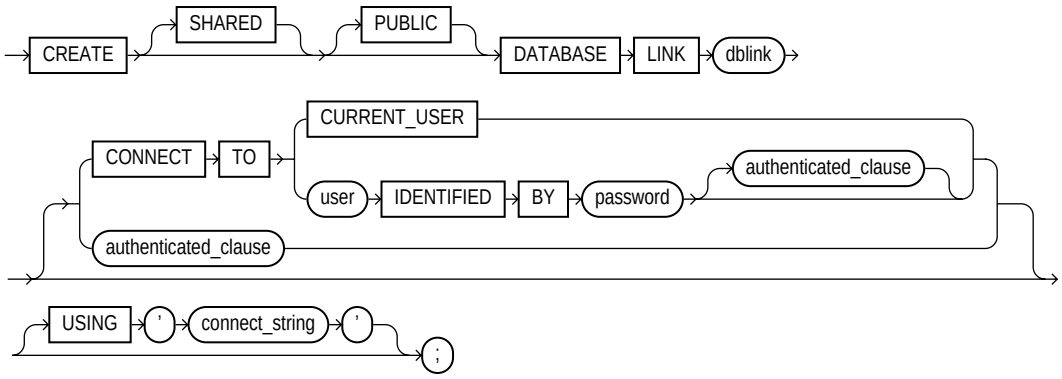
Prerequisites

To create a private database link, you must have CREATE DATABASE LINK system privilege. To create a public database link, you must have CREATE PUBLIC DATABASE LINK system privilege. Also, you must have CREATE SESSION privilege on the remote Oracle database.

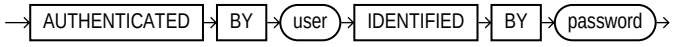
Net8 must be installed on both the local and remote Oracle databases.

To access non-Oracle systems you must use the Oracle Heterogeneous Services.

Syntax



authenticated_clause::=



Keyword and Parameters

SHARED

Specify SHARED to use a single network connection to create a public database link that can be shared between multiple users. This clause is available only with the multi-threaded server configuration.

> **See Also:** *Oracle8i Distributed Database Systems* for more information about shared database links

PUBLIC

Specify PUBLIC to create a public database link available to all users. If you omit this clause, the database link is private and is available only to you.

> **See Also:** The "PUBLIC Database Link Example" on page 9-32

dblink

Specify the complete or partial name of the database link. The value of the GLOBAL_NAMES initialization parameter determines whether the database link must have the same name as the database to which it connects.

The maximum number of database links that can be open in one session or one instance of an Oracle Parallel Server configuration depends on the value of the OPEN_LINKS and OPEN_LINKS_PER_INSTANCE initialization parameters.

Restriction: You cannot create a database link in another user's schema, and you cannot qualify *dblink* with the name of a schema. (Periods are permitted in names of database links, so Oracle interprets the entire name, such as ralph.linktosales, as the name of a database link in your schema rather than as a database link named linktosales in the schema ralph.)

See Also:

- "Referring to Objects in Remote Databases" on page 2-90 for guidelines for naming database links
- *Oracle8i Reference* for information on the GLOBAL_NAMES, OPEN_LINKS, and OPEN_LINKS_PER_INSTANCE initialization parameters

CONNECT TO

The CONNECT TO clause lets you enable a connection to the remote database.

CURRENT_USER Specify CURRENT_USER to create a **current user database link**. The current user must be a global user with a valid account on the remote database for the link to succeed.

If the database link is used directly, that is, not from within a stored object, then the current user is the same as the connected user.

When executing a stored object (such as a procedure, view, or trigger) that initiates a database link, CURRENT_USER is the username that owns the stored object, and not the username that called the object. For example, if the database link appears inside procedure scott.p (created by scott), and user jane calls procedure scott.p, the current user is scott.

However, if the stored object is an invoker-rights function, procedure, or package, the invoker's authorization ID is used to connect as a remote user. For example, if the privileged database link appears inside procedure `scott.p` (an invoker-rights procedure created by `scott`), and user `jane` calls procedure `scott.p`, then CURRENT_USER is `jane` and the procedure executes with Jane's privileges.

See Also:

- CREATE FUNCTION on page 9-43 for more information on invoker-rights functions

- "CURRENT_USER Example" on page 9-31

user IDENTIFIED BY *password*

Specify the username and password used to connect to the remote database (fixed user database link). If you omit this clause, the database link uses the username and password of each user who is connected to the database (connected user database link).

See Also: The "Fixed User Example" on page 9-32

authenticated_clause

Specify the username and password on the target instance. This clause authenticates the user to the remote server and is required for security. The specified username and password must be a valid username and password on the remote instance. The username and password are used only for authentication. No other operations are performed on behalf of this user.

You must specify this clause when using the SHARED clause.

USING '*connect string*'

Specify the service name of a remote database.

See Also: *Net8 Administrator's Guide* for information on specifying remote databases

Examples

CURRENT_USER Example The following statement defines a current-user database link:

```
CREATE DATABASE LINK sales.hq.acme.com
```

```
   CONNECT TO CURRENT_USER
   USING 'sales';
```

Fixed User Example The following statement defines a fixed-user database link named `sales.hq.acme.com`:

```
CREATE DATABASE LINK sales.hq.acme.com
   CONNECT TO scott IDENTIFIED BY tiger
   USING 'sales';
```

Once this database link is created, you can query tables in the schema `scott` on the remote database in this manner:

```
SELECT *
   FROM emp@sales.hq.acme.com;
```

You can also use DML statements to modify data on the remote database:

```
INSERT INTO accounts@sales.hq.acme.com(acc_no, acc_name, balance)
   VALUES (5001, 'BOWER', 2000);

UPDATE accounts@sales.hq.acme.com
   SET balance = balance + 500;

DELETE FROM accounts@sales.hq.acme.com
   WHERE acc_name = 'BOWER';
```

You can also access tables owned by other users on the same database. This statement assumes `scott` has access to Adam's `dept` table:

```
SELECT *
   FROM adams.dept@sales.hq.acme.com;
```

The previous statement connects to the user `scott` on the remote database and then queries Adam's `dept` table.

A synonym may be created to hide the fact that Scott's `emp` table is on a remote database. The following statement causes all future references to `emp` to access a remote `emp` table owned by `scott`:

```
CREATE SYNONYM emp
   FOR scott.emp@sales.hq.acme.com;
```

PUBLIC Database Link Example The following statement defines a shared public fixed user database link named `sales.hq.acme.com` that refers to user `scott` with password `tiger` on the database specified by the string service name '`sales`':

```
CREATE SHARED PUBLIC DATABASE LINK sales.hq.acme.com
   CONNECT TO scott IDENTIFIED BY tiger
   AUTHENTICATED BY anupam IDENTIFIED BY bhide
   USING 'sales';
```

CREATE DIMENSION

Purpose

Use the CREATE DIMENSION statement to create a **dimension**. A dimension defines a parent-child relationship between pairs of column sets, where all the columns of a column set must come from the same table. However, columns in one column set (or "level") can come from a different table than columns in another set. The optimizer uses these relationships with materialized views to perform **query rewrite**. The Summary Advisor uses these relationships to recommend creation of specific materialized views.

Note: Oracle does not automatically validate the relationships you declare when creating a dimension. To validate the relationships specified in the *hierarchy_clause* and the *join_clause*, you must run the DBMS_OLAP.validate_dimension procedure. For information on this procedure, see *Oracle8i Supplied PL/SQL Packages Reference*.

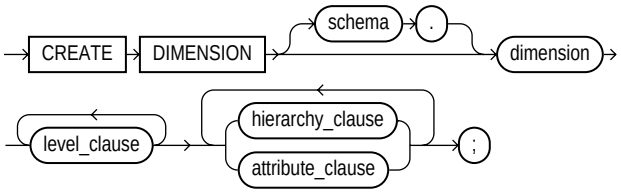
See Also:

- CREATE MATERIALIZED VIEW on page 9-88 for more information on materialized views
- *Oracle8i Data Warehousing Guide* for more information on query rewrite, the optimizer and the Summary Advisor

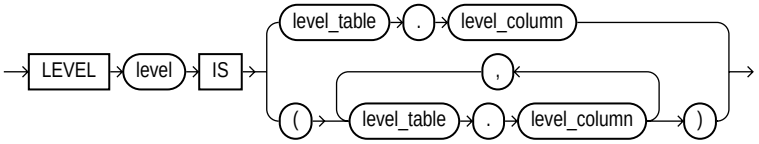
Prerequisites

To create a dimension in your own schema, you must have the CREATE DIMENSION system privilege. To create a dimension in another user's schema, you must have the CREATE ANY DIMENSION system privilege. In either case, you must have the SELECT object privilege on any objects referenced in the dimension.

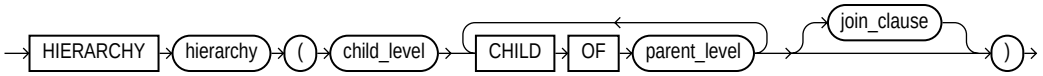
Syntax



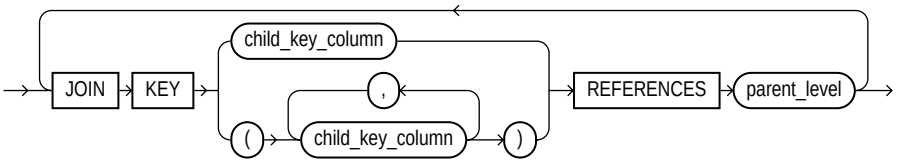
level_clause::=



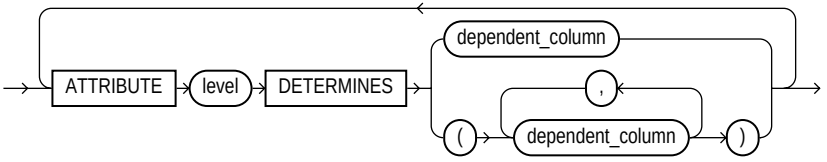
hierarchy_clause::=



join_clause::=



attribute_clause::=



Keywords and Parameters

schema

Specify the schema in which the dimension will be created. If you do not specify *schema*, Oracle creates the dimension in your own schema.

dimension

Specify the name of the dimension. The name must be unique within its schema.

level_clause

The *level_clause* defines a level in the dimension. A level defines dimension hierarchies and attributes.

level

Specify the name of the level

level_table.*level_column*

Specify the columns in the level. You can specify up to 32 columns. The tables you specify in this clause must already exist.

Restrictions:

- All of the columns in a level must come from the same table.
- If columns in different levels come from different tables, then you must specify the *join_clause*.
- The set of columns you specify must be unique to this level.
- The columns you specify cannot be specified in any other dimension.
- Each *level_column* must be non-null. (However, these columns need not have NOT NULL constraints.)

hierarchy_clause

The *hierarchy_clause* defines a linear hierarchy of levels in the dimension. Each hierarchy forms a chain of parent-child relationships among the levels in the dimension. Hierarchies in a dimension are independent of each other. They may (but need not) have columns in common.

Each level in the dimension should be specified at most once in this clause, and each level must already have been named in the *level_clause*.

hierarchy Specify the name of the hierarchy. This name must be unique in the dimension.

child_level Specify the name of a level that has an *n*:1 relationship with a parent level: the *level_columns* of *child_level* cannot be null, and each *child_level* value uniquely determines the value of the next named *parent_level*.

If the child *level_table* is different from the parent *level_table*, you must specify a join relationship between them in the *join_clause*.

parent_level Specify the name of a level.

join_clause

The *join_clause* lets you specify an inner equijoin relationship for a dimension whose columns are contained in multiple tables. This clause is required and permitted only when the columns specified in the hierarchy are not all in the same table.

Restrictions:

- The *child_key_columns* must be non-null and the parent key must be unique and non-null. You need not define constraints to enforce these conditions, but queries may return incorrect results if these conditions are not true.
- Each child key must join with a key in the *parent_level* table.
- Self-joins are not permitted (that is, the *child_key_columns* cannot be in the same table as *parent_level*).

child_key_column Specify one or more columns that are join-compatible with columns in the parent level.

If you do not specify the schema and table of each *child_column*, the schema and table are inferred from the CHILD OF relationship in the *hierarchy_clause*. If you do specify the schema and column of a *child_key_column*, the schema and table must match the schema and table of columns that comprise the child of *parent_level* in the *hierarchy_clause*.

Restrictions:

- All of the child-key columns must come from the same table.
- The number of child-key columns must match the number of columns in *parent_level*, and the columns must be joinable.
- Do not specify multiple child key columns unless the parent level consists of multiple columns.
- You can specify only one *join_clause* for a given pair of levels in the same hierarchy.

parent_level Specify the name of a level.

attribute_clause

The *attribute_clause* lets you specify the columns that are uniquely determined by a hierarchy level. The columns in *level* must all come from the same table as the *dependent_columns*. The *dependent_columns* need not have been specified in the *level_clause*.

For example, if the hierarchy levels are *city*, *state*, and *country*, then *city* might determine *mayor*, *state* might determine *governor*, and *country* might determine *president*.

Examples

CREATE DIMENSION Example This statement creates a time dimension on table time_tab, and creates a geog dimension on tables city, state, and country.

```
CREATE DIMENSION time
   LEVEL curDate         IS time_tab.curDate
   LEVEL month           IS time_tab.month
   LEVEL qtr             IS time_tab.qtr
   LEVEL year            IS time_tab.year
   LEVEL fiscal_week     IS time_tab.fiscal_week
   LEVEL fiscal_qtr      IS time_tab.fiscal_qtr
   LEVEL fiscal_year     IS time_tab.fiscal_year
   HIERARCHY month_rollup (
      curDate            CHILD OF
      month              CHILD OF
      qtr                CHILD OF
      year)
   HIERARCHY fiscal_year_rollup (
```

```
      curDate            CHILD OF
      fiscal_week        CHILD OF
      fiscal_qtr         CHILD OF
      fiscal_year )
   ATTRIBUTE curDate     DETERMINES (holiday, dayOfWeek)
   ATTRIBUTE month       DETERMINES (yr_ago_month, qtr_ago_month)
   ATTRIBUTE fiscal_qtr  DETERMINES yr_ago_qtr
   ATTRIBUTE year        DETERMINES yr_ago ;

CREATE DIMENSION geog
   LEVEL cityID         IS (city.city, city.state)
   LEVEL stateID        IS state.state
   LEVEL countryID      IS country.country
   HIERARCHY political_rollup (
      cityID            CHILD OF
      stateID           CHILD OF
      countryID
         JOIN KEY city.state REFERENCES stateID
         JOIN KEY state.country REFERENCES countryID);
```

CREATE DIRECTORY

Purpose

Use the CREATE DIRECTORY statement to create a directory object. A directory object specifies an alias for a directory on the server's file system where external binary file LOBs (BFILEs) are located. You can use directory names when referring to BFILEs in your PL/SQL code and OCI calls, rather than hard-coding the operating system pathname, thereby allowing greater file management flexibility.

All directories are created in a single namespace and are not owned by an individual's schema. You can secure access to the BFILEs stored within the directory structure by granting object privileges on the directories to specific users.

See Also:

- "Large Object (LOB) Datatypes" on page 2-16 for more information on BFILE objects
- GRANT on page 11-31 for more information on granting object privileges

Prerequisites

You must have CREATE ANY DIRECTORY system privileges to create directories.

When you create a directory, you are automatically granted the READ object privilege and can grant READ privileges to other users and roles. The DBA can also grant this privilege to other users and roles.

You must also create a corresponding operating system directory for file storage. Your system or database administrator must ensure that the operating system directory has the correct read permissions for Oracle processes.

Privileges granted for the directory are created independently of the permissions defined for the operating system directory. Therefore, the two may or may not correspond exactly. For example, an error occurs if user scott is granted READ privilege on the directory schema object, but the corresponding operating system directory does not have READ permission defined for Oracle processes.

Syntax

Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the directory database object if it already exists. You can use this clause to change the definition of an existing directory without dropping, re-creating, and regranting database object privileges previously granted on the directory.

Users who had previously been granted privileges on a redefined directory can still access the directory without being regranted the privileges

> **See Also:** DROP DIRECTORY on page 10-133 for information on removing a directory from the database

directory

Specify the name of the directory object to be created. The maximum length of *directory* is 30 bytes. You cannot qualify a directory object with a schema name.

> **Note:** Oracle does not verify that the directory you specify actually exists. Therefore, take care that you specify a valid directory in your operating system. In addition, if your operating system uses case-sensitive pathnames, be sure you specify the directory in the correct format. (However, you need not include a trailing slash at the end of the pathname.)

'path_name'

Specify the full pathname of the operating system directory on the server where the files are located. The single quotes are required, with the result that the path name is case sensitive.

Example

CREATE DIRECTORY Example The following statement redefines directory database object `bfile_dir` to enable access to `BFILE`s stored in the operating system directory `/private1/lob/files`:

```
CREATE OR REPLACE DIRECTORY bfile_dir AS '/private1/LOB/files';
```

CREATE FUNCTION

Purpose

Use the CREATE FUNCTION statement to create a standalone stored function or a call specification. (You can also create a function as part of a package using the CREATE PACKAGE statement.)

A **stored function** (also called a **user function**) is a set of PL/SQL statements you can call by name. Stored functions are very similar to procedures, except that a function returns a value to the environment in which it is called. User functions can be used as part of a SQL expression.

A **call specification** declares a Java method or a third-generation language (3GL) routine so that it can be called from SQL and PL/SQL. The call specification tells Oracle which Java method, or which named function in which shared library, to invoke when a call is made. It also tells Oracle what type conversions to make for the arguments and return value.

See Also:

- CREATE PROCEDURE on page 9-132 for a general discussion of procedures and functions
- "Examples" on page 9-50 for examples of creating functions
- CREATE PACKAGE on page 9-122 for information on creating packages
- ALTER FUNCTION on page 7-38 for information on modifying a function
- CREATE LIBRARY on page 9-86 for information on shared libraries
- DROP FUNCTION on page 10-134 for information on dropping a standalone function
- *Oracle8i Application Developer's Guide - Fundamentals* for more information about registering external functions

Prerequisites

Before a stored function can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

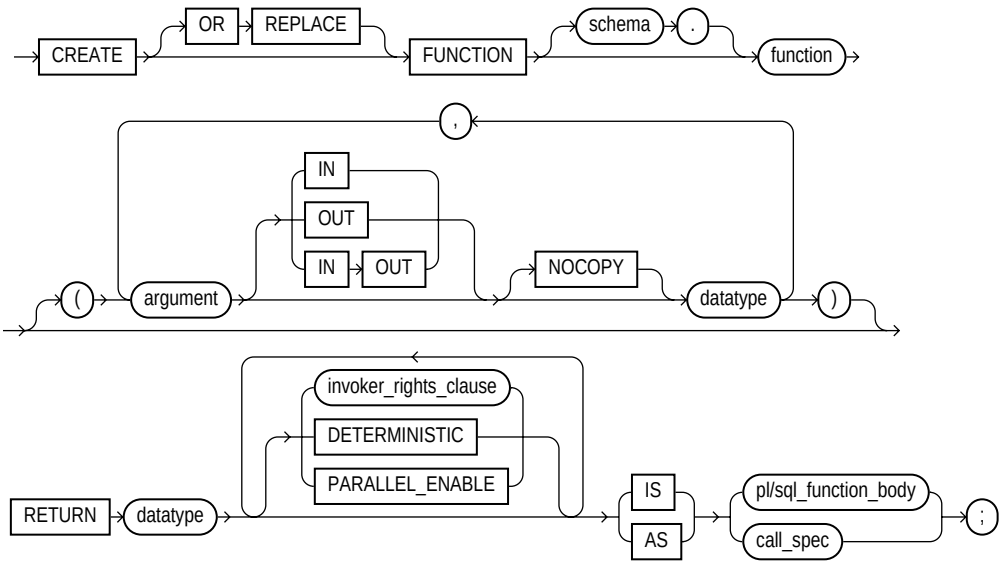
To create a function in your own schema, you must have the CREATE PROCEDURE system privilege. To create a function in another user's schema, you must have the CREATE ANY PROCEDURE system privilege. To replace a function in another user's schema, you must have the ALTER ANY PROCEDURE system privilege.

To invoke a call specification, you may need additional privileges (for example, EXECUTE privileges on C library for a C call specification).

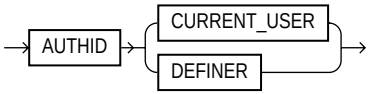
To embed a CREATE FUNCTION statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

> **See Also:** *PL/SQL User's Guide and Reference* or *Oracle8i Java Stored Procedures Developer's Guide* for more information on such prerequisites

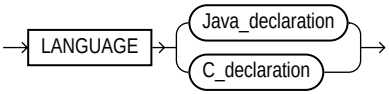
Syntax



invoker_rights_clause::=

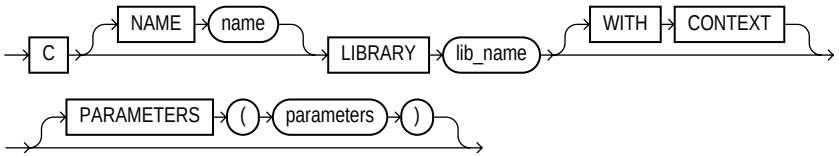


call_spec::=



Java_declaration::=

C_declaration::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the function if it already exists. Use this clause to change the definition of an existing function without dropping, re-creating, and regranting object privileges previously granted on the function. If you redefine a function, Oracle recompiles it.

Users who had previously been granted privileges on a redefined function can still access the function without being regranted the privileges.

If any function-based indexes depend on the function, Oracle marks the indexes DISABLED.

> **See Also:** ALTER FUNCTION on page 7-38 for information on recompiling functions

schema

Specify the schema to contain the function. If you omit *schema*, Oracle creates the function in your current schema.

function

Specify the name of the function to be created. If creating the function results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SHOW ERRORS command.

Restrictions on User-Defined Functions

User-defined functions cannot be used in situations that require an unchanging definition. Thus, you cannot use user-defined functions:

- In a CHECK constraint clause of a CREATE TABLE or ALTER TABLE statement
- In a DEFAULT clause of a CREATE TABLE or ALTER TABLE statement

In addition, when a function is called from within a query or DML statement, the function cannot:

- Have OUT or IN OUT parameters
- Commit or roll back the current transaction, create or roll back to a savepoint, or alter the session or the system. DDL statements implicitly commit the current transaction, so a user-defined function cannot execute any DDL statements.
- Write to the database, if the function is being called from a SELECT statement. However, a function called from a subquery in a DML statement can write to the database.
- Write to the same table that is being modified by the statement from which the function is called, if the function is called from a DML statement.

Except for the restriction on OUT and IN OUT parameters, Oracle enforces these restrictions not only for the function called directly from the SQL statement, but also for any functions that function calls, and on any functions called from the SQL statements executed by that function or any function it calls.

argument

Specify the name of an argument to the function. If the function does not accept arguments, you can omit the parentheses following the function name.

IN — Specify IN to indicate that you must supply a value for the argument when calling the function. This is the default.

`OUT` Specify `OUT` to indicate that the function will set the value of the argument.

`IN OUT` Specify `IN OUT` to indicate that a value for the argument can be supplied by you and may be set by the function.

`NOCOPY` Specify `NOCOPY` to instruct Oracle to pass this argument as fast as possible. This clause can significantly enhance performance when passing a large value like a record, an index-by table, or a varray to an `OUT` or `IN OUT` parameter. (IN parameter values are always passed `NOCOPY`.)

- When you specify `NOCOPY`, assignments made to a package variable may show immediately in this parameter (or assignments made to this parameter may show immediately in a package variable) if the package variable is passed as the actual assignment corresponding to this parameter.
- Similarly, changes made either to this parameter or to another parameter may be visible immediately through both names if the same variable is passed to both.
- If the function is exited with an unhandled exception, any assignment made to this parameter may be visible in the caller's variable.

These effects may or may not occur on any particular call. You should use `NOCOPY` only when these effects would not matter.

`datatype` Specify the datatype of an argument. An argument can have any datatype supported by PL/SQL.

The datatype cannot specify a length, precision, or scale. Oracle derives the length, precision, or scale of an argument from the environment from which the function is called.

RETURN

`datatype` Specify the datatype of the function's return value. Because every function must return a value, this clause is required. The return value can have any datatype supported by PL/SQL.

The datatype cannot specify a length, precision, or scale. Oracle derives the length, precision, or scale of the return value from the environment from which the function is called.

See Also: *PL/SQL User's Guide and Reference* for information on PL/SQL datatypes

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the function executes with the privileges and in the schema of the user who owns it or with the privileges and in the schema of CURRENT_USER.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the function.

AUTHID CURRENT_USER — Specify CURRENT_USER if you want the function to execute with the privileges of CURRENT_USER. This clause creates an "invoker-rights function."

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the function resides.

AUTHID DEFINER — Specify DEFINER if you want the function to execute with the privileges of the owner of the schema in which the function resides, and that external names resolve in the schema where the function resides. This is the default.

See Also:

- *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined
- *PL/SQL User's Guide and Reference*

DETERMINISTIC

DETERMINISTIC is an optimization hint that allows the system to use a saved copy of the function's return result (if such a copy is available). The saved copy could come from a materialized view, a function-based index, or a redundant call to the same function in the same SQL statement. The query optimizer can choose whether to use the saved copy or re-call the function.

The function should reliably return the same result value whenever it is called with the same values for its arguments. Therefore, do not define the function to use package variables or to access the database in any way that might affect the function's return result, because the results of doing so will not be captured if the system chooses not to call the function.

A function must be declared DETERMINISTIC in order to be called in the expression of a function-based index, or from the query of a materialized view if that view is marked REFRESH FAST or ENABLE QUERY REWRITE.

See Also:

- *Oracle8i Data Warehousing Guide* for information on materialized views
- CREATE INDEX on page 9-52 for information on function-based indexes

PARALLEL_ENABLE

PARALLEL_ENABLE is an optimization hint indicating that the function can be executed from a parallel execution server of a parallel query operation. The function should not use session state, such as package variables, as those variables may not be shared among the parallel execution servers.

See Also: *Oracle8i Application Developer's Guide - Fundamentals*

IS | AS

pl/sql_subprogram_body — Declare the function in a PL/SQL subprogram body.

See Also: *Oracle8i Application Developer's Guide - Fundamentals* for more information on PL/SQL subprograms

call_spec The *call_spec* lets you map a Java or C method name, parameter types, and return type to their SQL counterparts. In *Java_declaration*, '*string*' identifies the Java implementation of the method.

See Also:

- *Oracle8i Java Stored Procedures Developer's Guide*

- *Oracle8i Application Developer's Guide - Fundamentals* for an explanation of the parameters and semantics of the *C_declaration*

AS EXTERNAL AS EXTERNAL is an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle Corporation recommends that you use the AS LANGUAGE C syntax.

Examples

CREATE FUNCTION Examples The following statement creates the function get_bal.

```
CREATE FUNCTION get_bal(acc_no IN NUMBER)
   RETURN NUMBER
   IS acc_bal NUMBER(11,2);
   BEGIN
      SELECT balance
      INTO acc_bal
      FROM accounts
      WHERE account_id = acc_no;
      RETURN(acc_bal);
    END;
```

The get_bal function returns the balance of a specified account.

When you call the function, you must specify the argument acc_no, the number of the account whose balance is sought. The datatype of acc_no is number.

The function returns the account balance. The RETURN clause of the CREATE FUNCTION statement specifies the datatype of the return value to be NUMBER.

The function uses a SELECT statement to select the balance column from the row identified by the argument acc_no in the accounts table. The function uses a

RETURN statement to return this value to the environment in which the function is called.

The function created above can be used in a SQL statement. For example:

```
SELECT get_bal(100) FROM DUAL;
```

The following statement creates PL/SQL standalone function `get_val` that registers the C routine `c_get_val` as an external function. (The parameters have been omitted from this example.)

```
CREATE FUNCTION get_val
   ( x_val IN NUMBER,
   y_val IN NUMBER,
   image IN LONG RAW )
   RETURN BINARY_INTEGER AS LANGUAGE C
      NAME "c_get_val"
      LIBRARY c_utils
      PARAMETERS (...);
```

CREATE INDEX

Purpose

Use the CREATE INDEX statement to create an index on

- One or more columns of a table, a partitioned table, an index-organized table, or a cluster
- One or more scalar typed object attributes of a table or a cluster
- A nested table storage table for indexing a nested table column

An **index** is a schema object that contains an entry for each value that appears in the indexed column(s) of the table or cluster and provides direct, fast access to rows. Oracle supports several types of index:

- Conventional (B*-tree) indexes
- **Bitmap indexes**, which store rowids associated with a key value as a bitmap
- **Partitioned indexes**, which consist of partitions containing an entry for each value that appears in the indexed column(s) of the table.
- **Function-based indexes**, which are based on expressions. They enable you to construct queries that evaluate the value returned by an expression, which in turn may include functions (built-in or user-defined).
- **Domain indexes**, which are instances of an application-specific index of type .

See Also:

- *Oracle8i Concepts* for a discussion of indexes
- ALTER INDEX on page 7-40 for information on modifying an index
- DROP INDEX on page 10-136 for information on dropping an index

Prerequisites

To create an index in your own schema, one of the following conditions must be true:

- The table or cluster to be indexed must be in your own schema.
- You must have INDEX privilege on the table to be indexed.

- You must have CREATE ANY INDEX system privilege.

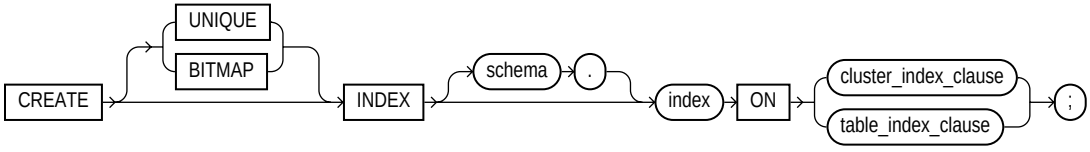
To create an index in another schema, you must have CREATE ANY INDEX system privilege. Also, the owner of the schema to contain the index must have either space quota on the tablespaces to contain the index or index partitions, or UNLIMITED TABLESPACE system privilege.

To create a domain index in your own schema, in addition to the prerequisites for creating a conventional index, you must also have EXECUTE privilege on the indextype. If you are creating a domain index in another user's schema, the index owner also must have EXECUTE privilege on the indextype and its underlying implementation type. Before creating a domain index, you should first define the indextype. See .

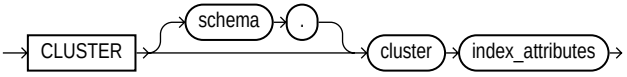
To create a function-based index in your own schema on your own table, in addition to the prerequisites for creating a conventional index, you must have the QUERY REWRITE system privilege. To create the index in another schema or on another schema's table, you must have the GLOBAL QUERY REWRITE privilege. In both cases, the table owner must also have the EXECUTE object privilege on the function(s) used in the function-based index. In addition, in order for Oracle to use function-based indexes in queries, the QUERY_REWRITE_ENABLED parameter must be set to `true`, and the QUERY_REWRITE_INTEGRITY parameter must be set to `trusted`.

See Also: CREATE INDEXTYPE on page 9-76

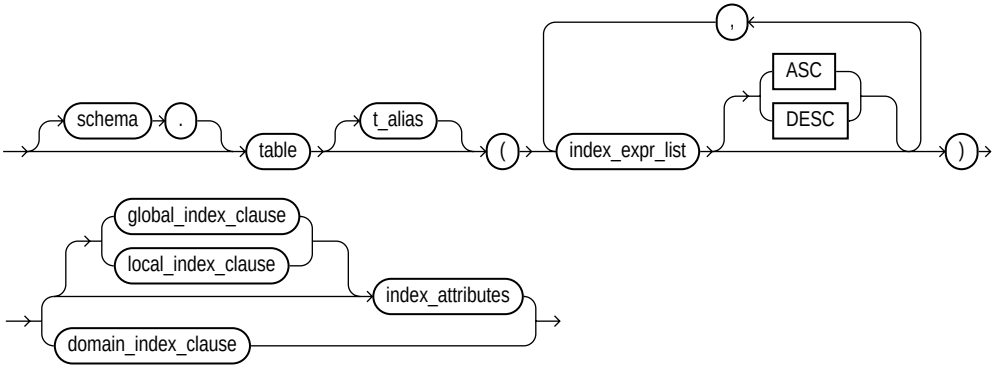
Syntax



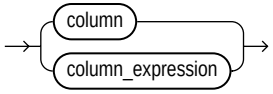
cluster_index_clause::=



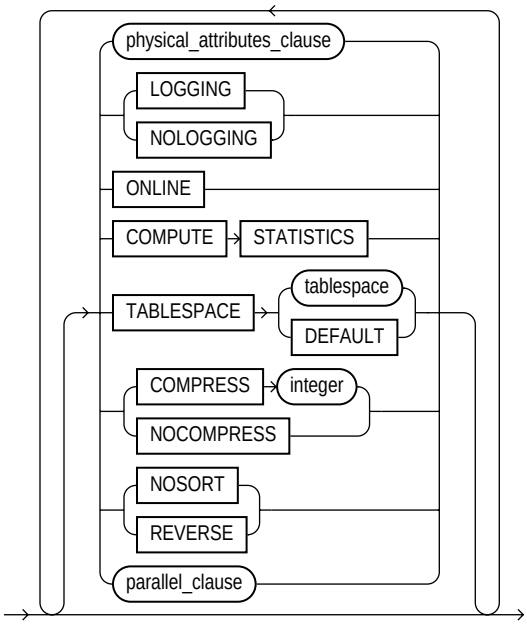
table_index_clause::=



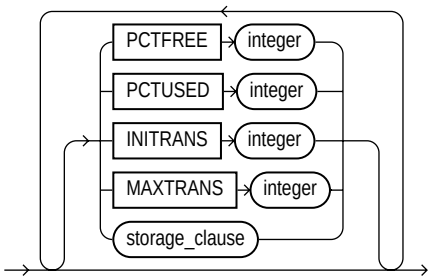
index_expr_list::=



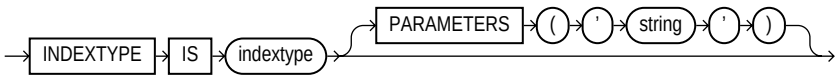
index_attributes::=



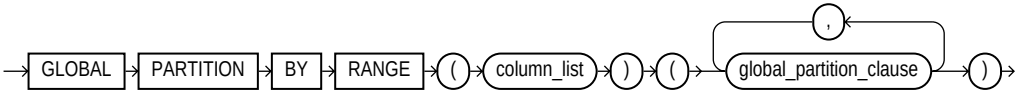
physical_attributes_clause::=



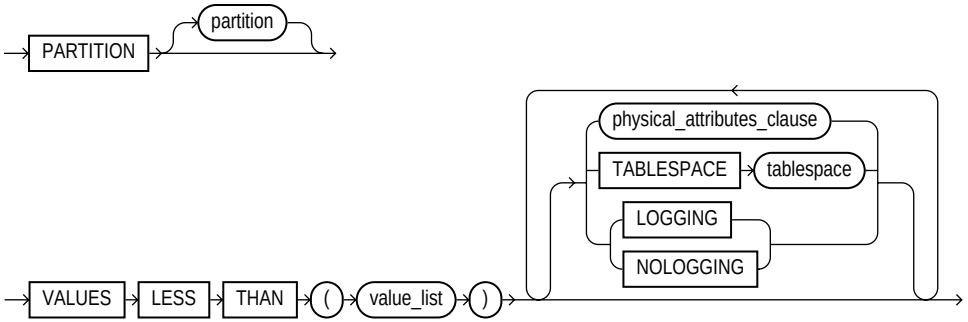
domain_index_clause::=



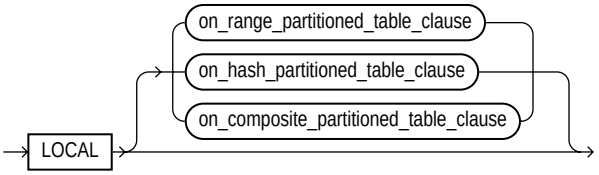
global_index_clause::=



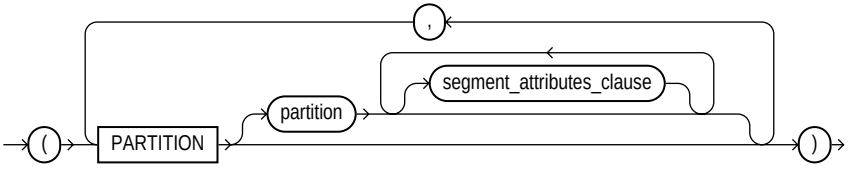
global_partition_clause::=



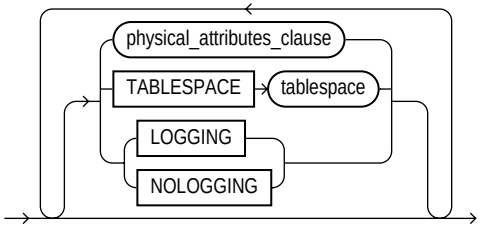
local_index_clauses::=



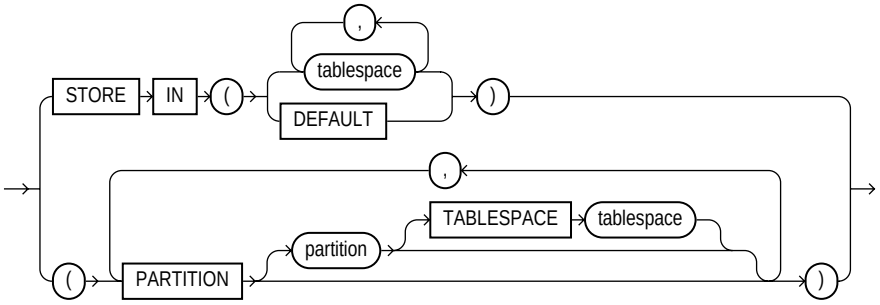
on_range_partitioned_table_clause::=



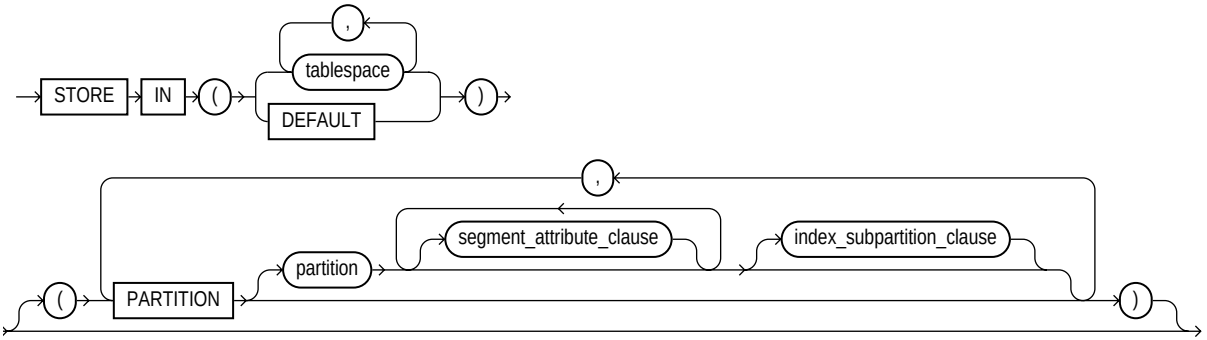
segment_attributes_clause::=



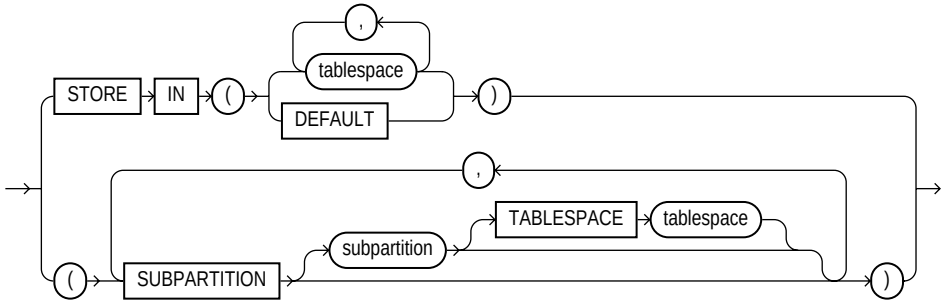
on_hash_partitioned_table_clause::=



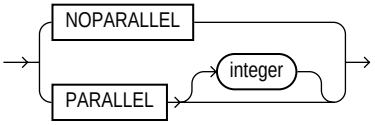
on_composite_partitioned_table_clause::=



index_subpartition_clause::=



parallel_clause::=



storage_clause: See storage_clause on page 11-129.

Keywords and Parameters

UNIQUE

Specify UNIQUE to indicate that the value of the column (or columns) upon which the index is based must be unique. If the index is local nonprefixed (see below), then the index key must contain the partitioning key.

Oracle recommends that you do not explicitly define UNIQUE indexes on tables. Uniqueness is strictly a logical concept and should be associated with the *definition* of a table. Therefore, define UNIQUE integrity constraints on the desired columns.

Restrictions:

- You cannot specify both UNIQUE and BITMAP.
- You cannot specify UNIQUE for a domain index.

> **See Also:** *constraint_clause* on page 8-136 for information on integrity constraints

BITMAP

Specify BITMAP to indicate that *index* is to be created as a bitmap, rather than as a B-tree. Bitmap indexes store the rowids associated with a key value as a bitmap. Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value. The internal representation of bitmaps is best suited for applications with low levels of concurrent transactions, such as data warehousing.

Restrictions:

- You cannot specify BITMAP when creating a global partitioned index or an index-organized table.
- You cannot specify both UNIQUE and BITMAP.
- You cannot specify BITMAP for a domain index.

> **See Also:** *Oracle8i Concepts* and *Oracle8i Performance Guide and Reference* for more information about using bitmap indexes

schema

Specify the schema to contain the index. If you omit *schema*, Oracle creates the index in your own schema.

index

Specify the name of the index to be created. An *index* can contain several partitions.

cluster_index_clause

Use the *cluster_index_clause* to identify the cluster for which a cluster index is to be created. If you do not qualify cluster with *schema*, Oracle assumes the cluster is in your current schema. You cannot create a cluster index for a hash cluster.

> **See Also:** CREATE CLUSTER on page 9-3

table_index_clause

Specify the table (and its attributes) on which you are defining the index. If you do not qualify *table* with *schema*, Oracle assumes the table is contained in your own schema.

You create an index on a nested table column by creating the index on the nested table storage table. Include the NESTED_TABLE_ID pseudocolumn of the storage table to create a UNIQUE index, which effectively ensures that the rows of a nested table value are distinct.

Restrictions:

- If the index is local, then *table* must be partitioned.
- If the table is index-organized, this statement creates a secondary index. You cannot specify BITMAP or REVERSE for this secondary index, and the combined size of the index key and the logical rowid should be less than half the block size.
- If *table* is a temporary table, the index will also be temporary with the same scope (session or transaction) as *table*. The following restrictions apply to indexes on temporary table:
  - The index cannot be a partitioned index or a domain index.
  - You cannot specify the *physical_attributes_clause* or the *parallel_clause*.
  - You cannot specify LOGGING, NOLOGGING, or TABLESPACE.

  See Also: CREATE TABLE on page 10-7 and *Oracle8i Concepts* for more information on temporary tables

t_alias Specify a correlation name (alias) for the table upon which you are building the index.

> **Note:** This alias is required if the *index_expression_list* references any object type attributes or object type methods. See "Function-based Index on Type Method Example" on page 9-73.

index_expr_list

The *index_expr_list* lets you specify the column or column expression upon which the index is based.

column Specify the name of a column in the table. A bitmap index can have a maximum of 30 columns. Other indexes can have as many as 32 columns.

Restriction: You cannot create an index on columns or attributes whose type is user-defined, LONG, LONG RAW, LOB, or REF, except that Oracle supports an index on REF type columns or attributes that have been defined with a SCOPE clause.

You can create an index on a scalar object attribute column or on the system-defined NESTED_TABLE_ID column of the nested table storage table. If you specify an object attribute column, the column name must be qualified with the table name. If you specify a nested table column attribute, it must be qualified with the outermost table name, the containing column name, and all intermediate attribute names leading to the nested table column attribute.

column_expression is an expression built from columns of *table*, constants, SQL functions, and user-defined functions. When you specify *column_expression*, you create a **function-based index**.

Name resolution of the function is based on the schema of the index creator. User-defined functions used in *column_expression* are fully name resolved during the CREATE INDEX operation.

After creating a function-based index, collect statistics on both the index and its base table using the ANALYZE statement. Oracle cannot use the function-based index until these statistics have been generated.

See Also: ANALYZE on page 8-96

Notes on function-based indexes:

- When you subsequently query a table that uses a function-based index, you must ensure in the query that *column_expression* is not null. However, Oracle will use a function-based index in a query even if the columns specified in the WHERE clause are in a different order than their order in the *column_expression* that defined the function-based index.

  See Also: The "Function-Based Index Example" on page 9-72

- If the function on which the index is based becomes invalid or is dropped, Oracle marks the index DISABLED. Queries on a DISABLED index fail if the optimizer chooses to use the index. DML operations on a DISABLED index fail unless the index is also marked UNUSABLE **and** the parameter SKIP_UNUSABLE_INDEXES is set to true.

  See Also: ALTER SESSION on page 7-105 for more information on this parameter

- Oracle's use of function-based indexes is also affected by the setting of the QUERY_REWRITE_ENABLED session parameter.

  See Also: ALTER SESSION on page 7-105

- If a public synonym for a function, package, or type is used in *column_expression*, and later an actual object with the same name is created in the table owner's schema, then Oracle will disable the function-based index. When you subsequently enable the function-based index using ALTER INDEX ... ENABLE or ALTER INDEX ... REBUILD, the function, package, or type used in the *column_expression* will continue to resolve to the function, package, or type to which the public synonym originally pointed. It will not resolve to the new function, package, or type.

- If the definition of a function-based index generates internal conversion to character data, use caution when changing NLS parameter settings. Function-based indexes use the current database settings for NLS parameters. If you reset these parameters at the session level, queries using the function-based index may return incorrect results. Two exceptions are the collation parameters (NLS_SORT and NLS_COMP). Oracle handles the conversions correctly even if these have been reset at the session level.

Restrictions on function-based indexes:

- Any user-defined function referenced in *column_expression* must be DETERMINISTIC.

- For a function-based globally partitioned index, the *column_expression* cannot be the partitioning key.

- All functions must be specified with parentheses, even if they have no parameters. Otherwise Oracle interprets them as column names.
- Any function you specify in *column_expression* must return a repeatable value. For example, you cannot specify the SYSDATE or USER function or the ROWNUM pseudocolumn.
- You cannot build a function-based index on LOB, REF, nested table, or varray columns. In addition, the function in *column_expression* cannot take as arguments any objects with attributes of type LOB, REF, nested table, or varray.
- The *column_expression* cannot contain any aggregate functions.

See Also: CREATE FUNCTION on page 9-43 and *PL/SQL User's Guide and Reference*

ASC | DESC

Use ASC or DESC to indicate whether the index should be created in ascending or descending order. Indexes on character data are created in ascending or descending order of the character values in the database character set.

Oracle treats descending indexes as if they were function-based indexes. You do not need the QUERY REWRITE or GLOBAL QUERY REWRITE privileges to create them, as you do with other function-based indexes. However, as with other function-based indexes, Oracle does not use descending indexes until you first analyze the index and the table on which the index is defined. See the *column_expression* clause of this statement.

Restriction: You cannot specify either of these clauses for a domain index. You cannot specify DESC for a reverse index. Oracle ignores DESC if index is bitmapped or if the COMPATIBLE initialization parameter is set to a value less than 8.1.0.

index_attributes

physical_attributes_clause

Use the *physical_attributes_clause* to establish values for physical and storage characteristics for the index. See CREATE TABLE on page 10-7.

Restriction: You cannot specify the PCTUSED parameter for an index.

PCTFREE

Specify the percentage of space to leave free for updates and insertions within each of the index's data blocks.

storage_clause

Use the storage_clause to establish the storage characteristics for the index.

See Also: *storage_clause* on page 11-129

TABLESPACE

Specify the name of the tablespace to hold the index, index partition, or index subpartition. If you omit this clause, Oracle creates the index in the default tablespace of the owner of the schema containing the index.

For a local index, you can specify the keyword DEFAULT in place of *tablespace*. New partitions or subpartitions added to the local index will be created in the same tablespace(s) as the corresponding partitions or subpartitions of the underlying table.

COMPRESS

Specify COMPRESS to enable key compression, which eliminates repeated occurrence of key column values and may substantially reduce storage. Use *integer* to specify the prefix length (number of prefix columns to compress).

- For unique indexes, the valid range of prefix length values is from 1 to the number of key columns minus 1. The default prefix length is the number of key columns minus 1.
- For nonunique indexes, the valid range of prefix length values is from 1 to the number of key columns. The default prefix length is the number of key columns.

Oracle compresses only nonpartitioned indexes that are nonunique or unique indexes of at least two columns.

Restriction: You cannot specify COMPRESS for a bitmap index.

NOCOMPRESS — Specify NOCOMPRESS to disable key compression. This is the default.

NOSORT — Specify NOSORT to indicate to Oracle that the rows are stored in the database in ascending order, so that Oracle does not have to sort the rows when creating the index. If the rows of the indexed column or columns are not stored in ascending order, Oracle returns an error. For greatest savings of sort time and space, use this clause immediately after the initial load of rows into a table.

Restrictions:

- You cannot specify REVERSE with this clause.
- You cannot use this clause to create a cluster, partitioned, or bitmap index.
- You cannot specify this clause for a secondary index on an index-organized table.

REVERSE — Specify REVERSE to store the bytes of the index block in reverse order, excluding the rowid. You cannot specify NOSORT with this clause.

You cannot reverse a bitmap index or an index-organized table.

LOGGING | NOLOGGING — Indicate whether the creation of the index will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file. It also specifies that subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against the index are logged or not logged. LOGGING is the default.

If *index* is nonpartitioned, this is the logging attribute of the index.

If *index* is partitioned, the logging attribute specified is

- The default value of all partitions specified in the CREATE statement (unless you specify LOGGING | NOLOGGING in the PARTITION description clause)
- The default value for the segments associated with the index partitions
- The default value for local index partitions or subpartitions added implicitly during subsequent ALTER TABLE ... ADD PARTITION operations

In NOLOGGING mode, data is modified with minimal logging (to mark new extents INVALID and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, since the redo data is not logged. Thus if you cannot afford to lose this index, it is important to take a backup after the NOLOGGING operation.

If the database is run in ARCHIVELOG mode, media recovery from a backup taken before the LOGGING operation will re-create the index. However, media recovery from a backup taken before the NOLOGGING operation will not re-create the index.

The logging attribute of the index is independent of that of its base table.

If you omit this clause, the logging attribute is that of the tablespace in which it resides.

> **See Also:** *Oracle8i Concepts* and *Oracle8i Parallel Server Concepts* for more information about logging and parallel DML

ONLINE

Specify ONLINE to indicate that DML operations on the table will be allowed during creation of the index.

Restriction: Parallel DML is not supported during online index building. If you specify ONLINE and then issue parallel DML statements, Oracle returns an error.

> **See Also:** *Oracle8i Concepts* for a description of online index building and rebuilding

COMPUTE STATISTICS

Specify COMPUTE STATISTICS to collect statistics at relatively little cost during the creation of an index. These statistics are stored in the data dictionary for ongoing use by the optimizer in choosing a plan of execution for SQL statements.

The types of statistics collected depend on the type of index you are creating.

> **Note:** If you create an index using another index (instead of a table), the original index might not provide adequate statistical information. Therefore, Oracle generally uses the base table to compute the statistics, which will improve the statistics but may negatively affect performance.

Additional methods of collecting statistics are available in PL/SQL packages and procedures.

See Also: *Oracle8i Supplied PL/SQL Packages Reference*

parallel_clause — Specify the *parallel_clause* if you want creation of the index to be parallelized.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL — Specify NOPARALLEL for serial execution. This is the default.

PARALLEL — Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

global_index_clause

The *global_index_clause* lets you specify that the partitioning of the index is user defined and is not equipartitioned with the underlying table. By default, nonpartitioned indexes are global indexes.

PARTITION BY RANGE — Specify PARTITION BY RANGE to indicate that the global index is partitioned on the ranges of values from the columns specified in *column_list*. You cannot specify this clause for a local index.

(column_list) — Specify the name of the column(s) of a table on which the index is partitioned. The *column_list* must specify a left prefix of the index column list.

You cannot specify more than 32 columns in *column_list*, and the columns cannot contain the ROWID pseudocolumn or a column of type ROWID.

Note: If your enterprise has or will have databases using different character sets, use caution when partitioning on character columns. The sort sequence of characters is not identical in all character sets.

See Also: *Oracle8i National Language Support Guide* for more information on character set support

PARTITION *partition*

The PARTITION clause lets you describe the individual partitions. The number of clauses determines the number of partitions. If you omit *partition*, Oracle generates a name with the form SYS_P*n*.

VALUES LESS THAN (*value_list*)

Specify the (noninclusive) upper bound for the current partition in a global index. The *value_list* is a comma-separated, ordered list of literal values corresponding to *column_list* in the *partition_by_range_clause*. Always specify MAXVALUE as the *value_list* of the last partition.

Restriction: You cannot specify this clause for a local index.

Note: If *index* is partitioned on a DATE column, and if the NLS date format does not specify the first two digits of the year, you must use the TO_DATE function with a 4-character format mask for the year. The NLS date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT.

See Also:

- *Oracle8i National Language Support Guide* for more information on these initialization parameters

- "Partitioned Table Example" on page 10-51

local_index_clauses

The *local_index_clauses* let you specify that the index is partitioned on the same columns, with the same number of partitions and the same partition bounds

as *table*. Oracle automatically maintains LOCAL index partitioning as the underlying table is repartitioned.

on_range_partitioned_table_clause — Specify the name and attributes of an index on a range-partitioned table.

PARTITION *partition* — Specify the names of the individual partitions. The number of clauses determines the number of partitions. For a local index, the number of index partitions must be equal to the number of the table partitions, and in the same order.

If you omit *partition*, Oracle generates a name that is consistent with the corresponding table partition. If the name conflicts with an existing index partition name, the form SYS_P*n* is used.

on_hash_partitioned_table_clause — Specify the name and attributes of an index on a hash-partitioned table. If you do not specify *partition*, Oracle uses the name of the corresponding base table partition, unless it conflicts with an explicitly specified name of another index partition. In this case, Oracle generates a name of the form SYS_P*nnn*.

You can optionally specify TABLESPACE for all index partitions or for one or more individual partitions. If you do not specify TABLESPACE at the index or partition level, Oracle stores each index partition in the same tablespace as the corresponding table partition.

on_composite_partitioned_table_clause — Specify the name and attributes of an index on a composite-partitioned table. The first STORE IN clause specifies the default tablespace for the index subpartitions. You can override this storage by specifying a different tablespace in the *index_subpartitioning_clause*.

If you do not specify TABLESPACE for subpartitions either in this clause or in the *index_subpartitioning_clause*, Oracle uses the tablespace specified for *index*. If you also do not specify TABLESPACE for *index*, Oracle stores the subpartition in the same tablespace as the corresponding table subpartition.

`STORE IN` The `STORE IN` clause lets you specify how index hash partitions (for a hash-partitioned index) or index subpartitions (for a composite-partitioned index) are to be distributed across various tablespaces. The number of tablespaces does not have to equal the number of index partitions. If the number of index partitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.

- `DEFAULT` The `DEFAULT` clause is valid only for a local index on a hash or composite-partitioned table. This clause overrides any tablespace specified at the index level for a partition or subpartition, and stores the index partition or subpartition in the same partition as the corresponding table partition or subpartition.
- *`index_subpartition_clause`* The *`index_subpartition_clause`* lets you specify one or more tablespaces in which to store all subpartitions in *`partition`* or one or more individual subpartitions in *`partition`*. The subpartition inherits all other attributes from *`partition`*. Attributes not specified for *`partition`* are inherited from *`index`*.

`domain_index_clause` Use the *`domain_index_clause`* to indicate that *index* is a domain index.

Restrictions:

- The *`index_expr_list`* can specify only a single column.
- You can define only one domain index on a column.
- You cannot specify a bitmap, unique, or function-based domain index.
- You cannot create a local domain index on a partitioned table.
- You cannot create a domain index on a partitioned table with row movement enabled.

column — Specify the table columns or object attributes on which the index is defined. Each *column* can have only one domain index defined on it.

Restrictions:

- You cannot create a domain index on a column of datatype REF, varray, nested table, LONG, or LONG RAW.
- You can create a domain index on a column of user-defined type, but not on an attribute of a column of user-defined type if that attribute itself is a user-defined type.

indextype — Specify the name of the indextype. This name should be a valid schema object that you have already defined.

See Also: CREATE INDEXTYPE on page 9-76

PARAMETERS '*string*' — Specify the parameter string that is passed uninterpreted to the appropriate indextype routine. The maximum length of the parameter string is 1000 characters.

Once the domain index is created, Oracle invokes this routine (see .) If the routine does not return successfully, the domain index is marked FAILED. The only operation supported on an failed domain index is DROP INDEX.

See Also: *Oracle8i Data Cartridge Developer's Guide* for information on these routines

Examples

PARALLEL Example The following statement creates an index using 10 parallel execution servers, 5 to scan scott.emp and another 5 to populate the emp_ix index:

```
CREATE INDEX emp_idx
   ON scott.emp (ename)
   PARALLEL 5;
```

COMPRESS Example To create an index with the COMPRESS clause, you might issue the following statement:

```
CREATE INDEX emp_idx2 ON emp(job, ename) COMPRESS 1;
```

The index will compress repeated occurrences of job column values.

NOLOGGING Example To quickly create an index in parallel on a table that was created using a fast parallel load (so all rows are already sorted), you might issue the following statement. (Oracle will choose the appropriate degree of parallelism.)

```
CREATE INDEX i_loc
   ON big_table (akey)
   NOSORT
   NOLOGGING
   PARALLEL;
```

Cluster Index Example To create an index for the employee cluster, issue the following statement:

```
CREATE INDEX ic_emp ON CLUSTER employee;
```

No index columns are specified, because the index is automatically built on all the columns of the cluster key. For cluster indexes, all rows are indexed.

NULL Example Consider the following statement:

```
SELECT ename FROM emp WHERE comm IS NULL;
```

The above query does not use an index created on the comm column unless it is a bitmap index.

Function-Based Index Example The following statements creates a function-based index on the emp table based on an uppercase evaluation of the ename column:

```
CREATE INDEX emp_i ON emp (UPPER(ename));
```

To ensure that Oracle will use the index rather than performing a full table scan, be sure that the value of the function is not null in subsequent queries. For example, the statement

```
SELECT * FROM emp WHERE UPPER(ename) IS NOT NULL
   ORDER BY UPPER(ename);
```

is guaranteed to use the index, but without the WHERE clause, Oracle may perform a full table scan.

In the next statements showing index creation and subsequent query, Oracle will use index emp_fi even though the columns are in reverse order in the query:

```
CREATE INDEX emp_fi ON emp(cola + colb);

SELECT * FROM emp WHERE colb + cola > 500;
```

Function-based Index on Type Method Example This example entails an object type rectangle containing two number attributes: length and width. The area() method computes the area of the rectangle.

```
CREATE TYPE rectangle AS OBJECT
( length NUMBER,
  width NUMBER,
  MEMBER FUNCTION area RETURN NUMBER DETERMINISTIC
);

CREATE OR REPLACE TYPE BODY rectangle AS
  MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
   RETURN (length*width);
  END;
END;
```

Now, if you create a table rectab of type rectangle, you can create a function-based index on the area() method as follows:

```
CREATE TABLE recttab OF rectangle;
CREATE INDEX area_idx ON recttab x (x.area());
```

You can use this index efficiently to evaluate a query of the form:

```
SELECT * FROM recttab x WHERE x.area() > 100;
```

Computing Statistics Example The following statement collects statistics on the nonpartitioned emp_indx index:

```
CREATE INDEX emp_indx ON emp(empno) COMPUTE STATISTICS;
```

The type of statistics collected depends on the type of index you are creating.

> **See Also:** *Oracle8i Concepts*

Partitioned Index Example The following statement creates a global prefixed index stock_ix on table stock_xactions with two partitions, one for each half of the alphabet. The index partition names are system generated:

```
CREATE INDEX stock_ix ON stock_xactions
  (stock_symbol, stock_series)
   GLOBAL PARTITION BY RANGE (stock_symbol)
     (PARTITION VALUES LESS THAN ('N') TABLESPACE ts3,
      PARTITION VALUES LESS THAN (MAXVALUE) TABLESPACE ts4);
```

Index on Hash-Partitioned Table Example. This statement creates a local index on the `item` column of the `sales` table. The `STORE IN` clause immediately following `LOCAL` indicates that `sales` is hash partitioned. Oracle will distribute the hash partitions between the `tbs1` and `tbs2` tablespaces:

```
CREATE INDEX sales_idx ON sales(item) LOCAL
   STORE IN (tbs1, tbs2);
```

Index on Composite-Partitioned Table Example. This statement creates a local index on the `sales` table, which is composite-partitioned. The `STORAGE` clause specifies default storage attributes for the index. The `STORE IN` clause specifies one or more default tablespaces for the index subpartitions. However, this default is overridden for the four subpartitions of partition `q3_1977`, because separate `TABLESPACE` is specified.

```
CREATE INDEX sales_idx ON sales(sale_date, item)
   STORAGE (INITIAL 1M, MAXEXTENTS UNLIMITED)
   LOCAL
   STORE IN (tbs1, tbs2, tbs3, tbs4, tbs5)
   (PARTITION q1_1997, PARTITION q2_1997,
    PARTITION q3_1997
      (SUBPARTITION q3_1997_s1 TABLESPACE ts2,
       SUBPARTITION q3_1997_s2 TABLESPACE ts4,
       SUBPARTITION q3_1997_s3 TABLESPACE ts6,
       SUBPARTITION q3_1997_s4 TABLESPACE ts8),
    PARTITION q4_1997,
    PARTITION q1_1998);
```

Bitmap Index Example To create a bitmap partitioned index on a table with four partitions, issue the following statement:

```
CREATE BITMAP INDEX partno_ix
   ON lineitem(partno)
   TABLESPACE ts1
   LOCAL (PARTITION quarter1 TABLESPACE ts2,
          PARTITION quarter2 STORAGE (INITIAL 10K NEXT 2K),
          PARTITION quarter3 TABLESPACE ts2,
          PARTITION quarter4);
```

Index on Nested Table Example In the following example, UNIQUE index uniq_proj_indx is created on storage table nested_project_table. Including pseudocolumn nested_table_id ensures distinct rows in nested table column projs_managed:

```
CREATE TYPE proj_type AS OBJECT
   (proj_num NUMBER, proj_name VARCHAR2(20));
CREATE TYPE proj_table_type AS TABLE OF proj_type;
CREATE TABLE employee ( emp_num NUMBER, emp_name CHAR(31),
   projs_managed proj_table_type )
   NESTED TABLE projs_managed STORE AS nested_project_table;
CREATE UNIQUE INDEX uniq_proj_indx
   ON nested_project_table ( NESTED_TABLE_ID, proj_num);
```

CREATE INDEXTYPE

Purpose

Use the CREATE INDEXTYPE statement to create an **indextype**, which is an object that specifies the routines that manage a domain (application-specific) index. Indextypes reside in the same namespace as tables, views, and other schema objects. This statement binds the indextype name to an implementation type, which in turn specifies and refers to user-defined index functions and procedures that implement the indextype.

> **See Also:** *Oracle8i Data Cartridge Developer's Guide* and *Oracle8i Concepts* for more information on implementing indextypes

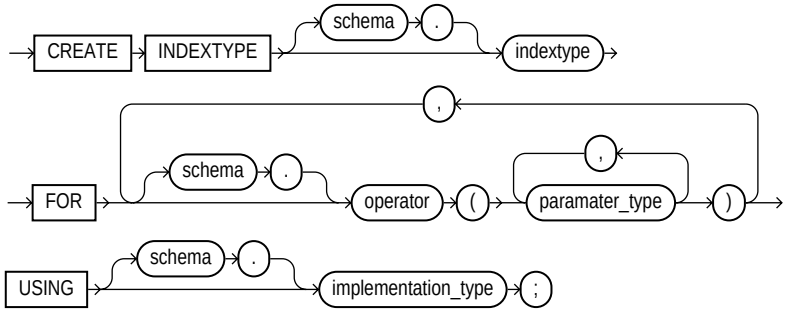
Prerequisites

To create an indextype in your own schema, you must have the CREATE INDEXTYPE system privilege. To create an indextype in another schema, you must have CREATE ANY INDEXTYPE system privilege. In either case, you must have the EXECUTE object privilege on the implementation type and the supported operators.

An indextype supports one or more operators, so before creating an indextype, you should first design the operator or operators to be supported and provide functional implementation for those operators.

> **See Also:** CREATE OPERATOR on page 9-115

Syntax



Keywords and Parameters

schema

Specify the name of the schema in which the indextype resides. If you omit *schema*, Oracle creates the indextype in your own schema.

indextype

Specify the name of the indextype to be created.

FOR

Use the FOR clause to specify the list of operators supported by the indextype.

schema	Specify the schema containing the operator. If you omit *schema*, Oracle assumes the operator is in your own schema.
operator	Specify the name of the operator supported by the indextype. All the operators listed in this clause should be valid operators.
parameter_type	Lists the types of parameters to the operator.

USING

The USING clause lets you specify the type that provides the implementation for the new indextype.

implementation_type	Specify the name of the type that implements the appropriate Oracle Data Cartridge interface (ODCI). ■ You must specify a valid type that implements the routines in the ODCI interface. ■ The implementation type must reside in the same schema as the indextype. **See Also:** *Oracle8i Data Cartridge Developer's Guide* for additional information on this interface

Example

CREATE INDEXTYPE Example The following statement creates an indextype named TextIndexType and specifies the `contains` operator that is supported by the indextype and the TextIndexMethods type that implements the index interface:

```
CREATE INDEXTYPE TextIndexType
   FOR contains (VARCHAR2, VARCHAR2)
   USING TextIndexMethods;
```

CREATE JAVA

Purpose

Use the CREATE JAVA statement to create a schema object containing a Java source, class, or resource.

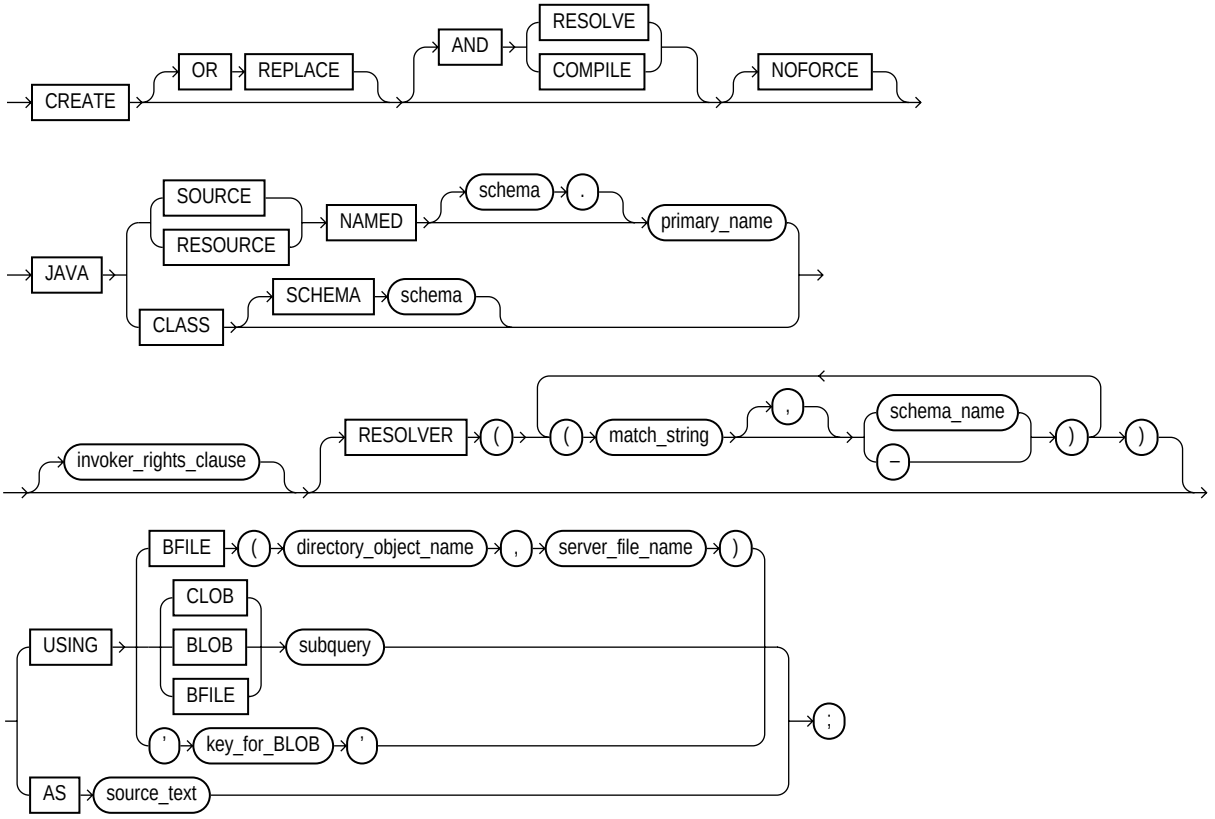
See Also:

- *Oracle8i Java Developer's Guide* for Java concepts
- *Oracle8i Java Stored Procedures Developer's Guide* for Java stored procedures
- *Oracle8i SQLJ Developer's Guide and Reference* for SQLJ
- *Oracle8i JDBC Developer's Guide and Reference* for JDBC
- *Oracle8i Enterprise JavaBeans Developer's Guide and Reference* for CORBA and EJB

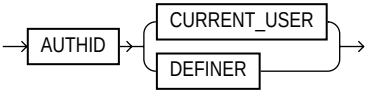
Prerequisites

To create or replace a schema object containing a Java source, class, or resource in your own schema, you must have CREATE PROCEDURE system privilege. To create such a schema object in another user's schema, you must have CREATE ANY PROCEDURE system privilege. To replace such a schema object in another user's schema, you must also have ALTER ANY PROCEDURE system privilege.

Syntax



invoker_rights_clause::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the schema object containing the Java class, source, or resource if it already exists. Use this clause to change the definition of an existing

object without dropping, re-creating, and regranting object privileges previously granted.

If you redefine a Java schema object and specify RESOLVE or COMPILE, Oracle recompiles or resolves the object. Whether or not the resolution or compilation is successful, Oracle invalidates classes that reference the Java schema object.

Users who had previously been granted privileges on a redefined function can still access the function without being regranted the privileges.

See Also: ALTER JAVA on page 7-58 for additional information

RESOLVE | COMPILE

RESOLVE and COMPILE are synonymous keywords. They specify that Oracle should attempt to resolve the Java schema object that is created if this statement succeeds.

- When applied to a class, resolution of referenced names to other class schema objects occurs.
- When applied to a source, source compilation occurs.

Restriction: You cannot specify this clause for a Java resource.

NOFORCE

Specify NOFORCE to roll back the results of this CREATE command if you have specified either RESOLVE or COMPILE, and the resolution or compilation fails. If you do not specify this option, Oracle takes no action if the resolution or compilation fails (that is, the created schema object remains).

JAVA SOURCE

Specify JAVA SOURCE to load a Java source file.

JAVA CLASS

Specify JAVA CLASS to load a Java class file.

JAVA RESOURCE

Specify JAVA RESOURCE to load a Java resource file.

NAMED

The NAMED clause is **required** for a Java source or resource. The *primary_name* must be enclosed in double quotation marks.

- For a Java source, this clause specifies the name of the schema object in which the source code is held. A successful CREATE JAVA SOURCE statement will also create additional schema objects to hold each of the Java classes defined by the source.
- For a Java resource, this clause specifies the name of the schema object to hold the Java resource.

Use double quotation marks to preserve lower- or mixed-case *primary_name*.

If you do not specify *schema*, Oracle creates the object in your own schema.

Restrictions:

- You cannot specify NAMED for a Java class.
- The *primary_name* cannot contain a database link.

SCHEMA *schema*

The SCHEMA clause applies only to a Java class. This optional clause specifies the schema in which the object containing the Java file will reside. If you do not specify this clause, Oracle creates the object in your own schema.

invoker_rights_clause

Use the *invoker_rights_clause* to indicate whether the methods of the class execute with the privileges and in the schema of the user who owns the class or with the privileges and in the schema of CURRENT_USER.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

AUTHID CURRENT_USER — CURRENT_USER indicates that the methods of the class execute with the privileges of CURRENT_USER. This clause is the default and creates an "invoker-rights class."

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the methods reside.

AUTHID DEFINER — DEFINER indicates that the methods of the class execute with the privileges of the owner of the schema in which the class resides, and that external names resolve in the schema where the class resides.

See Also:

- *Oracle8i Java Stored Procedures Developer's Guide*
- *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined

RESOLVER

The RESOLVER clause lets you specify a mapping of the fully qualified Java name to a Java schema object, where

- *match_string* is either a fully qualified Java name, a wildcard that can match such a Java name, or a wildcard that can match any name.
- *schema_name* designates a schema to be searched for the corresponding Java schema object.
- A dash (-) as an alternative to *schema_name* indicates that if *match_string* matches a valid Java name, Oracle can leave the name unresolved. The resolution succeeds, but the name cannot be used at run time by the class.

This mapping is stored with the definition of the schema objects created in this command for use in later resolutions (either implicit or in explicit ALTER ... RESOLVE statements).

USING

The USING clause determines a sequence of character (CLOB or BFILE) or binary (BLOB or BFILE) data for the Java class or resource. Oracle uses the sequence of characters to define one file for a Java class or resource, or one source file and one or more derived classes for a Java source.

BFILE — Specify the directory and filename of a previously created file on the operating system (*directory_object_name*) and server file (*server_file_name*) containing the sequence. BFILE is usually interpreted as a character sequence by CREATE JAVA SOURCE and as a binary sequence by CREATE JAVA CLASS or CREATE JAVA RESOURCE.

CLOB/BLOB/BFILE *subquery* — Specify a query that selects a single row and column of the type specified (CLOB, BLOB, or BFILE). The value of the column makes up the sequence of characters.

> **Note:** The USING clause implicitly supplies the keyword SELECT. Therefore, omit this keyword from the subquery.

key_for_BLOB — The *key_for_BLOB* clause supplies the following implicit query:

```
SELECT LOB FROM CREATE$JAVA$LOB$TABLE
   WHERE NAME = 'key_for_BLOB';
```

Restriction: To use this case, the table CREATE$JAVA$LOB$TABLE must exist in the current schema and must have a column LOB of type BLOB and a column NAME of type VARCHAR2.

AS *source_text*

Specify a sequence of characters for a Java or SQLJ source.

Examples

Java Class Example The following statement creates a schema object containing a Java class using the name found in a Java binary file:

```
CREATE JAVA CLASS USING BFILE (bfile_dir, 'Agent.class');
```

This example assumes the directory object bfile_dir, which points to the operating system directory containing the Java class Agent.class, already exists. In this example, the name of the class determines the name of the Java class schema object.

Java Source Example The following statement creates a Java source schema object:

```
CREATE JAVA SOURCE NAMED "Hello" AS
   public class Hello {
      public static String hello() {
         return "Hello World";   } };
```

Java Resource Example The following statement creates a Java resource schema object named apptext from a bfile:

```
CREATE JAVA RESOURCE NAMED "appText"
   USING BFILE (bfile_dir, 'textBundle.dat');
```

CREATE LIBRARY

Purpose

Use the CREATE LIBRARY statement to create a schema object associated with an operating-system shared library. The name of this schema object can then be used in the *call_spec* of CREATE FUNCTION or CREATE PROCEDURE statements, or when declaring a function or procedure in a package or type, so that SQL and PL/SQL can call to third-generation-language (3GL) functions and procedures.

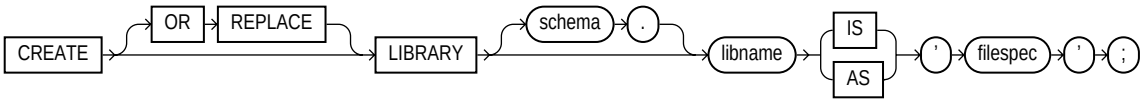
> **See Also:** CREATE FUNCTION on page 9-43 and *PL/SQL User's Guide and Reference* for more information on functions and procedures

Prerequisites

To create a library in your own schema, you must have the CREATE LIBRARY system privilege. To create a library in another user's schema, you must have the CREATE ANY LIBRARY system privilege. To use the procedures and functions stored in the library, you must have EXECUTE object privileges on the library.

The CREATE LIBRARY statement is valid only on platforms that support shared libraries and dynamic linking.

Syntax



filespec: See filespec on page 11-27.

Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the library if it already exists. Use this clause to change the definition of an existing library without dropping, re-creating, and regranting schema object privileges granted on it.

Users who had previously been granted privileges on a redefined library can still access the library without being regranted the privileges.

libname

Specify the name you with to create to represent this library when declaring a function or procedure with a *call_spec*.

'*filespec*'

Specify a string literal, enclosed in single quotes. This string should be the path or filename your operating system recognizes as naming the shared library.

The '*filespec*' is not interpreted during execution of the CREATE LIBRARY statement. The existence of the library file is not checked until an attempt is made to execute a routine from it.

Examples

CREATE LIBRARY Examples The following statement creates library ext_lib:

```
CREATE LIBRARY ext_lib AS '/OR/lib/ext_lib.so';
```

The following statement re-creates library ext_lib:

```
CREATE OR REPLACE LIBRARY ext_lib IS '/OR/newlib/ext_lib.so';
```

CREATE MATERIALIZED VIEW

Purpose

Use the CREATE MATERIALIZED VIEW statement to create a **materialized view**. The terms **snapshot** and **materialized view** are synonymous in Oracle documentation. This reference uses "materialized view" for consistency. Both refer to a database object that contains the results of a query of one or more tables.

The tables in the query are called **master tables** (a replication term) or **detail tables** (a data warehouse term). This reference uses "master tables" for consistency. The databases containing the master tables are called the **master databases**.

For replication purposes, materialized views allow you to maintain copies of remote data on your local node. The copies can be updatable with the Advanced Replication feature and are read-only without this feature. You can select data from a materialized view as you would from a table or view. In replication environments, the materialized views commonly created are **primary key**, **rowid**, and **subquery** materialized views.

For data warehousing purposes, the materialized views commonly created are **materialized aggregate views**, **single-table materialized aggregate views**, and **materialized join views**. All three types of materialized views can be used by query rewrite, an optimization technique that transforms a user request written in terms of master tables into a semantically equivalent request that includes one or more materialized view. In a data warehousing environment, all master tables must be local.

See Also:

- *Oracle8i Replication* for information on the types of materialized views used to support replication
- *Oracle8i Data Warehousing Guide* for information on the types of materialized views used to support data warehousing

Prerequisites

The privileges required to create a materialized view should be granted directly.

To create a materialized view **in your own schema**:

- You must have been granted either the CREATE MATERIALIZED VIEW or CREATE SNAPSHOT system privilege **an**d either the CREATE TABLE or CREATE ANY TABLE system privilege.
- You must also have access to any master tables of the materialized view that you do not own, either through a SELECT object privilege on each of the tables or through the SELECT ANY TABLE system privilege.

To create a materialized view **in another user's schema**:

- You must have the CREATE ANY MATERIALIZED VIEW or CREATE ANY SNAPSHOT system privilege **and** access to any master tables of the materialized view that you do not own, either through a SELECT object privilege on each of the tables or through the SELECT ANY TABLE system privilege.
- The owner or the materialized view must have the CREATE TABLE system privilege. The owner must also have access to any master tables of the materialized view that the schema owner does not own **and** to any materialized view logs defined on those master tables, either through a SELECT object privilege on each of the tables or through the SELECT ANY TABLE system privilege.

To create the materialized view **with query rewrite enabled**, in addition to the preceding privileges:

- The owner of the master tables must have the QUERY REWRITE system privilege.
- If you are not the owner of the master tables, you must have the GLOBAL QUERY REWRITE system privilege.
- If the schema owner does not own the master tables, then the schema owner must have the GLOBAL QUERY REWRITE privilege.

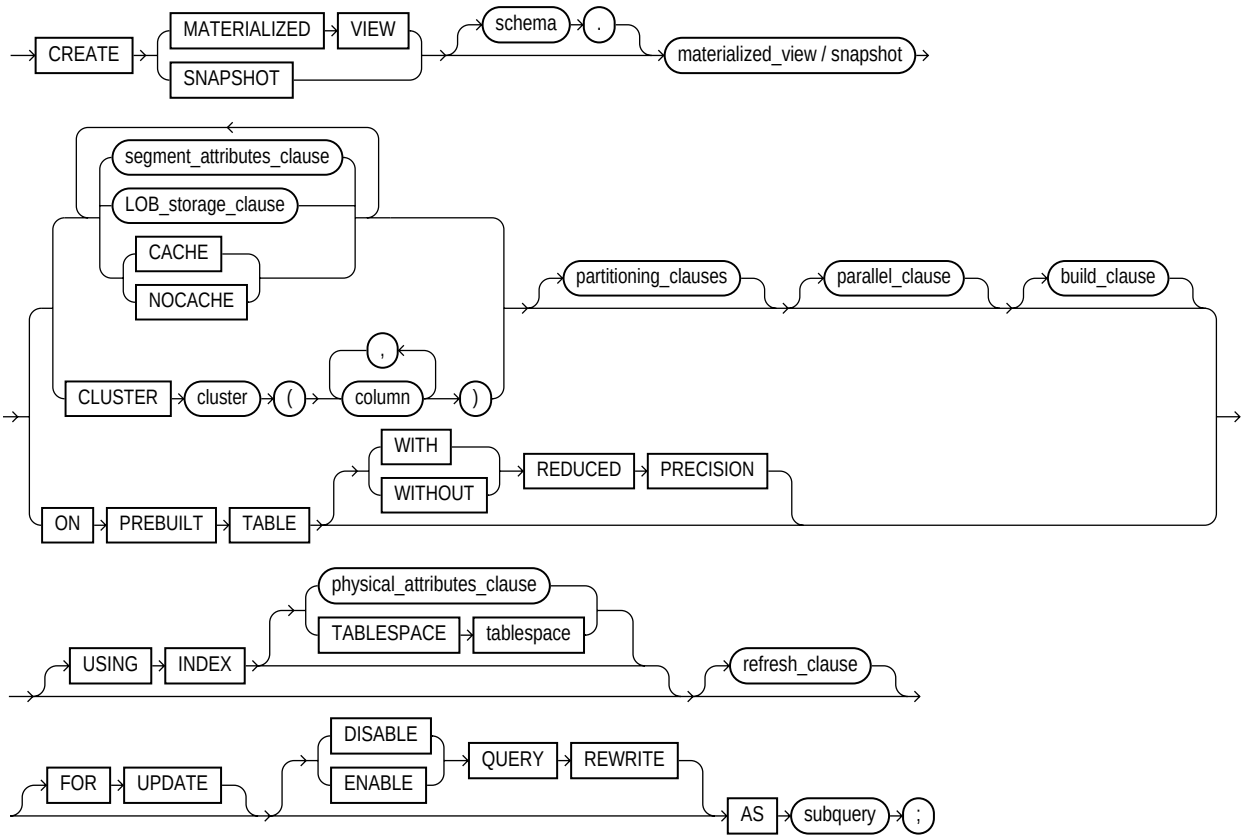
The user whose schema contains the materialized view must have sufficient quota in the target tablespace to store the materialized view's master table and index, or must have the UNLIMITED TABLESPACE system privilege.

When you create a materialized view, Oracle creates one internal table and at least one index, and may create one view, all in the schema of the materialized view. Oracle uses these objects to maintain the materialized view's data. You must have the privileges necessary to create these objects.

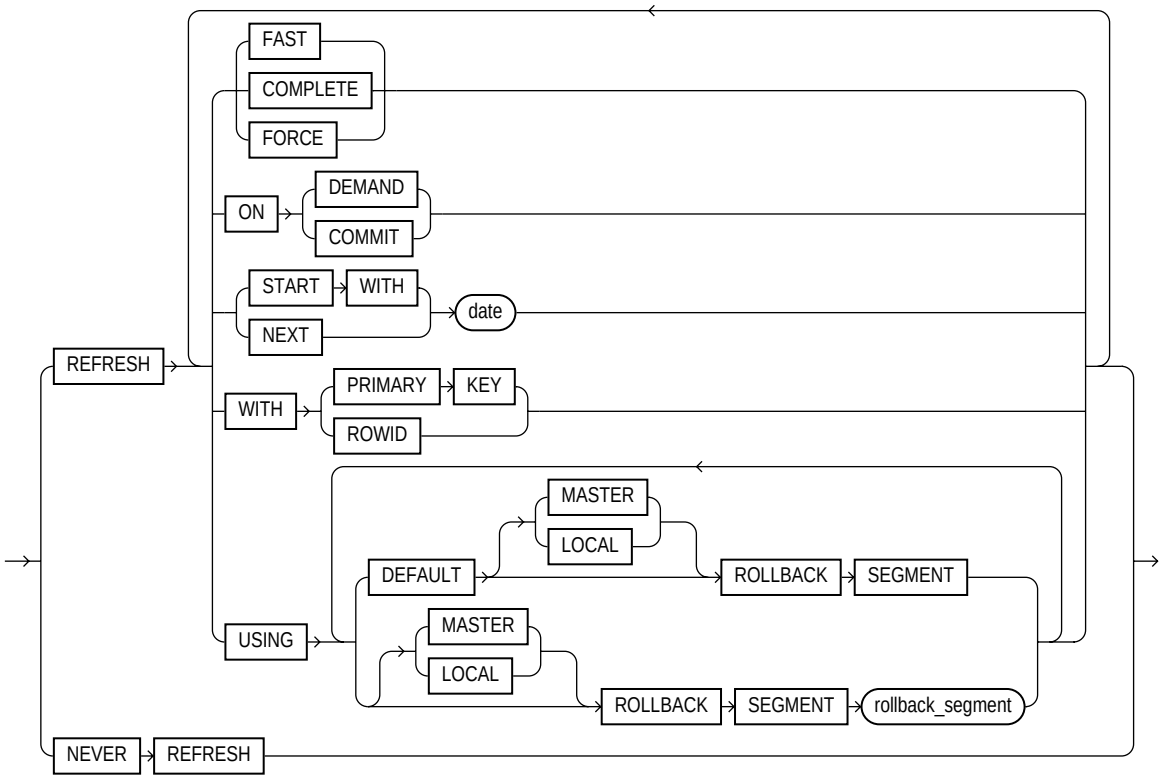
See Also:

- CREATE TABLE on page 10-7, CREATE VIEW on page 10-105, and CREATE INDEX on page 9-52 for information on these privileges
- *Oracle8i Replication* for information about the prerequisites that apply to creating replication materialized views
- *Oracle8i Data Warehousing Guide* for information about the prerequisites that apply to creating data warehousing materialized views

Syntax

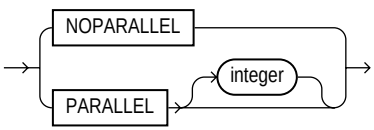


refresh_clause::=

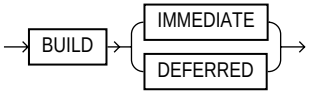


segment_attributes_clause: See CREATE TABLE on page 10-7.

parallel_clause::=



build_clause::=



subquery: See SELECT and subquery on page 11-88.

LOB_storage_clause: See CREATE TABLE on page 10-7.

partitioning_clauses: See CREATE TABLE on page 10-7.

Keywords and Parameters

schema

Specify the schema to contain the materialized view. If you omit *schema*, Oracle creates the materialized view in your schema.

materialized_view

Specify the name of the materialized view to be created. Oracle generates names for the table and indexes used to maintain the materialized view by adding a prefix or suffix to the materialized view name.

segment_attributes_clause

Use the *segment_attributes_clause* to establish values for the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters (or, when used in the USING INDEX clause, for the INITRANS and MAXTRANS parameters only), the storage characteristics for the materialized view, to assign a tablespace, and to specify whether logging is to occur.

See Also:

- CREATE TABLE on page 10-7 for information on the PCTFREE, PCTUSED, INITRANS, and MAXTRANS, TABLESPACE, and LOGGING | NOLOGGING parameters
- *storage_clause* on page 11-129 for information about storage characteristics

TABLESPACE

Specify the tablespace in which the materialized view is to be created. If you omit this clause, Oracle creates the materialized view in the default tablespace of the owner of the materialized view's schema.

LOB_storage_clause

The *LOB_storage_clause* lets you specify the LOB storage characteristics.

> **See Also:** CREATE TABLE on page 10-7 for detailed information about specifying the parameters of this clause

LOGGING | NOLOGGING

Specify LOGGING or NOLOGGING to establish the logging characteristics for the materialized view.

> **See Also:** CREATE TABLE on page 10-7 for a description of logging characteristics

CACHE | NOCACHE

For data that will be accessed frequently, CACHE specifies that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. NOCACHE specifies that the blocks are placed at the least recently used end of the LRU list.

> **Note:** NOCACHE has no effect on materialized views for which you specify KEEP in the *storage_clause*.

> **See Also:** CREATE TABLE on page 10-7 for information about specifying CACHE or NOCACHE

CLUSTER

Use the CLUSTER clause to create the materialized view as part of the specified cluster. A clustered materialized view uses the cluster's space allocation. Therefore, do not use the *physical_attributes_clause* or the TABLESPACE clause with the CLUSTER clause.

partitioning_clauses

The *partitioning_clauses* let you specify that the materialized view is partitioned on specified ranges of values or on a hash function. Partitioning of materialized views is the same as partitioning tables.

> **See Also:** CREATE TABLE on page 10-7

parallel_clause

The *parallel_clause* lets you indicate whether parallel operations will be supported for the materialized view and sets the default degree of parallelism for queries and DML on the materialized view after creation.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL — Specify NOPARALLEL for serial execution. This is the default.

PARALLEL — Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 10-41

build_clause

The *build_clause* lets you specify when to populate the materialized view.

`IMMEDIATE` Specify `IMMEDIATE` to indicate that the materialized view is populated immediately. This is the default.

`DEFERRED` Specify `DEFERRED` to indicate that the materialized view will be populated by the next `REFRESH` operation. The first (deferred) refresh must always be a complete refresh. Until then, the materialized view has a staleness value of `UNUSABLE`, so it cannot be used for query rewrite.

ON PREBUILT TABLE

The `ON PREBUILT TABLE` clause lets you register an existing table as a preinitialized materialized view. This is particularly useful for registering large materialized views in a data warehousing environment. The table must have the same name and be in the same schema as the resulting materialized view.

If the materialized view is dropped, the preexisting table reverts to its identity as a table.

Caution: This clause assumes that the table object reflects the materialization of a subquery. Oracle Corporation strongly recommends that you ensure that this assumption is true in order to ensure that the materialized view correctly reflects the data in its master tables.

Restrictions:

- Each column alias in *subquery* must correspond to a column in *table_name*, and corresponding columns must have matching datatypes.
- If you specify this clause, you cannot specify a `NOT NULL` constraint for any column that is unmanaged (that is, not referenced in *subquery*) unless you also specify a default value for that column.

`WITH REDUCED PRECISION` Specify `WITH REDUCED PRECISION` to authorize the loss of precision that will result if the precision of the table or materialized view columns do not exactly match the precision returned by *subquery*.

WITHOUT REDUCED PRECISION — Specify WITHOUT REDUCED PRECISION to require that the precision of the table or materialized view columns match exactly the precision returned by *subquery*, or the create operation will fail. This is the default.

USING INDEX

The USING INDEX clause lets you establish the value of INITRANS, MAXTRANS, and STORAGE parameters for the index Oracle uses to maintain the materialized view's data. If USING INDEX is not specified, then default values are used for the index.

Restriction: You cannot specify the PCTUSED or PCTFREE parameters in this clause.

refresh_clause

Use the *refresh_clause* to specify the default methods, modes, and times for Oracle to refresh the materialized view. If a materialized view's master tables are modified, the data in a materialized view must be updated to make the materialized view accurately reflect the data currently in its master tables. This clause lets you schedule the times and specify the method and mode for Oracle to refresh the materialized view.

Note: This clause only sets the default refresh options. For instructions on actually implementing the refresh, refer to *Oracle8i Replication* and *Oracle8i Data Warehousing Guide*.

FAST — Specify FAST to indicate the incremental refresh method, which performs the refresh according to the changes that have occurred to the master tables. The changes are stored either in the materialized view log associated with the master table (for conventional DML changes) or in the direct loader log (for direct-load INSERT operations).

You can create a materialized aggregate view even if you have not yet created materialized view logs for the underlying master tables. However, if you are creating any other type of materialized view, the CREATE statement will fail unless those materialized view logs already exist. (Oracle creates the direct loader log automatically when a direct-load INSERT takes place. No user intervention is needed.)

After create time, Oracle will perform the fast refresh for conventional DML only if the appropriate materialized view logs exist.

For both conventional DML changes and for direct-path loads, other conditions may restrict the eligibility of a materialized view for fast refresh.

Materialized views are not eligible for fast refresh if the defining query contains an analytic function.

See Also:

- *Oracle8i Replication* for restrictions on fast refresh in replication environments

- *Oracle8i Data Warehousing Guide* for restrictions on fast refresh in data warehouse environments

- "Analytic Functions" on page 4-8

`COMPLETE` Specify `COMPLETE` to indicate the complete refresh method, which is implemented by executing the materialized view's defining query. If you request a complete refresh, Oracle performs a complete refresh even if a fast refresh is possible.

`FORCE` Specify `FORCE` to indicate that when a refresh occurs, Oracle will perform a fast refresh if one is possible or a complete refresh otherwise. If you do not specify a refresh method (`FAST`, `COMPLETE`, or `FORCE`), `FORCE` is the default.

`ON COMMIT` Specify `ON COMMIT` to indicate that a fast refresh is to occur whenever Oracle commits a transaction that operates on a master table of the materialized view.

Restriction: This clause is supported only for materialized join views and single-table materialized aggregate views.

See Also: *Oracle8i Replication* and *Oracle8i Data Warehousing Guide*

ON DEMAND — Specify ON DEMAND to indicate that the materialized view will be refreshed on demand by calling one of the three DBMS_MVIEW refresh procedures. If you omit both ON COMMIT and ON DEMAND, ON DEMAND is the default.

See Also:

- *Oracle8i Supplied PL/SQL Packages Reference* for information on these procedures

- *Oracle8i Data Warehousing Guide* on the types of materialized views you can create by specifying REFRESH ON DEMAND

If you specify ON COMMIT or ON DEMAND, you cannot also specify START WITH or NEXT.

START WITH — Specify a date expression for the first automatic refresh time.

NEXT — Specify a date expression for calculating the interval between automatic refreshes.

Both the START WITH and NEXT values must evaluate to a time in the future. If you omit the START WITH value, Oracle determines the first automatic refresh time by evaluating the NEXT expression with respect to the creation time of the materialized view. If you specify a START WITH value but omit the NEXT value, Oracle refreshes the materialized view only once. If you omit both the START WITH and NEXT values, or if you omit the *refresh_clause* entirely, Oracle does not automatically refresh the materialized view.

WITH PRIMARY KEY — Specify WITH PRIMARY KEY to indicate that a primary key materialized view is to be created. This is the default, and should be used in all cases except those described for WITH ROWID. Primary key materialized views allow materialized view master tables to be reorganized without affecting the materialized view's ability to continue to fast refresh. The master table must contain an enabled primary key constraint.

See Also: *Oracle8i Replication* for detailed information about primary key materialized views

WITH ROWID — Specify WITH ROWID to indicate that a rowid materialized view is to be created. Rowid materialized views provide compatibility with master tables in releases of Oracle prior to 8.0.

You can also use rowid materialized views if the materialized view does not include all primary key columns of the master tables. Rowid materialized views must be based on a single remote table and cannot contain any of the following:

- Distinct or aggregate functions
- GROUP BY or CONNECT BY clauses
- Subqueries
- Joins
- Set operations

Rowid materialized views cannot be fast refreshed after a master table reorganization until a complete refresh has been performed.

USING ROLLBACK SEGMENT *rollback_segment*

Specify the remote rollback segment to be used during materialized view refresh, where *rollback_segment* is the name of the rollback segment to be used.

- DEFAULT specifies that Oracle will choose automatically which rollback segment to use. If you specify DEFAULT, you cannot specify *rollback_segment*.

  DEFAULT is most useful when modifying a materialized view.

  See Also: ALTER MATERIALIZED VIEW on page 7-61
- MASTER specifies the remote rollback segment to be used at the remote master site for the individual materialized view.
- LOCAL specifies the remote rollback segment to be used for the local refresh group that contains the materialized view.

  See Also: *Oracle8i Replication* for information on specifying the local materialized view rollback segment using the DBMS_REFRESH package.

If you do not specify MASTER or LOCAL, Oracle uses LOCAL by default. If you do not specify *rollback_segment*, Oracle automatically chooses the rollback segment to be used.

The master rollback segment is stored on a per-materialized-view basis and is validated during materialized view creation and refresh. If the materialized view is complex, the master rollback segment, if specified, is ignored.

NEVER REFRESH — Specify NEVER REFRESH to prevent the materialized view from being refreshed with any Oracle refresh mechanism or procedure. If you issue a REFRESH statement on the materialized view, Oracle returns an error.

FOR UPDATE

Specify FOR UPDATE to allow a subquery, primary key, or rowid materialized view to be updated. When used in conjunction with Advanced Replication, these updates will be propagated to the master.

See Also: *Oracle8i Replication*

QUERY REWRITE

The QUERY REWRITE clause lets you specify whether the materialized view is eligible to be used for query rewrite.

ENABLE — Specify ENABLE to enable the materialized view for query rewrite.

See Also: *Oracle8i Data Warehousing Guide* for more information on query rewrite

Notes:

- Query rewrite is disabled by default, so you must specify this clause to make materialized views eligible for query rewrite.
- Be sure to analyze the materialized view after you create it. Oracle needs the statistics generated by the ANALYZE operation to optimize query rewrite.

Restrictions:

- You can enable query rewrite only if all user-defined functions in the materialized view are `DETERMINISTIC`.
- You can enable query rewrite only if expressions in the statement are repeatable. For example, you cannot include `CURRENT_TIME` or `USER`, sequence values (such as the `CURRVAL` or `NEXTVAL` pseudocolumns), or the `SAMPLE` clause (which may sample different rows as the contents of the materialized view change).

  See Also: CREATE FUNCTION on page 9-43 and *Oracle8i Data Warehousing Guide*

`DISABLE` Specify `DISABLE` to indicate that the materialized view is not eligible for use by query rewrite. However, a disabled materialized view can be refreshed.

AS *subquery*

Specify the materialized view's defining query. When you create the materialized view, Oracle executes this query and places the results in the materialized view. This query is any valid SQL query. However, not all queries are fast refreshable, nor are all queries eligible for query rewrite.

Notes on the materialized view subquery:

- Oracle does not execute the query immediately if you specify `BUILD DEFERRED`.
- Oracle recommends that you qualify each table and view in the `FROM` clause of the materialized view query with the schema containing it.

  See Also: the AS *subquery* clause of CREATE TABLE on page 10-7 for some additional caveats

Restrictions on the materialized view subquery:

- A materialized view query can select from tables or views owned by the user `SYS`, but you cannot enable `QUERY REWRITE` on such a materialized view.
- You cannot refer to a user-defined type anywhere in the materialized view query.

- Materialized join views and materialized aggregate views with a GROUP BY clause cannot select from an index-organized table.
- Materialized views cannot contain columns of datatype LONG.
- If the subquery refers to a temporary table, you cannot create a materialized view log for this materialized view, nor can you specify the QUERY REWRITE clause of CREATE MATERIALIZED VIEW or ALTER MATERIALIZED VIEW.
- If the FROM list of the materialized view references another materialized view, you must control the refresh order of the materialized views manually. That is, you must refresh the materialized view depended upon and then the dependent materialized view in order to maintain freshness.

If you are creating a materialized view enabled for query rewrite:

- The subquery cannot contain (either directly or through a view) references to ROWNUM, USER, SYSDATE, remote tables, sequences, or PL/SQL functions that write or read database or package state.
- The materialized view and the master tables of the materialized view must be local.

If you want the materialized view to be eligible for fast refresh using a materialized view log, some additional restrictions may apply.

See Also:

- *Oracle8i Data Warehousing Guide* for more information on restrictions relating to data warehousing
- *Oracle8i Replication f*or more information on restrictions relating to replication

Examples

Materialized Aggregate View Examples The following statement creates and populates a materialized aggregate view and specifies the default refresh method, mode, and time:

```
CREATE MATERIALIZED VIEW mv1 REFRESH FAST ON COMMIT
   BUILD IMMEDIATE
   AS SELECT t.month, p.prod_name, SUM(f.sales) AS sum_sales
      FROM time t, product p, fact f
      WHERE f.curDate = t.curDate AND f.item = p.item
      GROUP BY t.month, p.prod_name;
```

The following statement creates and populates the materialized aggregate view `sales_by_month_by_state`. The materialized view will be populated with data as soon as the statement executes successfully. By default, subsequent refreshes will be accomplished by reexecuting the materialized view's query:

```
CREATE MATERIALIZED VIEW sales_by_month_by_state
   TABLESPACE my_ts PARALLEL (10)
   ENABLE QUERY REWRITE
   BUILD IMMEDIATE
   REFRESH COMPLETE
   AS SELECT t.month, g.state, SUM(f.sales) AS sum_sales
      FROM fact f, time t, geog g
      WHERE f.cur_date = t.cur_date AND f.city_id = g.city_id
      GROUP BY month, state;
```

Prebuilt Materialized View Example The following statement creates a materialized aggregate view for the preexisting summary table, `sales_sum_table`:

```
CREATE TABLE sales_sum_table
   (month DATE, state VARCHAR2(25), sales NUMBER);

CREATE MATERIALIZED VIEW sales_sum_table
   ON PREBUILT TABLE
   ENABLE QUERY REWRITE
   AS SELECT t.month, g.state, SUM(f.sales) AS sum_sales
      FROM fact f, time t, geog g
      WHERE f.cur_date = t.cur_date AND f.city_id = g.city_id
      GROUP BY month, state;
```

In this example, the materialized view has the same name as the prebuilt table and also has the same number of columns with the same datatypes as the prebuilt table.

Materialized Join View Example The following statement creates the materialized join view `mjv`:

```
CREATE MATERIALIZED VIEW mjv
   REFRESH FAST
      AS SELECT l.rowid as l_rid, l.pk, l.ofk, l.c1, l.c2,
         o.rowid as o_rid, o.pk, o.cfk, o.c1, o.c2,
         c.rowid as c_rid, c.pd, c.c1, c.c2
      FROM l, o, c
      WHERE l.ofk = o.pk(+) AND o.ofk = c.pk(+);
```

Subquery Materialized View Example The following statement creates a subquery materialized view based on the orders and customers tables in the sales schema at a remote database:

```
CREATE MATERIALIZED VIEW sales.orders FOR UPDATE
   AS SELECT * FROM sales.orders@dbs1.acme.com o
   WHERE EXISTS
     (SELECT * FROM sales.customers@dbs1.acme.com c
      WHERE o.c_id = c.c_id);
```

Primary Key Example The following statement creates the primary-key materialized view human_genome:

```
CREATE MATERIALIZED VIEW human_genome
   REFRESH FAST START WITH SYSDATE NEXT  SYSDATE + 1/4096
   WITH PRIMARY KEY
   AS SELECT * FROM genome_catalog;
```

Rowid Example The following statement creates a rowid materialized view:

```
CREATE MATERIALIZED VIEW emp_data REFRESH WITH ROWID
   AS SELECT * FROM emp_table73;
```

Periodic Refresh Example The following statement creates the primary key materialized view emp_sf and populates it with data from scott's employee table in New York:

```
CREATE MATERIALIZED VIEW emp_sf
   PCTFREE 5 PCTUSED 60
   TABLESPACE users
   STORAGE (INITIAL 50K NEXT 50K)
   REFRESH FAST NEXT sysdate + 7
   AS SELECT * FROM scott.emp@ny;
```

The statement does not include a START WITH parameter, so Oracle determines the first automatic refresh time by evaluating the NEXT value using the current SYSDATE. Provided that a materialized view log currently exists for the employee table in New York, Oracle performs a fast refresh of the materialized view every 7 days, beginning 7 days after the materialized view is created.

Because the materialized view conforms to the conditions for fast refresh, Oracle will perform a fast refresh. The above statement also establishes storage characteristics that Oracle uses to maintain the materialized view.

Automatic Refresh Times Example The following statement creates the complex materialized view `all_emps` that queries the employee tables in Dallas and Baltimore:

```
CREATE MATERIALIZED VIEW all_emps
   PCTFREE 5 PCTUSED 60
   TABLESPACE users
   STORAGE INITIAL 50K NEXT 50K
   USING INDEX STORAGE (INITIAL 25K NEXT 25K)
   REFRESH START WITH ROUND(SYSDATE + 1) + 11/24
   NEXT NEXT_DAY(TRUNC(SYSDATE, 'MONDAY') + 15/24
   AS SELECT * FROM fran.emp@dallas
         UNION
      SELECT * FROM marco.emp@balt;
```

Oracle automatically refreshes this materialized view tomorrow at 11:00 am and subsequently every Monday at 3:00 pm. The default refresh method is FORCE. `all_emps` contains a UNION, which is not supported for fast refresh, so Oracle will automatically perform a complete refresh.

The above statement also establishes storage characteristics for both the materialized view and the index that Oracle uses to maintain it:

- The first *storage_clause* establishes the sizes of the first and second extents of the materialized view as 50 kilobytes each.
- The second *storage_clause* (appearing with the USING INDEX clause) establishes the sizes of the first and second extents of the index as 25 kilobytes each.

Rollback Segment Example The following statement creates the primary key materialized view `sales_emp` with rollback segment `master_seg` at the remote master and rollback segment `snap_seg` for the local refresh group that contains the materialized view:

```
CREATE MATERIALIZED VIEW sales_emp
   REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 7
   USING MASTER ROLLBACK SEGMENT master_seg
   LOCAL ROLLBACK SEGMENT snap_seg
   AS SELECT * FROM bar;
```

The following statement is incorrect and generates an error because it specifies a segment name with a DEFAULT rollback segment:

```
CREATE MATERIALIZED VIEW bogus
   REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 7
```

```
USING DEFAULT ROLLBACK SEGMENT snap_seg
AS SELECT * FROM faux;
```

CREATE MATERIALIZED VIEW LOG

Purpose

Use the CREATE MATERIALIZED VIEW LOG statement to create a **materialized view log**, which is a table associated with the master table of a materialized view. The terms **snapshot** and **materialized view** are synonymous. Both refer to a table that contains the results of a query of one or more tables, each of which may be located on the same or on a remote database.

When DML changes are made to the master table's data, Oracle stores rows describing those changes in the materialized view log and then uses the materialized view log to refresh materialized views based on the master table. This process is called a *fast refresh*. Without a materialized view log, Oracle must reexecute the materialized view query to refresh the materialized view. This process is called a *complete refresh*. Usually, a fast refresh takes less time than a complete refresh.

A materialized view log is located in the master database in the same schema as the master table. You need only a single materialized view log for a master table. Oracle can use this materialized view log to perform fast refreshes for all fast-refreshable materialized views based on the master table.

To fast refresh a materialized join view (a materialized view containing a join), you must create a materialized view log for each of its master tables.

See Also:

- CREATE MATERIALIZED VIEW on page 9-88, *Oracle8i Concepts*, *Oracle8i Data Warehousing Guide* and *Oracle8i Replication* for information on materialized views in general
- ALTER MATERIALIZED VIEW LOG on page 7-76 for information on modifying a materialized view log
- DROP MATERIALIZED VIEW LOG on page 10-145 for information on dropping a materialized view log
- *Oracle8i Concepts* for information on using direct loader logs

Prerequisites

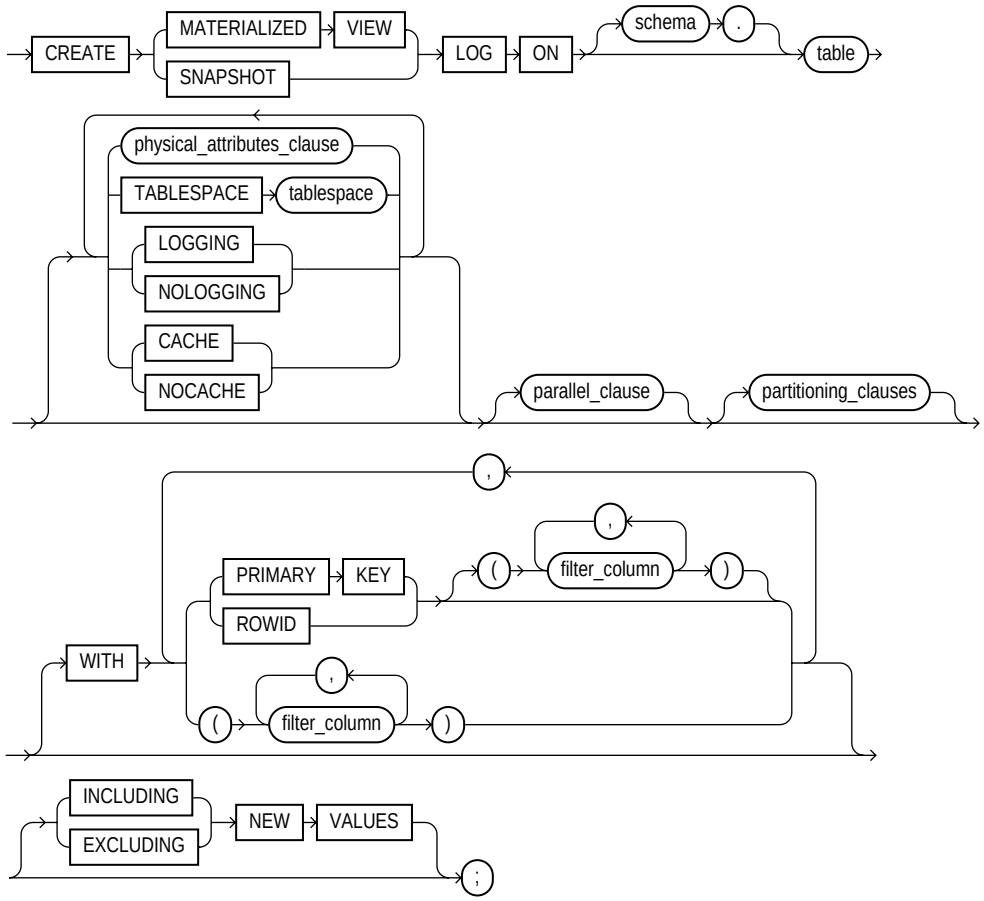
The privileges required to create a materialized view log directly relate to the privileges necessary to create the underlying objects associated with a materialized view log.

- If you own the master table, you can create an associated materialized view log if you have the CREATE TABLE privilege.
- If you are creating a materialized view log for a table in another user's schema, you must have the CREATE ANY TABLE and COMMENT ANY TABLE privileges, as well as either the SELECT privilege for the master table or SELECT ANY TABLE.

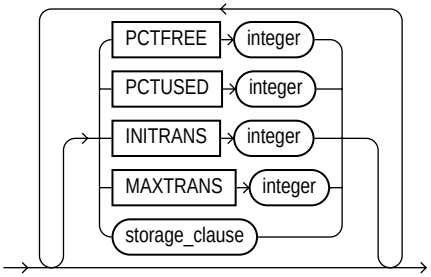
In either case, the owner of the materialized view log must have sufficient quota in the tablespace intended to hold the materialized view log or must have the UNLIMITED TABLESPACE system privilege.

> **See Also:** *Oracle8i Data Warehousing Guide* for more information about the prerequisites for creating a materialized view log

Syntax

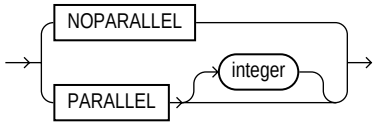

CREATE
MATERIALIZED
VIEW
SNAPSHOT
LOG
ON
schema
.
table
physical_attributes_clause
TABLESPACE
tablespace
LOGGING
NOLOGGING
CACHE
NOCACHE
parallel_clause
partitioning_clauses
WITH
,
PRIMARY
KEY
ROWID
(
filter_column
)
INCLUDING
EXCLUDING
NEW
VALUES
;

physical_attributes_clause::=



storage_clause: See storage_clause on page 11-129.

parallel_clause::=



partitioning_clauses: See table_properties CREATE TABLE on page 10-34.

Keywords and Parameters

schema

Specify the schema containing the materialized view log's master table. If you omit *schema*, Oracle assumes the master table is contained in your own schema. Oracle creates the materialized view log in the schema of its master table. You cannot create a materialized view log for a table in the schema of the user SYS.

table

Specify the name of the master table for which the materialized view log is to be created. You cannot create a materialized view log for a view.

physical_attributes_clause

Use the *physical_attributes_clause* to establish values for physical and storage characteristics for the materialized view log.

> **See Also:** CREATE TABLE on page 10-7 and *storage_clause* on page 11-129

TABLESPACE

Specify the tablespace in which the materialized view log is to be created. If you omit this clause, Oracle creates the materialized view log in the default tablespace of the owner of the materialized view log's schema.

LOGGING | NOLOGGING

Specify either LOGGING or NOLOGGING to establish the logging characteristics for the materialized view log.

> **See Also:** CREATE TABLE on page 10-7 for a description of logging characteristics

CACHE | NOCACHE

For data that will be accessed frequently, CACHE specifies that the blocks retrieved for this log are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. NOCACHE specifies that the blocks are placed at the least recently used end of the LRU list.

Note: NOCACHE has no effect on materialized view logs for which you specify KEEP in the *storage_clause*.

> **See Also:** ALTER TABLE on page 8-2 for information about specifying CACHE or NOCACHE

parallel_clause

The *parallel_clause* lets you indicate whether parallel operations will be supported for the materialized view log.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL — Specify NOPARALLEL for serial execution. This is the default.

PARALLEL — Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

See Also: "Notes on the parallel_clause" for CREATE TABLE on page 10-41

partitioning_clauses

Use the *partitioning_clauses* to indicate that the materialized view log is partitioned on specified ranges of values or on a hash function. Partitioning of materialized view logs is the same as partitioning of tables, as described in CREATE TABLE on page 10-7.

WITH

Use the WITH clause to indicate whether the materialized view log should record the primary key, the rowid, or both the primary key and rowid when rows in the master are updated.

This clause also specifies whether the materialized view log records filter columns, which are non-primary-key columns referenced by subquery materialized views.

If you omit this clause, primary key values are stored by default. Primary key values are implicitly stored when you specify a filter column list by itself. However, primary key values are not implicitly stored if you specify only ROWID or ROWID (filter_column) at create time.

PRIMARY KEY — Specify PRIMARY KEY to indicate that the primary key of all rows updated should be recorded in the materialized view log. The primary key of updated rows in the master table must be recorded in the materialized view log.

ROWID	Specify ROWID to indicate that the rowid of all rows updated should be recorded in the materialized view log. The rowid must be recorded in the materialized view log.
filter_column	Specify a comma-separated list that specifies the filter columns to be recorded in the materialized view log. For fast-refreshable primary-key materialized views defined with subqueries, all filter columns referenced by the defining subquery must be recorded in the materialized view log.

Restrictions:

- You can specify only one PRIMARY KEY, one ROWID, and one filter column list specification per materialized view log.
- Because PRIMARY KEY is implicitly included in *filter_column*, you cannot specify either of the following combinations:

```
ADD PRIMARY KEY,(filter_column)
ADD (filter_column), PRIMARY KEY
```

NEW VALUES

The NEW VALUES clause lets you indicate whether Oracle saves both old and new values in the materialized view log.

INCLUDING	Specify INCLUDING to save both new and old values in the log. If this log is for a table on which you have a single-table materialized aggregate view, and if you want the materialized view to be eligible for fast refresh, you must specify INCLUDING.
EXCLUDING	Specify EXCLUDING to disable the recording of new values in the log. This is the default. You can use this clause to avoid the overhead of recording new values. However, do not use this clause if you have a fast-refreshable single-table materialized aggregate view defined on this table.

Examples

Primary Key Examples The following statement creates a materialized view log on an employee table that records only primary key values:

```
CREATE MATERIALIZED VIEW LOG ON emp WITH PRIMARY KEY;
```

Oracle can use this materialized view log to perform a fast refresh on any simple primary key materialized view subsequently created on the emp table.

The following statement also creates a materialized view log that record only the primary keys of updated rows:

```
CREATE MATERIALIZED VIEW LOG ON emp
   PCTFREE 5
   TABLESPACE users
   STORAGE (INITIAL 10K NEXT 10K);
```

ROWID Example The following statement creates a materialized view log that records both the primary keys and the rowids of updated rows:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID, PRIMARY KEY;
```

Filter Column Example The following statement creates a materialized view log that records primary keys and updates to the filter column zip:

```
CREATE MATERIALIZED VIEW LOG ON address WITH (zip);
```

NEW VALUES Example The following example creates a master table, then creates a materialized view log that specifies INCLUDING NEW VALUES:

```
CREATE TABLE agg
   (u NUMBER, a NUMBER, b NUMBER, c NUMBER, d NUMBER);

CREATE MATERIALIZED VIEW LOG ON agg
   WITH ROWID (u,a,b,c,d)
   INCLUDING NEW VALUES;
```

You could create the following materialized aggregate view to use the agg log:

```
CREATE MATERIALIZED VIEW sn0
   REFRESH FAST ON COMMIT
   AS SELECT SUM(b+c), COUNT(*), a, d, COUNT(b+c)
         FROM agg
         GROUP BY a,d;
```

This materialized view is eligible for fast refresh because the log it uses includes both old and new values.

CREATE OPERATOR

Purpose

Use the CREATE OPERATOR statement to create a new operator and define its bindings.

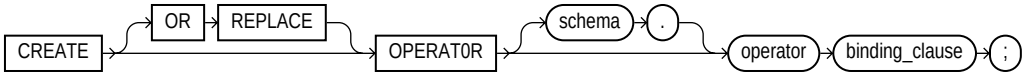
Operators can be referenced by indextypes and by DML and query SQL statements. The operators, in turn, reference functions, packages, types, and other user-defined objects.

> **See Also:** *Oracle8i Data Cartridge Developer's Guide* and *Oracle8i Concepts* for a discussion of these dependencies, and of operators in general

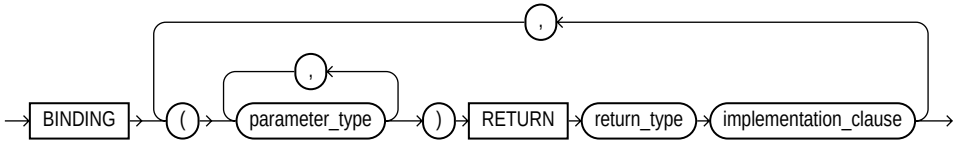
Prerequisites

To create an operator in your own schema, you must have CREATE OPERATOR system privilege. To create an operator in another schema, you must have the CREATE ANY OPERATOR system privilege. In either case, you must also have EXECUTE privilege on the functions and operators referenced.

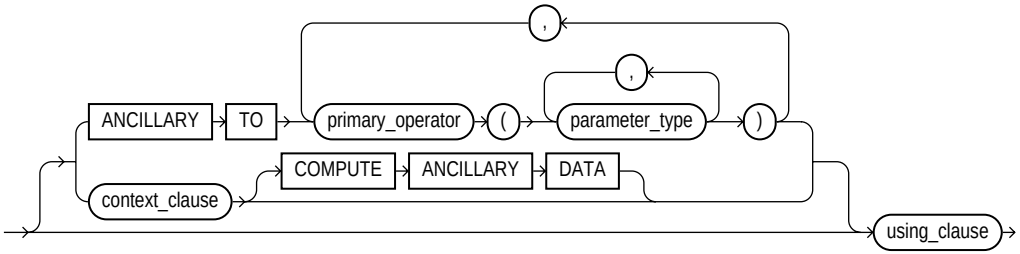
Syntax



binding_clause::=

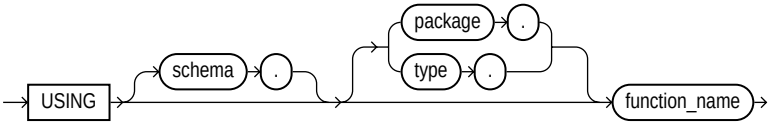


implementation_clause::=



context_clause::=

using_clause::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to replace the definition of the operator schema object.

Restriction: You can replace the definition only if the operator has no dependent objects (for example, indextypes supporting the operator).

schema

Specify the schema containing the operator. If you omit *schema*, Oracle assumes the operator is in your own schema.

operator

Specify the name of the operator to be created.

binding_clause

Use the *binding_clause* to specify one or more parameter datatypes (*parameter_type*) for binding the operator to a function. The signature of each binding (that is, the sequence of the datatypes of the arguments to the corresponding function) must be unique according to the rules of overloading.

The *parameter_type* can itself be an object type. If it is, you can optionally qualify it with its schema.

Restriction: You cannot specify a *parameter_type* of REF, LONG, or LONG RAW.

> **See Also:** *PL/SQL User's Guide and Reference* for more information about overloading

RETURN *return_type*	Specify the return datatype for the binding. The *return_type* can itself be an object type. If so, you can optionally qualify it with its schema. **Restriction:** You cannot specify a *return_type* of REF, LONG, or LONG RAW.	
implementation_clause	ANCILLARY TO *primary_operator*	Use the ANCILLARY TO clause to indicate that the operator binding is ancillary to the specified primary operator binding (*primary_operator*). If you specify this clause, do not specify a previous binding with just one number parameter.
	context_clause	Specify the name of the implementation type used by the functional implementation of the operator as a scan context.
	COMPUTE ANCILLARY DATA	Specify COMPUTE ANCILLARY DATA to indicate that the operator binding computes ancillary data.
using_clause	The *using_clause* lets you specify the function that provides the implementation for the binding.	
	function_name	Specify the name of the function. The function can be a standalone function, packaged function, type method, or a synonym for any of these.

Example

CREATE OPERATOR Example This example creates an operator called MERGE in the scott schema with two bindings. The first binding is for merging two VARCHAR2 values and returning a VARCHAR2 result. The second binding is for merging two geometries into a single geometry. The corresponding functional implementations for the bindings are also specified.

```
CREATE OPERATOR scott.merge
BINDING (varchar2, varchar2) RETURN varchar2
         USING text.merge,
   (spatial.geo, spatial.geo) RETURN spatial.geo
         USING spatial.merge;
```

CREATE OUTLINE

Purpose

Use the CREATE OUTLINE statement to create a **stored outline**, which is a set of attributes used by the optimizer to generate an execution plan. You can then instruct the optimizer to use a set of outlines to influence the generation of execution plans whenever a particular SQL statement is issued, regardless of changes in factors that can affect optimization. You can also modify an outline so that it takes into account changes in these factors.

You enable or disable the use of stored outlines dynamically for an individual session or for the system.

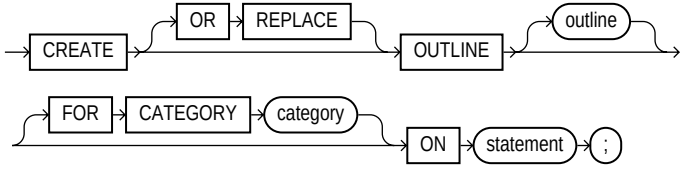
See Also:

- *Oracle8i Performance Guide and Reference*
- ALTER OUTLINE on page 7-83 for information on modifying an outline
- ALTER SESSION on page 7-105 and ALTER SYSTEM on page 7-127 for information on dynamically enabling and disabling stored outlines

Prerequisites

To create an outline, you must have the CREATE ANY OUTLINE system privilege.

Syntax



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to replace an existing outline with a new outline of the same name.

outline

Specify the unique name to be assigned to the stored outline. If you do not specify *outline*, the system generates an outline name.

FOR CATEGORY *category*

Specify an optional name used to group stored outlines. For example, you could specify a category of outlines for end-of-week use and another for end-of-quarter use. If you do not specify *category*, the outline is stored in the DEFAULT category.

ON *statement*

Specify the SQL statement for which Oracle will create an outline when the statement is compiled. You can specify any one of the following statements:

- SELECT
- DELETE
- UPDATE
- INSERT ... SELECT
- CREATE TABLE ... AS SELECT

> **Note:** You can specify multiple outlines for a single statement, but each outline for the same statement must be in a different category.

Example

CREATE OUTLINE Example The following statement creates a stored outline by compiling the ON statement. The outline is called salaries and is stored in the category special.

```
CREATE OUTLINE salaries FOR CATEGORY special
   ON SELECT ename, sal FROM emp;
```

When this same SELECT statement is subsequently compiled, if the USE_STORED_OUTLINES parameter is set to special, Oracle generates the same execution plan as was generated when the outline salaries was created.

CREATE PACKAGE

Purpose

Use the CREATE PACKAGE statement to create the specification for a stored **package**, which is an encapsulated collection of related procedures, functions, and other program objects stored together in the database. The **specification** declares these objects.

See Also:

- CREATE FUNCTION on page 9-43 and CREATE PROCEDURE on page 9-132 for information on creating standalone functions and procedures
- ALTER PACKAGE on page 7-85 for information on modifying a package
- DROP PACKAGE on page 10-150 for information on dropping a package
- *Oracle8i Application Developer's Guide - Fundamentals* and *Oracle8i Supplied PL/SQL Packages Reference* for detailed discussions of packages and how to use them

Prerequisites

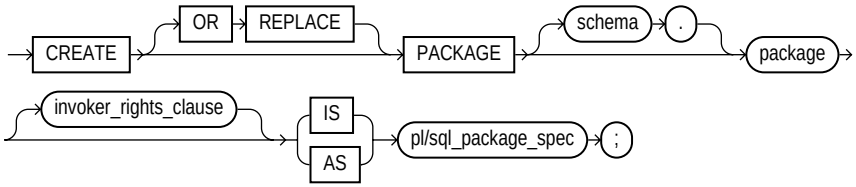
Before a package can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

To create a package in your own schema, you must have CREATE PROCEDURE system privilege. To create a package in another user's schema, you must have CREATE ANY PROCEDURE system privilege.

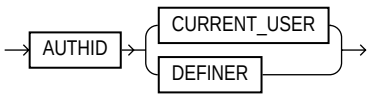
To embed a CREATE PACKAGE statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

See Also: *PL/SQL User's Guide and Reference*

Syntax



invoker_rights_clause::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the package specification if it already exists. Use this clause to change the specification of an existing package without dropping, re-creating, and regranting object privileges previously granted on the package. If you change a package specification, Oracle recompiles it.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

If any function-based indexes depend on the package, Oracle marks the indexes DISABLED.

> **See Also:** ALTER PACKAGE on page 7-85 for information on recompiling package specifications

schema

Specify the schema to contain the package. If you omit *schema*, Oracle creates the package in your own schema.

package

Specify the name of the package to be created.

If creating the package results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SHOW ERRORS command.

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the functions and procedures in the package execute with the privileges and in the schema of the user who owns it or with the privileges and in the schema of CURRENT_USER. This specification applies to the corresponding package body as well.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the package.

AUTHID CURRENT_USER — Specify CURRENT_USER to indicate that the package executes with the privileges of CURRENT_USER. This clause creates an "invoker-rights package."

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the package resides.

AUTHID DEFINER — Specify DEFINER to indicate that the package executes with the privileges of the owner of the schema in which the package resides and that external names resolve in the schema where the package resides. This is the default.

See Also:

- *PL/SQL User's Guide and Reference*
- *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined

pl/sql_package_spec

Specify the package specification, which can contain type definitions, cursor declarations, variable declarations, constant declarations, exception declarations, PL/SQL subprogram specifications, and call specifications (declarations of a C or Java routine expressed in PL/SQL).

See Also:

- *PL/SQL User's Guide and Reference* for more information on PL/SQL package program units
- *Oracle8i Supplied PL/SQL Packages Reference* for information on Oracle supplied packages
- "Restrictions on User-Defined Functions" on page 9-46 for a list of restrictions on user-defined functions in a package

Example

CREATE PACKAGE Example The following SQL statement creates the specification of the `emp_mgmt` package:

```
CREATE PACKAGE emp_mgmt AS
   FUNCTION hire(ename VARCHAR2, job VARCHAR2, mgr NUMBER,
                 sal NUMBER, comm NUMBER, deptno NUMBER)
      RETURN NUMBER;
   FUNCTION create_dept(dname VARCHAR2, loc VARCHAR2)
      RETURN NUMBER;
   PROCEDURE remove_emp(empno NUMBER);
   PROCEDURE remove_dept(deptno NUMBER);
   PROCEDURE increase_sal(empno NUMBER, sal_incr NUMBER);
   PROCEDURE increase_comm(empno NUMBER, comm_incr NUMBER);
      no_comm EXCEPTION;
      no_sal EXCEPTION;
END emp_mgmt;
```

The specification for the `emp_mgmt` package declares the following public program objects:

- the functions `hire` and `create_dept`
- the procedures `remove_emp`, `remove_dept`, `increase_sal`, and `increase_comm`
- the exceptions `no_comm` and `no_sal`

All of these objects are available to users who have access to the package. After creating the package, you can develop applications that call any of the package's public procedures or functions or raise any of the package's public exceptions.

Before you can call this package's procedures and functions, you must define these procedures and functions in the package body. For an example of a CREATE

PACKAGE BODY statement that creates the body of the `emp_mgmt` package, see CREATE PACKAGE BODY on page 9-127.

CREATE PACKAGE BODY

Purpose

Use the CREATE PACKAGE BODY statement to create the body of a stored **package**, which is an encapsulated collection of related procedures, stored functions, and other program objects stored together in the database. The **body** defines these objects.

Packages are an alternative to creating procedures and functions as standalone schema objects.

See Also:

- CREATE FUNCTION on page 9-43 and CREATE PROCEDURE on page 9-132 for information on creating standalone functions and procedures
- CREATE PACKAGE on page 9-122 for a discussion of packages, including how to create packages
- "Examples" on page 9-129 for some illustrations
- ALTER PACKAGE on page 7-85 for information on modifying a package
- DROP PACKAGE on page 10-150 for information on removing a package from the database

Prerequisites

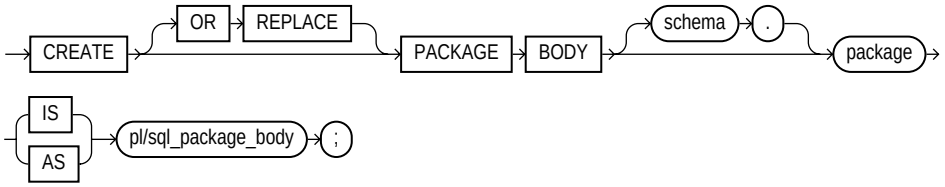
Before a package can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

To create a package in your own schema, you must have CREATE PROCEDURE system privilege. To create a package in another user's schema, you must have CREATE ANY PROCEDURE system privilege.

To embed a CREATE PACKAGE BODY statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

See Also: *PL/SQL User's Guide and Reference*

Syntax



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the package body if it already exists. Use this clause to change the body of an existing package without dropping, re-creating, and regranting object privileges previously granted on it. If you change a package body, Oracle recompiles it.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

> **See Also:** ALTER PACKAGE on page 7-85 for information on recompiling package bodies

schema

Specify the schema to contain the package. If you omit *schema*, Oracle creates the package in your current schema.

package

Specify the name of the package to be created.

pl/sql_package_body

Specify the package body, which can contain PL/SQL subprogram bodies or call specifications (declarations of a C or Java routine expressed in PL/SQL).

See Also:

- *Oracle8i Application Developer's Guide - Fundamentals* for more information on writing PL/SQL or C package program units
- *Oracle8i Java Stored Procedures Developer's Guide* for information on JAVA package program units
- "Restrictions on User-Defined Functions" on page 9-46 for a list of restrictions on user-defined functions in a package

Examples

CREATE PACKAGE BODY Example This SQL statement creates the body of the `emp_mgmt` package:

```
CREATE PACKAGE BODY emp_mgmt AS
   tot_emps  NUMBER;
   tot_depts NUMBER;

FUNCTION hire
   (ename VARCHAR2,
    job VARCHAR2,
    mgr NUMBER,
    sal NUMBER,
    comm NUMBER,
    deptno NUMBER)

RETURN NUMBER IS
   new_empno NUMBER(4);
BEGIN
   SELECT empseq.NEXTVAL
      INTO new_empno
      FROM DUAL;
   INSERT INTO emp
      VALUES (new_empno, ename, job, mgr, sal, comm, deptno,
         tot_emps := tot_emps + 1;
   RETURN(new_empno);
END;

FUNCTION create_dept(dname VARCHAR2, loc VARCHAR2)
   RETURN NUMBER IS
      new_deptno NUMBER(4);
   BEGIN
      SELECT deptseq.NEXTVAL
```

```
         INTO new_deptno
         FROM dual;
      INSERT INTO dept
         VALUES (new_deptno, dname, loc);
            tot_depts := tot_depts + 1;
      RETURN(new_deptno);
   END;

PROCEDURE remove_emp(empno NUMBER) IS
   BEGIN
      DELETE FROM emp
      WHERE emp.empno = remove_emp.empno;
         tot_emps := tot_emps - 1;
   END;

PROCEDURE remove_dept(deptno NUMBER) IS
   BEGIN
      DELETE FROM dept
      WHERE dept.deptno = remove_dept.deptno;
         tot_depts := tot_depts - 1;
      SELECT COUNT(*)
         INTO tot_emps
         FROM emp;
         /* In case Oracle deleted employees from the EMP table
         to enforce referential integrity constraints, reset
         the value of the variable TOT_EMPS to the total
         number of employees in the EMP table. */
   END;

PROCEDURE increase_sal(empno NUMBER, sal_incr NUMBER) IS
   curr_sal NUMBER(7,2);
   BEGIN
      SELECT sal
      INTO curr_sal
      FROM emp
      WHERE emp.empno = increase_sal.empno;
      IF curr_sal IS NULL
         THEN RAISE no_sal;
      ELSE
         UPDATE emp
         SET sal = sal + sal_incr
         WHERE empno = empno;
      END IF;
   END;
```

```
PROCEDURE increase_comm(empno NUMBER, comm_incr NUMBER) IS
      curr_comm NUMBER(7,2);
   BEGIN
      SELECT comm
      INTO curr_comm
      FROM emp
      WHERE emp.empno = increase_comm.empno
      IF curr_comm IS NULL
         THEN RAISE no_comm;
      ELSE
         UPDATE emp
         SET comm = comm + comm_incr;
      END IF;
   END;

END emp_mgmt;
```

This package body corresponds to the package specification in the example of the CREATE PACKAGE statement earlier in this chapter. The package body defines the public program objects declared in the package specification:

- The functions `hire` and `create_dept`
- The procedures `remove_emp`, `remove_dept`, `increase_sal`, and `increase_comm`

These objects are declared in the package specification, so they can be called by application programs, procedures, and functions outside the package. For example, if you have access to the package, you can create a procedure `increase_all_comms` separate from the `emp_mgmt` package that calls the `increase_comm` procedure.

These objects are defined in the package body, so you can change their definitions without causing Oracle to invalidate dependent schema objects. For example, if you subsequently change the definition of `hire`, Oracle need not recompile `increase_all_comms` before executing it.

The package body in this example also declares private program objects, the variables `tot_emps` and `tot_depts`. These objects are declared in the package body rather than the package specification, so they are accessible to other objects in the package, but they are not accessible outside the package. For example, you cannot develop an application that explicitly changes the value of the variable `tot_depts`. However, the function `create_dept` is part of the package, so `create_dept` can change the value of tot_depts.

CREATE PROCEDURE

Purpose

Use the CREATE PROCEDURE statement to create a standalone stored procedure or a call specification.

A **procedure** is a group of PL/SQL statements that you can call by name. A **call specification ("call spec")** declares a Java method or a third-generation language (3GL) routine so that it can be called from SQL and PL/SQL. The call spec tells Oracle which Java method to invoke when a call is made. It also tells Oracle what type conversions to make for the arguments and return value.

Stored procedures offer advantages in the areas of development, integrity, security, performance, and memory allocation.

See Also:

- *Oracle8i Application Developer's Guide - Fundamentals* for more information on stored procedures, including how to call stored procedures
- CREATE FUNCTION on page 9-43 for information specific to functions, which are similar in many ways
- CREATE PACKAGE on page 9-122 for information on creating packages. (The CREATE PROCEDURE statement creates a procedure as a standalone schema object. You can also create a procedure as part of a package.
- ALTER PROCEDURE on page 7-88 and DROP PROCEDURE on page 10-152 for information on modifying and dropping a standalone procedure
- CREATE LIBRARY on page 9-86 for more information about shared libraries
- *Oracle8i Application Developer's Guide - Fundamentals* for more information about registering external procedures

Prerequisites

Before creating a procedure, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depends on your operating system.

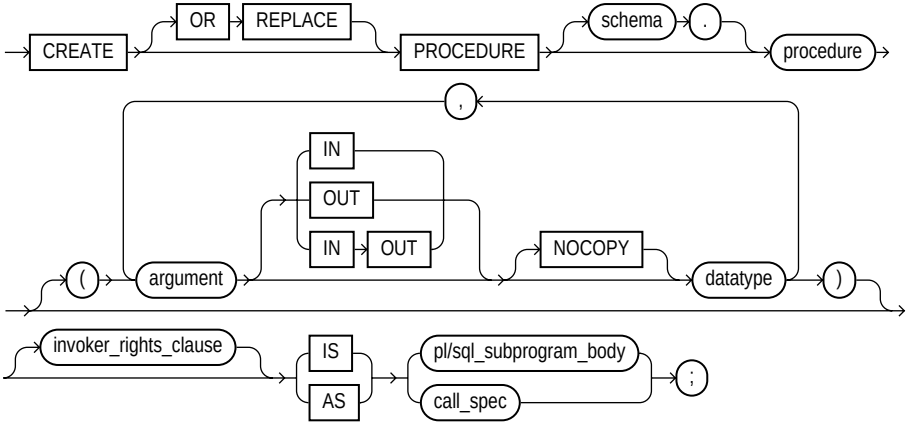
To create a procedure in your own schema, you must have the CREATE PROCEDURE system privilege. To create a procedure in another user's schema, you must have CREATE ANY PROCEDURE system privilege. To replace a procedure in another schema, you must have the ALTER ANY PROCEDURE system privilege.

To invoke a call spec, you may need additional privileges (for example, EXECUTE privileges on the C library for a C call spec).

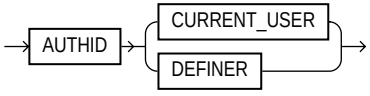
To embed a CREATE PROCEDURE statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

> **See Also:** *PL/SQL User's Guide and Reference* or *Oracle8i Java Stored Procedures Developer's Guide* for more information on such prerequisites

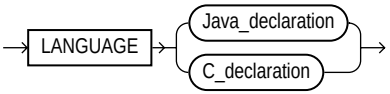
Syntax



invoker_rights_clause::=

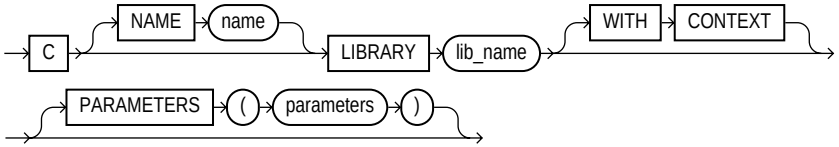


call_spec::=



Java_declaration::=

C_declaration::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the procedure if it already exists. Use this clause to change the definition of an existing procedure without dropping, re-creating, and regranting object privileges previously granted on it. If you redefine a procedure, Oracle recompiles it.

Users who had previously been granted privileges on a redefined procedure can still access the procedure without being regranted the privileges.

If any function-based indexes depend on the package, Oracle marks the indexes DISABLED.

> **See Also:** ALTER PROCEDURE on page 7-88 for information on recompiling procedures

schema

Specify the schema to contain the procedure. If you omit *schema*, Oracle creates the procedure in your current schema.

procedure

Specify the name of the procedure to be created.

If creating the procedure results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

argument

argument — Specify the name of an argument to the procedure. If the procedure does not accept arguments, you can omit the parentheses following the procedure name.

IN — Specify IN to indicate that you must specify a value for the argument when calling the procedure.

OUT — Specify OUT to indicate that the procedure passes a value for this argument back to its calling environment after execution.

IN OUT — Specify IN OUT to indicate that you must specify a value for the argument when calling the procedure and that the procedure passes a value back to its calling environment after execution.

If you omit IN, OUT, and IN OUT, the argument defaults to IN.

NOCOPY — Specify NOCOPY to instruct Oracle to pass this argument as fast as possible. This clause can significantly enhance performance when passing a large value like a record, an index-by table, or a varray to an OUT or IN OUT parameter. (IN parameter values are always passed NOCOPY.)

- When you specify NOCOPY, assignments made to a package variable may show immediately in this parameter (or assignments made to this parameter may show immediately in a package variable) if the package variable is passed as the actual assignment corresponding to this parameter.
- Similarly, changes made either to this parameter or to another parameter may be visible immediately through both names if the same variable is passed to both.
- If the procedure is exited with an unhandled exception, any assignment made to this parameter may be visible in the caller's variable.

These effects may or may not occur on any particular call. You should use NOCOPY only when these effects would not matter.

datatype Specify the datatype of the argument. An argument can have any datatype supported by PL/SQL.

Datatypes cannot specify length, precision, or scale. For example, VARCHAR2(10) is not valid, but VARCHAR2 is valid. Oracle derives the length, precision, and scale of an argument from the environment from which the procedure is called.

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the procedure executes with the privileges and in the schema of the user who owns it or with the privileges and in the schema of CURRENT_USER.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the procedure.

AUTHID CURRENT_USER Specify CURRENT_USER to indicate that the procedure executes with the privileges of CURRENT_USER. This clause creates an "invoker-rights procedure."

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the procedure resides.

AUTHID DEFINER Specify DEFINER to indicate that the procedure executes with the privileges of the owner of the schema in which the procedure resides, and that external names resolve in the schema where the procedure resides. This is the default.

See Also:

- *PL/SQL User's Guide and Reference*
- *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined

IS | AS

pl/sql_subprogram_body — In the PL/SQL subprogram body, declare the procedure in a PL/SQL subprogram body.

See Also: *Oracle8i Application Developer's Guide - Fundamentals* for more information on PL/SQL subprograms

call_spec — Use the *call_spec* to map a Java or C method name, parameter types, and return type to their SQL counterparts.

In *Java_declaration*, '*string*' identifies the Java implementation of the method.

See Also:

- *Oracle8i Java Stored Procedures Developer's Guide*

- *Oracle8i Application Developer's Guide - Fundamentals* for an explanation of the parameters and semantics of the *C_declaration*

AS EXTERNAL — The AS EXTERNAL clause is an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle Corporation recommends that you use the AS LANGUAGE C syntax.

Examples

CREATE PROCEDURE Example The following statement creates the procedure `credit` in the schema `sam`:

```
CREATE PROCEDURE sam.credit (acc_no IN NUMBER, amount IN NUMBER) AS
   BEGIN
      UPDATE accounts
      SET balance = balance + amount
      WHERE account_id = acc_no;
   END;
```

The `credit` procedure credits a specified bank account with a specified amount. When you call the procedure, you must specify the following arguments:

ACC_NO	is the number of the bank account to be credited. The argument's datatype is NUMBER.
AMOUNT	is the amount of the credit. The argument's datatype is NUMBER.

The procedure uses an UPDATE statement to increase the value in the balance column of the accounts table by the value of the argument amount for the account identified by the argument acc_no.

In the following example, external procedure c_find_root expects a pointer as a parameter. Procedure find_root passes the parameter by reference using the BY REF phrase:

```
CREATE PROCEDURE find_root
   ( x IN REAL )
   IS LANGUAGE C
      NAME "c_find_root"
      LIBRARY c_utils
      PARAMETERS ( x BY REF );
```

CREATE PROFILE

Purpose

Use the CREATE PROFILE statement to create a **profile**, which is a set of limits on database resources. If you assign the profile to a user, that user cannot exceed these limits.

> **See Also:** *Oracle8i Administrator's Guide* for a detailed description and explanation of how to use password management and protection

Prerequisites

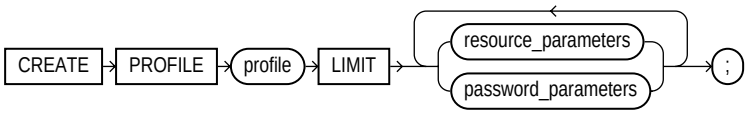
You must have CREATE PROFILE system privilege.

To specify resource limits for a user, you must:

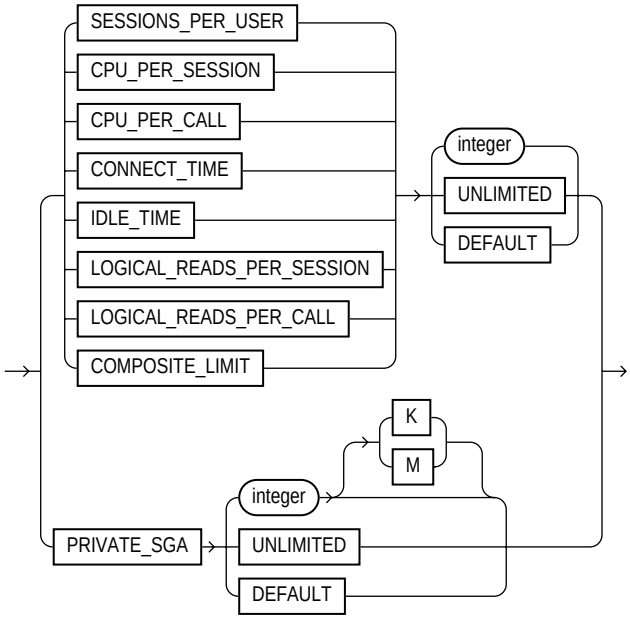
- Enable resource limits dynamically with the ALTER SYSTEM statement or with the initialization parameter RESOURCE_LIMIT. (This parameter does not apply to password resources. Password resources are always enabled.)
- Create a profile that defines the limits using the CREATE PROFILE statement
- Assign the profile to the user using the CREATE USER or ALTER USER statement

> **See Also:**
>
> - ALTER SYSTEM on page 7-127 for information on enabling resource limits dynamically
> - *Oracle8i Reference* for information on the RESOURCE_LIMIT parameter
> - CREATE USER on page 10-99 and ALTER USER on page 8-88 for information on profiles

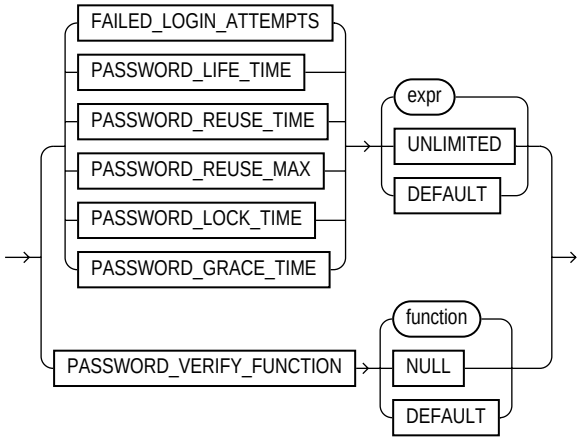
Syntax



resource_parameters::=



password_parameters::=



Keywords and Parameters

profile

Specify the name of the profile to be created. Use profiles to limit the database resources available to a user for a single call or a single session.

Oracle enforces resource limits in the following ways:

- If a user exceeds the CONNECT_TIME or IDLE_TIME session resource limit, Oracle rolls back the current transaction and ends the session. When the user process next issues a call, Oracle returns an error.
- If a user attempts to perform an operation that exceeds the limit for other session resources, Oracle aborts the operation, rolls back the current statement, and immediately returns an error. The user can then commit or roll back the current transaction, and must then end the session.
- If a user attempts to perform an operation that exceeds the limit for a single call, Oracle aborts the operation, rolls back the current statement, and returns an error, leaving the current transaction intact.

Notes:

- You can use fractions of days for all parameters that limit time, with days as units. For example, 1 hour is 1/24 and 1 minute is 1/1440.
- You can specify resource limits for users regardless of whether the resource limits are enabled. However, Oracle does not enforce the limits until you enable them.

UNLIMITED

When specified with a resource parameter, indicates that a user assigned this profile can use an unlimited amount of this resource. When specified with a password parameter, indicates that no limit has been set for the parameter.

DEFAULT

Specify DEFAULT if you want to omit a limit for this resource in this profile. A user assigned this profile is subject to the limit for this resource specified in the DEFAULT profile. The DEFAULT profile initially defines unlimited resources. You can change those limits with the ALTER PROFILE statement.

Any user who is not explicitly assigned a profile is subject to the limits defined in the DEFAULT profile. Also, if the profile that is explicitly assigned to a user omits limits for some resources or specifies DEFAULT for some limits, the user is subject to the limits on those resources defined by the DEFAULT profile.

resource_parameters

SESSIONS_PER_USER	Specify the number of concurrent sessions to which you want to limit the user.
CPU_PER_SESSION	Specify the CPU time limit for a session, expressed in hundredth of seconds.
CPU_PER_CALL	Specify the CPU time limit for a call (a parse, execute, or fetch), expressed in hundredths of seconds.
CONNECT_TIME	Specify the total elapsed time limit for a session, expressed in minutes.
IDLE_TIME	Specify the permitted periods of continuous inactive time during a session, expressed in minutes. Long-running queries and other operations are not subject to this limit.
LOGICAL_READS_PER_SESSION	Specify the permitted number of data blocks read in a session, including blocks read from memory and disk.
LOGICAL_READS_PER_CALL	Specify the permitted the number of data blocks read for a call to process a SQL statement (a parse, execute, or fetch).
PRIVATE_SGA	Specify the amount of private space a session can allocate in the shared pool of the system global area (SGA), expressed in bytes. Use K or M to specify this limit in kilobytes or megabytes.

> **Note:** This limit applies only if you are using multi-threaded server architecture. The private space for a session in the SGA includes private SQL and PL/SQL areas, but not shared SQL and PL/SQL areas.

COMPOSITE_LIMIT — Specify the total resources cost for a session, expressed in *service units*. Oracle calculates the total service units as a weighted sum of CPU_PER_SESSION, CONNECT_TIME, LOGICAL_READS_PER_SESSION, and PRIVATE_SGA.

> **See Also:** ALTER RESOURCE COST on page 7-95 for information on how to specify the weight for each session resource

password_parameters

FAILED_LOGIN_ATTEMPTS — Specify the number of failed attempts to log in to the user account before the account is locked.

PASSWORD_LIFE_TIME — Specify the number of days the same password can be used for authentication. The password expires if it is not changed within this period, and further connections are rejected.

PASSWORD_REUSE_TIME — Specify the number of days before which a password cannot be reused. If you set PASSWORD_REUSE_TIME to an integer value, then you must set PASSWORD_REUSE_MAX to UNLIMITED.

PASSWORD_REUSE_MAX — Specify the number of password changes required before the current password can be reused. If you set PASSWORD_REUSE_MAX to an integer value, then you must set PASSWORD_REUSE_TIME to UNLIMITED.

PASSWORD_LOCK_TIME — Specify the number of days an account will be locked after the specified number of consecutive failed login attempts.

PASSWORD_GRACE_TIME — Specify the number of days after the grace period begins during which a warning is issued and login is allowed. If the password is not changed during the grace period, the password expires.

PASSWORD_VERIFY_FUNCTION — The PASSWORD_VERIFY_FUNCTION clause lets allows a PL/SQL password complexity verification script to be passed as an argument to the CREATE PROFILE statement. Oracle provides a default script, but you can create your own routine or use third-party software instead.

- *function* — Specify the name of the password complexity verification routine.
- NULL — Specify NULL to indicate that no password verification is performed.

Restrictions on password parameters:

- If PASSWORD_REUSE_TIME is set to an integer value, PASSWORD_REUSE_MAX must be set to UNLIMITED. If PASSWORD_REUSE_MAX is set to an integer value, PASSWORD_REUSE_TIME must be set to UNLIMITED.
- If both PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX are set to UNLIMITED, then Oracle uses neither of these password resources.
- If PASSWORD_REUSE_MAX is set to DEFAULT and PASSWORD_REUSE_TIME is set to UNLIMITED, then Oracle uses the PASSWORD_REUSE_MAX value defined in the DEFAULT profile.
- If PASSWORD_REUSE_TIME is set to DEFAULT and PASSWORD_REUSE_MAX is set to UNLIMITED, then Oracle uses the PASSWORD_REUSE_TIME value defined in the DEFAULT profile.
- If both PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX are set to DEFAULT, then Oracle uses whichever value is defined in the DEFAULT profile.

Examples

CREATE PROFILE Example The following statement creates the profile prof:

```
CREATE PROFILE prof
  LIMIT PASSWORD_REUSE_MAX DEFAULT
        PASSWORD_REUSE_TIME UNLIMITED;
```

Setting Resource Limits Example The following statement creates the profile system_manager:

```
CREATE PROFILE system_manager
   LIMIT SESSIONS_PER_USER    UNLIMITED
   CPU_PER_SESSION            UNLIMITED
   CPU_PER_CALL               3000
   CONNECT_TIME               45
   LOGICAL_READS_PER_SESSION  DEFAULT
   LOGICAL_READS_PER_CALL     1000
   PRIVATE SGA                15K
   COMPOSITE_LIMIT            5000000;
```

If you then assign the system_manager profile to a user, the user is subject to the following limits in subsequent sessions:

- The user can have any number of concurrent sessions.

- In a single session, the user can consume an unlimited amount of CPU time.
- A single call made by the user cannot consume more than 30 seconds of CPU time.
- A single session cannot last for more than 45 minutes.
- In a single session, the number of data blocks read from memory and disk is subject to the limit specified in the DEFAULT profile.
- A single call made by the user cannot read more than 1000 data blocks from memory and disk.
- A single session cannot allocate more than 15 kilobytes of memory in the SGA.
- In a single session, the total resource cost cannot exceed 5 million service units. The formula for calculating the total resource cost is specified by the ALTER RESOURCE COST statement.
- Since the system_manager profile omits a limit for IDLE_TIME and for password limits, the user is subject to the limits on these resources specified in the DEFAULT profile.

Setting Password Limits Example The following statement creates profile myprofile with password profile limits values set:

```
CREATE PROFILE myprofile LIMIT
   FAILED_LOGIN_ATTEMPTS 5
   PASSWORD_LIFE_TIME 60
   PASSWORD_REUSE_TIME 60
   PASSWORD_REUSE_MAX UNLIMITED
   PASSWORD_VERIFY_FUNCTION verify_function
   PASSWORD_LOCK_TIME 1/24
   PASSWORD_GRACE_TIME 10;
```

CREATE ROLE

Purpose

Use the CREATE ROLE statement to create a **role**, which is a set of privileges that can be granted to users or to other roles. You can use roles to administer database privileges. You can add privileges to a role and then grant the role to a user. The user can then enable the role and exercise the privileges granted by the role.

A role contains all privileges granted to the role and all privileges of other roles granted to it. A new role is initially empty. You add privileges to a role with the GRANT statement.

When you create a role that is NOT IDENTIFIED or is IDENTIFIED EXTERNALLY or BY *password*, Oracle grants you the role with ADMIN OPTION. However, when you create a role IDENTIFIED GLOBALLY, Oracle does not grant you the role.

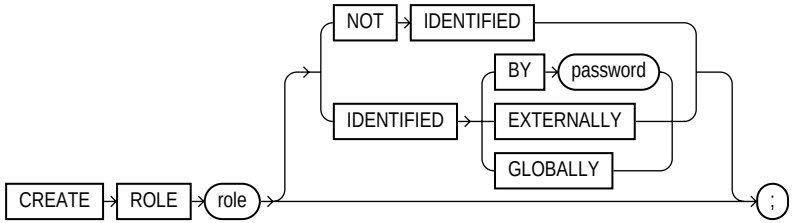
See Also:

- GRANT on page 11-31 for information on granting roles
- ALTER USER on page 8-88 for information on enabling roles
- ALTER ROLE on page 7-98 for information on modifying a role
- DROP ROLE on page 10-156 for information on removing a role from the database
- SET ROLE on page 11-122 for information on enabling and disabling roles for the current session
- *Oracle8i Distributed Database Systems* for a detailed description and explanation of using global roles

Prerequisites

You must have CREATE ROLE system privilege.

Syntax



Keywords and Parameters

role

Specify the name of the role to be created. Oracle recommends that the role contain at least one single-byte character regardless of whether the database character set also contains multibyte characters.

Some roles are defined by SQL scripts provided on your distribution media.

> **See Also:** GRANT on page 11-31 for a list of these predefined roles

NOT IDENTIFIED

Specify NOT IDENTIFIED to indicate that this role is authorized by the database and that no password is required to enable the role.

IDENTIFIED

Use the IDENTIFIED clause to indicate that a user must be authorized by the specified method before the role is enabled with the SET ROLE statement:

BY *password* — The BY *password* clause lets you create a **local user** and indicates that the user must specify the password to Oracle when enabling the role. The password can contain only single-byte characters from your database character set regardless of whether this character set also contains multibyte characters.

EXTERNALLY — Specify EXTERNALLY to create an **external user** and indicates that a user must be authorized by an external service (such as an operating system or third-party service) before enabling the role.

Depending on the operating system, the user may have to specify a password to the operating system before the role is enabled.

`GLOBALLY` Specify `GLOBALLY` to create a **global user** and indicates that a user must be authorized to use the role by the enterprise directory service before the role is enabled with the `SET ROLE` statement, or at login.

If you omit both the `NOT IDENTIFIED` clause and the `IDENTIFIED` clause, the role defaults to `NOT IDENTIFIED`.

Examples

CREATE ROLE Example The following statement creates global role `vendor`:

```
CREATE ROLE vendor IDENTIFIED GLOBALLY;
```

The following statement creates the role `teller`:

```
CREATE ROLE teller
   IDENTIFIED BY cashflow;
```

Users who are subsequently granted the `teller` role must specify the password `cashflow` to enable the role with the `SET ROLE` statement.

CREATE ROLLBACK SEGMENT

Purpose

Use the CREATE ROLLBACK SEGMENT statement to create a **rollback segment**, which is an object that Oracle uses to store data necessary to reverse, or undo, changes made by transactions.

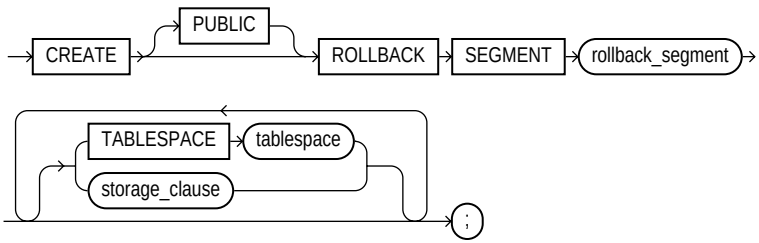
See Also:

- ALTER ROLLBACK SEGMENT on page 7-100 for information on altering a rollback segment
- DROP ROLLBACK SEGMENT on page 10-157 for information on removing a rollback segment

Prerequisites

You must have CREATE ROLLBACK SEGMENT system privilege.

Syntax



storage_clause: See storage_clause on page 11-129.

Keyword and Parameters

PUBLIC

Specify PUBLIC to indicate that the rollback segment is public and is available to any instance. If you omit this clause, the rollback segment is private and is available only to the instance naming it in its initialization parameter ROLLBACK_SEGMENTS.

rollback_segment

Specify the name of the rollback segment to be created.

TABLESPACE

Use the TABLESPACE clause to identify the tablespace in which the rollback segment is created. If you omit this clause, Oracle creates the rollback segment in the SYSTEM tablespace.

Restriction: You cannot create a rollback segment in a tablespace that is system managed (that is, during creation you specified EXTENT MANAGEMENT LOCAL AUTOALLOCATE).

Notes:

- A tablespace can have multiple rollback segments. Generally, multiple rollback segments improve performance.
- The tablespace must be online for you to add a rollback segment to it.
- When you create a rollback segment, it is initially offline. To make it available for transactions by your Oracle instance, bring it online using the ALTER ROLLBACK SEGMENT statement. To bring it online automatically whenever you start up the database, add the segment's name to the value of the ROLLBACK_SEGMENTS initialization parameter.

See Also:

- CREATE TABLESPACE on page 10-56
- *Oracle8i Administrator's Guide* for more information on creating rollback segments and making them available

storage_clause

The *storage_clause* lets you specify the characteristics for the rollback segment.

Notes:

- The OPTIMAL parameter of the *storage_clause* is of particular interest, because it applies only to rollback segments.
- You cannot specify the PCTINCREASE parameter of the *storage_clause* with CREATE ROLLBACK SEGMENT.

See Also: *storage_clause* on page 11-129

Examples

CREATE ROLLBACK SEGMENT Example The following statement creates a rollback segment with default storage values in the system tablespace:

```
CREATE ROLLBACK SEGMENT rbs_2
   TABLESPACE system;
```

The above statement is equivalent to the following:

```
CREATE ROLLBACK SEGMENT rbs_2
   TABLESPACE system
   STORAGE
   ( INITIAL 10K
     NEXT 10K
     MAXEXTENTS UNLIMITED);
```

CREATE SCHEMA

Purpose

Use the CREATE SCHEMA to create multiple tables and views and perform multiple grants in a single transaction.

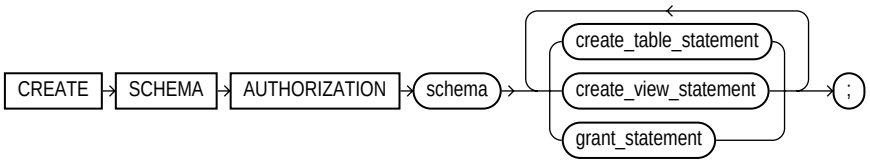
To execute a CREATE SCHEMA statement, Oracle executes each included statement. If all statements execute successfully, Oracle commits the transaction. If any statement results in an error, Oracle rolls back all the statements.

> **Note:** This statement does not actually create a schema. Oracle automatically creates a schema when you create a user (see CREATE USER on page 10-99). This statement lets you populate your schema with tables and views and grant privileges on those objects without having to issue multiple SQL statements in multiple transactions.

Prerequisites

The CREATE SCHEMA statement can include CREATE TABLE, CREATE VIEW, and GRANT statements. To issue a CREATE SCHEMA statement, you must have the privileges necessary to issue the included statements.

Syntax



Keyword and Parameters

schema

Specify the name of the schema. The schema name must be the same as your Oracle username.

create_table_statement

Specify a CREATE TABLE statement to be issued as part of this CREATE SCHEMA statement. Do not end this statement with a semicolon (or other terminator character).

See Also: CREATE TABLE on page 10-7

create_view_statement

Specify a CREATE VIEW statement to be issued as part of this CREATE SCHEMA statement. Do not end this statement with a semicolon (or other terminator character).

See Also: CREATE VIEW on page 10-105

grant_statement

Specify a GRANT *object_privileges* statement to be issued as part of this CREATE SCHEMA statement. Do not end this statement with a semicolon (or other terminator character).

See Also: GRANT on page 11-31

The CREATE SCHEMA statement supports the syntax of these statements only as defined by standard SQL, rather than the complete syntax supported by Oracle.

The order in which you list the CREATE TABLE, CREATE VIEW, and GRANT statements is unimportant. The statements within a CREATE SCHEMA statement can reference existing objects or objects you create in other statements within the same CREATE SCHEMA statement.

Restriction: The syntax of the *parallel_clause* is allowed for a CREATE TABLE statement in CREATE SCHEMA, but parallelism is **not** used when creating the objects.

See Also: the parallel_clause of CREATE TABLE on page 10-40

Example

CREATE SCHEMA Example The following statement creates a schema named blair for the user Blair, creates the table sox, creates the view red_sox, and grants SELECT privilege on the red_sox view to the user waites.

```
CREATE SCHEMA AUTHORIZATION blair
```

```
CREATE TABLE sox
   (color VARCHAR2(10)  PRIMARY KEY, quantity NUMBER)
CREATE VIEW red_sox
   AS SELECT color, quantity FROM sox WHERE color = 'RED'
GRANT select ON red_sox TO waites;
```

CREATE SEQUENCE

Purpose

Use the CREATE SEQUENCE statement to create a **sequence**, which is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values.

When a sequence number is generated, the sequence is incremented, independent of the transaction committing or rolling back. If two users concurrently increment the same sequence, the sequence numbers each user acquires may have gaps because sequence numbers are being generated by the other user. One user can never acquire the sequence number generated by another user. Once a sequence value is generated by one user, that user can continue to access that value regardless of whether the sequence is incremented by another user.

Sequence numbers are generated independently of tables, so the same sequence can be used for one or for multiple tables. It is possible that individual sequence numbers will appear to be skipped, because they were generated and used in a transaction that ultimately rolled back. Additionally, a single user may not realize that other users are drawing from the same sequence.

Once a sequence is created, you can access its values in SQL statements with the CURRVAL pseudocolumn (which returns the current value of the sequence) or the NEXTVAL pseudocolumn (which increments the sequence and returns the new value).

See Also:

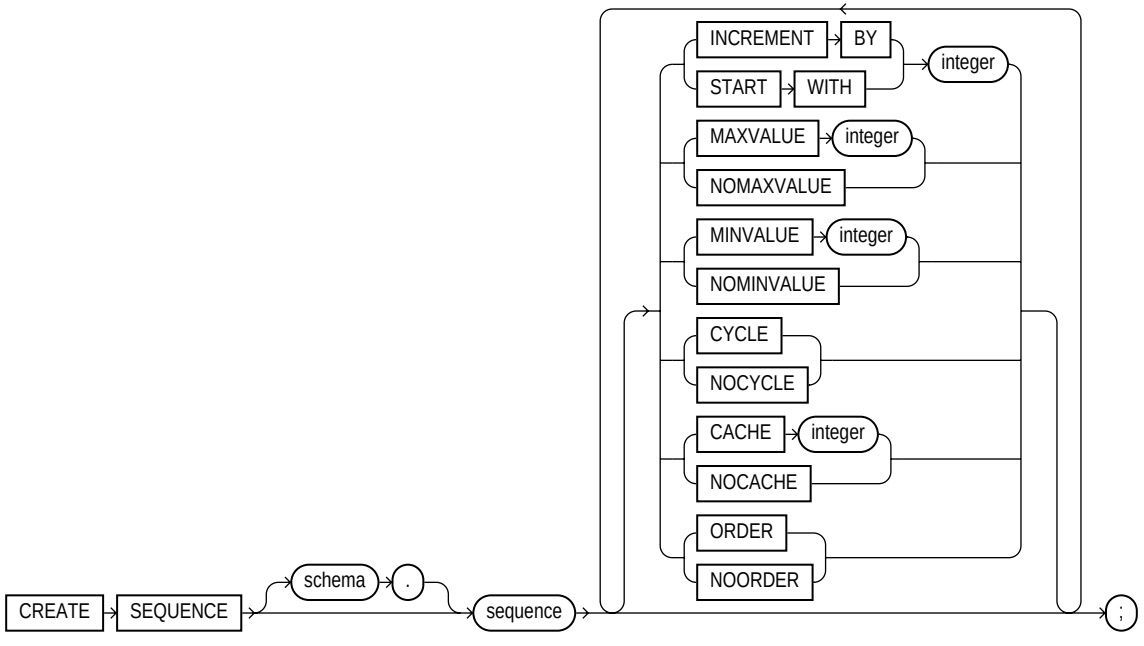
- "Pseudocolumns" on page 2-59 for more information on using the CURRVAL and NEXTVAL
- "How to Use Sequence Values" on page 2-61for information on using sequences
- ALTER SEQUENCE on page 7-103 or DROP SEQUENCE on page 11-3 for information on modifying or dropping a sequence

Prerequisites

To create a sequence in your own schema, you must have CREATE SEQUENCE privilege.

To create a sequence in another user's schema, you must have CREATE ANY SEQUENCE privilege.

Syntax



Keywords and Parameters

schema

Specify the schema to contain the sequence. If you omit *schema*, Oracle creates the sequence in your own schema.

sequence

Specify the name of the sequence to be created.

If you specify none of the following clauses, you create an ascending sequence that starts with 1 and increases by 1 with no upper limit. Specifying only INCREMENT BY -1 creates a descending sequence that starts with -1 and decreases with no lower limit.

- **To create a sequence that increments without bound**, for ascending sequences, omit the MAXVALUE parameter or specify NOMAXVALUE. For descending sequences, omit the MINVALUE parameter or specify the NOMINVALUE.
- **To create a sequence that stops at a predefined limit**, for an ascending sequence, specify a value for the MAXVALUE parameter. For a descending sequence, specify a value for the MINVALUE parameter. Also specify the NOCYCLE. Any attempt to generate a sequence number once the sequence has reached its limit results in an error.
- **To create a sequence that restarts after reaching a predefined limit**, specify values for both the MAXVALUE and MINVALUE parameters. Also specify the CYCLE. If you do not specify MINVALUE, then it defaults to NOMINVALUE (that is, the value 1).

Sequence Parameters

INCREMENT BY *integer*

Specify the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. This value can have 28 or fewer digits. The absolute of this value must be less than the difference of MAXVALUE and MINVALUE. If this value is negative, then the sequence descends. If the increment is positive, then the sequence ascends. If you omit this clause, the interval defaults to 1.

START WITH *integer*

Specify the first sequence number to be generated. Use this clause to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum. For ascending sequences, the default value is the sequence's minimum value. For descending sequences, the default value is the sequence's maximum value. This integer value can have 28 or fewer digits.

> **Note:** This value is not necessarily the value to which an ascending cycling sequence cycles after reaching its maximum or minimum value.

MAXVALUE *integer*

Specify the maximum value the sequence can generate. This integer value can have 28 or fewer digits. MAXVALUE must be equal to or greater than START WITH and must be greater than MINVALUE.

NOMAXVALUE — Specify NOMAXVALUE to indicate a maximum value of 10^{27} for an ascending sequence or -1 for a descending sequence. This is the default.

MINVALUE *integer* — Specify the sequence's minimum value. This integer value can have 28 or fewer digits. MINVALUE must be less than or equal to START WITH and must be less than MAXVALUE.

NOMINVALUE — Specify NOMINVALUE to indicate a minimum value of 1 for an ascending sequence or $-(10^{26})$ for a descending sequence. This is the default.

CYCLE — Specify CYCLE to indicate that the sequence continues to generate values after reaching either its maximum or minimum value. After an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum, it generates its maximum.

NOCYCLE — Specify NOCYCLE to indicate that the sequence cannot generate more values after reaching its maximum or minimum value. This is the default.

CACHE *integer* — Specify how many values of the sequence Oracle preallocates and keeps in memory for faster access. This integer value can have 28 or fewer digits. The minimum value for this parameter is 2. For sequences that cycle, this value must be less than the number of values in the cycle. You cannot cache more values than will fit in a given cycle of sequence numbers. Therefore, the maximum value allowed for CACHE must be less than the value determined by the following formula:

```
(CEIL (MAXVALUE - MINVALUE)) / ABS (INCREMENT)
```

If a system failure occurs, all cached sequence values that have not been used in committed DML statements are lost. The potential number of lost values is equal to the value of the CACHE parameter.

NOCACHE — Specify NOCACHE to indicate that values of the sequence are not preallocated.

If you omit both CACHE and NOCACHE, Oracle caches 20 sequence numbers by default.

ORDER	Specify ORDER to guarantee that sequence numbers are generated in order of request. You may want to use this clause if you are using the sequence numbers as timestamps. Guaranteeing order is usually not important for sequences used to generate primary keys. ORDER is necessary only to guarantee ordered generation if you are using Oracle with the Parallel Server option in parallel mode. If you are using exclusive mode, sequence numbers are always generated in order.
NOORDER	Specify NOORDER if you do not want to guarantee sequence numbers are generated in order of request. This is the default.

Example

CREATE SEQUENCE Example The following statement creates the sequence eseq:

```
CREATE SEQUENCE eseq
  INCREMENT BY 10;
```

The first reference to `eseq.nextval` returns 1. The second returns 11. Each subsequent reference will return a value 10 greater than the one previous.

10

SQL Statements: CREATE SYNONYM to DROP ROLLBACK SEGMENT

This chapter contains the following SQL statements:

- DROP FUNCTION
- DROP INDEX
- DROP INDEXTYPE
- DROP JAVA
- DROP LIBRARY
- DROP MATERIALIZED VIEW
- DROP MATERIALIZED VIEW LOG
- DROP OPERATOR
- DROP OUTLINE
- DROP PACKAGE
- DROP PROCEDURE
- DROP PROFILE
- DROP ROLE
- DROP ROLLBACK SEGMENT

CREATE SYNONYM

Purpose

Use the CREATE SYNONYM statement to create a **synonym**, which is an alternative name for a table, view, sequence, procedure, stored function, package, materialized view, Java class schema object, or another synonym.

Synonyms provide both data independence and location transparency. Synonyms permit applications to function without modification regardless of which user owns the table or view and regardless of which database holds the table or view.

Table 10–1 lists the SQL statements in which you can refer to synonyms.

Table 10–1 Using Synonyms

DML Statements	DDL Statements
SELECT	AUDIT
INSERT	NOAUDIT
UPDATE	GRANT
DELETE	REVOKE
EXPLAIN PLAN	COMMENT
LOCK TABLE	

See Also: *Oracle8i Concepts* for general information on synonyms

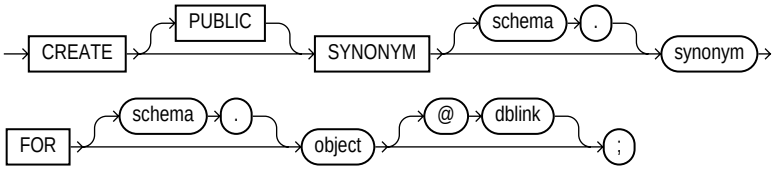
Prerequisites

To create a private synonym in your own schema, you must have CREATE SYNONYM system privilege.

To create a private synonym in another user's schema, you must have CREATE ANY SYNONYM system privilege.

To create a PUBLIC synonym, you must have CREATE PUBLIC SYNONYM system privilege.

Syntax



Keywords and Parameters

PUBLIC

Specify PUBLIC to create a public synonym. Public synonyms are accessible to all users.

Oracle uses a public synonym only when resolving references to an object if the object is not prefaced by a schema and the object is not followed by a database link.

If you omit this clause, the synonym is private and is accessible only within its schema. A private synonym name must be unique in its schema.

schema

Specify the schema to contain the synonym. If you omit *schema*, Oracle creates the synonym in your own schema. You cannot specify a schema for the synonym if you have specified PUBLIC.

synonym

Specify the name of the synonym to be created.

Caution: The functional maximum length of the synonym name is 32 bytes. Names longer than 30 bytes are permitted for Java functionality only. If you specify a name longer than 30 bytes, Oracle encrypts the name and places a representation of the encryption in the data dictionary. The actual encryption is not accessible, and you cannot use either your original specification or the data dictionary representation as the synonym name.

FOR *object*

Specify the object for which the synonym is created. If you do not qualify object with *schema*, Oracle assumes that the schema object is in your own schema. The schema object can be of the following types:

- Table or object table
- View or object view
- Sequence
- Stored procedure, function, or package
- Materialized view
- Java class schema object
- Synonym

The schema object need not currently exist and you need not have privileges to access the object.

Restrictions:

- The schema object cannot be contained in a package.

You cannot create a synonym for an object type.

dblink

You can use a complete or partial *dblink* to create a synonym for a schema object on a remote database where the object is located. If you specify *dblink* and omit *schema*, the synonym refers to an object in the schema specified by the database link. Oracle Corporation recommends that you specify the schema containing the object in the remote database.

If you omit *dblink*, Oracle assumes the object is located on the local database.

Restriction: You cannot specify *dblink* for a Java class synonym.

See Also:

- "Referring to Objects in Remote Databases" on page 2-90 for more information on referring to database links
- CREATE DATABASE LINK on page 9-28 for more information on creating database links

Examples

CREATE SYNONYM Examples To define the synonym market for the table market_research in the schema scott, issue the following statement:

```
CREATE SYNONYM market
   FOR scott.market_research;
```

To create a PUBLIC synonym for the emp table in the schema scott on the remote SALES database, you could issue the following statement:

```
CREATE PUBLIC SYNONYM emp
   FOR scott.emp@sales;
```

A synonym may have the same name as the base table, provided the base table is contained in another schema.

Resolution of Synonyms Example Oracle attempts to resolve references to objects at the schema level before resolving them at the PUBLIC synonym level. For example, assume the schemas scott and blake each contain tables named dept and the user SYSTEM creates a PUBLIC synonym named dept for blake.dept. If the user scott then issues the following statement, Oracle returns rows from scott.dept:

```
SELECT * FROM dept;
```

To retrieve rows from blake.dept, the user scott must preface dept with the schema name:

```
SELECT * FROM blake.dept;
```

If the user adam's schema does not contain an object named dept, then adam can access the dept table in blake's schema by using the public synonym dept:

```
SELECT * FROM dept;
```

CREATE TABLE

Purpose

Use the CREATE TABLE statement to create one of the following types of tables:

- A **relational table** is the basic structure to hold user data.
- An **object table** is a table that uses an object type for a column definition. An object table is a table explicitly defined to hold object instances of a particular type.

You can also create an object type and then use it in a column when creating a relational table.

Tables are created with no data unless a query is specified. You can add rows to a table with the INSERT statement. After creating a table, you can define additional columns, partitions, and integrity constraints with the ADD clause of the ALTER TABLE statement. You can change the definition of an existing column or partition with the MODIFY clause of the ALTER TABLE statement.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* and CREATE TYPE on page 10-80 for more information about creating objects

Prerequisites

To create a **relational table** in your own schema, you must have system privilege. To create a table in another user's schema, you must have CREATE ANY TABLE system privilege. Also, the owner of the schema to contain the table must have either space quota on the tablespace to contain the table or UNLIMITED TABLESPACE system privilege.

In addition to the table privileges above, to create an **object table** (or a relational table with an object type column, the owner of the table must have the EXECUTE object privilege in order to access all types referenced by the table, or you must have the EXECUTE ANY TYPE system privilege. These privileges must be granted explicitly and not acquired through a role.

Additionally, if the table owner intends to grant access to the table to other users, the owner must have been granted the EXECUTE privileges to the referenced types with the GRANT OPTION, or have the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. Without these privileges, the table owner has insufficient privileges to grant access on the table to other users.

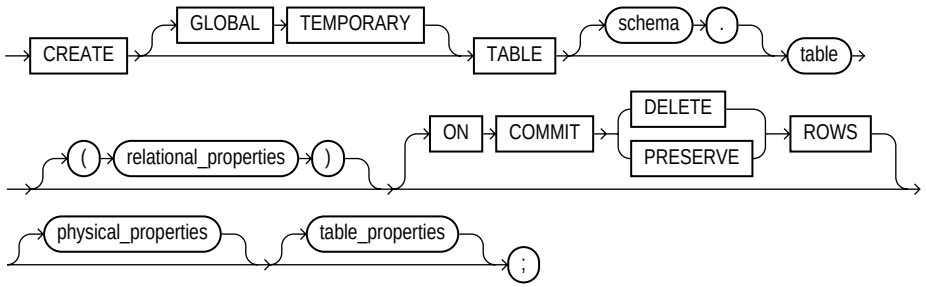
To enable a UNIQUE or PRIMARY KEY constraint, you must have the privileges necessary to create an index on the table. You need these privileges because Oracle creates an index on the columns of the unique or primary key in the schema containing the table.

See Also:

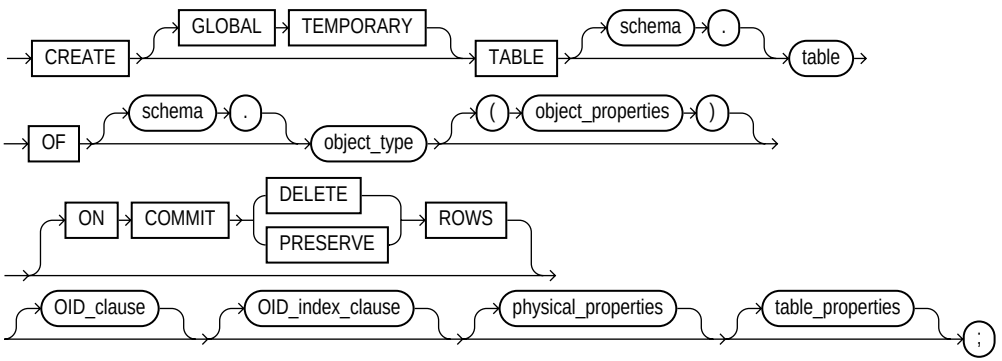
- CREATE INDEX on page 9-52
- *Oracle8i Application Developer's Guide - Fundamentals* for more information about the privileges required to create tables using types

Syntax

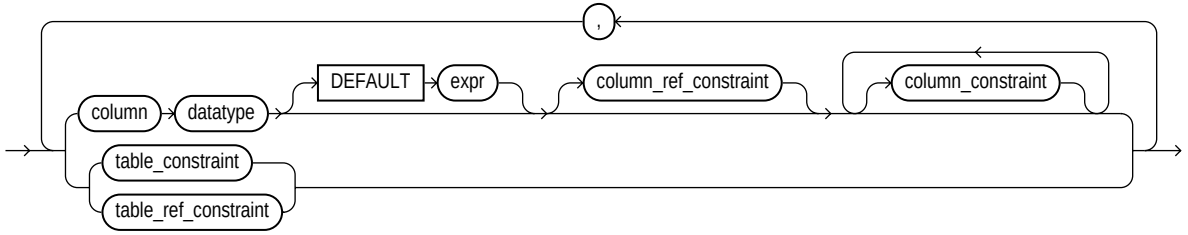
relational_table::=



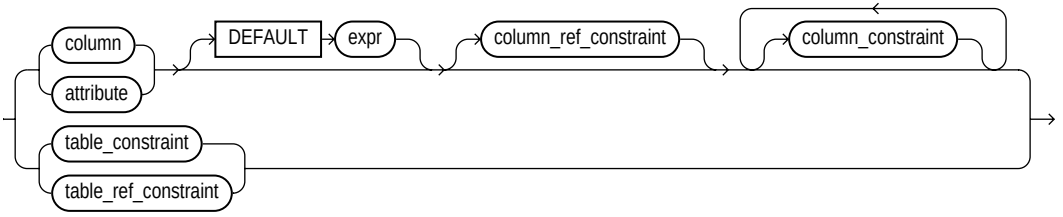
object_table::=



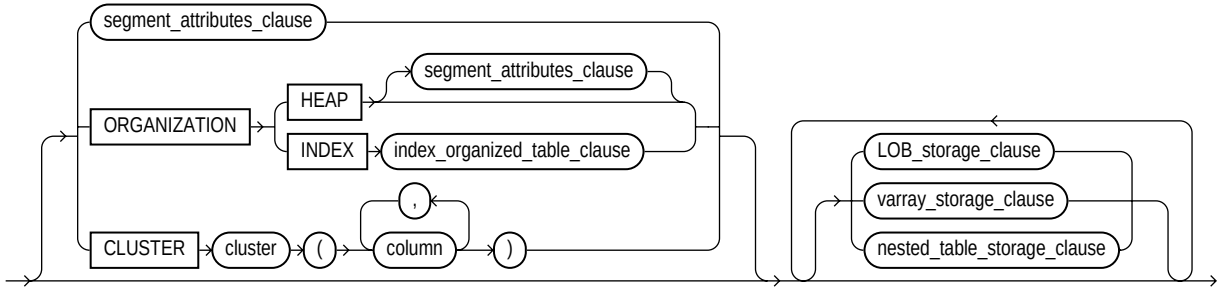
relational_properties::=



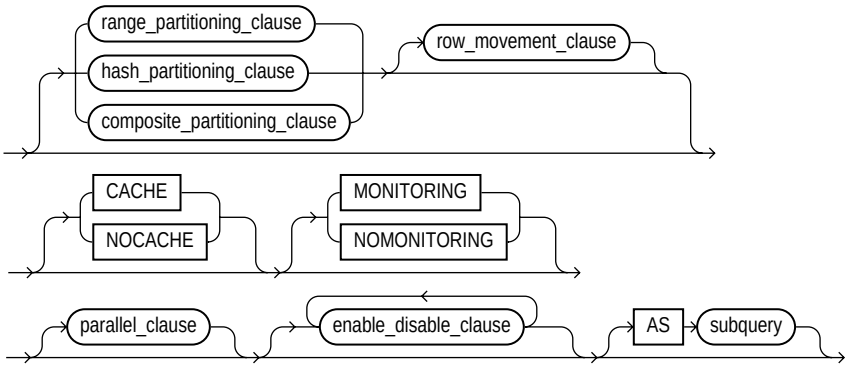
object_properties::=



physical_properties::=



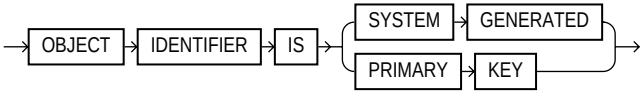
table_properties::=



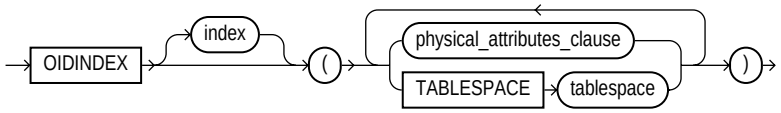
subquery::= See SELECT and subquery on page 11-88.

table_constraint, **column_constraint**, **table_ref_constraint**, **column_ref_constraint**, **constraint_state**: See the *constraint_clause* on page 8-136

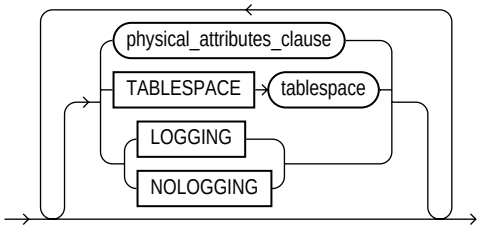
OID_clause::=



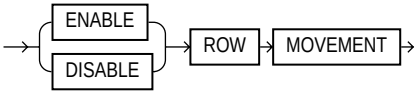
OID_index_clause::=



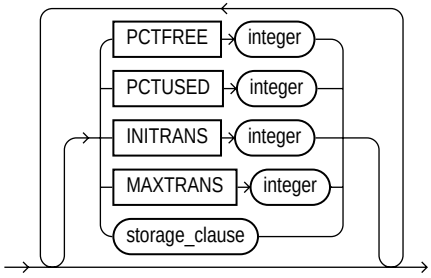
segment_attributes_clause::=



row_movement_clause::=

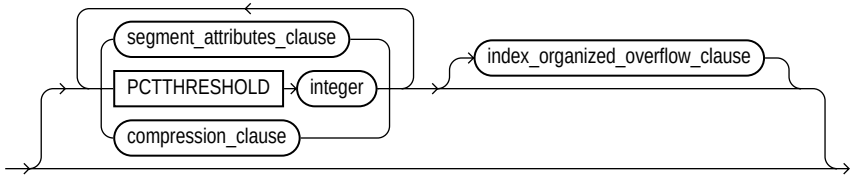


physical_attributes_clause::=

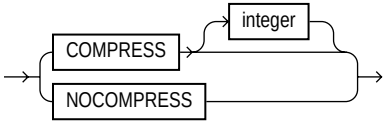


storage_clause: See the *storage_clause* on page 11-129.

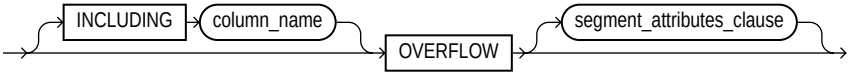
index_organized_table_clause::=



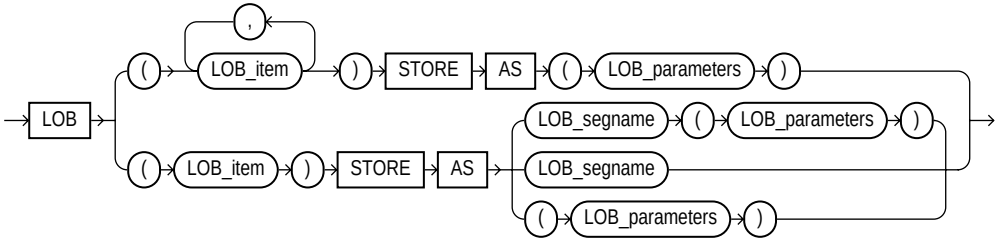
compression_clause::=



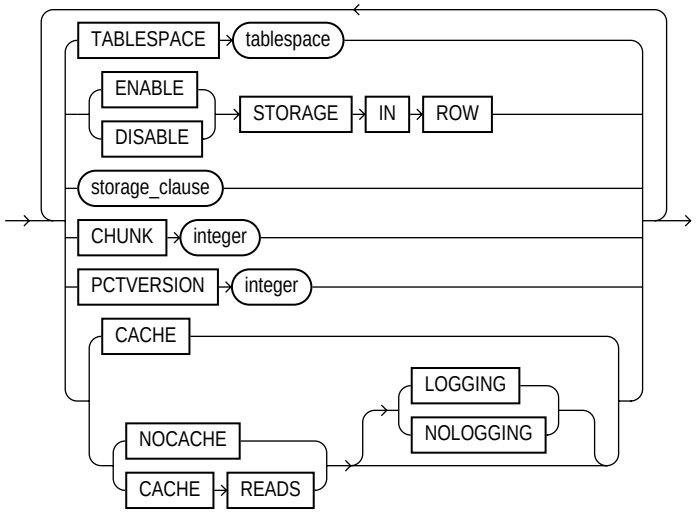
index_organized_overflow_clause::=



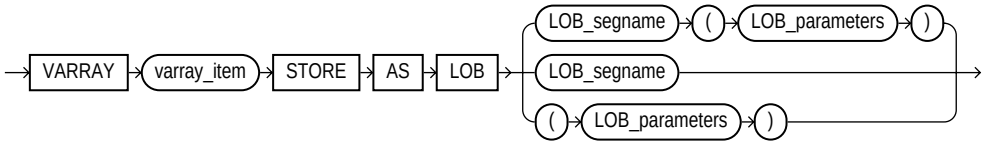
LOB_storage_clause::=



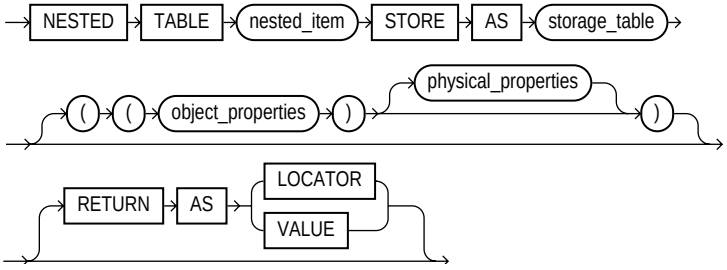
LOB_parameters::=



varray_storage_clause::=

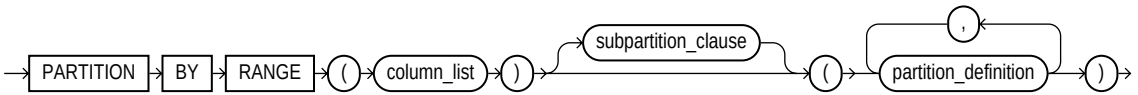


nested_table_storage_clause::=

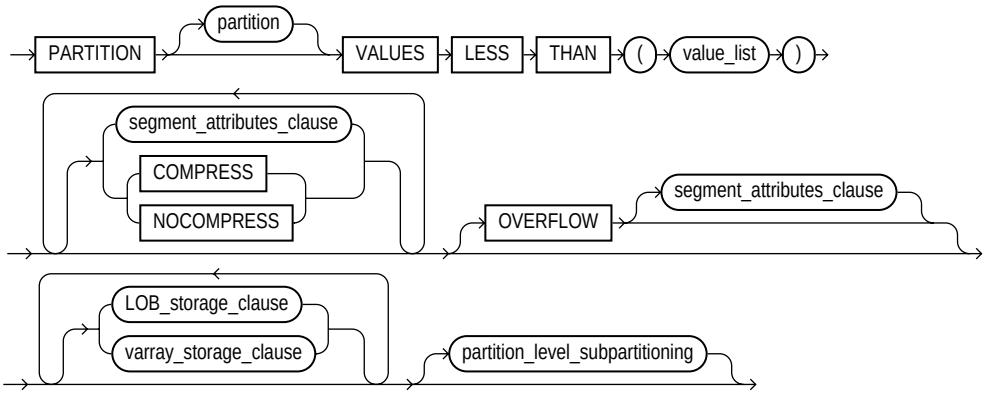


range_partitioning_clause::=

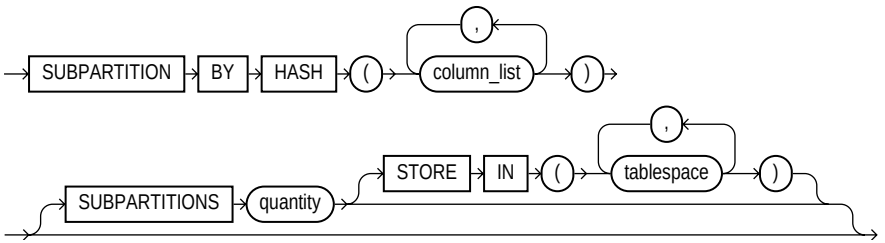
composite_partitioning_clause::=



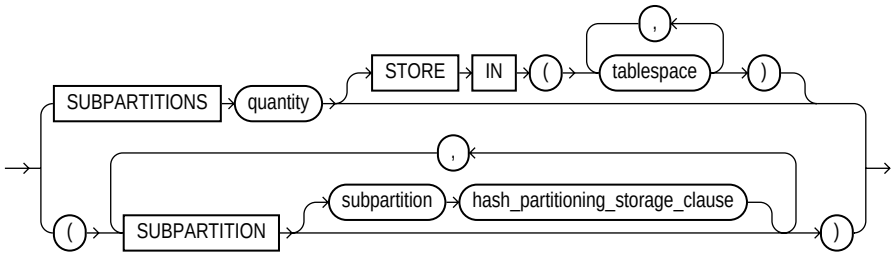
partition_definition::=



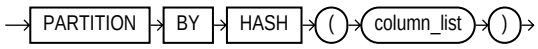
subpartition_clause::=

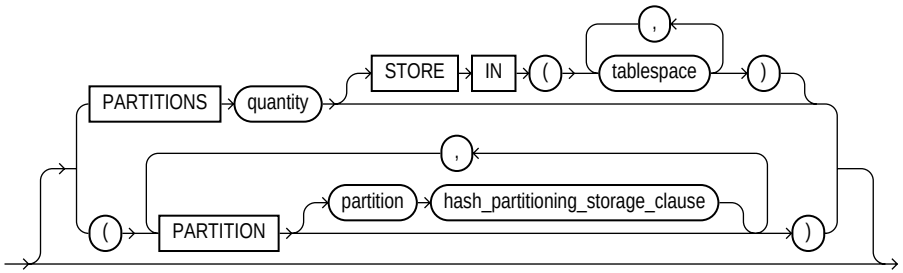


partition_level_subpartitioning::=

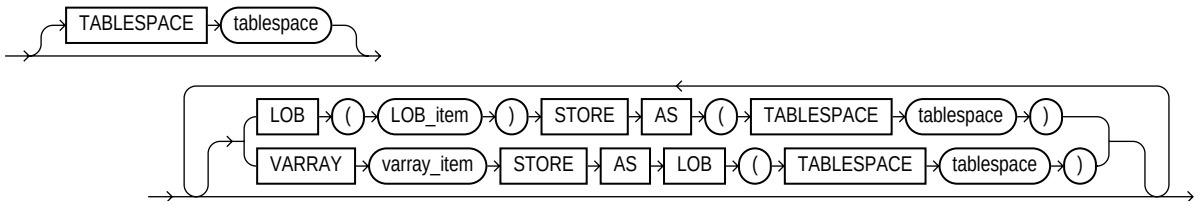


hash_partitioning_clause::=

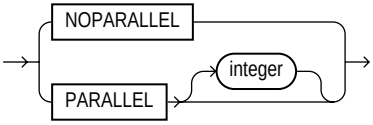




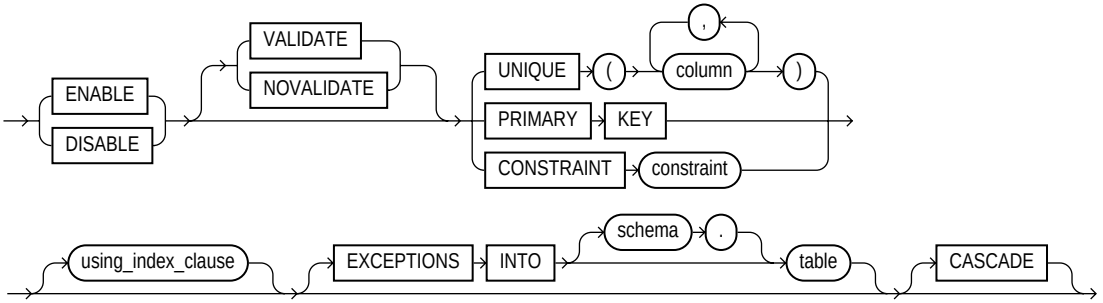
hash_partitioning_storage_clause::=



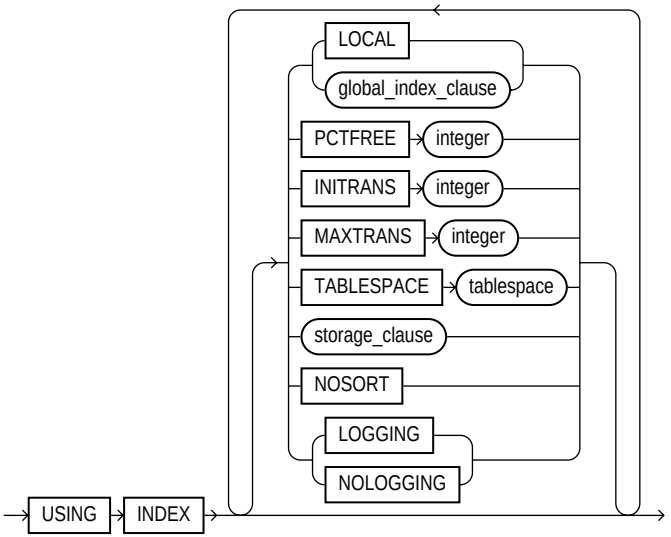
parallel_clause::=



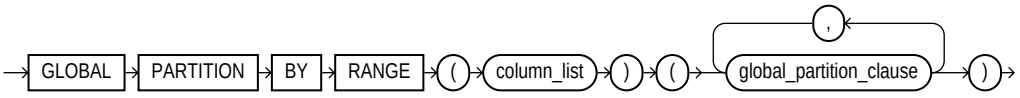
enable_disable_clause::=



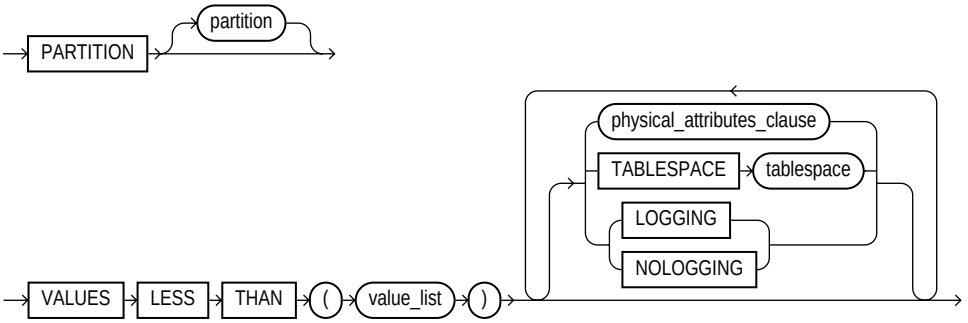
using_index_clause::=



global_index_clause::=



global_partition_clause::=



Keywords and Parameters

GLOBAL TEMPORARY

Specify GLOBAL TEMPORARY to indicate that the table is temporary and that its definition is visible to all sessions. The data in a temporary table is visible only to the session that inserts the data into the table.

A temporary table has a definition that persists the same as the definitions of regular tables, but it contains either **session-specific** or **transaction-specific** data. You specify whether the data is session- or transaction-specific with the ON COMMIT keywords (below).

See Also: *Oracle8i Concepts* for information on temporary tables

Restrictions:

- Temporary tables cannot be partitioned, index-organized, or clustered.
- You cannot specify any referential integrity (foreign key) constraints on temporary tables.
- Temporary tables cannot contain columns of nested table or varray type.

- You cannot specify the following clauses of the *LOB_storage_clause*: TABLESPACE, *storage_clause*, LOGGING or NOLOGGING, MONITORING or NOMONITORING, or *LOB_index_clause*.
- Parallel DML and parallel queries are not supported for temporary tables. (Parallel hints are ignored. Specification of the *parallel_clause* returns an error.)
- You cannot specify the *segment_attributes_clause*, *nested_table_storage_clause*, or *parallel_clause*.
- Distributed transactions are not supported for temporary tables.

schema

Specify the schema to contain the table. If you omit *schema*, Oracle creates the table in your own schema.

table

Specify the name of the table (or object table) to be created.

OF *object_type*

The OF clause lets you explicitly creates an object table of type *object_type*. The columns of an object table correspond to the top-level attributes of type *object_type*. Each row will contain an object instance, and each instance will be assigned a unique, system-generated object identifier (OID) when a row is inserted. If you omit *schema*, Oracle creates the object table in your own schema.

Objects residing in an object table are referenceable.

See Also:

- CREATE TYPE on page 10-80 for more information about creating objects
- "User-Defined Type Categories" on page 2-24, "User-Defined Functions" on page 4-128, "Expressions" on page 5-2, CREATE TYPE on page 10-80, and *Oracle8i Administrator's Guide* for more information about using REF types

relational_properties

column Specify the name of a column of the table.

If you also specify AS *subquery*, you can omit *column* and *datatype* unless you are creating an index-organized table (IOT). If you specify AS *subquery* when creating an IOT, you must specify *column*, and you must omit *datatype*.

The absolute maximum number of columns in a table is 1000. However, when you create an object table (or a relational table with columns of object, nested table, varray, or REF type), Oracle maps the columns of the user-defined types to relational columns, creating in effect "hidden columns" that count toward the 1000-column limit. For details on how Oracle calculates the total number of columns in such a table, please refer to *Oracle8i Administrator's Guide*.

datatype Specify the datatype of a column.

See Also: "Datatypes" on page 2-2for information on Oracle-supplied datatypes

Restrictions:

- You cannot specify a LOB column or a column of type VARRAY for a partitioned index-organized table (IOT). The datatypes for nonpartitioned IOTs are not restricted.
- You can specify a column of type ROWID, but Oracle does not guarantee that the values in such columns are valid rowids.

Note: You can omit *datatype* under these conditions:

- If you also specify AS *subquery*. (If you are creating an index-organized table and you specify AS *subquery*, you **must** omit the datatype.)
- If the statement also designates the column as part of a foreign key in a referential integrity constraint. (Oracle automatically assigns to the column the datatype of the corresponding column of the referenced key of the referential integrity constraint.)

DEFAULT

The DEFAULT clause lets you specify a value to be assigned to the column if a subsequent INSERT statement omits a value for the column. The datatype of the expression must match the datatype of the column. The column must also be long enough to hold this expression.

Restriction: A DEFAULT expression cannot contain references to other columns, the pseudocolumns CURRVAL, NEXTVAL, LEVEL, and ROWNUM, or date constants that are not fully specified.

See Also: "Expressions" on page 5-2 for the syntax of *expr*

table_ref_constraint

and

column_ref_constraint

These clauses let you further describe a column of type REF. The only difference between these clauses is that you specify *table_ref* from the table level, so you must identify the REF column or attribute you are defining. You specify *column_ref* after you have already identified the REF column or attribute.

See Also: *constraint_clause* on page 8-136 for syntax and description of these constraints

column_constraint

Use the *column_constraint* to define an integrity constraint as part of the column definition.

You can create UNIQUE, PRIMARY KEY, and REFERENCES constraints on scalar attributes of object type columns. You can also create NOT NULL constraints on object type columns, and CHECK constraints that reference object type columns or any attribute of an object type column.

See Also: the syntax description of *column_constraint* in the *constraint_clause* on page 8-136

table_constraint

Use the *table_constraint* to define an integrity constraint as part of the table definition.

See Also: the syntax description of *table_constraint* in the *constraint_clause* on page 8-136

Note: You must specify a PRIMARY KEY constraint for an index-organized table, and it cannot be DEFERRABLE.

---

object_properties

The properties of object tables are essentially the same as those of relational tables. However, instead of specifying columns, you specify attributes of the object.

attribute Specify the qualified column name of an item in an object.

ON COMMIT

The ON COMMIT clause is relevant only if you are creating a temporary table. This clause specifies whether the data in the temporary table persists for the duration of a transaction or a session.

DELETE ROWS Specify DELETE ROWS for a transaction-specific temporary table (this is the default). Oracle will truncate the table (delete all its rows) after each commit.

PRESERVE ROWS Specify PRESERVE ROWS for a session-specific temporary table. Oracle will truncate the table (delete all its rows) when you terminate the session.

OID_clause

The *OID_clause* lets you specify whether the object identifier (OID) of the object table should be system generated or should be based on the primary key of the table. The default is SYSTEM GENERATED.

Restrictions:

- You cannot specify OBJECT IDENTIFIER IS PRIMARY KEY unless you have already specified a PRIMARY KEY constraint for the table.
- You cannot specify this clause for a nested table.

> **Note:** A primary key OID is locally (but not necessarily globally) unique. If you require a globally unique identifier, you must ensure that the primary key is globally unique.

OID_index_clause

This clause is relevant only if you have specified the *OID_clause* as SYSTEM GENERATED. It specifies an index, and optionally its storage characteristics, on the hidden object identifier column.

index Specify the name of the index on the hidden system-generated object identifier column. If not specified, Oracle generates a name.

physical_properties

segment_attributes_clause

physical_attributes_clause The *physical_attributes_clause* lets you specify the value of the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and the storage characteristics of the table.

- For a nonpartitioned table, each parameter and storage characteristic you specify determines the actual physical attribute of the segment associated with the table.
- For partitioned tables, the value you specify for the parameter or storage characteristic is the default physical attribute of the segments associated with all partitions specified in this CREATE statement (and in subsequent ALTER TABLE ... ADD PARTITION statements), unless you explicitly override that value in the PARTITION clause of the statement that creates the partition.

PCTFREE *integer* Specify the percentage of space in each data block of the table, object table OID index, or partition reserved for future updates to the table's rows. The value of PCTFREE must be a value from 0 to 99. A value of 0 allows the entire block to be filled by inserts of new rows. The default value is 10. This value reserves 10% of each block for updates to existing rows and allows inserts of new rows to fill a maximum of 90% of each block.

PCTFREE has the same function in the PARTITION description and in the statements that create and alter clusters, indexes, materialized views, and materialized view logs. The combination of PCTFREE and PCTUSED determines whether new rows will be inserted into existing data blocks or into new blocks.

PCTUSED *integer* Specify the minimum percentage of used space that Oracle maintains for each data block of the table, object table OID index, or index-organized table overflow data segment. A block becomes a candidate for row insertion when its used space falls below PCTUSED. PCTUSED is specified as a positive integer from 0 to 99 and defaults to 40.

PCTUSED has the same function in the PARTITION description and in the statements that create and alter clusters, materialized views, and materialized view logs.

PCTUSED is not a valid table storage characteristic for an index-organized table (ORGANIZATION INDEX).

The sum of PCTFREE and PCTUSED must be equal to or less than 100. You can use PCTFREE and PCTUSED together to utilize space within a table more efficiently.

> **See Also:** *Oracle8i Performance Guide and Reference* for information on the performance effects of different values PCTUSED and PCTFREE

INITRANS *integer*

Specify the initial number of transaction entries allocated within each data block allocated to the table, object table OID index, partition, LOB index segment, or overflow data segment. This value can range from 1 to 255 and defaults to 1. In general, you should not change the INITRANS value from its default.

Each transaction that updates a block requires a transaction entry in the block. The size of a transaction entry depends on your operating system.

This parameter ensures that a minimum number of concurrent transactions can update the block and helps avoid the overhead of dynamically allocating a transaction entry.

The INITRANS parameter serves the same purpose in the PARTITION description, clusters, indexes, materialized views, and materialized view logs as in tables. The minimum and default INITRANS value for a cluster or index is 2, rather than 1.

MAXTRANS *integer*

Specify the maximum number of concurrent transactions that can update a data block allocated to the table, object table OID index, partition, LOB index segment, or index-organized overflow data segment. This limit does not apply to queries. This value can range from 1 to 255 and the default is a function of the data block size. You should not change the MAXTRANS value from its default.

If the number of concurrent transactions updating a block exceeds the INITRANS value, Oracle dynamically allocates transaction entries in the block until either the MAXTRANS value is exceeded or the block has no more free space.

The MAXTRANS parameter serves the same purpose in the PARTITION description, clusters, materialized views, and materialized view logs as in tables.

storage_clause

The *storage_clause* lets you specify storage characteristics for the table, object table OID index, partition, LOB storage, LOB index segment, or index-organized table overflow data segment. This clause has performance ramifications for large tables. Storage should be allocated to minimize dynamic allocation of additional space.

See Also: the *storage_clause* on page 11-129

TABLESPACE

Specify the tablespace in which Oracle creates the table, object table OID index, partition, LOB storage, LOB index segment, or index-organized table overflow data segment. If you omit TABLESPACE, then Oracle creates that item in the default tablespace of the owner of the schema containing the table.

For heap-organized tables with one or more LOB columns, if you omit the TABLESPACE clause for LOB storage, Oracle creates the LOB data and index segments in the tablespace where the table is created.

However, for an index-organized table with one or more LOB columns, if you omit TABLESPACE, the LOB data and index segments are created in the tablespace in which the primary key index segment of the index-organized table is created.

For nonpartitioned tables, the value specified for TABLESPACE is the actual physical attribute of the segment associated with the table. For partitioned tables, the value specified for TABLESPACE is the default physical attribute of the segments associated with all partitions specified in the CREATE statement (and on subsequent ALTER TABLE ... ADD PARTITION statements), unless you specify TABLESPACE in the PARTITION description.

See Also: CREATE TABLESPACE on page 10-56 for more information on tablespaces

LOGGING | NOLOGGING

Specify whether the creation of the table (and any indexes required because of constraints), partition, or LOB storage characteristics will be logged in the redo log file (LOGGING) or not (NOLOGGING).The logging attribute of the table is independent of that of its indexes.

This attribute also specifies that subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against the table, partition, or LOB storage are logged (LOGGING) or not logged (NOLOGGING).

For a table or table partition, if you omit this clause, the logging attribute of the table or table partition defaults to the logging attribute of the tablespace in which it resides.

For LOBs, if you omit this clause,

- If you specify CACHE, then LOGGING is used (because you cannot have CACHE NOLOGGING).
- If you specify NOCACHE or CACHE READS, the logging attribute defaults to the logging attribute of the tablespace in which it resides.

NOLOGGING does not apply to LOBs that are stored inline with row data. That is, if you specify NOLOGGING for LOBs with values less than 4000 bytes and you have not disabled STORAGE IN ROW, Oracle ignores the NOLOGGING specification and treats the LOB data the same as other table data.

For nonpartitioned tables, the value specified for LOGGING is the actual physical attribute of the segment associated with the table. For partitioned tables, the logging attribute value specified is the default physical attribute of the segments associated with all partitions specified in the CREATE statement (and in subsequent ALTER TABLE ... ADD PARTITION statements), unless you specify LOGGING | NOLOGGING in the PARTITION description.

In NOLOGGING mode, data is modified with minimal logging (to mark new extents INVALID and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not fully logged. Therefore, if you cannot afford to lose this table, you should take a backup after the NOLOGGING operation.

The size of a redo log generated for an operation in NOLOGGING mode is significantly smaller than the log generated with the LOGGING attribute set.

If the database is run in ARCHIVELOG mode, media recovery from a backup taken before the LOGGING operation restores the table. However, media recovery from a backup taken before the NOLOGGING operation does not restore the table.

See Also: *Oracle8i Concepts* and *Oracle8i Administrator's Guide* for more information about logging and parallel DML

RECOVERABLE | UNRECOVERABLE

These keywords are deprecated and have been replaced with LOGGING and NOLOGGING, respectively. Although RECOVERABLE and UNRECOVERABLE are supported for backward compatibility, Oracle Corporation strongly recommends that you use the LOGGING and NOLOGGING keywords.

Restrictions:

- You cannot specify RECOVERABLE for partitioned tables or LOB storage characteristics.
- You cannot specify UNRECOVERABLE for a partitioned or index-organized tables.
- You can specify UNRECOVERABLE only with AS *subquery*.

ORGANIZATION

The ORGANIZATION clause lets you specify the order in which the data rows of the table are stored.

HEAP — HEAP indicates that the data rows of *table* are stored in no particular order. This is the default.

INDEX — INDEX indicates that *table* is created as an index-organized table. In an index-organized table, the data rows are held in an index defined on the primary key for the table.

index_organized_table_clause

Use the *index_organized_table_clause* to instruct Oracle to maintain the table rows (both primary key column values and non-key column values) in a B*-tree index built on the primary key. Index-organized tables are therefore best suited

for primary key-based access and manipulation. An index-organized table is an alternative to

- A nonclustered table indexed on the primary key by using the CREATE INDEX statement
- A clustered table stored in an indexed cluster that has been created using the CREATE CLUSTER statement that maps the primary key for the table to the cluster key

Restrictions:

- You cannot specify a column of type ROWID for an index-organized table.
- A partitioned index-organized table cannot contain columns of LOB or varray type. (This restriction does not apply to nonpartitioned index-organized tables.)

> **Note:** You must specify a primary key for an index-organized table, because the primary key uniquely identifies a row. The primary key cannot be DEFERRABLE. Use the primary key instead of the rowid for directly accessing index-organized rows.

PCTTHRESHOLD *integer*

Specify the percentage of space reserved in the index block for an index-organized table row. All trailing columns of a row, starting with the column that causes the specified threshold to be exceeded, are stored in the overflow segment. PCTTHRESHOLD must be a value from 1 to 50. If you do not specify PCTTHRESHOLD, the default is 50.

Restrictions:

- PCTTHRESHOLD must be large enough to hold the primary key.
- You cannot specify PCTTHRESHOLD for individual partitions of an index-organized table.

See Also: the INCLUDING clause of the *index_organized_table_clause*

compression_clause

The *compression_clause* lets you enable or disable key compression.

COMPRESS	Specify COMPRESS to enable key compression, which eliminates repeated occurrence of primary key column values in index-organized tables. Use *integer* to specify the prefix length (number of prefix columns to compress). The valid range of prefix length values is from 1 to the number of primary key columns minus 1. The default prefix length is the number of primary key columns minus 1. **Restriction:** At the partition level, you can specify COMPRESS, but you cannot specify the prefix length with *integer*.
NOCOMPRESS	Specify NOCOMPRESS to disable key compression in index-organized tables. This is the default.

index_organized_overflow_clause

The *index_organized_overflow_clause* lets you instruct Oracle that index-organized table data rows exceeding the specified threshold are placed in the data segment specified in this clause.

- When you create an index-organized table, Oracle evaluates the maximum size of each column to estimate the largest possible row. If an overflow segment is needed but you have not specified OVERFLOW, Oracle raises an error and does not execute the CREATE TABLE statement. This checking function guarantees that subsequent DML operations on the index-organized table will not fail because an overflow segment is lacking.
- All physical attributes and storage characteristics you specify in this clause after the OVERFLOW keyword apply only to the overflow segment of the table. Physical attributes and storage characteristics for the index-organized table itself, default values for all its partitions, and values for individual partitions must be specified before this keyword.
- If the index-organized table contains one or more LOB columns, the LOBs will be stored out-of-line unless you specify OVERFLOW, even if they would otherwise be small enough be to stored inline.

`INCLUDING` *`column_name`* Specify a column at which to divide an index-organized table row into index and overflow portions. The primary key columns are always stored in the index. *`column_name`* can be either the last primary-key column or any non-primary-key column. All non-primary-key columns that follow *`column_name`* are stored in the overflow data segment.

Restriction: You cannot specify this clause for individual partitions of an index-organized table.

Note: If an attempt to divide a row at *`column_name`* causes the size of the index portion of the row to exceed the `PCTTHRESHOLD` value (either specified or default), Oracle breaks up the row based on the `PCTTHRESHOLD` value.

CLUSTER

The `CLUSTER` clause indicates that the table is to be part of *`cluster`*. The columns listed in this clause are the table columns that correspond to the cluster's columns. Generally, the cluster columns of a table are the column or columns that make up its primary key or a portion of its primary key.

See Also: CREATE CLUSTER on page 9-3

Specify one column from the table for each column in the cluster key. The columns are matched by position, not by name.

A clustered table uses the cluster's space allocation. Therefore, do not use the `PCTFREE`, `PCTUSED`, `INITRANS`, or `MAXTRANS` parameters, the `TABLESPACE` clause, or the *`storage_clause`* with the `CLUSTER` clause.

Restrictions: Object tables and tables containing LOB columns cannot be part of a cluster.

`LOB_storage_clause`

The *`LOB_storage_clause`* lets you specify the storage attributes of LOB data segments.

- For a nonpartitioned table (that is, when specified in the *`physical_properties`* clause without any of the partitioning clauses), this clause specifies the table's storage attributes of LOB data segments.

- For a partitioned table specified at the table level (that is, when specified in the *physical_properties* clause along with one of the partitioning clauses), this clause specifies the default storage attributes for LOB data segments associated with each partition or subpartition. These storage attributes apply to all partitions or subpartitions unless overridden by a *LOB_storage_clause* at the partition or subpartition level.
- For an individual partition of a partitioned table (that is, when specified as part of a *partition_definition*), this clause specifies the storage attributes of the data segments of that partition or the default storage attributes of any subpartitions of this partition. A partition-level *LOB_storage_clause* overrides a table-level *LOB_storage_clause*.
- For an individual subpartition of a partitioned table (that is, when specified as part of a *subpartition_clause*), this clause specifies the storage attributes of the data segments of this subpartition. A subpartition-level *LOB_storage_clause* overrides both partition-level and table-level *LOB_storage_clauses*.

Restriction: You cannot specify the *LOB_index_clause* if *table* is partitioned.

See Also:

- *Oracle8i Application Developer's Guide - Large Objects (LOBs)* for detailed information about LOBs, including guidelines for creating gigabyte LOBs
- "LOB Column Example" on page 10-50

LOB_item — Specify the LOB column name or LOB object attribute for which you are explicitly defining tablespace and storage characteristics that are different from those of the table. Oracle automatically creates a system-managed index for each *LOB_item* you create.

LOB_segname — Specify the name of the LOB data segment. You cannot use *LOB_segname* if you specify more than one *LOB_item*.

LOB_parameters — The *LOB_parameters* clause lets you specify various elements of LOB storage.

ENABLE STORAGE IN ROW — If you enable storage in row, the LOB value is stored in the row (inline) if its length is less than approximately 4000 bytes minus system control information. This is the default.

Restriction: For an index-organized table, you cannot specify this parameter unless you have specified an OVERFLOW segment in the *index_organized_table_clause*.

DISABLE STORAGE IN ROW — If you disable storage in row, the LOB value is stored outside of the row regardless of the length of the LOB value.

The LOB locator is always stored in the row regardless of where the LOB value is stored. You cannot change the value of STORAGE IN ROW once it is set except by moving the table.

> **See Also:** *move_table_clause* of ALTER TABLE on page 8-25.

CHUNK *integer* — Specify the number of bytes to be allocated for LOB manipulation. If *integer* is not a multiple of the database block size, Oracle rounds up (in bytes) to the next multiple. For example, if the database block size is 2048 and *integer* is 2050, Oracle allocates 4096 bytes (2 blocks). The maximum value is 32768 (32K), which is the largest Oracle block size allowed. The default CHUNK size is one Oracle database block.

You cannot change the value of CHUNK once it is set.

> **Note:** The value of CHUNK must be less than or equal to the value of NEXT (either the default value or that specified in the *storage_clause*). If CHUNK exceeds the value of NEXT, Oracle returns an error.

PCTVERSION *integer* — Specify the maximum percentage of overall LOB storage space used for creating new versions of the LOB. The default value is 10, meaning that older versions of the LOB data are not overwritten until 10% of the overall LOB storage space is used.

LOB_index_clause — This clause is deprecated as of Oracle8*i*. Oracle generates an index for each LOB column. Oracle names and manages the LOB indexes internally.

Although it is still possible for you to specify this clause, Oracle Corporation strongly recommends that you no longer do so. In any event, do not put the LOB index in a different tablespace from the LOB data.

See Also: *Oracle8i Migration* for information on how Oracle manages LOB indexes in tables migrated from earlier versions

varray_storage_clause

The *varray_storage_clause* lets you specify separate storage characteristics for the LOB in which a varray will be stored. In addition, if you specify this clause, Oracle will always store the varray in a LOB, even if it is small enough to be stored inline.

- For a nonpartitioned table (that is, when specified in the *physical_properties* clause without any of the partitioning clauses), this clause specifies the storage attributes of the varray's LOB data segments.
- For a partitioned table specified at the table level (that is, when specified in the *physical_properties* clause along with one of the partitioning clauses), this clause specifies the default storage attributes for the varray's LOB data segments associated with each partition (or its subpartitions, if any).
- For an individual partition of a partitioned table (that is, when specified as part of a *partition_definition*), this clause specifies the storage attributes of the varray's LOB data segments of that partition or the default storage attributes of the varray's LOB data segments of any subpartitions of this partition. A partition-level *varray_storage_clause* overrides a table-level *varray_storage_clause*.
- For an individual subpartition of a partitioned table (that is, when specified as part of a *subpartition_clause*), this clause specifies the storage attributes of the varray's data segments of this subpartition. A subpartition-level

varray_storage_clause overrides both partition-level and table-level *varray_storage_clauses*.

Restriction: You cannot specify the TABLESPACE parameter of *LOB_parameters* as part of this clause. The LOB tablespace for a varray defaults to the containing table's tablespace.

nested_table_storage_clause

The *nested_table_storage_clause* lets you to specify separate storage characteristics for a nested table, which in turn enables you to define the nested table as an index-organized table. The storage table is created in the same tablespace as its parent table (using the default storage characteristics) and stores the nested table values of the column for which it was created.

You must include this clause when creating a table with columns or column attributes whose type is a nested table. (Clauses within this clause that function the same way they function for parent object tables are not repeated here.)

Restrictions:

- You cannot specify this clause for a temporary table.
- You cannot specify the *OID_clause*.
- You cannot specify TABLESPACE (as part of the *segment_attributes_clause*) for a nested table. The tablespace is always that of the parent table.
- At create time, you cannot specify (as part of *object_properties*) a *table_ref_constraint*, *column_ref_constraint*, or referential constraint for the attributes of a nested table. However, you can modify a nested table to add such constraints using ALTER TABLE.
- You cannot query or perform DML statements on the storage table directly, but you can modify the nested table column storage characteristics by using the name of storage table in an ALTER TABLE statement.

> **See Also:** ALTER TABLE on page 8-2 for information about modifying nested table column storage characteristics

nested_item Specify the name of a column (or a top-level attribute of the table's object type) whose type is a nested table.

storage_table Specify the name of the table where the rows of *nested_item* reside. For a nonpartitioned table, the storage table is created in the same schema and the same tablespace as the parent table. For a partitioned table, the storage table is created in the default tablespace of the schema.

Restriction: You cannot partition the storage table of a nested table.

You cannot query or perform DML statements on *storage_table* directly, but you can modify its storage characteristics by specifying its name in an ALTER TABLE statement.

See Also: ALTER TABLE on page 8-2 for information about modifying nested table column storage characteristics

RETURN AS Specify what Oracle returns as the result of a query.

- VALUE returns a copy of the nested table itself.
- LOCATOR returns a collection locator to the copy of the nested table.

Note: The locator is scoped to the session and cannot be used across sessions. Unlike a LOB locator, the collection locator cannot be used to modify the collection instance.

If you do not specify the *segment_attributes_clause* or the *LOB_storage_clause*, the nested table is heap organized and is created with default storage characteristics.

table_properties

range_partitioning_clause

Use the *range_partitioning_clause* to partition the table on ranges of values from *column_list*. For an index-organized table, *column_list* must be a subset of the primary key columns of the table.

column_list Specify an ordered list of columns used to determine into which partition a row belongs (the **partitioning key**).

Restriction: The columns in *column_list* can be of any built-in datatype except ROWID, LONG, or LOB.

hash_partitioning_clause

Use the *hash_partitioning_clause* to specify that the table is to be partitioned using the hash method. Oracle assigns rows to partitions using a hash function on values found in columns designated as the partitioning key.

> **See Also:** *Oracle8i Concepts* for more information on hash partitioning

column_list Specify an ordered list of columns used to determine into which partition a row belongs (**the partitioning key**).

Restrictions:

- You cannot specify more than 16 columns in *column_list*.
- The *column_list* cannot contain the ROWID or UROWID pseudocolumns.
- The columns in *column_list* can be of any built-in datatype except ROWID, LONG, or LOB.

> **Note:** If your enterprise has or will have databases using different character sets, use caution when partitioning on character columns. The sort sequence of characters is not identical in all character sets.

> **See Also:** *Oracle8i National Language Support Guide* for more information on character set support

You can specify hash partitioning in one of two ways:

- You can specify the number of partitions. In this case, Oracle assigns partition names of the form SYS_P*nnn*. The STORE IN clause specifies one or more tablespaces where the hash partitions are to be stored. The number of tablespaces does not have to equal the number of partitions. If the number of partitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.
- Alternatively, you can specify individual partitions by name. The TABLESPACE clause specifies where the partition should be stored.

Note: The only attribute you can specify for hash partitions (or subpartitions) is TABLESPACE. Hash partitions inherit all other attributes from table-level defaults. Hash subpartitions inherit any attributes specified at the partition level, and inherit all other attributes from the table-level defaults.

Tablespace storage specified at the table level is overridden by tablespace storage specified at the partition level, which in turn is overridden by tablespace storage specified at the subpartition level.

composite_partitioning_clause

Use the composite_partitioning_clause to first partition *table* by range, and then partition the partitions further into hash subpartitions. This combination of range partitioning and hash subpartitioning is called **composite partitioning**.

subpartition_clause — Use the *subpartition_clause* to indicate that Oracle should subpartition by hash each partition in *table*. The subpartitioning *column_list* is unrelated to the partitioning key, but is subject to the same restrictions.

SUBPARTITIONS *quantity* — Specify the default number of subpartitions in each partition of *table*, and optionally one or more tablespaces in which they are to be stored.

The default value is 1. If you do not specify the *subpartition_clause* here, Oracle will create each partition with one hash subpartition unless you subsequently specify the *partition_level_hash_subpartitioning* clause.

partition_definition

PARTITION *partition* — Specify the physical partition attributes. If *partition* is omitted, Oracle generates a name with the form SYS_P*n* for the partition. The *partition* must conform to the rules for naming schema objects and their part as described in "Schema Object Naming Rules" on page 2-83.

Notes:

- You can specify up to 64K-1 partitions and 64K-1 subpartitions. For a discussion of factors that might impose practical limits less than this number, please refer to *Oracle8i Administrator's Guide*.
- You can create a partitioned table with just one partition. Note, however, that a partitioned table with one partition is different from a nonpartitioned table. For instance, you cannot add a partition to a nonpartitioned table.

VALUES LESS THAN *value_list*

Specify the noninclusive upper bound for the current partition. *value_list* is an ordered list of literal values corresponding to *column_list* in the *partition_by_range_clause*. You can substitute the keyword MAXVALUE for any literal in *value_list*. MAXVALUE specifies a maximum value that will always sort higher than any other value, including NULL.

Specifying a value other than MAXVALUE for the highest partition bound imposes an implicit integrity constraint on the table.

See Also: *Oracle8i Concepts* for more information about partition bounds

Note: If *table* is partitioned on a DATE column, and if the NLS date format does not specify the first two digits of the year, you must use the TO_DATE function with the YYYY 4-character format mask for the year. (The RRRR format mask is not supported.) The NLS date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT.

See Also:

- *Oracle8i National Language Support Guide* for more information on these initialization parameters

- "Partitioned Table Example" on page 10-51

LOB_storage_clause The *LOB_storage_clause* lets you specify LOB storage characteristics for one or more LOB items in this partition. If you do not specify the *LOB_storage_clause* for a LOB item, Oracle generates a name for each LOB data partition. The system-generated names for LOB data and LOB index partitions take the form SYS_LOB_P*n* and SYS_IL_P*n*, respectively, where P stands for "partition" and *n* is a system-generated number.

varray_storage_clause The *varray_storage_clause* lets you specify storage characteristics for one or more varray items in this partition.

partition_level_subpartitioning The *partition_level_subpartitioning* clause lets you specify hash subpartitions for *partition*. This clause overrides the default settings established in the *subpartition_clause*.

Restriction: You can specify this clause only for a composite-partitioned table.

- You can specify individual subpartitions by name, and optionally the tablespace where each should be stored, or
- You can specify the number of subpartitions (and optionally one or more tablespaces where they are to be stored). In this case, Oracle assigns subpartition names of the form SYS_SUBPnnn. The number of tablespaces does not have to equal the number of subpartitions. If the number of partitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.

row_movement_clause

The *row_movement_clause* lets you specify whether a row can be moved to a different partition or subpartition because of a change to one or more of its key values during an update operation.

Restriction: You can specify this clause only for a partitioned table.

ENABLE Specify ENABLE to allow Oracle to move a row to a different partition or subpartition as the result of an update to the partitioning or subpartitioning key.

> **Caution:** Moving a row in the course of an UPDATE operation changes that row's ROWID.

DISABLE — Specify DISABLE if you want Oracle to return an error if an update to a partitioning or subpartitioning key would result in a row moving to a different partition or subpartition. This is the default.

CACHE | NOCACHE | CACHE READS

CACHE — For data that will be accessed frequently, specify CACHE to indicate that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This clause is useful for small lookup tables.

As a parameter in the *LOB_storage_clause*, CACHE specifies that Oracle places the LOB values in the buffer cache for faster access.

Restriction: You cannot specify CACHE for an index-organized table. However, index-organized tables implicitly provide CACHE behavior.

NOCACHE — For data that will not be accessed frequently, specify NOCACHE to indicate that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the default.

As a parameter in the *LOB_storage_clause*, NOCACHE specifies that the LOB value either is not brought into the buffer cache or is brought into the buffer cache and placed at the least recently used end of the LRU list. (The latter is the default behavior.)

Restriction: You cannot specify NOCACHE for an index-organized table.

> **Note:** NOCACHE has no effect on tables for which you specify KEEP in the *storage_clause*.

CACHE READS — CACHE READS applies only to LOB storage. It specifies that LOB values are brought into the buffer cache only during read operations, but not during write operations.

MONITORING | NOMONITORING

MONITORING — Specify MONITORING if you want modification statistics to be collected on this table. These statistics are estimates of the number of rows affected by DML statements over a particular period of time. They are available for use by the optimizer or for analysis by the user.

Restriction: You cannot specify MONITORING for a temporary table.

NOMONITORING — Specify NOMONITORING if you do not want Oracle to collect modification statistics on the table. This is the default.

Restriction: You cannot specify NOMONITORING for a temporary table.

parallel_clause

The *parallel_clause* lets you parallelize creation of the table and set the default degree of parallelism for queries and DML on the table after creation.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior.

NOPARALLEL — Specify NOPARALLEL for serial execution. This is the default.

PARALLEL — Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Notes on the *parallel_clause*

- If *table* contains any columns of LOB or user-defined object type, this statement as well as subsequent INSERT, UPDATE, or DELETE operations on *table* are executed serially without notification. Subsequent queries, however, will be executed in parallel.
- For partitioned index-organized tables, CREATE TABLE ... AS SELECT is executed serially, as are subsequent DML operations. Subsequent queries, however, will be executed in parallel.
- A parallel hint overrides the effect of the *parallel_clause*.
- DML statements and CREATE TABLE ... AS SELECT statements that reference remote objects can run in parallel. However, the "remote object" must really be on a remote database. The reference cannot loop back to an object on the local database (for example, by way of a synonym on the remote database pointing back to an object on the local database).

> **See Also:** *Oracle8i Performance Guide and Reference*, *Oracle8i Concepts*, and *Oracle8i Parallel Server Concepts* for more information on parallelized operations

enable_disable_clause

The *enable_disable_clause* lets you specify whether Oracle should apply a constraint. By default, constraints are created in ENABLE VALIDATE state.

Restrictions:

- To enable or disable any integrity constraint, you must have defined the constraint in this or a previous statement.
- You cannot enable a referential integrity constraint unless the referenced unique or primary key constraint is already enabled.

> **See Also:** *constraint_clause* on page 8-136 for more information on constraints

ENABLE

Specify ENABLE if you want the constraint to be applied to all new data in the table.

- VALIDATE additionally specifies that all old data also complies with the constraint. An enabled validated constraint guarantees that all data is and will continue to be valid.

  If any row in the table violates the integrity constraint, the constraint remains disabled and Oracle returns an error. If all rows comply with the constraint, Oracle enables the constraint. Subsequently, if new data violates the constraint, Oracle does not execute the statement and returns an error indicating the integrity constraint violation.

  If you place a primary key constraint in ENABLE VALIDATE mode, the validation process will verify that the primary key columns contain no nulls. To avoid this overhead, mark each column in the primary key NOT NULL before enabling the table's primary key constraint. (For optimal results, do this before entering data into the column.)

- NOVALIDATE ensures that all new DML operations on the constrained data comply with the constraint. This clause does not ensure that existing data in the table complies with the constraint and therefore does not require a table lock.

- If you specify neither VALIDATE nor NOVALIDATE, the default is VALIDATE.

- If you enable a unique or primary key constraint, and if no index exists on the key, Oracle creates a unique index. This index is dropped if the constraint is subsequently disabled, so Oracle rebuilds the index every time the constraint is enabled.

  To avoid rebuilding the index and eliminate redundant indexes, create new primary key and unique constraints initially disabled. Then create (or use existing) nonunique indexes to enforce the constraint. Oracle does not drop a nonunique index when the constraint is disabled, so subsequent ENABLE operations are facilitated.

- If you change the state of any single constraint from ENABLE NOVALIDATE to ENABLE VALIDATE, the operation can be performed in parallel, and does not block reads, writes, or other DDL operations.

  Restriction: You cannot enable a foreign key that references a unique or primary key that is disabled.

DISABLE

Specify DISABLE to disable the integrity constraint. Disabled integrity constraints appear in the data dictionary along with enabled constraints. If you do not specify this clause when creating a constraint, Oracle automatically enables the constraint.

- DISABLE VALIDATE disables the constraint and drops the index on the constraint, but keeps the constraint valid. This feature is most useful in data warehousing situations, where the need arises to load into a range-partitioned table a quantity of data with a distinct range of values in the unique key. In such situations, the disable validate state enables you to save space by not having an index. You can then load data from a nonpartitioned table into a partitioned table using the *exchange_partition_clause* of the ALTER TABLE statement or using SQL*Loader. All other modifications to the table (inserts, updates, and deletes) by other SQL statements are disallowed.

  If the unique key coincides with the partitioning key of the partitioned table, disabling the constraint saves overhead and has no detrimental effects. If the unique key does not coincide with the partitioning key, Oracle performs automatic table scans during the exchange to validate the constraint, which might offset the benefit of loading without an index.

- DISABLE NOVALIDATE signifies that Oracle makes no effort to maintain the constraint (because it is disabled) and cannot guarantee that the constraint is true (because it is not being validated).

  See Also: *Oracle8i Performance Guide and Reference* for information on when to use this setting

  You cannot drop a table whose primary key is being referenced by a foreign key even if the foreign key constraint is in DISABLE NOVALIDATE state. Further, the optimizer can use constraints in DISABLE NOVALIDATE state.

- If you specify neither VALIDATE nor NOVALIDATE, the default is NOVALIDATE.

- If you disable a unique or primary key constraint that is using a unique index, Oracle drops the unique index.

UNIQUE — The UNIQUE clause lets you enable or disable the unique constraint defined on the specified column or combination of columns.

PRIMARY KEY — The PRIMARY KEY clause lets you enable or disable the table's primary key constraint.

CONSTRAINT — The CONSTRAINT clause lets you enable or disable the integrity constraint named *constraint*.

using_index_clause

The *using_index_clause* lets you specify parameters for the index Oracle creates to enforce a unique or primary key constraint. Oracle gives the index the same name as the constraint.

You can choose the values of the INITRANS, MAXTRANS, TABLESPACE, STORAGE, and PCTFREE parameters for the index. These parameters are described earlier in this statement. If *table* is partitioned, you can specify a locally or globally partitioned index for the unique or primary key constraint.

> **See Also:** CREATE INDEX on page 9-52 for a description of LOCAL and the *global_index_clause*, and for a description of NOSORT and LOGGING | NOLOGGING in relation to indexes

Restriction: Use these parameters only when enabling unique and primary key constraints.

EXCEPTIONS INTO

Specify a table into which Oracle places the rowids of all rows violating the constraint. If you omit schema, Oracle assumes the exceptions table is in your own schema. If you omit this clause altogether, Oracle assumes that the table is named `EXCEPTIONS`. The exceptions table must be on your local database.

You can create the `EXCEPTIONS` table using one of these scripts:

- `UTLEXCPT.SQL` uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.)
- `UTLEXPT1.SQL` uses universal rowids, so it can accommodate rows from both conventional and index-organized tables.

If you create your own exceptions table, it must follow the format prescribed by one of these two scripts.

> **See Also:** *Oracle8i Migration* for compatibility issues related to the use of these scripts

---

> **Note:** If you are collecting exceptions from index-organized tables based on primary keys (rather than universal rowids), you must create a separate exceptions table for each index-organized table to accommodate its primary-key storage. You create multiple exceptions tables with different names by modifying and resubmitting the script.

> **See Also:**
>
> - The `DBMS_IOT` package in *Oracle8i Supplied PL/SQL Packages Reference* for information on the SQL scripts
> - *Oracle8i Performance Guide and Reference f*or information on eliminating migrated and chained rows

CASCADE

Specify `CASCADE` to disable any integrity constraints that depend on the specified integrity constraint. To disable a primary or unique key that is part of a referential integrity constraint, you must specify this clause.

Restriction: You can specify `CASCADE` only if you have specified `DISABLE`.

AS *subquery*

Specify a subquery to determine the contents of the table. The rows returned by the subquery are inserted into the table upon its creation.

For object tables, *subquery* can contain either one expression corresponding to the table type, or the number of top-level attributes of the table type.

> **See Also:** SELECT and subquery on page 11-88

Restrictions:

- The number of columns in the table must equal the number of expressions in the subquery.
- The column definitions can specify only column names, default values, and integrity constraints, not datatypes.
- You cannot define a referential integrity constraint in a CREATE TABLE statement that contains AS *subquery*. Instead, you must create the table without the constraint and then add it later with an ALTER TABLE statement.

If you specify the *parallel_clause* in this statement, Oracle will ignore any value you specify for the INITIAL storage parameter, and will instead use the value of the NEXT parameter.

> **See Also:** *storage_clause* on page 11-129 for information on these parameters

Oracle derives datatypes and lengths from the subquery. Oracle also follows the following rules for integrity constraints:

- Oracle automatically defines any NOT NULL constraints on columns in the new table that existed on the corresponding columns of the selected table if the subquery selects the column rather than an expression containing the column.
- If a CREATE TABLE statement contains both AS *subquery* and a CONSTRAINT clause or an ENABLE clause with the EXCEPTIONS INTO clause, Oracle ignores AS *subquery*. If any rows violate the constraint, Oracle does not create the table and returns an error.

If all expressions in *subquery* are columns, rather than expressions, you can omit the columns from the table definition entirely. In this case, the names of the columns of table are the same as the columns in *subquery*.

You can use *subquery* in combination with the TO_LOB function to convert the values in a LONG column in another table to LOB values in a column of the table you are creating.

See Also:

- *Oracle8i Migration* for a discussion of why and when to copy LONGs to LOBs
- "Conversion Functions" on page 4-5 for a description of how to use the TO_LOB function

---

Note: If *subquery* returns (in part or totally) the equivalent of an existing materialized view, Oracle may use the materialized view (for query rewrite) in place of one or more tables specified in *subquery*.

See Also: *Oracle8i Data Warehousing Guide* for more information on materialized views and query rewrite

---

order_by_clause The ORDER BY clause lets you order rows returned by the statements.

See also: SELECT and subquery on page 11-88 for more information on the *order_by_clause*

Note: When specified with CREATE TABLE, this clause does not necessarily order data cross the entire table. (For example, it does not order across partitions.) Specify this clause if you intend to create an index on the same key as the ORDER BY key column. Oracle will cluster data on the ORDER BY key so that it corresponds to the index key.

Examples

General Example To define the emp table owned by scott, you could issue the following statement:

```
CREATE TABLE scott.emp
   (empno     NUMBER         CONSTRAINT pk_emp PRIMARY KEY,
    ename     VARCHAR2(10)   CONSTRAINT nn_ename NOT NULL
```

```
                          CONSTRAINT upper_ename
   CHECK (ename = UPPER(ename)),
       job        VARCHAR2(9),
       mgr        NUMBER       CONSTRAINT fk_mgr
                               REFERENCES scott.emp(empno),
       hiredate  DATE          DEFAULT SYSDATE,
       sal       NUMBER(10,2)  CONSTRAINT ck_sal
   CHECK (sal > 500),
          comm      NUMBER(9,0)   DEFAULT NULL,
          deptno    NUMBER(2)     CONSTRAINT nn_deptno NOT NULL
                                  CONSTRAINT fk_deptno
                                  REFERENCES scott.dept(deptno) )
   PCTFREE 5 PCTUSED 75;
```

This table contains eight columns. The `empno` column is of datatype `NUMBER` and has an associated integrity constraint named `pk_emp`. The `hiredate` column is of datatype `DATE` and has a default value of `SYSDATE`, and so on.

This table definition specifies a `PCTFREE` of 5 and a `PCTUSED` of 75, which is appropriate for a relatively static table. The definition also defines integrity constraints on some columns of the `emp` table.

Temporary Table Example The following statement creates a temporary table `flight_schedule` for use in an automated airline reservation scheduling system. Each client has its own session and can store temporary schedules. The temporary schedules are deleted at the end of the session.

```
CREATE GLOBAL TEMPORARY TABLE flight_schedule (
   startdate DATE,
   enddate DATE,
   cost NUMBER)
   ON COMMIT PRESERVE ROWS;
```

Storage Example To define the sample table `salgrade` in the `human_resource` tablespace with a small storage capacity and limited allocation potential, issue the following statement:

```
CREATE TABLE salgrade
   ( grade  NUMBER  CONSTRAINT pk_salgrade
                    PRIMARY KEY
                    USING INDEX TABLESPACE users_a,
     losal  NUMBER,
     hisal  NUMBER )
   TABLESPACE human_resource
   STORAGE (INITIAL     6144
```

```
   NEXT        6144
   MINEXTENTS     1
   MAXEXTENTS     5 );
```

The above statement also defines a primary key constraint on the grade column and specifies that the index Oracle creates to enforce this constraint is created in the users_a tablespace.

> **See Also:** The *constraint_clause* on page 8-136 for more examples of defining integrity constraints.

PARALLEL Example The following statement creates a table using an optimum number of parallel execution servers to scan scott.emp and to populate emp_dept:

```
CREATE TABLE emp_dept
   PARALLEL
   AS SELECT * FROM scott.emp
   WHERE deptno = 10;
```

Using parallelism speeds up the creation of the table because Oracle uses parallel execution servers to create the table. After the table is created, querying the table is also faster, because the same degree of parallelism is used to access the table.

NOPARALLEL Example The following statement creates a table serially. Subsequent DML and queries on the table will also be serially executed.

```
CREATE TABLE emp_dept
   AS SELECT * FROM scott.emp
   WHERE deptno = 10;
```

ENABLE VALIDATE Example The following statement creates the dept table, defines a primary key constraint, and places it in ENABLE VALIDATE state:

```
CREATE TABLE dept
   (deptno  NUMBER (2) PRIMARY KEY,
    dname   VARCHAR2(10),
    loc     VARCHAR2(9) )
   TABLESPACE user_a;
```

DISABLE Example The following statement creates the dept table and defines a disabled primary key constraint:

```
CREATE TABLE dept
   (deptno   NUMBER (2)   PRIMARY KEY DISABLE,
```

```
    dname    VARCHAR2(10),
    loc      VARCHAR2(9) );
```

EXCEPTIONS INTO Example The following example creates the `order_exceptions` table to hold rows from an index-organized table `orders` that violate integrity constraint `check_orders`:

```
CREATE TABLE orders
   (ord_num  NUMBER PRIMARY KEY,
    ord_quantity NUMBER)
   ORGANIZATION INDEX;

EXECUTE DBMS_IOT.BUILD_EXCEPTIONS_TABLE
   ('SCOTT', 'ORDERS', 'ORDER_EXCEPTIONS');

ALTER TABLE orders
   ADD CONSTRAINT CHECK_ORDERS CHECK (ord_quantity > 0)
   EXCEPTIONS INTO ORDER_EXCEPTIONS;
```

To specify an exception table, you must have the privileges necessary to insert rows into the table. To examine the identified exceptions, you must have the privileges necessary to query the exceptions table.

See Also:

- INSERT on page 11-51
- SELECT and subquery on page 11-88 for information on the privileges necessary to insert rows into tables

Nested Table Example The following statement creates relational table `employee` with a nested table column `projects`:

```
CREATE TABLE employee
   (empno NUMBER, name CHAR(31), projects PROJ_TABLE_TYPE)
   NESTED TABLE projects STORE AS nested_proj_table(
      (PRIMARY KEY (nested_table_id, pno)) ORGANIZATION INDEX)
   RETURN AS LOCATOR;
```

LOB Column Example The following statement creates table `lob_tab` with two LOB columns and specifies the LOB storage characteristics:

```
CREATE TABLE lob_tab (col1 BLOB, col2 CLOB)
   STORAGE (INITIAL 256 NEXT 256)
   LOB (col1, col2) STORE AS
      (TABLESPACE lob_seg_ts
```

```
        STORAGE (INITIAL 6144 NEXT 6144)
        CHUNK 4000
        NOCACHE LOGGING);
```

In the example, Oracle rounds the value of CHUNK up to 4096 (the nearest multiple of the block size of 2048).

Index-Organized Table Example The following statement creates an index-organized table:

```
CREATE TABLE docindex
  ( token                 CHAR(20),
    doc_oid               INTEGER,
    token_frequency       SMALLINT,
    token_occurrence_data  VARCHAR2(512),
       CONSTRAINT pk_docindex PRIMARY KEY (token, doc_oid) )
  ORGANIZATION INDEX TABLESPACE text_collection
  PCTTHRESHOLD 20 INCLUDING token_frequency
  OVERFLOW TABLESPACE text_collection_overflow;
```

Partitioned Table Example The following statement creates a table with three partitions:

```
CREATE TABLE stock_xactions
      (stock_symbol CHAR(5),
       stock_series CHAR(1),
       num_shares NUMBER(10),
       price NUMBER(5,2),
       trade_date DATE)
   STORAGE (INITIAL 100K NEXT 50K) LOGGING
   PARTITION BY RANGE (trade_date)
     (PARTITION sx1992 VALUES LESS THAN (TO_DATE('01-JAN-1993','DD-MON-YYYY'))
        TABLESPACE ts0 NOLOGGING,
      PARTITION sx1993 VALUES LESS THAN (TO_DATE('01-JAN-1994','DD-MON-YYYY'))
        TABLESPACE ts1,
      PARTITION sx1994 VALUES LESS THAN (TO_DATE('01-JAN-1995','DD-MON-YYYY'))
        TABLESPACE ts2);
```

See Also: *Oracle8i Administrator's Guide* for information about partitioned table maintenance operations

Partitioned Table with LOB Columns Example This statement creates a partitioned table pt with two partitions p1 and p2, and three LOB columns, b, c, and d:

```
CREATE TABLE PT (A NUMBER, B BLOB, C CLOB, D CLOB)
   LOB (B,C,D) STORE AS (STORAGE (NEXT 20M))
   PARTITION BY RANGE (A)
   (PARTITION P1 VALUES LESS THAN (10) TABLESPACE TS1
      LOB (B,D) STORE AS (TABLESPACE TSA STORAGE (INITIAL 20M)),
   PARTITION P2 VALUES LESS THAN (20)
      LOB (B,C) STORE AS (TABLESPACE TSB)
   TABLESPACE TSX;
```

Partition p1 will be in tablespace `ts1`. The LOB data partitions for `b` and `d` will be in tablespace `tsa`. The LOB data partition for `c` will be in tablespace `ts1`. The storage attribute `INITIAL` is specified for LOB columns `b` and `d`; other attributes will be inherited from the default table-level specification. The default LOB storage attributes not specified at the table level will be inherited from the tablespace `tsa` for columns `b` and `d` and tablespace `ts1` for column `c`. LOB index partitions will be in the same tablespaces as the corresponding LOB data partitions. Other storage attributes will be based on values of the corresponding attributes of the LOB data partitions and default attributes of the tablespace where the index partitions reside.

Partition `p2` will be in the default tablespace `tsx`. The LOB data for `b` and `c` will be in tablespace `tsb`. The LOB data for `d` will be in tablespace `tsx`. The LOB index for columns `b` and `c` will be in tablespace `tsb`. The LOB index for column `d` will be in tablespace `tsx`.

Hash-Partitioned Table Example This statement creates a table partitioned by hash on columns containing data about chemicals. The hash partitions are stored in tablespaces `tbs1`, `tbs2`, `tbs3`, and `tbs4`:

```
CREATE TABLE exp_data (
      d DATE, temperature NUMBER, Fe2O3_concentration NUMBER,
      HCl_concentration NUMBER, Au_concentration NUMBER,
      amps NUMBER, observation VARCHAR(4000))
   PARTITION BY HASH (HCl_concentration, Au_concentration)
   PARTITIONS 32 STORE IN (tbs1, tbs2, tbs3, tbs4);
```

Composite-Partitioned Table Example This statement creates a composite-partitioned table. The range partitioning facilitates data and partition pruning by sale date. The hash subpartitioning enables subpartition elimination for queries by a specific item number. Most of the partitions consist of 8 subpartitions. However, the partition covering the slowest quarter will have 4 subpartitions, and the partition covering the busiest quarter will have 16 subpartitions.

```
CREATE TABLE sales (item INTEGER, qty INTEGER,
                    store VARCHAR(30),
```

```
                  dept NUMBER, sale_date DATE)
   PARTITION BY RANGE (sale_date)
   SUBPARTITION BY HASH(item)
   SUBPARTITIONS 8
   STORE IN (tbs1, tbs2, tbs3, tbs4, tbs5, tbs6, tbs7, tbs8)
   (PARTITION q1_1997
      VALUES LESS THAN (TO_DATE('01-apr-1997', 'dd-mon-yyyy')),
    PARTITION q2_1997
      VALUES LESS THAN (TO_DATE('01-jul-1997', 'dd-mon-yyyy')),
    PARTITION q3_1997
      VALUES LESS THAN (TO_DATE('01-oct-1997', 'dd-mon-yyyy'))
      (SUBPARTITION q3_1997_s1 TABLESPACE ts1,
       SUBPARTITION q3_1997_s2 TABLESPACE ts3,
       SUBPARTITION q3_1997_s3 TABLESPACE ts5,
       SUBPARTITION q3_1997_s4 TABLESPACE ts7),
    PARTITION q4_1997
      VALUES LESS THAN (TO_DATE('01-jan-1998', 'dd-mon-yyyy'))
      SUBPARTITIONS 16
      STORE IN (tbs1, tbs3, tbs5, tbs7, tbs8, tbs9, tbs10,
                tbs11),
    PARTITION q1_1998
      VALUES LESS THAN (TO_DATE('01-apr-1998', 'dd-mon-yyyy')));
```

Object Table Examples Consider object type `dept_t`:

```
CREATE TYPE dept_t AS OBJECT
   ( dname VARCHAR2(100),
     address VARCHAR2(200) );
```

Object table dept holds department objects of type `dept_t`:

```
CREATE TABLE dept OF dept_t;
```

The following statement creates object table `salesreps` with a user-defined object type, `salesrep_t`:

```
CREATE OR REPLACE TYPE salesrep_t AS OBJECT
  ( repId NUMBER,
    repName VARCHAR2(64));
CREATE TABLE salesreps OF salesrep_t;
```

Nested Table Example The following statement creates relational table `employee` with a nested table column `projects`:

```
CREATE TABLE employee (empno NUMBER, name CHAR(31),
   projects PROJ_TABLE_TYPE)
```

```
   NESTED TABLE projects STORE AS nested_proj_table;
```

REF Example The following example creates object type dept_t and object table dept to store instances of all departments. A table with a scoped REF is then created.

```
CREATE TYPE dept_t AS OBJECT
   ( dname   VARCHAR2(100),
     address VARCHAR2(200) );

CREATE TABLE dept OF dept_t;

CREATE TABLE emp
   ( ename   VARCHAR2(100),
     enumber NUMBER,
     edept   REF dept_t SCOPE IS dept );
```

The following statement creates a table with a REF column which has a referential constraint defined on it:

```
CREATE TABLE emp
   ( ename   VARCHAR2(100),
     enumber NUMBER,
     edept   REF dept_t REFERENCES dept);
```

User-Defined OID Example This example creates an object type and a corresponding object table whose OID is primary key based:

```
CREATE TYPE emp_t AS OBJECT (empno NUMBER, address CHAR(30));
CREATE TABLE emp OF emp_t (empno PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;
```

You can subsequently reference the emp object table in either of the following two ways:

```
CREATE TABLE dept (dno NUMBER
                   mgr_ref REF emp_t SCOPE IS emp);
CREATE TABLE dept (
    dno NUMBER,
    mgr_ref REF emp_t CONSTRAINT mgr_in_emp REFERENCES emp);
```

Constraints on Type Columns Example

```
CREATE TYPE address AS OBJECT
  ( hno    NUMBER,
    street VARCHAR2(40),
```

```
    city   VARCHAR2(20),
    zip    VARCHAR2(5),
    phone  VARCHAR2(10) );

CREATE TYPE person AS OBJECT
  ( name        VARCHAR2(40),
    dateofbirth DATE,
    homeaddress address,
    manager     REF person );

CREATE TABLE persons OF person
  ( homeaddress NOT NULL
      UNIQUE (homeaddress.phone),
      CHECK (homeaddress.zip IS NOT NULL),
      CHECK (homeaddress.city <> 'San Francisco') );
```

PARALLEL Example The following statement creates a table using 10 parallel execution servers, 5 to scan `scott.emp` and another 5 to populate `emp_dept`:

```
CREATE TABLE emp_dept
   PARALLEL (5)
   AS SELECT * FROM scott.emp
   WHERE deptno = 10;
```

CREATE TABLESPACE

Purpose

Use the CREATE TABLESPACE statement to create a **tablespace**, which is an allocation of space in the database that can contain persistent schema objects.

When you create a tablespace, it is initially a read-write tablespace. You can subsequently use the ALTER TABLESPACE statement to take the tablespace offline or online, add datafiles to it, or make it a read-only tablespace.

You can also drop a tablespace from the database with the DROP TABLESPACE statement.

You can use the CREATE TEMPORARY TABLESPACE statement to create tablespaces that contain schema objects only for the duration of a session.

See Also:

- *Oracle8i Concepts* for information on tablespaces
- ALTER TABLESPACE on page 8-67 for information on modifying tablespaces
- DROP TABLESPACE on page 11-10 for information on dropping tablespaces
- CREATE TEMPORARY TABLESPACE on page 10-63

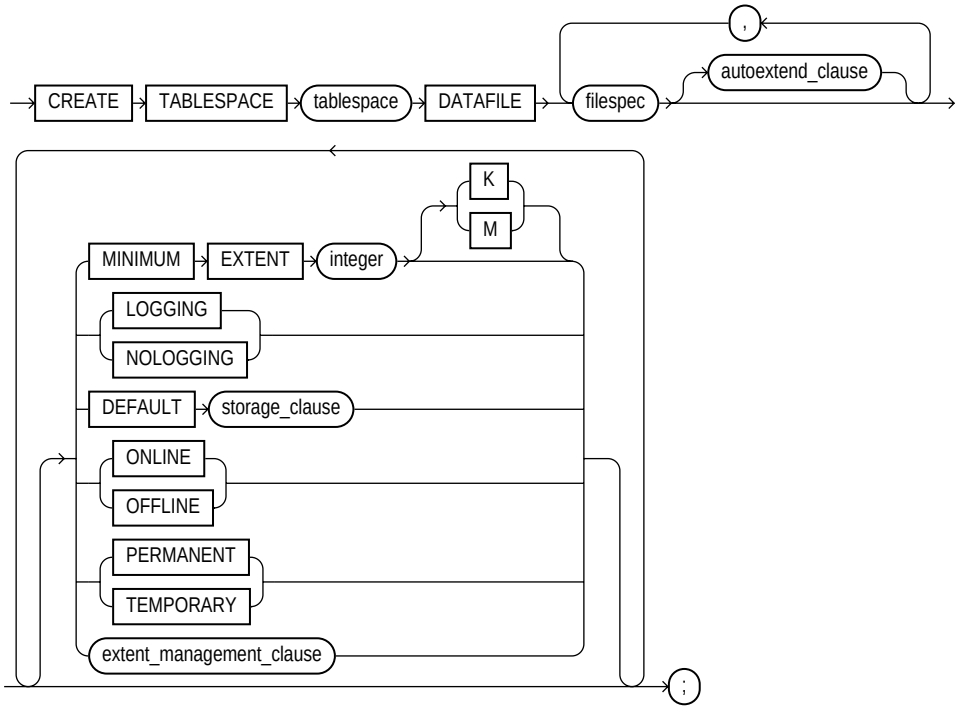
Prerequisites

You must have CREATE TABLESPACE system privilege. Also, the SYSTEM tablespace must contain at least two rollback segments including the SYSTEM rollback segment.

Before you can create a tablespace, you must create a database to contain it, and the database must be open.

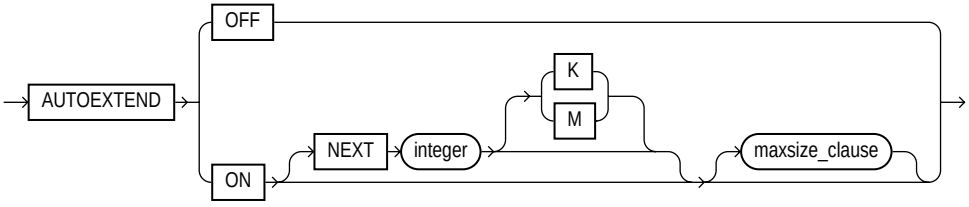
See Also: CREATE DATABASE on page 9-21

Syntax

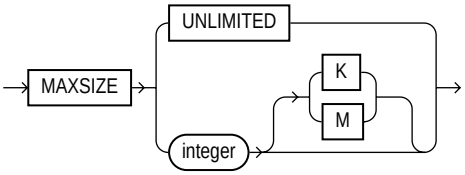


filespec: See filespec on page 11-27.

autoextend_clause::=

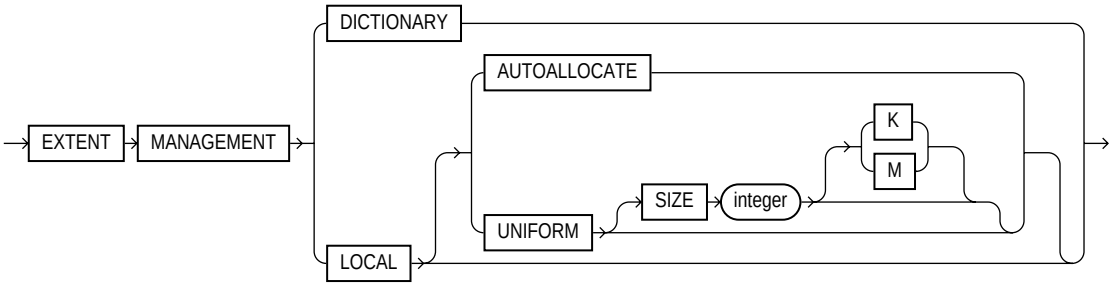


maxsize_clause::=



storage_clause: See storage_clause on page 11-129.

extent_management_clause::=



Keywords and Parameters

tablespace

Specify the name of the tablespace to be created.

DATAFILE *filespec*

Specify the datafile or files to make up the tablespace.

Note: For operating systems that support raw devices, the REUSE keyword of *filespec* has no meaning when specifying a raw device as a datafile. Such a CREATE TABLESPACE statement will succeed whether or not you specify REUSE.

See Also: filespec on page 11-27

autoextend_clause

Use the *autoextend_clause* to enable or disable the automatic extension of the datafile.

OFF	Specify OFF to disable autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER TABLESPACE AUTOEXTEND statements.
ON	Specify ON to enable autoextend.
NEXT *integer*	Specify the disk space to allocate to the datafile when more extents are required.
maxsize_clause	The *maxsize_clause* lets you specify the maximum disk space allowed for allocation to the datafile. ■ *integer*: Specify in bytes the maximum disk space allowed for allocation to the tempfile. Use K or M to specify this space in kilobytes or megabytes. ■ UNLIMITED: Specify UNLIMITED to set no limit on allocating disk space to the datafile.

MINIMUM EXTENT *integer*

Specify the minimum size of an extent in the tablespace. This clause lets you control free space fragmentation in the tablespace by ensuring that every used or free extent size in a tablespace is at least as large as, and is a multiple of, *integer*.

Note: This clause is not relevant for a dictionary-managed temporary tablespace.

See Also: *Oracle8i Concepts* for more information about using MINIMUM EXTENT to control fragmentation

LOGGING | NOLOGGING

Specify the default logging attributes of all tables, indexes, and partitions within the tablespace. LOGGING is the default.

The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

Only the following operations support the NOLOGGING mode:

- **DML**: direct-load INSERT (serial or parallel), Direct Loader (SQL*Loader)
- **DDL**: CREATE TABLE ... AS SELECT, CREATE INDEX, ALTER INDEX ... REBUILD, ALTER INDEX ... REBUILD PARTITION, ALTER INDEX ... SPLIT PARTITION, ALTER TABLE ... SPLIT PARTITION, and ALTER TABLE ... MOVE PARTITION

In NOLOGGING mode, data is modified with minimal logging (to mark new extents INVALID and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not logged. Therefore, if you cannot afford to lose the object, you should take a backup after the NOLOGGING operation.

DEFAULT *storage_clause*

Specify the default storage parameters for all objects created in the tablespace. For a dictionary-managed temporary tablespace, Oracle considers only the NEXT parameter of the *storage_clause*.

> **See Also:** *storage_clause* on page 11-129 for information on storage parameters

ONLINE | OFFLINE

ONLINE — Specify ONLINE to make the tablespace available immediately after creation to users who have been granted access to the tablespace. This is the default.

OFFLINE — Specify OFFLINE to make the tablespace unavailable immediately after creation.

The data dictionary view DBA_TABLESPACES indicates whether each tablespace is online or offline.

PERMANENT | TEMPORARY

PERMANENT — Specify PERMANENT if the tablespace will be used to hold permanent objects. This is the default.

TEMPORARY Specify TEMPORARY if the tablespace will be used only to hold temporary objects, for example, segments used by implicit sorts to handle ORDER BY clauses.

Restriction: If you specify TEMPORARY, you cannot specify EXTENT MANAGEMENT LOCAL.

extent_management_clause

The *extent_management_clause* lets you specify how the extents of the tablespace will be managed.

Note: Once you have specified extent management with this clause, you can change extent management only by migrating the tablespace.

DICTIONARY Specify DICTIONARY if you want the tablespace to be managed using dictionary tables. This is the default.

LOCAL Specify LOCAL if you want the tablespace to be locally managed. Locally managed tablespaces have some part of the tablespace set aside for a bitmap.

- AUTOALLOCATE specifies that the tablespace is system managed. Users cannot specify an extent size.
- UNIFORM specifies that the tablespace is managed with uniform extents of SIZE bytes. Use *K* or *M* to specify the extent size in kilobytes or megabytes. The default SIZE is 1 megabyte.

  See Also: *Oracle8i Concepts* for a discussion of locally managed tablespaces

If you do not specify either AUTOALLOCATE or UNIFORM, then AUTOALLOCATE is the default.

Restriction: If you specify LOCAL, you cannot specify DEFAULT *storage_clause*, MINIMUM EXTENT, or TEMPORARY.

See Also: *Oracle8i Migration* for information on changing extent management by migrating tablespaces

Examples

DEFAULT Storage Example This statement creates a tablespace named tabspace_2 with one datafile:

```
CREATE TABLESPACE tabspace_2
   DATAFILE 'diska:tabspace_file2.dat' SIZE 20M
   DEFAULT STORAGE (INITIAL 10K NEXT 50K
                    MINEXTENTS 1 MAXEXTENTS 999)
   ONLINE;
```

AUTOEXTEND Example This statement creates a tablespace named tabspace_3 with one datafile. When more space is required, 50 kilobyte extents will be added up to a maximum size of 10 megabytes:

```
CREATE TABLESPACE tabspace_5
   DATAFILE 'diskb:tabspace_file3.dat' SIZE 500K REUSE
   AUTOEXTEND ON NEXT 500K MAXSIZE 10M;
```

MINIMUM EXTENT Example This statement creates tablespace tabspace_5 with one datafile and allocates every extent as a multiple of 64K:

```
CREATE TABLESPACE tabspace_3
   DATAFILE 'tabspace_file5.dbf' SIZE 2M
   MINIMUM EXTENT 64K
   DEFAULT STORAGE (INITIAL 128K NEXT 128K)
   LOGGING;
```

Locally Managed Example In the following statement, we assume that the database block size is 2K.

```
CREATE TABLESPACE tbs_1 DATAFILE 'file_1.f' SIZE 10M
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 128K;
```

This statement creates a locally managed tablespace in which every extent is 128K and each bit in the bit map describes 64 blocks.

CREATE TEMPORARY TABLESPACE

Purpose

Use the CREATE TEMPORARY TABLESPACE statement to create a **temporary tablespace**, which is an allocation of space in the database that can contain schema objects for the duration of a session.

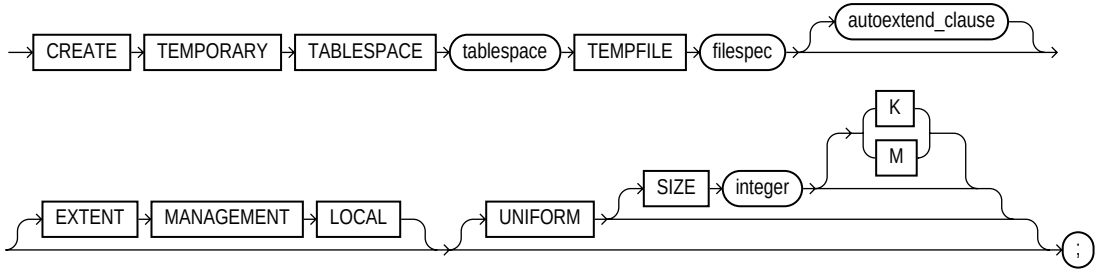
To create a tablespace to contain persistent schema objects, use the CREATE TABLESPACE statement.

See Also: CREATE TABLESPACE on page 10-56

Prerequisites

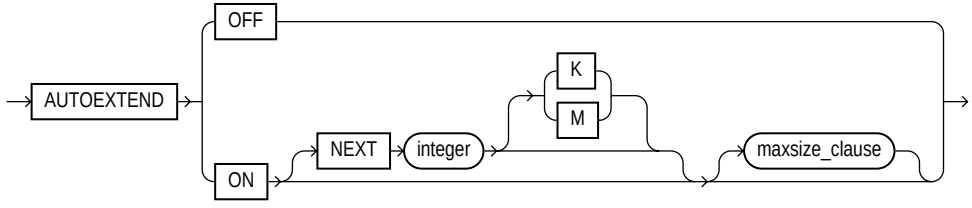
You must have the CREATE TABLESPACE system privilege.

Syntax

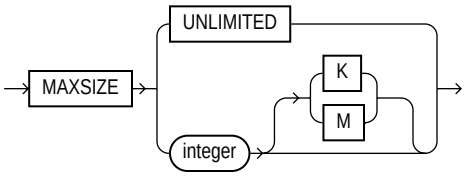


filespec: See *filespec* on page 11-27.

autoextend_clause::=



maxsize_clause::=



Keywords and Parameters

tablespace

Specify the name of the temporary tablespace.

TEMPFILE *filespec*

Specify the tempfiles that make up the tablespace.

Note: Media recovery does not recognize tempfiles.

See Also: *filespec* on page 11-27

autoextend_clause

The *autoextend_clause* lets you enable or disable the automatic extension of the tempfile.

OFF	Specify OFF to disable autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER TABLESPACE AUTOEXTEND statements.
ON	Specify ON to enable autoextend.
NEXT *integer*	Specify the disk space to allocate to the tempfile when more extents are required.
maxsize_clause	The *maxsize_clause* lets you specify the maximum disk space allowed for allocation to the tempfile.

- *integer*: Specify in bytes the maximum disk space allowed for allocation to the tempfile. Use K or M to specify this space in kilobytes or megabytes.
- UNLIMITED: Specify UNLIMITED to set no limit on allocating disk space to the tempfile.

EXTENT MANAGEMENT LOCAL

The EXTENT MANAGEMENT clause lets you specify that the tablespace is locally managed, meaning that some part of the tablespace is set aside for a bitmap.

UNIFORM *integer* — Specify the size of the extents of the temporary tablespace in bytes. All extents of temporary tablespaces are the same size (uniform). If you do not specify this clause, Oracle uses uniform extents of 1M.

SIZE *integer* — Specify in bytes the size of the tablespace extents. Use K or M to specify the size in kilobytes or megabytes.

If you do not specify SIZE, Oracle uses the default extent size of 1M.

See Also: *Oracle8i Concepts* for a discussion of locally managed tablespaces

Example

Temporary Tablespace Example This statement creates a temporary tablespace in which each extent is 16M.

```
CREATE TEMPORARY TABLESPACE tbs_1 TEMPFILE 'file_1.f'
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 16M;
```

If we assume the default database block size of 2K, and that each bit in the map represents one extent, then each bit maps 8,000 blocks.

CREATE TRIGGER

Purpose

Use the CREATE TRIGGER statement to create and enable a **database trigger**, which is

- A stored PL/SQL block associated with a table, a schema, or the database
- An anonymous PL/SQL block or a call to a procedure implemented in PL/SQL or Java

Oracle automatically executes a trigger when specified conditions occur.

When you create a trigger, Oracle enables it automatically. You can subsequently disable and enable a trigger with the DISABLE and ENABLE clause of the ALTER TRIGGER or ALTER TABLE statement.

See Also:

- *Oracle8i Concepts* for a description of the various types of triggers
- *Oracle8i Application Developer's Guide - Fundamentals* for more information on how to design triggers for the above purposes
- ALTER TRIGGER on page 8-76 and ALTER TABLE on page 8-2 for information on enabling, disabling, and compiling triggers
- DROP TRIGGER on page 11-13 for information on dropping a trigger

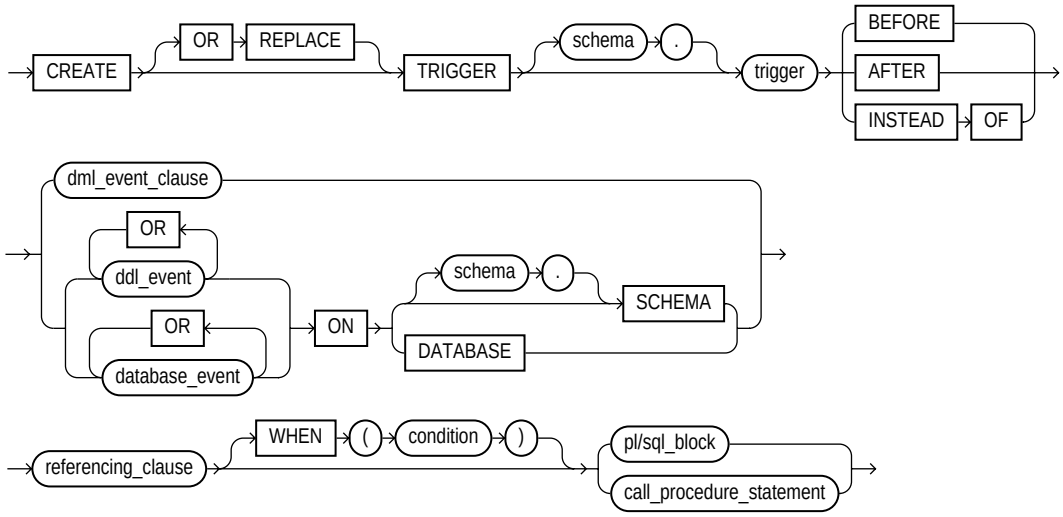
Prerequisites

Before a trigger can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

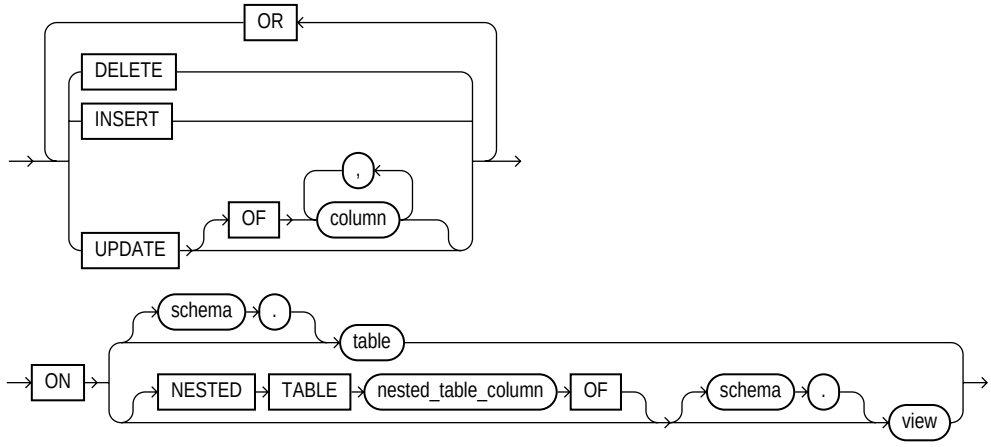
- To create a trigger in your own schema on a table in your own schema or on your own schema (SCHEMA), you must have the CREATE TRIGGER privilege.
- To create a trigger in any schema on a table in any schema, or on another user's schema (*schema*.SCHEMA), you must have the CREATE ANY TRIGGER privilege.
- In addition to the preceding privileges, to create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER system privilege.

If the trigger issues SQL statements or calls procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner, rather than acquired through roles.

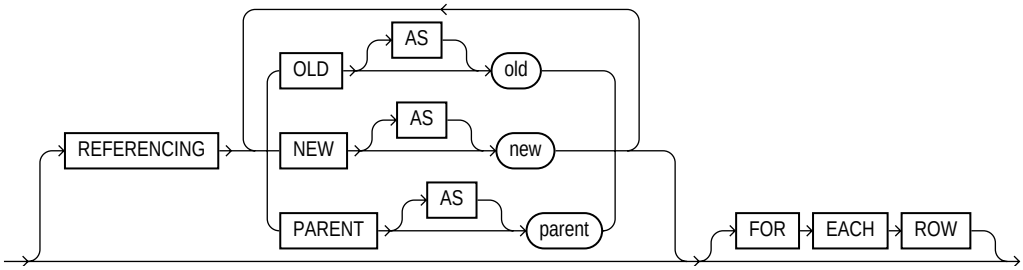
Syntax



dml_event_clause::=



referencing_clause::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the trigger if it already exists. Use this clause to change the definition of an existing trigger without first dropping it.

schema

Specify the schema to contain the trigger. If you omit *schema*, Oracle creates the trigger in your own schema.

trigger

Specify the name of the trigger to be created.

If a trigger produces compilation errors, it is still created, but it fails on execution. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS. This means it effectively blocks all triggering DML statements until it is disabled, replaced by a version without compilation errors, or dropped.

Note: If you create a trigger on a base table of a materialized view, you must ensure that the trigger does not fire during a refresh of the materialized view. (During refresh, the DBMS_SNAPSHOT procedure I_AM_A_REFRESH returns TRUE.)

BEFORE

Specify BEFORE to cause Oracle to fire the trigger before executing the triggering event. For row triggers, this is a separate firing before each affected row is changed.

Restrictions:

- You cannot specify a BEFORE trigger on a view or an object view.
- When defining a BEFORE trigger for LOB columns, you can read the :OLD value but not the :NEW value. You cannot write either the :OLD or the :NEW value.

AFTER

Specify AFTER to cause Oracle to fire the trigger after executing the triggering event. For row triggers, this is a separate firing after each affected row is changed.

Restrictions:

- You cannot specify an AFTER trigger on a view or an object view.
- When defining an AFTER trigger for LOB columns, you can read the :OLD value but not the :NEW value. You cannot write either the :OLD or the :NEW value.

Note: When you create a materialized view log for a table, Oracle implicitly creates an AFTER ROW trigger on the table. This trigger inserts a row into the materialized view log whenever an INSERT, UPDATE, or DELETE statement modifies the table's data. You cannot control the order in which multiple row triggers fire. Therefore, you should not write triggers intended to affect the content of the materialized view.

See Also: CREATE MATERIALIZED VIEW LOG on page 9-107 for more information on materialized view logs

INSTEAD OF

Specify INSTEAD OF to cause Oracle to fire the trigger instead of executing the triggering event. By default, INSTEAD OF triggers are activated for each row.

If a view is inherently updatable and has INSTEAD OF triggers, the triggers take preference. In other words, Oracle fires the triggers instead of performing DML on the view.

Restrictions:

- INSTEAD OF is a valid clause only for views. You cannot specify an INSTEAD OF trigger on a table.

- If a view has INSTEAD OF triggers, any views created on it must have INSTEAD OF triggers, even if the views are inherently updatable.
- When defining INSTEAD OF triggers for LOB columns, you can read both the :OLD and the :NEW value, but you cannot write either the :OLD or the :NEW values.

> **Note:** You can create multiple triggers of the same type (BEFORE, AFTER, or INSTEAD OF) that fire for the same statement on the same table. The order in which Oracle fires these triggers is indeterminate. If your application requires that one trigger be fired before another of the same type for the same statement, combine these triggers into a single trigger whose trigger action performs the trigger actions of the original triggers in the appropriate order.

dml_event_clause

The *dml_event_clause* lets you specify one of three DML statements that can cause the trigger to fire. Oracle fires the trigger in the existing user transaction.

DELETE — Specify DELETE if you want Oracle to fire the trigger whenever a DELETE statement removes a row from the table or an element from a nested table.

INSERT — Specify INSERT if you want Oracle to fire the trigger whenever an INSERT statement adds a row to table or an element to a nested table.

UPDATE — Specify UPDATE if you want Oracle to fire the trigger whenever an UPDATE statement changes a value in one of the columns specified after OF. If you omit OF, Oracle fires the trigger whenever an UPDATE statement changes a value in any column of the table or nested table.

For an UPDATE trigger, you can specify object type, varray, and REF columns after OF to indicate that the trigger should be fired whenever an UPDATE statement changes a value in one of the columns. However, you cannot change the values of these columns in the body of the trigger itself.

Note: Using OCI functions or the DBMS_LOB package to update LOB values or LOB attributes of object columns does not cause Oracle to fire triggers defined on the table containing the columns or the attributes.

Restrictions:

- You cannot specify OF with UPDATE for an INSTEAD OF trigger. Oracle fires INSTEAD OF triggers whenever an UPDATE changes a value in any column of the view.
- You cannot specify nested table or LOB columns with OF.

See Also: AS *subquery* of CREATE VIEW on page 10-105 for a list of constructs that prevent inserts, updates, or deletes on a view

Performing DML operations directly on nested table columns does not cause Oracle to fire triggers defined on the table containing the nested table column

ddl_event

Specify one or more types of DDL statements that can cause the trigger to fire. You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. You can create BEFORE and AFTER triggers for these events. Oracle fires the trigger in the existing user transaction. The following values are valid:

ALTER	Specify ALTER to fire the trigger whenever an ALTER statement modifies a database object in the data dictionary. **Restriction:** The trigger will not be fired by an ALTER DATABASE statement.
ANALYZE	Specify ANALYZE to fire the trigger whenever Oracle collects or deletes statistics or validates the structure of a database object.
ASSOCIATE STATISTICS	Specify ASSOCIATE STATISTICS to fire the trigger whenever Oracle associates a statistics type with a database object.
AUDIT	Specify AUDIT to fire the trigger whenever Oracle tracks the occurrence of a SQL statement or tracks operations on a schema object.
COMMENT	Specify COMMENT to fire the trigger whenever a comment on a database object is added to the data dictionary.

CREATE	Specify CREATE to fire the trigger whenever a CREATE statement adds a new database object to the data dictionary. **Restriction:** The trigger will not be fired by a CREATE DATABASE or CREATE CONTROLFILE statement.
DISASSOCIATE STATISTICS	Specify DISASSOCIATE STATISTICS to fire the trigger whenever Oracle disassociates a statistics type from a database object.
DROP	Specify DROP to fire the trigger whenever a DROP statement removes a database object from the data dictionary.
GRANT	Specify GRANT to fire the trigger whenever a user grants system privileges or roles or object privileges to another user or to a role.
NOAUDIT	Specify NOAUDIT to fire the trigger whenever a NOAUDIT statement instructs Oracle to stop tracking a SQL statement or operations on a schema object.
RENAME	Specify RENAME to fire the trigger whenever a RENAME statement change the name of a database object.
REVOKE	Specify REVOKE to fire the trigger whenever a REVOKE statement removes system privileges or roles or object privileges from a user or role.
TRUNCATE	Specify TRUNCATE to fire the trigger whenever a TRUNCATE statement removes the rows from a table or cluster and resets its storage characteristics.
DDL	Specify DDL to fire the trigger whenever any of the preceding DDL statements is issued.

Restriction: You cannot specify as a triggering event any DDL operation performed through a PL/SQL procedure.

database_event

Specify one or more particular states of the database that can cause the trigger to fire. You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. For each of these triggering events, Oracle opens an autonomous transaction scope, fires the trigger, and commits any separate transaction (regardless of any existing user transaction).

> **See Also:** *PL/SQL User's Guide and Reference* for more information on autonomous transaction scope

SERVERERROR	Specify SERVERERROR to fire the trigger whenever a server error message is logged. The following errors do not cause a SERVERERROR trigger to fire: ■ ORA-01403: data not found ■ ORA-01422: exact fetch returns more than requested number of rows ■ ORA-01423: error encountered while checking for extra rows in exact fetch ■ ORA-01034: ORACLE not available ■ ORA-04030: out of process memory
LOGON	Specify LOGON to fire the trigger whenever a client application logs onto the database.
LOGOFF	Specify LOGOFF to fire the trigger whenever a client applications logs off the database.
STARTUP	Specify STARTUP to fire the trigger whenever the database is opened.
SHUTDOWN	Specify SHUTDOWN to fire the trigger whenever an instance of the database is shut down.

Notes:

- Only AFTER triggers are relevant for LOGON, STARTUP, and SERVERERROR.
- Only BEFORE triggers are relevant for LOGOFF and SHUTDOWN.
- AFTER STARTUP and BEFORE SHUTDOWN triggers apply only to DATABASE.

ON *table* | *view*

The ON clause lets you determine the database object on which the trigger is to be created.

[schema.] table | *view*	Specify the *schema* and *table* or *view* name of one of the following on which the trigger is to be created: ■ Table or view ■ Object table or object view ■ A column of nested-table type If you omit *schema*, Oracle assumes the table is in your own schema. You can create triggers on index-organized tables. **Restriction:** You cannot create a trigger on a table in the schema SYS.
NESTED TABLE	Specify the *nested_table_column* of a view upon which the trigger is being defined. Such a trigger will fire only if the DML operates on the elements of the nested table. **Restriction:** You can specify NESTED TABLE only for INSTEAD OF triggers.
DATABASE	Specify DATABASE to define the trigger on the entire database.
SCHEMA	Specify SCHEMA to define the trigger on the current schema.

referencing_clause

The *referencing_clause* lets you specify correlation names. You can use correlation names in the PL/SQL block and WHEN condition of a row trigger to refer specifically to old and new values of the current row. The default correlation names are OLD and NEW. If your row trigger is associated with a table named OLD or NEW, use this clause to specify different correlation names to avoid confusion between the table name and the correlation name.

- If the trigger is defined on a **nested table**, OLD and NEW refer to the row of the nested table, and PARENT refers to the current row of the parent table.
- If the trigger is defined on an object table or view, OLD and NEW refer to object instances.

Restriction: This clause is valid only for DML event triggers (not DDL or database event triggers).

FOR EACH ROW Specify FOR EACH ROW to designate the trigger as a row trigger. Oracle fires a row trigger once for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the WHEN condition.

> **Note:** This clause is applies only to DML events, not to DDL or database events.

Except for INSTEAD OF triggers, if you omit this clause, the trigger is a statement trigger. Oracle fires a statement trigger only once when the triggering statement is issued if the optional trigger constraint is met.

INSTEAD OF trigger statements are implicitly activated for each row.

WHEN

Specify the trigger restriction, which is a SQL condition that must be satisfied for Oracle to fire the trigger. See the syntax description of condition in "Conditions" on page 5-15. This condition must contain correlation names and cannot contain a query.

Restrictions:

- You can specify a trigger restriction only for a row trigger. Oracle evaluates this condition for each row affected by the triggering statement.
- You cannot specify trigger restrictions for INSTEAD OF trigger statements.
- You can reference object columns or their attributes, or varray, nested table, or LOB columns. You cannot invoke PL/SQL functions or methods in the trigger restriction.

pl/sql_block

Specify the PL/SQL block that Oracle executes to fire the trigger.

The PL/SQL block of a database trigger can contain one of a series of built-in functions in the SYS schema designed solely to extract system event attributes. These functions can be used **only** in the PL/SQL block of a database trigger.

Restrictions:

- The PL/SQL block of a trigger cannot contain transaction control SQL statements (`COMMIT`, `ROLLBACK`, `SAVEPOINT`, and `SET CONSTRAINT`) if the block is executed within the same transaction.
- You can reference and use LOB columns in the trigger action inside the PL/SQL block, but you cannot modify their values within the trigger action.

See Also:

- *PL/SQL User's Guide and Reference* for information on PL/SQL, including how to write PL/SQL blocks
- *Oracle8i Application Developer's Guide - Fundamentals* for information on these functions
- *Oracle8i Application Developer's Guide - Fundamentals*

call_procedure_statement

The *call_procedure_statement* lets you call a stored procedure, rather than specifying inline the trigger code as a PL/SQL block. The syntax of this statement is the same as that for CALL on page 8-128, with the following exceptions:

- You cannot specify the `INTO` clause of `CALL`, because it applies only to functions.
- You cannot specify bind variables in *expr*.
- To reference columns of tables on which the trigger is being defined, you must specify :`NEW` and :`OLD`.

See Also: "Calling a Procedure in a Trigger Body Example" on page 10-77

Examples

DML Trigger Example This example creates a `BEFORE` statement trigger named `emp_permit_changes` in the schema `scott`. You would write such a trigger to place restrictions on DML statements issued on this table (such as when such statements could be issued).

```
CREATE TRIGGER scott.emp_permit_changes
    BEFORE
    DELETE OR INSERT OR UPDATE
    ON scott.emp
```

```
    pl/sql block
```

Oracle fires this trigger whenever a DELETE, INSERT, or UPDATE statement affects the emp table in the schema scott. The trigger emp_permit_changes is a BEFORE statement trigger, so Oracle fires it once before executing the triggering statement.

DML Trigger Example with Restriction This example creates a BEFORE row trigger named salary_check in the schema scott. The PL/SQL block might specify, for example, that the employee's salary must fall within the established salary range for the employee's job:

```
CREATE TRIGGER scott.salary_check
    BEFORE
    INSERT OR UPDATE OF sal, job ON scott.emp
    FOR EACH ROW
    WHEN (new.job <> 'PRESIDENT')
       pl/sql_block
```

Oracle fires this trigger whenever one of the following statements is issued:

- an INSERT statement that adds rows to the emp table
- an UPDATE statement that changes values of the sal or job columns of the emp table

salary_check is a BEFORE row trigger, so Oracle fires it before changing each row that is updated by the UPDATE statement or before adding each row that is inserted by the INSERT statement.

salary_check has a trigger restriction that prevents it from checking the salary of the company president.

Calling a Procedure in a Trigger Body Example You could create the salary_check trigger described in the preceding example by calling a procedure instead of providing the trigger body in a PL/SQL block. Assume you have defined a procedure scott.salary_check, which verifies that an employee's salary in in an appropriate range. Then you could create the trigger salary_check as follows:

```
CREATE TRIGGER scott.salary_check
   BEFORE INSERT OR UPDATE OF sal, job ON scott.emp
   FOR EACH ROW
   WHEN (new.job <> 'PRESIDENT')
   CALL check_sal(:new.job, :new.sal, :new.ename);
```

The procedure check_sal could be implemented in PL/SQL, C, or Java. Also, you can specify :OLD values in the CALL clause instead of :NEW values.

Database Event Trigger Example This example creates a trigger to log all errors. The PL/SQL block does some special processing for a particular error (invalid logon, error number 1017. This trigger is an AFTER statement trigger, so it is fired after an unsuccessful statement execution (such as unsuccessful logon).

```
CREATE TRIGGER log_errors AFTER SERVERERROR ON DATABASE
   BEGIN
      IF (IS_SERVERERROR (1017)) THEN
         <special processing of logon error>
      ELSE
         <log error number>
      END IF;
   END;
```

DDL Trigger Example This example creates an AFTER statement trigger on any DDL statement CREATE. Such a trigger can be used to audit the creation of new data dictionary objects in your schema.

```
CREATE TRIOGGER audit_db_object AFTER CREATE
   ON SCHEMA
      pl/sql_block
```

INSTEAD OF Trigger Example In this example, customer data is stored in two tables. The object view all_customers is created as a UNION of the two tables, customers_sj and customers_pa. An INSTEAD OF trigger is used to insert values.

```
CREATE TABLE customers_sj
  ( cust    NUMBER(6),
    address VARCHAR2(50),
    credit   NUMBER(9,2)  );

CREATE TABLE customers_pa
  ( cust    NUMBER(6),
    address VARCHAR2(50),
    credit   NUMBER(9,2) );

CREATE TYPE customer_t AS OBJECT
  ( cust    NUMBER(6),
    address   VARCHAR2(50),
    credit    NUMBER(9,2),
    location   VARCHAR2(20)  );
```

```
CREATE VIEW all_customers (cust)
    AS SELECT customer_t (cust, address, credit, 'SAN_JOSE')
    FROM   customers_sj
  UNION ALL
    SELECT customer_t (cust, address, credit, 'PALO_ALTO')
    FROM   customers_pa;

CREATE TRIGGER instrig INSTEAD OF INSERT ON all_customers
   FOR EACH ROW
     BEGIN
      IF (:new.cust.location = 'SAN_JOSE') THEN
        INSERT INTO customers_sj
        VALUES (:new.cust.cust, :new.cust.address,:new.cust.credit);
      ELSE
       INSERT INTO customers_pa
       VALUES (:new.cust.cust, :new.cust.address, :new.cust.credit);
      END IF;
    END;
```

CREATE TYPE

Purpose

Use the CREATE TYPE statement to create the specification of an **object type**, named varying array (**varray**), **nested table type**, or an **incomplete object type**. You create object types with the CREATE TYPE and the CREATE TYPE BODY statements. The CREATE TYPE statement specifies the name of the object type, its attributes, methods, and other properties. The CREATE TYPE BODY statement contains the code for the methods in the type.

> **Note:** If you create an object type for which the type specification declares only attributes but no methods, you need not specify a type body.

Oracle implicitly defines a constructor method for each user-defined type that you create. A **constructor** is a system-supplied procedure that is used in SQL statements or in PL/SQL code to construct an instance of the type value. The name of the constructor method is the same as the name of the user-defined type.

The parameters of the object type constructor method are the data attributes of the object type. They occur in the same order as the attribute definition order for the object type. The parameters of a nested table or varray constructor are the elements of the nested table or the varray.

An **incomplete type** is a type created by a forward type definition. It is called "incomplete" because it has a name but no attributes or methods. It can be referenced by other types, and so can be used to define types that refer to each other. However, you must fully specify the type before you can use it to create a table or an object column or a column of a nested table type.

See Also:

- CREATE TYPE BODY on page 10-93 for information on creating the member methods of a type
- *PL/SQL User's Guide and Reference*, *Oracle8i Application Developer's Guide - Fundamentals*, and *Oracle8i Concepts* for more information about objects, incomplete types, varrays, and nested tables

Prerequisites

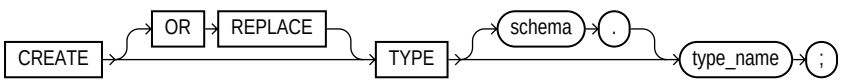
To create a type in your own schema, you must have the CREATE TYPE system privilege. To create a type in another user's schema, you must have the CREATE ANY TYPE system privilege. You can acquire these privileges explicitly or be granted them through a role.

The owner of the type must either be explicitly granted the EXECUTE object privilege in order to access all other types referenced within the definition of the type, or the type owner must be granted the EXECUTE ANY TYPE system privilege. The owner **cannot** obtain these privileges through roles.

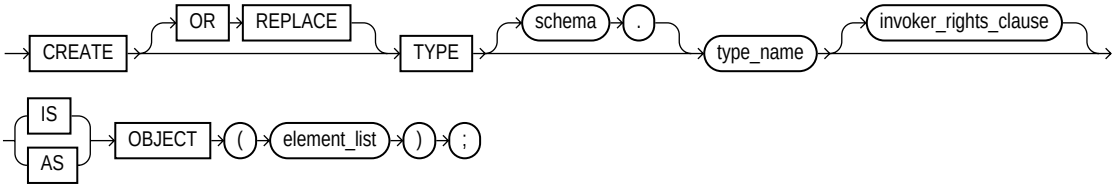
If the type owner intends to grant other users access to the type, the owner must be granted the EXECUTE object privilege to the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. Otherwise, the type owner has insufficient privileges to grant access on the type to other users.

Syntax

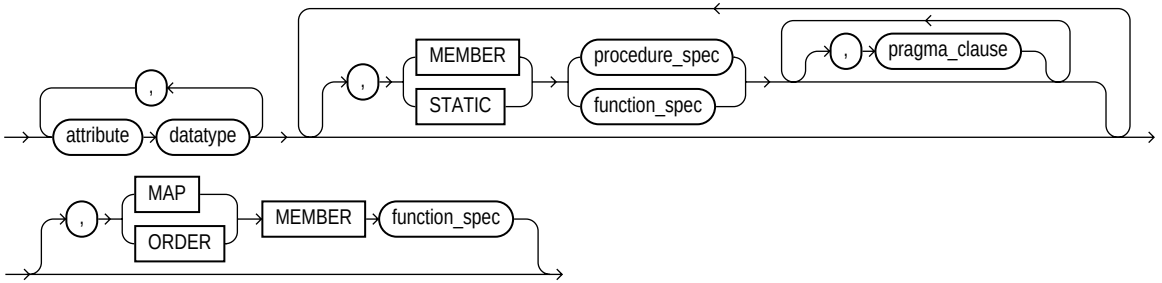
create_incomplete_type::=



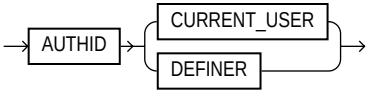
create_object_type::=



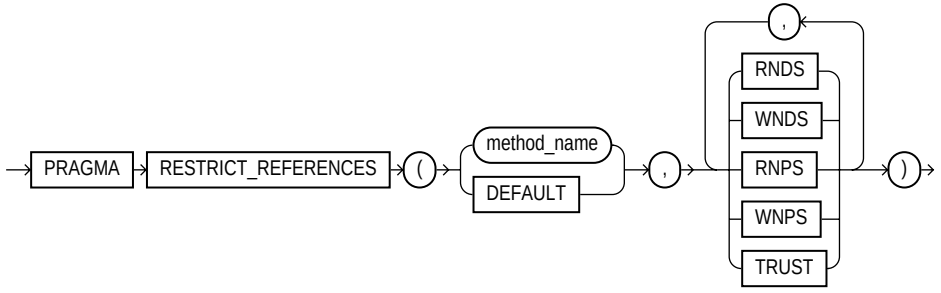
element_list::=



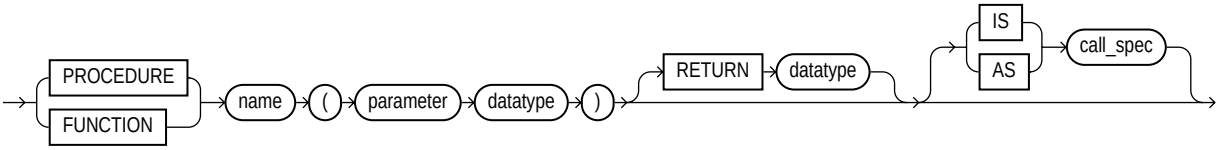
invoker_rights_clause::=



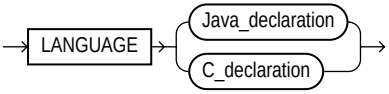
pragma_clause::=



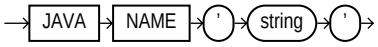
procedure_spec or function_spec::=



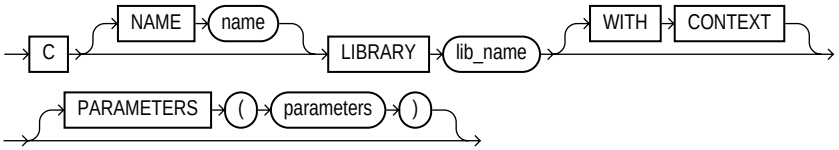
call_spec::=



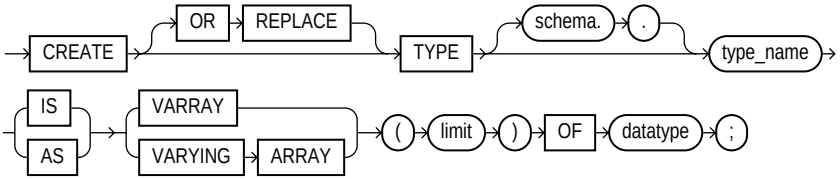
Java_declaration::=



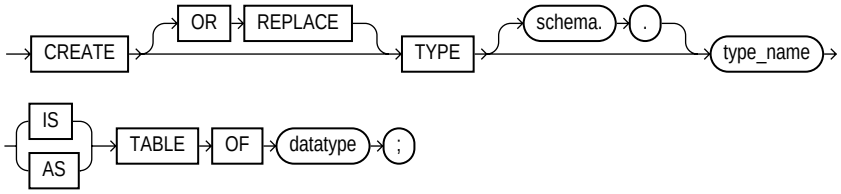
C_declaration::=



create_varray_type::=



create_nested_table_type::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the type if it already exists. Use this clause to change the definition of an existing type without first dropping it.

Users previously granted privileges on the re-created object type can use and reference the object type without being granted privileges again.

If any function-based indexes depend on the type, Oracle marks the indexes DISABLED.

schema

Specify the schema to contain the type. If you omit *schema*, Oracle creates the type in your current schema.

type_name

Specify the name of an object type, a nested table type, or a varray type.

If creating the type results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

create_object_type

Use the *create_object_type* clause to create a user-defined object type (rather than an incomplete type). The variables that form the data structure are called **attributes**. The member subprograms that define the object's behavior are called **methods**. AS OBJECT is required when creating an object type.

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the member functions and procedures of the object type execute with the privileges and in the schema of the user who owns the object type or with the privileges and in the schema of CURRENT_USER. This specification applies to the corresponding type body as well.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

Restriction: You can specify this clause only for an object type, not for a nested table or varray type.

AUTHID CURRENT_USER	Specify CURRENT_USER if you want the member functions and procedures of the object type to execute with the privileges of CURRENT_USER. This clause creates an **invoker-rights type**. This clause also indicates that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the type resides.
AUTHID DEFINER	Specify DEFINER if you want the member functions and procedures of the object type to execute with the privileges of the owner of the schema in which the functions and procedures reside, and that external names resolve in the schema where the member functions and procedures reside. This is the default.

See Also:

- *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined
- *PL/SQL User's Guide and Reference*

element_list

datatype — Specify the attribute's Oracle built-in datatype or user-defined type.

Restrictions:

- You cannot specify attributes of type ROWID, LONG, or LONG ROW.
- You cannot create an object with NCLOB, NCHAR, or NVARCHAR2 attributes, but you can specify parameters of these datatypes in methods.
- You cannot specify a datatype of UROWID for a user-defined object type.
- If you specify an object of type REF, the target object must have an object identifier.

See Also: "Datatypes" on page 2-2 for a list of possible datatypes

attribute — Specify, for an object type, the name of an object attribute. Attributes are data items with a name and a type specifier that form the structure of the object. You must specify at least one attribute for each object type.

MEMBER *procedure_spec* or *function_spec* — Specify a function or procedure subprogram associated with the object type that is referenced as an attribute. Typically, you invoke member methods in a "selfish" style, such as `object_expression.method()`. This class of method has an implicit first argument referenced as SELF in the method's body, which represents the object on which the method has been invoked.

STATIC *procedure_spec* or *function_spec* — Specify a function or procedure subprogram associated with the object type. Unlike member methods, static methods do not have any implicit parameters (that is, SELF is not referenceable in their body). They are typically invoked as type_name.method().

For both member and static methods, you must specify a corresponding method body in the object type body for each procedure or function specification.

The RETURN clause is valid only for a function. The syntax shown is an abbreviated form.

If this subprogram does not include the declaration of the procedure or function, you must issue a corresponding CREATE TYPE BODY statement.

See Also:

- *PL/SQL User's Guide and Reference* for information about method invocation and methods

- CREATE PROCEDURE on page 9-132 and CREATE FUNCTION on page 9-43 for the full syntax with all possible clauses

- CREATE TYPE BODY on page 10-93

- "Restrictions on User-Defined Functions" on page 9-46 for a list of restrictions on user-defined functions

call_spec — Specify the call specification ("call spec") that maps a Java or C method name, parameter types, and return type to their SQL counterparts. If all the member methods in the type have been defined in this clause, you need not issue a corresponding CREATE TYPE BODY statement.

In *Java_declaration*, '*string*' identifies the Java implementation of the method.

See Also:

- *Oracle8i Java Stored Procedures Developer's Guide*

- *Oracle8i Application Developer's Guide - Fundamentals* for an explanation of the parameters and semantics of the *C_declaration*

pragma_clause — The *pragma_clause* lets you specify a compiler directive.

PRAGMA RESTRICT_REFERENCES — The PRAGMA RESTRICT_REFERENCES compiler directive denies member functions read / write access to database tables, packaged variables, or both, and thereby helps to avoid side effects.

See Also: *Oracle8i Application Developer's Guide - Fundamentals*

method_name — Specify the name of the MEMBER function or procedure to which the pragma is being applied.

DEFAULT	Specify DEFAULT to apply the pragma to all methods in the type for which a pragma has not been explicitly specified.
WNDS	Specify WNDS to invoke the constraint **writes no database state** (does not modify database tables).
WNPS	Specify WNPS to invoke the constraint **writes no package state** (does not modify packaged variables).
RNDS	Specify RNDS to invoke the constraint **reads no database state** (does not query database tables).
RNPS	Specify RNPS to invoke the constraint **reads no package state** (does not reference packages variables).
TRUST	TRUST indicates that the restrictions listed in the pragma are not actually to be enforced, but are simply trusted to be true.

MAP MEMBER *function_spec*

This clause lets you specify a member function (MAP method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the map method is null, the map method returns null and the method is not invoked.

An object specification can contain only one map method, which must be a function. The result type must be a predefined SQL scalar type, and the map function can have no arguments other than the implicit SELF argument.

Note: If *type_name* will be referenced in queries involving sorts (through an ORDER BY, GROUP BY, DISTINCT, or UNION clause) or joins, and you want those queries to be parallelized, you must specify a MAP member function.

ORDER MEMBER *function_spec*

This clause lets you specify a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

If either argument to the order method is null, the order method returns null and the method is not invoked.

When instances of the same object type definition are compared in an ORDER BY clause, the order method *function_spec* is invoked.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

You can define either a MAP method or an ORDER method in a type specification, but not both. If you declare either method, you can compare object instances in SQL.

If neither a MAP nor an ORDER method is specified, only comparisons for equality or inequality can be performed. Therefore object instances cannot be ordered. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method needs to be specified to determine the equality of two object types.

Use MAP if you are performing extensive sorting or hash join operations on object instances. MAP is applied once to map the objects to scalar values and then the scalars are used during sorting and merging. A MAP method is more efficient than an ORDER method, which must invoke the method for each object comparison. You must use a MAP method for hash joins. You cannot use an ORDER method because the hash mechanism hashes on the object value.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* for more information about object value comparisons

create_varray_type

The *create_varray_type* lets you create the type as an ordered set of elements, each of which has the same datatype. You must specify a name and a maximum limit of zero or more. The array limit must be an integer literal. Oracle does not support anonymous varrays.

The type name for the objects contained in the varray must be one of the following:

- A built-in datatype,

- A REF, or
- An object type.

Restrictions:

- A collection type cannot contain any other collection type, either directly or indirectly. That is, a varray type cannot contain any elements that are or contain varrays or nested tables.
- You cannot create varray types of LOB datatypes.

create_nested_table_type

The *create_nested_table_type* lets you create a named nested table of type *datatype*.

- When *datatype* is an object type, the nested table type describes a table whose columns match the name and attributes of the object type.
- When *datatype* is a scalar type, then the nested table type describes a table with a single, scalar type column called "column_value".

Restrictions:

- A collection type cannot contain any other collection type, either directly or indirectly. That is, a nested table type cannot contain any elements that are or contain varrays or nested tables.
- You cannot specify NCLOB for *datatype*. However, you can specify CLOB or BLOB.

Examples

Object Type Example The following example creates object type person_t with LOB attributes:

```
CREATE TYPE person_t AS OBJECT
  (name    CHAR(20),
   resume  CLOB,
   picture BLOB);
```

Varray Type Example The following statement creates members_type as a varray type with 100 elements:

```
CREATE TYPE members_type AS VARRAY(100) OF CHAR(5);
```

Nested Table Type Example The following example creates a named table type `project_table` of object type `project_t`:

```
CREATE TYPE project_t AS OBJECT
  (pno CHAR(5),
   pname CHAR(20),
   budgets DEC(7,2));

CREATE TYPE project_table AS TABLE OF project_t;
```

Constructor Example The following example invokes method constructor `col.getbar()`:

```
CREATE TYPE foo AS OBJECT (a1 NUMBER,
                  MEMBER FUNCTION getbar RETURN NUMBER,);
CREATE TABLE footab(col foo);

SELECT col.getbar() FROM footab;
```

Unlike function invocations, method invocations require parentheses, even when the methods do not have additional arguments.

The next example invokes the system-defined constructor to construct the `foo_t` object and insert it into the `foo_tab` table:

```
CREATE TYPE foo_t AS OBJECT (a1 NUMBER, a2 NUMBER);
CREATE TABLE foo_tab (b1 NUMBER, b2 foo_t);
INSERT INTO foo_tab VALUES (1, foo_t(2,3));
```

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* and *PL/SQL User's Guide and Reference* for more information about constructors

Static Method Example The following example changes the definition of the `employee_t` type to associate it with the `construct_emp` function:

```
CREATE OR REPLACE TYPE employee_t AS OBJECT(
   empid RAW(16),
   ename CHAR(31),
   dept   REF department_t,
   STATIC function construct_emp
      (name VARCHAR2, dept REF department_t)
   RETURN employee_t
  );
```

This statement requires the following type body statement (PL/SQL is shown in italics):

```
CREATE OR REPLACE TYPE BODY employee_t IS
   STATIC FUNCTION construct_emp
      (name varchar2, dept REF department_t)
   RETURN employee_t IS
      BEGIN
         return employee_t(SYS_GUID(),name,dept);
      END;
   END;
```

This type and type body definition allows the following operation:

```
INSERT INTO emptab
   VALUES (employee_t.construct_emp('John Smith', NULL));
```

CREATE TYPE BODY

Purpose

Use the CREATE TYPE BODY to define or implement the member methods defined in the object type specification. You create object types with the CREATE TYPE and the CREATE TYPE BODY statements. The CREATE TYPE statement specifies the name of the object type, its attributes, methods, and other properties. The CREATE TYPE BODY statement contains the code for the methods in the type.

For each method specified in an object type specification for which you did not specify the *call_spec*, you must specify a corresponding method body in the object type body.

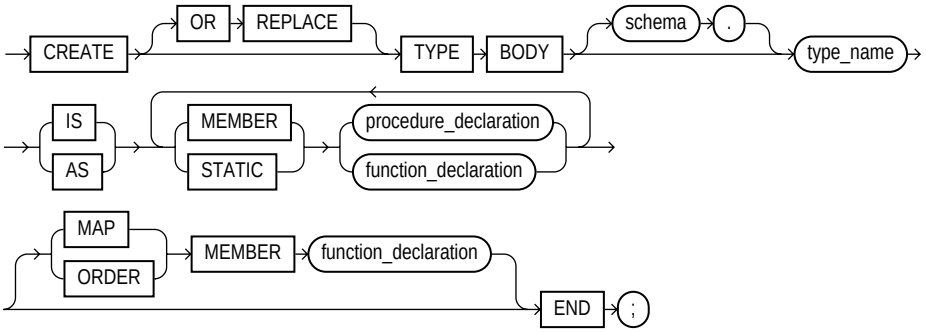
> **See Also:** CREATE TYPE on page 10-80 and ALTER TYPE on page 8-79 for information on creating and modifying a type specification

Prerequisites

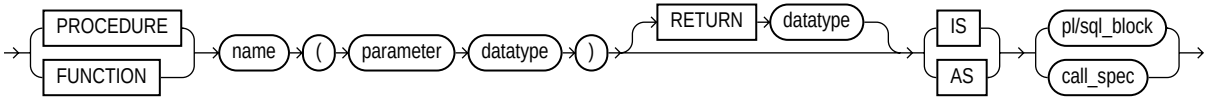
Every member declaration in the CREATE TYPE specification for object types must have a corresponding construct in the CREATE TYPE or CREATE TYPE BODY statement.

To create or replace a type body in your own schema, you must have the CREATE TYPE or the CREATE ANY TYPE system privilege. To create an object type in another user's schema, you must have the CREATE ANY TYPE system privileges. To replace an object type in another user's schema, you must have the DROP ANY TYPE system privileges.

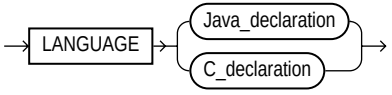
Syntax



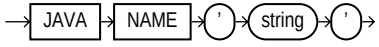
procedure_declaration | function_declaration::=



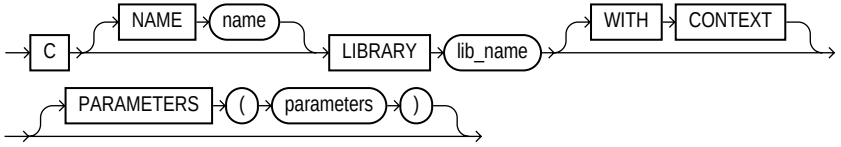
call_spec::=



Java_declaration::=



C_declaration::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the type body if it already exists. Use this clause to change the definition of an existing type body without first dropping it.

Users previously granted privileges on the re-created object type body can use and reference the object type body without being granted privileges again.

You can use this clause to add new member subprogram definitions to specifications added with the ALTER TYPE ... REPLACE statement.

schema

Specify the schema to contain the type body. If you omit *schema*, Oracle creates the type body in your current schema.

type_name

Specify the name of an object type.

IS | AS

MEMBER | STATIC — Specify the type of method function or procedure subprogram associated with the object type specification.

You must define a corresponding method name, optional parameter list, and (for functions) a return type in the object type specification for each procedure or function declaration.

procedure_declaration — Declare a procedure subprogram.

function_declaration — Declare a function subprogram.

See Also:

- CREATE TYPE on page 10-80 for a list of restrictions on user-defined functions

- *PL/SQL User's Guide and Reference* for information about overloading subprogram names within a package

- CREATE PROCEDURE on page 9-132, CREATE FUNCTION on page 9-43, and *Oracle8i Application Developer's Guide - Fundamentals*

MAP | ORDER

MAP MEMBER — Specify MAP MEMBER to declare or implement a member function (MAP method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and specifies an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the map method is null, the map method returns null and the method is not invoked.

An object type body can contain only one map method, which must be a function. The map function can have no arguments other than the implicit SELF argument.

ORDER MEMBER — Specify ORDER MEMBER to specify a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

If either argument to the order method is null, the order method returns null and the method is not invoked.

When instances of the same object type definition are compared in an ORDER BY clause, Oracle invokes the order method *function_spec*.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

You can declare either a MAP method or an ORDER method, but not both. If you declare either method, you can compare object instances in SQL.

If you do not declare either method, you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal.

procedure_declaration | *function_declaration*

Declare a procedure or function subprogram. The RETURN clause is valid only for a function. The syntax shown is an abbreviated form.

See Also: CREATE PROCEDURE on page 9-132 and CREATE FUNCTION on page 9-43 for the full syntax with all possible clauses

pl/sql_block

Declare the procedure or function.

See Also: *PL/SQL User's Guide and Reference*

call_spec

Specify the call specification ("call spec") that maps a Java or C method name, parameter types, and return type to their SQL counterparts.

In *Java_declaration*, '*string*' identifies the Java implementation of the method.

See Also:

- *Oracle8i Java Stored Procedures Developer's Guide*

- *Oracle8i Application Developer's Guide - Fundamentals* for an explanation of the parameters and semantics of *the C_declaration*

AS EXTERNAL

AS EXTERNAL is an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle Corporation recommends that you use the *call_spec* syntax with the *C_declaration*.

Examples

Creating a Type Body Example The following object type body implements member subprograms for `rational`. (PL/SQL is shown in italics.)

```
CREATE TYPE BODY rational
  IS
   MAP MEMBER FUNCTION rat_to_real RETURN REAL IS
      BEGIN
         RETURN numerator/denominator;
      END;

   MEMBER PROCEDURE normalize IS
      gcd NUMBER := integer_operations.greatest_common_divisor
                    (numerator, denominator);
      BEGIN
         numerator := numerator/gcd;
         denominator := denominator/gcd;
      END;

   MEMBER FUNCTION plus(x rational) RETURN rational IS
      r rational := rational_operations.make_rational
                     (numerator*x.denominator +
                      x.numerator*denominator,
                      denominator*x.denominator);
      BEGIN
         RETURN r;
      END;

   END;
```

CREATE USER

Purpose

Use the CREATE USER statement to create and configure a database **user**, or an account through which you can log in to the database and establish the means by which Oracle permits access by the user.

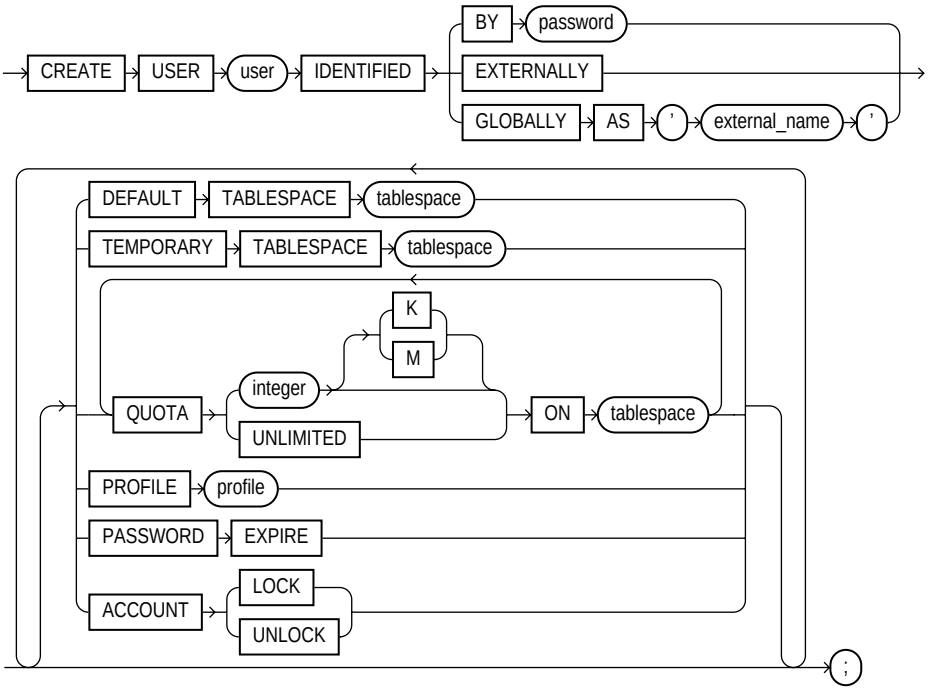
Note: You can enable a user to connect to Oracle through a proxy (that is, an application or application server). For syntax and discussion, refer to ALTER USER on page 8-88.

Prerequisites

You must have CREATE USER system privilege. When you create a user with the CREATE USER statement, the user's privilege domain is empty. To log on to Oracle, a user must have CREATE SESSION system privilege. Therefore, after creating a user, you should grant the user at least the CREATE SESSION privilege.

See Also: GRANT on page 11-31

Syntax



Keywords and Parameters

user

Specify the name of the user to be created. This name can contain only characters from your database character set and must follow the rules described in the section "Schema Object Naming Rules" on page 2-83. Oracle recommends that the user name contain at least one single-byte character regardless of whether the database character set also contains multi-byte characters.

IDENTIFIED

The `IDENTIFIED` clause lets you indicate how Oracle authenticates the user.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* and your operating system specific documentation for more information

BY *password*

The BY *password* clause lets you creates a **local user** and indicates that the user must specify *password* to log on. Passwords can contain only single-byte characters from your database character set regardless of whether this character set also contains multibyte characters.

Passwords must follow the rules described in the section "Schema Object Naming Rules" on page 2-83, unless you are using Oracle's password complexity verification routine. That routine requires a more complex combination of characters than the normal naming rules permit. You implement this routine with the UTLPWDMG.SQL script, which is further described in *Oracle8i Administrator's Guide.*

> **See Also:** *Oracle8i Administrator's Guide* to for a detailed description and explanation of how to use password management and protection

EXTERNALLY

Specify EXTERNALLY to create an **external user** and indicate that a user must be authenticated by an external service (such as an operating system or a third-party service). Doing so causes Oracle to rely on the login authentication of the operating system to ensure that a specific operating system user has access to a specific database user.

> **Caution:** Oracle strongly recommends that you do not use IDENTIFIED EXTERNALLY with operating systems that have inherently weak login security. For more information, see *Oracle8i Administrator's Guide.*

GLOBALLY AS '*external_name*'

The GLOBALLY clause lets you create a **global user** and indicates that a user must be authenticated by the enterprise directory service. The '*external_name*' string can take one of two forms:

- The X.509 name at the enterprise directory service that identifies this user. It should be of the form 'CN=*username,other_attributes'*, where *other_attributes* is the rest of the user's distinguished name (DN) in the directory.
- A null string (' ') indicating that the enterprise directory service will map authenticated global users to the appropriate database schema with the appropriate roles

Note: You can control the ability of an application server to connect as the specified user and to activate that user's roles using the ALTER USER statement.

See Also:

- *Oracle Advanced Security Administrator's Guide* for more information on global users

- ALTER USER on page 8-88

DEFAULT TABLESPACE

Specify the default tablespace for objects that the user creates. If you omit this clause, objects default to the SYSTEM tablespace.

See Also: CREATE TABLESPACE on page 10-56 for more information on tablespaces

TEMPORARY TABLESPACE

Specify the tablespace for the user's temporary segments. If you omit this clause, temporary segments default to the SYSTEM tablespace.

QUOTA

Use the QUOTA clause to allow the user to allocate space in the tablespace and optionally establishes a quota of *integer* bytes. Use K or M to specify the quota in kilobytes or megabytes. This quota is the maximum space in the tablespace the user can allocate.

A CREATE USER statement can have multiple QUOTA clauses for multiple tablespaces.

UNLIMITED allows the user to allocate space in the tablespace without bound.

PROFILE

Specify the the profile you want to reassign to the user. The profile limits the amount of database resources the user can use. If you omit this clause, Oracle assigns the DEFAULT profile to the user.

See Also: GRANT on page 11-31 and CREATE PROFILE on page 9-139

PASSWORD EXPIRE

Specify PASSWORD EXPIRE if you want the user's password to expire. This setting forces the user (or the DBA) to change the password before the user can log in to the database.

ACCOUNT Clause

ACCOUNT LOCK	Specify ACCOUNT LOCK to lock the user's account and disables access.
ACCOUNT UNLOCK	Specify ACCOUNT UNLOCK to unlock the user's account and enables access to the account.

Examples

Creating a User Example If you create a new user with PASSWORD EXPIRE, the user's password must be changed before attempting to log in to the database. You can create the user sidney by issuing the following statement:

```
CREATE USER sidney
    IDENTIFIED BY welcome
    DEFAULT TABLESPACE cases_ts
    QUOTA 10M ON cases_ts
    TEMPORARY TABLESPACE temp_ts
    QUOTA 5M ON system
    PROFILE engineer
    PASSWORD EXPIRE;
```

The user sidney has the following characteristics:

- The password welcome
- Default tablespace cases_ts, with a quota of 10 megabytes
- Temporary tablespace temp_ts
- Access to the tablespace SYSTEM, with a quota of 5 megabytes
- Limits on database resources defined by the profile engineer
- An expired password, which must be changed before sidney can log in to the database

To create a user accessible only by the operating system account george, prefix george by the value of the initialization parameter OS_AUTHENT_PREFIX. For

example, if this value is "ops$", you can create the user ops$george with the following statement:

```
CREATE USER ops$george
    IDENTIFIED EXTERNALLY
    DEFAULT TABLESPACE accs_ts
    TEMPORARY TABLESPACE temp_ts
    QUOTA UNLIMITED ON accs_ts;
```

The user ops$george has the following additional characteristics:

- Default tablespace accs_ts
- Default temporary tablespace temp_ts
- Unlimited space on the tablespaces accs_ts and temp_ts
- Limits on database resources defined by the DEFAULT profile

The following example creates user cindy as a global user:

```
CREATE USER cindy
   IDENTIFIED GLOBALLY AS 'CN=cindy,OU=division1,O=oracle,C=US'
   DEFAULT TABLESPACE legal_ts
   QUOTA 20M ON legal_ts
   PROFILE lawyer;
```

CREATE VIEW

Purpose

Use the CREATE VIEW statement to define a **view**, which is a logical table based on one or more tables or views. A view contains no data itself. The tables upon which a view is based are called **base tables**.

You can also create an **object view** or a relational view that supports LOB and object datatypes (object types, REFs, nested table, or varray types) on top of the existing view mechanism. An object view is a view of a user-defined type, where each row contains objects, each object with a unique object identifier.

See Also:

- *Oracle8i Concepts, Oracle8i Application Developer's Guide - Fundamentals,* and *Oracle8i Administrator's Guide*for information on various types of views and their uses
- ALTER VIEW on page 8-94 for information on modifying a view
- DROP VIEW on page 11-21 for information on removing a view from the database

Prerequisites

To create a view in your own schema, you must have CREATE VIEW system privilege. To create a view in another user's schema, you must have CREATE ANY VIEW system privilege.

The owner of the schema containing the view must have the privileges necessary to either select, insert, update, or delete rows from all the tables or views on which the view is based. The owner must be granted these privileges directly, rather than through a role.

To use the basic constructor method of an object type when creating an object view, one of the following must be true:

- The object type must belong to the same schema as the view to be created.
- You must have EXECUTE ANY TYPE system privileges.
- You must have the EXECUTE object privilege on that object type.

See Also: SELECT and subquery on page 11-88, INSERT on page 11-51, UPDATE on page 11-141, and DELETE on page 10-115 for information on the privileges required by the owner of a view on the base tables or views of the view being created

Partition Views

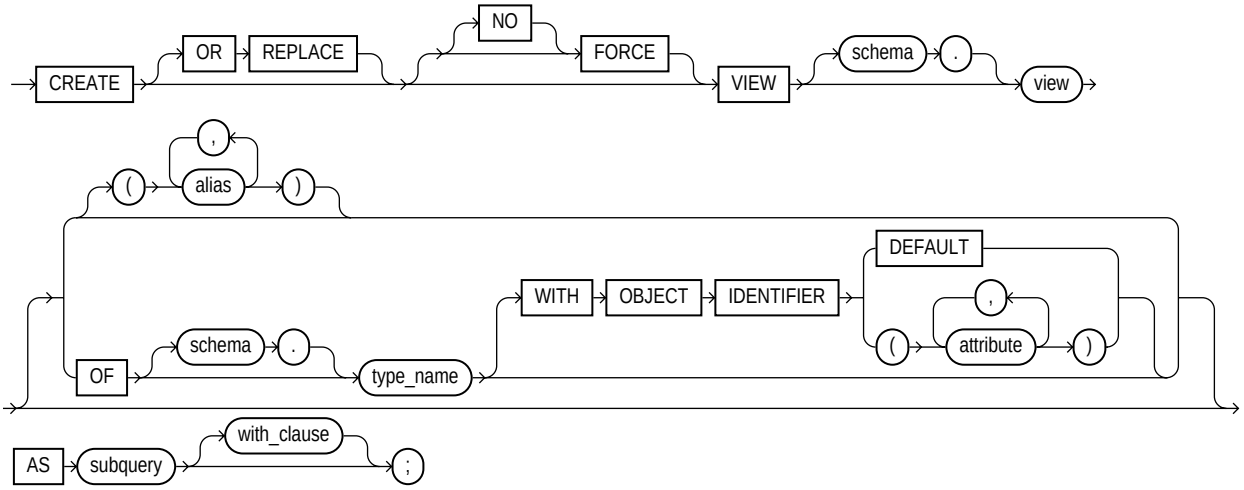
Partition views were introduced in Release 7.3 to provide partitioning capabilities for applications requiring them. Partition views are supported in Oracle8*i* so that you can upgrade applications from Release 7.3 without any modification. In most cases, subsequent to migration to Oracle8*i* you will want to migrate partition views into partitions.

With Oracle8*i*, you can use the CREATE TABLE statement to create partitioned tables easily. Partitioned tables offer the same advantages as partition views, while also addressing their shortcomings. Oracle recommends that you use partitioned tables rather than partition views in most operational environments.

See Also:

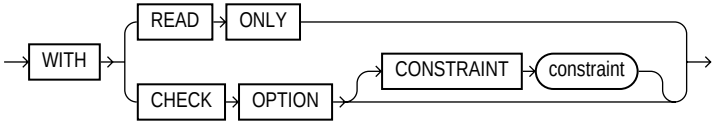
- *Oracle8i Concepts* for more information on the shortcomings of partition views
- *Oracle8i Administrator's Guide* for information on migrating partition views into partitions
- CREATE TABLE on page 10-7 for more information about partitioned tables

Syntax



subquery: See SELECT and subquery on page 11-88.

with_clause::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the view if it already exists. You can use this clause to change the definition of an existing view without dropping, re-creating, and regranting object privileges previously granted on it.

INSTEAD OF triggers defined in the view are dropped when a view is re-created.

If any materialized views are dependent on *view*, those materialized views will be marked UNUSABLE and will require a full refresh to restore them to a usable state. Invalid materialized views cannot be used by query rewrite and cannot be refreshed until they are recompiled.

See Also:

- ALTER MATERIALIZED VIEW on page 7-61 for information on refreshing invalid materialized views
- *Oracle8i Concepts* for information on materialized views in general
- CREATE TRIGGER on page 10-66 for more information about the INSTEAD OF clause

FORCE

Specify FORCE if you want to create the view regardless of whether the view's base tables or the referenced object types exist or the owner of the schema containing the view has privileges on them. These conditions must be true before any SELECT, INSERT, UPDATE, or DELETE statements can be issued against the view.

NO FORCE

Specify NOFORCE if you want to create the view only if the base tables exist and the owner of the schema containing the view has privileges on them. This is the default.

schema

Specify the schema to contain the view. If you omit *schema*, Oracle creates the view in your own schema.

view

Specify the name of the view or the object view.

Restriction: If a view has INSTEAD OF triggers, any views created on it must have INSTEAD OF triggers, even if the views are inherently updatable.

alias

Specify names for the expressions selected by the view's query. The number of aliases must match the number of expressions selected by the view. Aliases must follow the rules for naming Oracle schema objects. Aliases must be unique within the view.

If you omit the aliases, Oracle derives them from the columns or column aliases in the view's query. For this reason, you **must** use aliases if the view's query contains expressions rather than only column names.

Restriction: You cannot specify an alias when creating an object view.

See Also: "Syntax for Schema Objects and Parts in SQL Statements" on page 2-88

OF *type_name*

Use this clause to explicitly creates an object view of type *type_name*. The columns of an object view correspond to the top-level attributes of type *type_name*. Each row will contain an object instance and each instance will be associated with an object identifier (OID) as specified in the WITH OBJECT IDENTIFIER clause. If you omit *schema*, Oracle creates the object view in your own schema.

WITH OBJECT IDENTIFIER — The WITH OBJECT IDENTIFIER lets you specify the attributes of the object type that will be used as a key to identify each row in the object view. In most cases these attributes correspond to the primary-key columns of the base table. You must ensure that the attribute list is unique and identifies exactly one row in the view.

If you try to dereference or pin a primary key REF that resolves to more than one instance in the object view, Oracle raises an error.

Note: The 8.0 syntax WITH OBJECT OID is replaced with this syntax for clarity. The keywords WITH OBJECT OID are supported for backward compatibility, but Oracle Corporation recommends that you use the new syntax WITH OBJECT IDENTIFIER.

If the object view is defined on an object table or an object view, you can omit this clause or specify DEFAULT.

DEFAULT — Specify DEFAULT if you want Oracle to use the intrinsic object identifier of the underlying object table or object view to uniquely identify each row.

attribute — Specify an attribute of the object type from which Oracle should create the object identifier for the object view.

See Also: CREATE TYPE on page 10-80 for more information about creating objects

AS *subquery*

Specify a subquery that identifies columns and rows of the table(s) that the view is based on. The subquery's select list can contain up to 1000 expressions.

If you create views that refer to remote tables and views, the database links you specify must have been created using the CONNECT TO clause of the CREATE DATABASE LINK statement, and you must qualify them with schema name in the view query.

Restrictions on the view query:

- The view query cannot select the CURRVAL or NEXTVAL pseudocolumns.
- If the view query selects the ROWID, ROWNUM, or LEVEL pseudocolumns, those columns must have aliases in the view query.
- If the view query uses an asterisk (*) to select all columns of a table, and you later add new columns to the table, the view will not contain those columns until you re-create the view by issuing a CREATE OR REPLACE VIEW statement.
- For object views, the number of elements in the view subquery select list must be the same as the number of top-level attributes for the object type. The datatype of each of the selecting elements must be the same as the corresponding top-level attribute.
- You cannot specify the SAMPLE clause.

The preceding restrictions apply to materialized views as well.

- If you want the view to be inherently updatable, it must not contain any of the following constructs:
  - A set operator
  - A DISTINCT operator
  - An aggregate or analytic function
  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
  - A collection expression in a SELECT list
  - A subquery in a SELECT list
  - Joins (with some exceptions). See *Oracle8i Administrator's Guide* for details.
- If an inherently updatable view contains pseudocolumns or expressions, the UPDATE statement must not refer to any of these pseudocolumns or expressions.

- If you want a join view to be updatable, all of the following conditions must be true:
  - The DML statement must affect only one table underlying the join.
  - For an UPDATE statement, all columns updated must be extracted from a key-preserved table. If the view has the CHECK OPTION, join columns and columns taken from tables that are referenced more than once in the view must be shielded from UPDATE.
  - For a DELETE statement, the join can have one and only one key-preserved table. That table can appear more than once in the join, unless the view has the CHECK OPTION.
  - For an INSERT statement, all columns into which values are inserted must come from a key-preserved table, and the view must not have the CHECK OPTION.

See Also:

Oracle8i Administrator's Guide for more information on updatable views

Oracle8i Application Developer's Guide - Fundamentals for more information about updating object views or relational views that support object types

with_clause

Use the *with_clause* to restrict the subquery in one of the following ways:

WITH READ ONLY	Specify WITH READ ONLY if you want no deletes, inserts, or updates to be performed through the view.
WITH CHECK OPTION	Specify WITH CHECK OPTION to guarantee that inserts and updates performed through the view will result in rows that the view query can select. The CHECK OPTION cannot make this guarantee if: ■ There is a subquery in the query of this view or any view on which this view is based or ■ INSERT, UPDATE, or DELETE operations are performed using INSTEAD OF triggers.

CONSTRAINT *constraint* — Specify the name of the CHECK OPTION constraint. If you omit this identifier, Oracle automatically assigns the constraint a name of the form SYS_C*n*, where *n* is an integer that makes the constraint name unique within the database.

Examples

Basic View Example The following statement creates a view of the emptable named dept20. The view shows the employees in Department 20 and their annual salary:

```
CREATE VIEW dept20
    AS SELECT ename, sal*12 annual_salary
        FROM emp
        WHERE deptno = 20;
```

The view declaration need not define a name for the column based on the expression sal*12, because the subquery uses a column alias (annual_salary) for this expression.

Updatable View Example The following statement creates an updatable view named clerk of all clerks in the emp table. Only the employees' IDs, names, and department numbers are visible in this view and only these columns can be updated in rows identified as clerks:

```
CREATE VIEW clerk (id_number, person, department, position)
    AS SELECT empno, ename, deptno, job
       FROM emp
       WHERE job = 'CLERK'
    WITH CHECK OPTION CONSTRAINT wco;
```

Because of the CHECK OPTION, you cannot subsequently insert a new row into clerk if the new employee is not a clerk.

Join View Example A join view is one whose view query contains a join. If at least one column in the join has a unique index, then it may be possible to modify one base table in a join view. You can query USER_UPDATABLE_COLUMNS to see whether the columns in a join view are updatable. For example:

```
CREATE VIEW ed AS
   SELECT e.empno, e.ename, d.deptno, d.loc
      FROM emp e, dept d
   WHERE e.deptno = d.deptno
```

```
View created.

SELECT column_name, updatable
  FROM user_updatable_columns
  WHERE table_name = 'ED';

COLUMN_NAME     UPD
--------------- ---
ENAME           YES
DEPTNO          NO
EMPNO           YES
LOC             NO

INSERT INTO ed (ENAME, EMPNO) values ('BROWN', 1234);
```

In the above example, there is a unique index on the `deptno` column of the `dept` table. You can insert, update or delete a row from the `emp` base table, because all the columns in the view mapping to the `emp` table are marked as updatable and because the primary key of `emp` is included in the view.

Note: You cannot insert into the table using the view unless the view contains all `NOT NULL` columns of all tables in the join, unless you have specified `DEFAULT` values for the `NOT NULL` columns.

See Also: *Oracle8i Application Developer's Guide - Fundamentals* for more information on updating join views.

Read-Only View Example The following statement creates a read-only view named `clerk` of all clerks in the `emp` table. Only the employees' IDs, names, department numbers, and jobs are visible in this view:

```
CREATE VIEW clerk (id_number, person, department, position)
     AS SELECT empno, ename, deptno, job
       FROM emp
       WHERE job = 'CLERK'
    WITH READ ONLY;
```

Object View Example The following example creates object view `emp_object_view` of `employee_type`:

```
CREATE TYPE employee_type AS OBJECT
```

```
  ( empno       NUMBER(4),
    ename       VARCHAR2(20),
    job         VARCHAR2(9),
    mgr         NUMBER(4),
    hiredate    DATE,
    sal         NUMBER(7,2),
    comm        NUMBER(7,2)  );

CREATE OR REPLACE VIEW emp_object_view OF employee_type
  WITH OBJECT IDENTIFIER (empno)
  AS SELECT empno, ename, job, mgr, hiredate, sal, comm
     FROM emp;
```

DELETE

Purpose

Use the DELETE statement to remove rows from a table, a partitioned table, a view's base table, or a view's partitioned base table.

Prerequisites

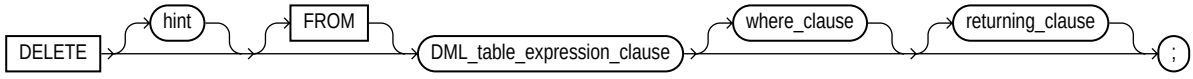
For you to delete rows from a table, the table must be in your own schema or you must have DELETE privilege on the table.

For you to delete rows from the base table of a view, the owner of the schema containing the view must have DELETE privilege on the base table. Also, if the view is in a schema other than your own, you must be granted DELETE privilege on the view.

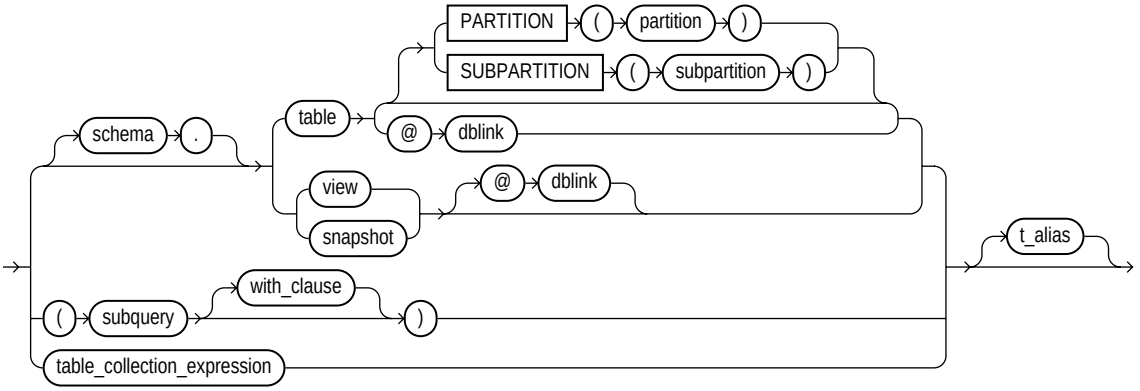
The DELETE ANY TABLE system privilege also allows you to delete rows from any table or table partition, or any view's base table.

If the SQL92_SECURITY initialization parameter is set to `true`, then you must have SELECT privilege on the table to perform a DELETE that references table columns (such as the columns in a *`where_clause`*).

Syntax

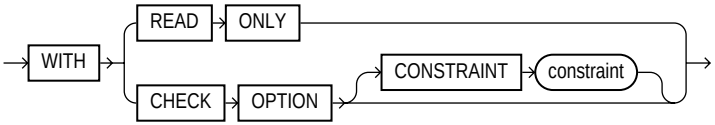


DML_table_expression_clause::=



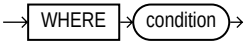
subquery: See SELECT and subquery on page 11-88.

with_clause::=

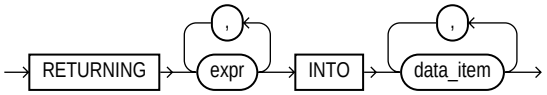


table_collection_expression::=

where_clause::=



returning_clause::=



Keywords and Parameters

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

> **See Also:** "Hints" on page 2-67 and *Oracle8i Performance Guide and Reference* for the syntax and description of hints

DML_table_expression_clause

schema

Specify the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table | *view* | *snapshot* | *subquery*

Specify is the name of a table or view, or the column or columns resulting from a subquery, from which the rows are to be deleted. If you specify *view*, Oracle deletes rows from the view's base table.

If table (or the base table of view) contains one or more domain index columns, this statements executes the appropriate indextype delete routine.

See Also: *Oracle8i Data Cartridge Developer's Guide* for more information on these routines

PARTITION *(partition_name)* and SUBPARTITION *(subpartition_name)*

Issuing a DELETE statement against a table fires any DELETE triggers defined on the table.

All table or index space released by the deleted rows is retained by the table and index.

Specify the name of the partition or subpartition within *table* targeted for deletes.

You need not specify the partition name when deleting values from a partitioned table. However, in some cases, specifying the partition name is more efficient than a complicated *where_clause*.

dblink Specify the complete or partial name of a database link to a remote database where the table or view is located. You can delete rows from a remote table or view only if you are using Oracle's distributed functionality.

> **See Also:** "Referring to Objects in Remote Databases" on page 2-90 for information on referring to database links

If you omit *dblink*, Oracle assumes that the table or view is located on the local database.

with_clause Use the *with_clause* to restrict the subquery in one of the following ways:

- WITH READ ONLY indicates that the subquery cannot be updated.
- WITH CHECK OPTION indicates that Oracle prohibits any changes to that table that would produce rows that are not included in the subquery.

  See Also: "WITH CHECK OPTION Example" on page 11-108

table_collection_expression The *table_collection_expression* lets you inform Oracle that the collection value expression should be treated as a table. You can use a *table_collection_expression* to delete only those rows that also exist in another table.

For *collection_expression*, specify a subquery that selects a nested table column from *table* or *view*.

Note: In earlier releases of Oracle, *table_collection_expression* was expressed as "THE *subquery*". That usage is now deprecated.

Restrictions on the DML_table_expression_clause:

- You cannot execute this statement if *table* (or the base table of *view*) contains any domain indexes marked LOADING or FAILED.
- You cannot specify the ORDER BY clause in the subquery of the *DML_query_expression_clause*.
- You cannot delete from a view except through INSTEAD OF triggers if the view's defining query contains one of the following constructs:
  - A set operator

  - A DISTINCT operator
  - An aggregate or analytic function
  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
  - A collection expression in a SELECT list
  - A subquery in a SELECT list
  - Joins (with some exceptions). See *Oracle8i Administrator's Guide* for details.
- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, the DELETE statement will fail unless the SKIP_UNUSABLE_INDEXES parameter has been set to `true`.

See Also: ALTER SESSION on page 7-105

where_clause

Use the *where_clause* to delete only rows that satisfy the condition. The condition can reference the table and can contain a subquery. You can delete rows from a remote table or view only if you are using Oracle's distributed functionality.

See Also: "Conditions" on page 5-15 for the syntax of *condition*

Note: If this clause contains a *subquery* that refers to remote objects, the DELETE operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if the *subquery* in the *DML_query_expression_clause* refers to any remote objects, the UPDATE operation will run serially without notification.

See Also: "parallel_clause" for CREATE TABLE on page 10-40

If you omit *dblink*, Oracle assumes that the table or view is located on the local database.

If you omit the *where_clause*, Oracle deletes all rows of the table or view.

t_alias — Provide a **correlation name** for the table, view, subquery, or collection value to be referenced elsewhere in the statement. Table aliases are generally used in DELETE statements with correlated queries.

> **Note:** This alias is required if the *DML_query_expression_clause* references any object type attributes or object type methods.

returning_clause

The returning clause retrieves the rows affected by a DML (INSERT, UPDATE, or DELETE) statement. You can specify this clause for tables and snapshots, and for views with a single base table.

- When operating on a single row, a DML statement with a *returning_clause* can retrieve column expressions using the affected row, rowid, and REFs to the affected row and store them in host variables or PL/SQL variables.
- When operating on multiple rows, a DML statement with the *returning_clause* stores values from expressions, rowids, and REFs involving the affected rows in bind arrays.

expr	Each item in the *expr* list must be a valid expression syntax.
INTO	The INTO clause indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item* list.
data_item	Each *data_item* is a host variable or PL/SQL variable that stores the retrieved *expr* value.

For each expression in the RETURNING list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the INTO list.

Restrictions:

- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.
- You cannot specify this clause for a view on which an INSTEAD OF trigger has been defined.

> **See Also:** *PL/SQL User's Guide and Reference* for information on using the BULK COLLECT clause to return multiple values to collection variables

Note: This clause lets you return values from deleted columns, and thereby eliminate the need to issue a SELECT statement following the DELETE statement.

Examples

Basic Examples The following statement deletes all rows from a table named temp_assign.

```
DELETE FROM temp_assign;
```

The following statement deletes from the emp table all sales staff who made less than $100 commission last month:

```
DELETE FROM emp
    WHERE JOB = 'SALESMAN'
    AND COMM < 100;
```

The following statement has the same effect as the preceding example, but uses a subquery:

```
DELETE FROM (select * from emp)
    WHERE JOB = 'SALESMAN'
    AND COMM < 100;
```

Remote Database Example The following statement deletes all rows from the accounts table owned by the user blake on a database accessible by the database link dallas:

```
DELETE FROM blake.accounts@dallas;
```

Nested Table Example The following example deletes rows of nested table projs where the department number is either 123 or 456, or the department's budget is greater than 456.78:

```
DELETE THE(SELECT projs
           FROM dept d WHERE d.dno = 123) AS p
   WHERE p.pno IN (123, 456) OR p.budgets > 456.78;
```

Partition Example The following example removes rows from partition nov98 of the sales table:

```
DELETE FROM sales PARTITION (nov98)
   WHERE amount_of_sale != 0;
```

RETURNING Clause Example The following example returns column `sal` from the deleted rows and stores the result in bind array :1:

```
DELETE FROM emp
   WHERE job = 'SALESMAN' AND COMM < 100
   RETURNING sal INTO :1;
```

DISASSOCIATE STATISTICS

Purpose

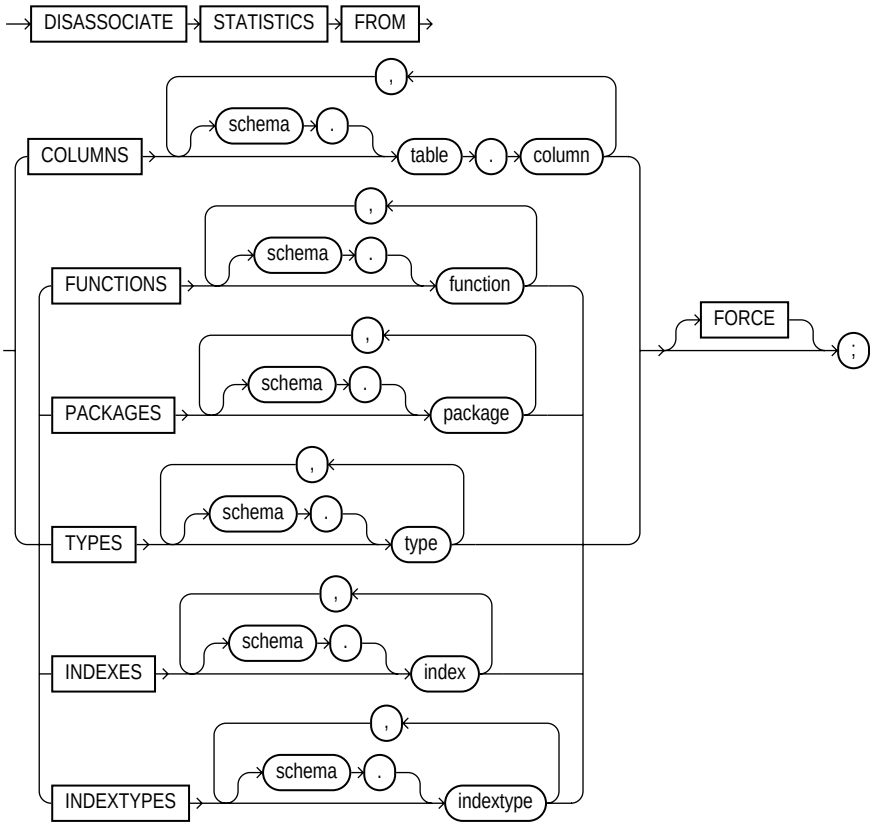
Use the `DISASSOCIATE STATISTICS` statement to disassociate a statistics type (or default statistics) from columns, standalone functions, packages, types, domain indexes, or indextypes.

> **See Also:** ASSOCIATE STATISTICS on page 8-110 for more information on statistics type associations

Prerequisites

To issue this statement, you must have the appropriate privileges to alter the base object (table, function, package, type, domain index, or indextype).

Syntax



Keywords and Parameters

FROM COLUMNS | FUNCTIONS | PACKAGES | TYPES | INDEXES | INDEXTYPES

Specify one or more columns, standalone functions, packages, types, domain indexes, or indextypes from which you are disassociating statistics.

If you do not specify *schema*, Oracle assumes the object is in your own schema.

If you have collected user-defined statistics on the object, the statement fails unless you specify FORCE.

FORCE

Specify FORCE to delete the association regardless of whether any statistics exist for the object using the statistics type. If statistics do exist, the statistics are deleted before the association is deleted.

> **Note:** When you drop an object with which a statistics type has been associated, Oracle automatically disassociates the statistics type with the FORCE option and drops all statistics that have been collected with the statistics type.

Example

Dissociating Statistics Example This statement disassociates statistics from the pack package in the hr schema:

```
DISASSOCIATE STATISTICS FROM PACKAGES hr.pack;
```

DROP CLUSTER

Purpose

Use the DROP CLUSTER clause to remove a cluster from the database.

You cannot uncluster an individual table. Instead you must perform these steps:

1. Create a new table with the same structure and contents as the old one, but with no CLUSTER clause.
2. Drop the old table.
3. Use the RENAME statement to give the new table the name of the old one.
4. Grant privileges on the new unclustered table, as grants on the old clustered table do not apply.

> **See Also:** CREATE TABLE on page 10-7, DROP TABLE on page 11-7, RENAME on page 11-71, GRANT on page 11-31 for information on these steps

Prerequisites

The cluster must be in your own schema or you must have the DROP ANY CLUSTER system privilege.

Syntax

Keywords and Parameters

schema

Specify the schema containing the cluster. If you omit *schema*, Oracle assumes the cluster is in your own schema.

cluster

Specify the name of the cluster to be dropped. Dropping a cluster also drops the cluster index and returns all cluster space, including data blocks for the index, to the appropriate tablespace(s).

INCLUDING TABLES

Specify INCLUDING TABLES to drop all tables that belong to the cluster.

CASCADE CONSTRAINTS

Specify CASCADE CONSTRAINTS to drop all referential integrity constraints from tables outside the cluster that refer to primary and unique keys in tables of the cluster. If you omit this clause and such referential integrity constraints exist, Oracle returns an error and does not drop the cluster.

Example

DROP CLUSTER Example This statement drops a cluster named geography, all its tables, and any referential integrity constraints that refer to primary or unique keys in those tables:

```
DROP CLUSTER geography
    INCLUDING TABLES
    CASCADE CONSTRAINTS;
```

DROP CONTEXT

Purpose

Use the DROP CONTEXT statement to remove a context namespace from the database.

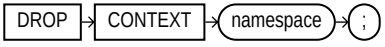
Note: Removing a context namespace does not invalidate any context under that namespace that has been set for a user session. However, the context will be invalid the next time the user attempts to set that context.

See Also: CREATE CONTEXT on page 9-13 and *Oracle8i Concepts* for more information on contexts

Prerequisites

You must have the DROP ANY CONTEXT system privilege.

Syntax



Keywords and Parameters

namespace

Specify the name of the context namespace to drop. You cannot drop the built-in namespace USERENV.

Example

DROP CONTEXT Example The following statement drops the context created in CREATE CONTEXT on page 9-13:

```
DROP CONTEXT hr_context;
```

DROP DATABASE LINK

Purpose

Use the DROP DATABASE LINK statement to remove a database link from the database.

> **See Also:** CREATE DATABASE LINK on page 9-28 for information on creating database links

Prerequisites

To drop a private database link, the database link must be in your own schema. To drop a PUBLIC database link, you must have the DROP PUBLIC DATABASE LINK system privilege.

Syntax

Keywords and Parameters

PUBLIC

You must specify PUBLIC to drop a PUBLIC database link.

dblink

Specify the name of the database link to be dropped.

Restriction: You cannot drop a database link in another user's schema, and you cannot qualify *dblink* with the name of a schema. The reason is that periods are permitted in names of database links. Therefore, Oracle interprets the entire name, such as ralph.linktosales, as the name of a database link in your schema rather than as a database link named linktosales in the schema ralph.

Example

Dropping a Database Link Example The following statement drops a private database link named `boston`:

```
DROP DATABASE LINK boston;
```

DROP DIMENSION

Purpose

Use the `DROP DIMENSION` statement to remove the named dimension.

> **Note:** This statement does not invalidate materialized views that use relationships specified in dimensions. However, requests that have been rewritten by query rewrite may be invalidated, and subsequent operations on such views may execute more slowly.

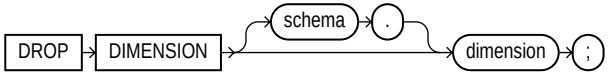
See Also:

- CREATE DIMENSION on page 9-34 for information on creating a dimension
- ALTER DIMENSION on page 7-34 for information on modifying a dimension
- *Oracle8i Concepts*

Prerequisites

The dimension must be in your own schema or you must have the `DROP ANY DIMENSION` system privilege to use this statement.

Syntax



Keywords and Parameters

schema

Specify the name of the schema in which the dimension is located. If you omit *schema*, Oracle assumes the dimension is in your own schema.

dimension

Specify the name of the dimension you want to drop. The dimension must already exist.

Example

DROP DIMENSION Example This example drops the `time` dimension:

```
DROP DIMENSION time;
```

DROP DIRECTORY

Purpose

Use the `DROP DIRECTORY` statement to remove a directory object from the database.

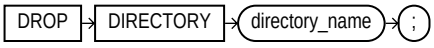
> **See Also:** CREATE DIRECTORY on page 9-40 for information on creating a directory

Prerequisites

To drop a directory you must have the `DROP ANY DIRECTORY` system privilege.

> **Caution:** Do not drop a directory when files in the associated file system are being accessed by PL/SQL or OCI programs.

Syntax



Keywords and Parameters

directory_name

Specify the name of the directory database object to be dropped.

Oracle removes the directory object, but does not delete the associated operating system directory on the server's file system.

Example

DROP DIRECTORY Example The following statement drops the directory object `bfile_dir`:

```
DROP DIRECTORY bfile_dir;
```

DROP FUNCTION

Purpose

Use the DROP FUNCTION statement to remove a standalone stored function from the database.

> **Note:** Do not use this statement to remove a function that is part of a package. Instead, either drop the entire package using the DROP PACKAGE statement or redefine the package without the function using the CREATE PACKAGE statement with the OR REPLACE clause.

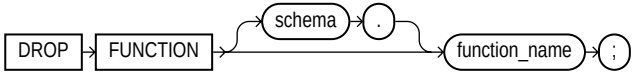
See Also:

- CREATE FUNCTION on page 9-43 for information on creating a function
- ALTER FUNCTION on page 7-38 for information on modifying a function

Prerequisites

The function must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the function. If you omit *schema*, Oracle assumes the function is in your own schema.

function_name

Specify the name of the function to be dropped.

Oracle invalidates any local objects that depend on, or call, the dropped function. If you subsequently reference one of these objects, Oracle tries to recompile the object and returns an error if you have not re-created the dropped function.

If any statistics types are associated with the function, Oracle disassociates the statistics types with the FORCE option and drops any user-defined statistics collected with the statistics type.

See Also:

- *Oracle8i Concepts* for more information on how Oracle maintains dependencies among schema objects, including remote objects
- ASSOCIATE STATISTICS on page 8-110 and DISASSOCIATE STATISTICS on page 10-123 for more information on statistics type associations

Example

DROP FUNCTION Example The following statement drops the function `new_acct` in the schema `riddley` and invalidates all objects that depend upon `new_acct`:

```
DROP FUNCTION riddley.new_acct;
```

DROP INDEX

Purpose

Use the `DROP INDEX` statement to remove an index or domain index from the database.

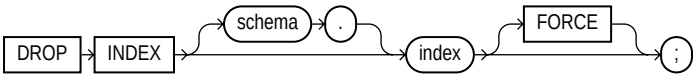
See Also:

- CREATE INDEX on page 9-52 for information on creating an index
- ALTER INDEX on page 7-40 for information on modifying an index
- The *domain_index_clause* of CREATE INDEX on page 9-52 for more information on domain indexes

Prerequisites

The index must be in your own schema or you must have the `DROP ANY INDEX` system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the index. If you omit *schema*, Oracle assumes the index is in your own schema.

index

Specify the name of the index to be dropped. When the index is dropped, all data blocks allocated to the index are returned to the index's tablespace.

If you drop a **domain index**:

- Oracle invokes the appropriate indextype drop routine. For information on these routines, see *Oracle8i Data Cartridge Developer's Guide.*
- In addition, if any statistics are associated with the domain index, Oracle disassociates the statistics types with the FORCE clause and removes the user-defined statistics collected with the statistics type.

> **See Also:** ASSOCIATE STATISTICS on page 8-110 and DISASSOCIATE STATISTICS on page 10-123 for more information on statistics type associations

If you drop a global partitioned index, a range-partitioned, or a hash-partitioned index, all the index partitions are also dropped. If you drop a a composite-partitioned index, all the index partitions and subpartitions are also dropped.

FORCE

FORCE applies only to domain indexes. This clause drops the domain index even if the indextype routine invocation returns an error or the index is marked LOADING. Without FORCE, you cannot drop a domain index if its indextype routine invocation returns an error or the index is marked LOADING.

Example

DROP INDEX Example This statement drops an index named monolith:

```
DROP INDEX monolith;
```

DROP INDEXTYPE

Purpose

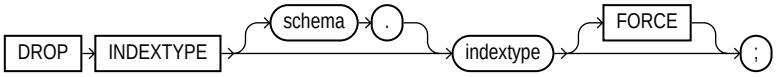
Use the DROP INDEXTYPE statement to drop an indextype, as well as any association with a statistics type.

> **See Also:** CREATE INDEXTYPE on page 9-76 for more information on indextypes

Prerequisites

The indextype must be in your own schema or you must have the DROP ANY INDEXTYPE system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the indextype. If you omit *schema*, Oracle assumes the indextype is in your own schema.

indextype

Specify the name of the indextype to be dropped.

If any statistics types have been associated with indextype, Oracle disassociates the statistics type from the indextype and drops any statistics that have been collected using the statistics type.

See Also: ASSOCIATE STATISTICS on page 8-110 and DISASSOCIATE STATISTICS on page 10-123 for more information on statistics associations

FORCE

Specify FORCE to drop the indextype even if the indextype is currently being referenced by one or more domain indexes, and marks those domain indexes INVALID. Without FORCE, you cannot drop an indextype if any domain indexes reference the indextype.

Example

DROP INDEXTYPE Example The following statement drops the indextype textindextype and marks INVALID any domain indexes defined on this indextype:

```
DROP INDEXTYPE textindextype FORCE;
```

DROP JAVA

Purpose

Use the DROP JAVA statement to drop a Java source, class, or resource schema object.

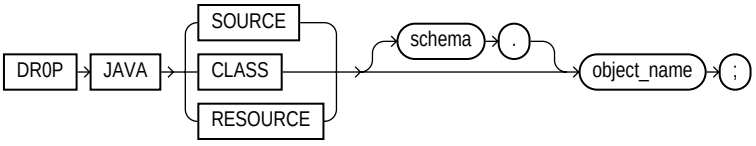
See Also:

- CREATE JAVA on page 9-79 for information on creating Java objects
- *Oracle8i Java Stored Procedures Developer's Guide* for more information on resolving Java sources, classes, and resources

Prerequisites

The Java source, class, or resource must be in your own schema or you must have the DROP ANY PROCEDURE system privilege. You also must have the EXECUTE object privilege on Java classes to use this command.

Syntax



Keywords and Parameters

JAVA SOURCE

Specify SOURCE to drop a Java source schema object and all Java class schema objects derived from it.

JAVA CLASS

Specify CLASS to drop a Java class schema object.

JAVA RESOURCE

Specify RESOURCE to drop a Java resource schema object.

object_name

Specify the name of an existing Java class, source, or resource schema object. Enclose the *object_name* in double quotation marks to preserve lower- or mixed-case names.

Example

DROP JAVA CLASS Example The following statement drops the Java class `MyClass`:

```
DROP JAVA CLASS "MyClass";
```

DROP LIBRARY

Purpose

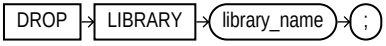
Use the DROP LIBRARY statement to remove an external procedure library from the database.

> **See Also:** CREATE LIBRARY on page 9-86 for information on creating a library

Prerequisites

You must have the DROP LIBRARY system privilege.

Syntax



Keywords and Parameters

library_name

Specify the name of the external procedure library being dropped.

Example

DROP LIBRARY Example The following statement drops the ext_procs library:

```
DROP LIBRARY ext_procs;
```

DROP MATERIALIZED VIEW

Purpose

Use the DROP MATERIALIZED VIEW statement to remove an existing materialized view from the database.

The terms "snapshot" and "materialized view" are synonymous.

See Also:

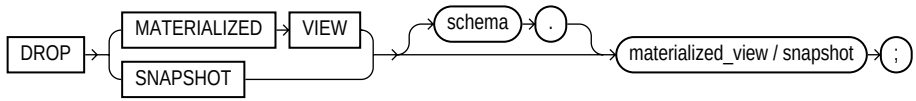
- CREATE MATERIALIZED VIEW on page 9-88 for more information on materialized views, including a description of the various types of materialized views
- ALTER MATERIALIZED VIEW on page 7-61 for information on modifying a materialized view
- *Oracle8i Replication* for information on materialized views in a replication environment
- *Oracle8i Data Warehousing Guide* for information on materialized views in a data warehousing environment

Prerequisites

The materialized view must be in your own schema or you must have the DROP ANY MATERIALIZED VIEW (or DROP ANY SNAPSHOT) system privilege. You must also have the privileges to drop the internal table, views, and index that Oracle uses to maintain the materialized view's data.

See Also: DROP TABLE on page 11-7, DROP VIEW on page 11-21, and DROP INDEX on page 10-136 for information on privileges required to drop objects that Oracle uses to maintain the materialized view

Syntax



Keywords and Parameters

schema

Specify the schema containing the materialized view. If you omit *schema*, Oracle assumes the materialized view is in your own schema.

materialized_view

Specify the name of the existing materialized view to be dropped.

- If you drop a simple materialized view that is the least recently refreshed materialized view of a master table, Oracle automatically purges from the detail table's materialized view log only the rows needed to refresh the dropped materialized view.
- If you drop a detail table, Oracle does not automatically drop materialized views based on the table. However, Oracle returns an error when it tries to refresh a materialized view based on a detail table that has been dropped.
- If you drop a materialized view, any compiled requests that were rewritten to use the materialized view will be invalidated and recompiled automatically. If the materialized view was prebuilt on a table, the table is not dropped, but it can no longer be maintained by the materialized view refresh mechanism.

Examples

DROP MATERIALIZED VIEW Examples The following statement drops the materialized view `parts` owned by the user `hq`:

```
DROP SNAPSHOT hq.parts;
```

The following statement drops the `sales_by_month` materialized view and the underlying table of the materialized view (unless the underlying table was registered in the `CREATE MATERIALIZED VIEW` statement with the `ON PREBUILT TABLE` clause):

```
DROP MATERIALIZED VIEW sales_by_month;
```

DROP MATERIALIZED VIEW LOG

Purpose

Use the `DROP MATERIALIZED VIEW LOG` statement to remove a materialized view log from the database.

The terms "snapshot" and "materialized view" are synonymous.

See Also:

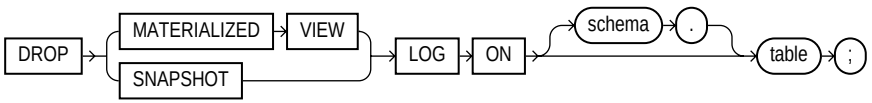
- CREATE MATERIALIZED VIEW on page 9-88 and ALTER MATERIALIZED VIEW on page 7-61 for more information on materialized views
- CREATE MATERIALIZED VIEW LOG on page 9-107 for information on materialized view logs
- *Oracle8i Replication* for information on materialized views in a replication environment
- *Oracle8i Data Warehousing Guide* for information on materialized views in a data warehousing environment

Prerequisites

A materialized view log consists of a table and a trigger. To drop a materialized view log, you must have the privileges needed to drop a table.

See Also: DROP TABLE on page 11-7

Syntax



Keywords and Parameters

schema

Specify the schema containing the materialized view log and its master table. If you omit *schema*, Oracle assumes the materialized view log and master table are in your own schema.

table

Specify the name of the detail table associated with the materialized view log to be dropped.

After you drop a materialized view log, some materialized views based on the materialized view log's detail table can no longer be fast refreshed. These materialized views include rowid materialized views, primary key materialized views, and subquery materialized views.

> **See Also:** *Oracle8i Data Warehousing Guide* for a description of the types of materialized views

Example

DROP MATERIALIZED VIEW LOG Example The following statement drops the materialized view log on the `parts` master table:

```
DROP MATERIALIZED VIEW LOG ON parts;
```

DROP OPERATOR

Purpose

Use the DROP OPERATOR statement to drop a user-defined operator.

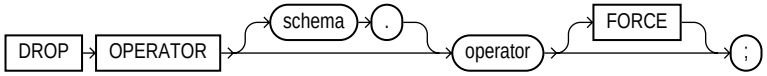
See Also:

- CREATE OPERATOR on page 9-115 for information on creating operators
- "User-Defined Operators" on page 3-16 and *Oracle8i Data Cartridge Developer's Guide* for more information on operators in general

Prerequisites

The operator must be in your schema or you must have the DROP ANY OPERATOR system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the operator. If you omit *schema*, Oracle assumes the operator is in your own schema.

operator

Specify the name of the operator to be dropped.

FORCE

Specify FORCE to drop the operator even if it is currently being referenced by one or more schema objects (indextypes, packages, functions, procedures, and so on), and marks those dependent objects INVALID. Without FORCE, you cannot drop an operator if any schema objects reference it.

Example

DROP OPERATOR Example The following statement drops the operator `merge`:

```
DROP OPERATOR ordsys.merge;
```

Because the FORCE clause is not specified, this operation will fail if any of the bindings of this operator are referenced by an indextype.

DROP OUTLINE

Purpose

Use the DROP OUTLINE statement to drop a stored outline.

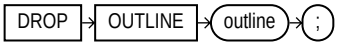
See Also:

- CREATE OUTLINE on page 9-119 for information on creating an outline
- *Oracle8i Performance Guide and Reference* for more information on outlines in general

Prerequisites

To drop an outline, you must have the DROP ANY OUTLINE system privilege.

Syntax



Keywords and Parameters

outline

Specify the name of the outline to be dropped.

After the outline is dropped, if the SQL statement for which the stored outline was created is compiled, the optimizer generates a new execution plan without the influence of the outline.

Example

DROP OUTLINE Example The following statement drops the stored outline called salaries.

```
DROP OUTLINE salaries;
```

DROP PACKAGE

Purpose

Use the DROP PACKAGE statement to remove a stored package from the database. This statement drops the body and specification of a package.

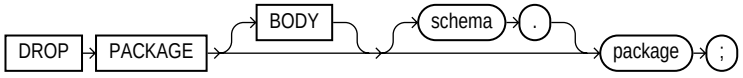
Note: Do not use this statement to remove a single object from a package. Instead, re-create the package without the object using the CREATE PACKAGE and CREATE PACKAGE BODY statements with the OR REPLACE clause.

See Also: CREATE PACKAGE on page 9-122

Prerequisites

The package must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

BODY

Specify BODY to drop only the body of the package. If you omit this clause, Oracle drops both the body and specification of the package.

When you drop only the body of a package but not its specification, Oracle does not invalidate dependent objects. However, you cannot call one of the procedures or stored functions declared in the package specification until you re-create the package body.

schema

Specify the schema containing the package. If you omit *schema*, Oracle assumes the package is in your own schema.

package

Specify the name of the package to be dropped.

Oracle invalidates any local objects that depend on the package specification. If you subsequently reference one of these objects, Oracle tries to recompile the object and returns an error if you have not re-created the dropped package.

If any statistics types are associated with the package, Oracle disassociates the statistics types with the FORCE clause and drops any user-defined statistics collected with the statistics types.

See Also:

- *Oracle8i Concepts* for information on how Oracle maintains dependencies among schema objects, including remote objects
- ASSOCIATE STATISTICS on page 8-110 and DISASSOCIATE STATISTICS on page 10-123

Example

DROP PACKAGE Example The following statement drops the specification and body of the banking package, invalidating all objects that depend on the specification:

```
DROP PACKAGE banking;
```

DROP PROCEDURE

Purpose

Use the DROP PROCEDURE statement to remove a standalone stored procedure from the database. Do not use this statement to remove a procedure that is part of a package. Instead, either drop the entire package using the DROP PACKAGE statement, or redefine the package without the procedure using the CREATE PACKAGE statement with the OR REPLACE clause.

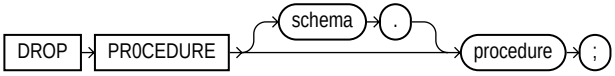
See Also:

- CREATE PROCEDURE on page 9-132 for information on creating a procedure
- ALTER PROCEDURE on page 7-88 for information on modifying a procedure

Prerequisites

The procedure must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the procedure. If you omit *schema*, Oracle assumes the procedure is in your own schema.

procedure

Specify the name of the procedure to be dropped.

When you drop a procedure, Oracle invalidates any local objects that depend upon the dropped procedure. If you subsequently reference one of these objects, Oracle

tries to recompile the object and returns an error message if you have not re-created the dropped procedure.

> **See Also:** *Oracle8i Concepts* for information on how Oracle maintains dependencies among schema objects, including remote objects

Example

DROP PROCEDURE Example The following statement drops the procedure `transfer` owned by the user `kerner` and invalidates all objects that depend upon `transfer`:

```
DROP PROCEDURE kerner.transfer
```

DROP PROFILE

Purpose

Use the DROP PROFILE statement to remove a profile from the database.

See Also:

- CREATE PROFILE on page 9-139 for information on creating a profile
- ALTER PROFILE on page 7-91 for information on modifying a profile

Prerequisites

You must have the DROP PROFILE system privilege.

Syntax

Keywords and Parameters

profile

Specify the name of the profile to be dropped.

Restriction: You cannot drop the DEFAULT profile.

CASCADE

Specify CASCADE to deassign the profile from any users to whom it is assigned. Oracle automatically assigns the DEFAULT profile to such users. You must specify this clause to drop a profile that is currently assigned to users.

Example

DROP PROFILE Example The following statement drops the profile engineer:

```
DROP PROFILE engineer CASCADE;
```

Oracle drops the profile `engineer` and assigns the `DEFAULT` profile to any users currently assigned the `engineer` profile.

DROP ROLE

Purpose

Use the DROP ROLE statement to remove a role from the database. When you drop a role, Oracle revokes it from all users and roles to whom it has been granted and removes it from the database.

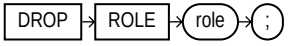
See Also:

- CREATE ROLE on page 9-146 for information on creating roles
- ALTER ROLE on page 7-98 for information on changing the authorization needed to enable a role
- SET ROLE on page 11-122 for information on disabling roles for the current session

Prerequisites

You must have been granted the role with the ADMIN OPTION or you must have the DROP ANY ROLE system privilege.

Syntax



Keywords and Parameters

role

Specify the name of the role to be dropped.

Example

DROP ROLE Example To drop the role florist, issue the following statement:

```
DROP ROLE florist;
```

DROP ROLLBACK SEGMENT

Purpose

Use the DROP ROLLBACK SEGMENT to remove a rollback segment from the database. When you drop a rollback segment, all space allocated to the rollback segment returns to the tablespace.

See Also:

- CREATE ROLLBACK SEGMENT on page 9-149 for information on creating a rollback segment
- ALTER ROLLBACK SEGMENT on page 7-100 for information on modifying a rollback segment
- CREATE TABLESPACE on page 10-56

Prerequisites

You must have the DROP ROLLBACK SEGMENT system privilege.

Syntax

Keywords and Parameters

rollback_segment

Specify the name the rollback segment to be dropped.

Restrictions:

- You can drop a rollback segment only if it is offline. To determine whether a rollback segment is offline, query the data dictionary view DBA_ROLLBACK_SEGS. Offline rollback segments have the value AVAILABLE in the STATUS column. You can take a rollback segment offline with the OFFLINE clause of the ALTER ROLLBACK SEGMENT statement.
- You cannot drop the SYSTEM rollback segment.

Example

DROP ROLLBACK SEGMENT Example The following statement drops the rollback segment `accounting`:

```
DROP ROLLBACK SEGMENT accounting;
```

11

SQL Statements: DROP SEQUENCE to UPDATE

This chapter contains the following SQL statements:

- DROP SEQUENCE
- DROP SYNONYM
- DROP TABLE
- DROP TABLESPACE
- DROP TRIGGER
- DROP TYPE
- DROP TYPE BODY
- DROP USER
- DROP VIEW
- EXPLAIN PLAN
- `filespec`
- GRANT
- INSERT
- LOCK TABLE
- NOAUDIT
- RENAME
- REVOKE

- ROLLBACK
- SAVEPOINT
- SELECT and subquery
- SET CONSTRAINT[S]
- SET ROLE
- SET TRANSACTION
- `storage_clause`
- TRUNCATE
- UPDATE

DROP SEQUENCE

Purpose

Use the DROP SEQUENCE statement to remove a sequence from the database.

You can also use this statement to restart a sequence by dropping and then re-creating it. For example, if you have a sequence with a current value of 150 and you would like to restart the sequence with a value of 27, you can drop the sequence and then re-create it with the same name and a START WITH value of 27.

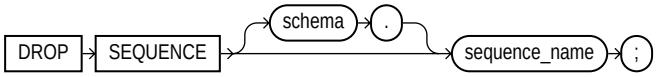
See Also:

- CREATE SEQUENCE on page 9-155 for information on creating a sequence
- ALTER SEQUENCE on page 7-103 for more information on modifying a sequence

Prerequisites

The sequence must be in your own schema or you must have the DROP ANY SEQUENCE system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the sequence. If you omit *schema*, Oracle assumes the sequence is in your own schema.

sequence_name

Specify the name of the sequence to be dropped.

Example

DROP SEQUENCE Example The following statement drops the sequence ESEQ owned by the user `elly`. To issue this statement, you must either be connected as user `elly` or have DROP ANY SEQUENCE system privilege:

```
DROP SEQUENCE elly.eseq;
```

DROP SYNONYM

Purpose

Use the DROP SYNONYM statement to remove a synonym from the database, or to change the definition of a synonym by dropping and re-creating it.

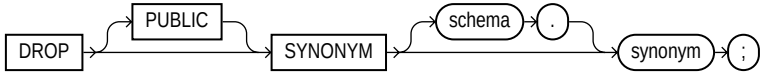
> **See Also:** CREATE SYNONYM on page 10-3 for more information on synonyms

Prerequisites

To drop a private synonym, either the synonym must be in your own schema or you must have the DROP ANY SYNONYM system privilege.

To drop a PUBLIC synonym, you must have the DROP PUBLIC SYNONYM system privilege.

Syntax



Keywords and Parameters

PUBLIC

You must specify PUBLIC to drop a public synonym. You cannot specify *schema* if you have specified PUBLIC.

schema

Specify the schema containing the synonym. If you omit *schema*, Oracle assumes the synonym is in your own schema.

synonym

Specify the name of the synonym to be dropped.

If you drop a synonym for a materialized view, or its containing table or snapshot, or any of its dependent tables, the materialized view will be invalidated.

Example

DROP SYNONYM Example To drop a synonym named `market`, issue the following statement:

```
DROP SYNONYM market;
```

DROP TABLE

Purpose

Use the DROP TABLE statement to remove a table or an object table and all its data from the database.

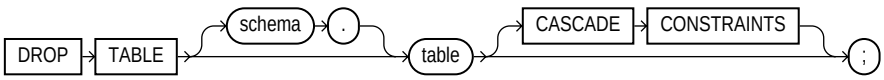
See Also:

- CREATE TABLE on page 10-7 for information on creating tables
- ALTER TABLE on page 8-2 for information on modifying tables

Prerequisites

The table must be in your own schema or you must have the DROP ANY TABLE system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the table. If you omit *schema*, Oracle assumes the table is in your own schema.

table

Specify the name of the table, object table, or index-organized table to be dropped. Oracle automatically performs the following operations:

- Removes all rows from the table (as if the rows were deleted).
- Drops all the table's indexes and domain indexes, regardless of who created them or whose schema contains them.

- If you drop a **range-partitioned or hash-partitioned** table, all the table partitions are also dropped. If you drop a composite-partitioned table, all the partitions and subpartitions are also dropped.
- For a **domain index**, this statement invokes the appropriate drop routines.

  See Also: *Oracle8i Data Cartridge Developer's Guide* for more information on these routines

- If any **statistic types** are associated with the table, Oracle disassociates the statistics types with the FORCE clause and removes any user-defined statistics collected with the statistics type.

  See Also: ASSOCIATE STATISTICS on page 8-110 and DISASSOCIATE STATISTICS on page 10-123 for more information on statistics type associations

- If the table is not part of a **cluster**, Oracle returns all data blocks allocated to the table and its indexes to the tablespaces containing the table and its indexes.

  Note: To drop a cluster and all its the tables, use the DROP CLUSTER statement with the INCLUDING TABLES clause to avoid dropping each table individually. See DROP CLUSTER on page 10-126.

- If the table is a **base table for a view**, a container or master table of a materialized view, or if it is referenced in a stored procedure, function, or package, Oracle invalidates these dependent objects but does not drop them. You cannot use these objects unless you re-create the table or drop and re-create the objects so that they no longer depend on the table.
- If you choose to **re-create** the table, it must contain all the columns selected by the queries originally used to define the materialized views/snapshots and all the columns referenced in the stored procedures, functions, or packages. Any users previously granted object privileges on the views, stored procedures, functions, or packages need not be regranted these privileges.
- If the table is a **detail table for a materialized view**, the materialized view can still be queried, but it cannot be refreshed unless the table is re-created so that it contains all the columns selected by the materialized view's query.

- If the table has a **materialized view log**, Oracle drops this log and any other direct-load INSERT refresh information associated with the table.

CASCADE CONSTRAINTS

Specify CASCADE CONSTRAINTS to drop all referential integrity constraints that refer to primary and unique keys in the dropped table. If you omit this clause, and such referential integrity constraints exist, Oracle returns an error and does not drop the table.

Example

DROP TABLE Example The following statement drops the test_data table:

```
DROP TABLE test_data;
```

DROP TABLESPACE

Purpose

Use the DROP TABLESPACE statement to remove a tablespace from the database.

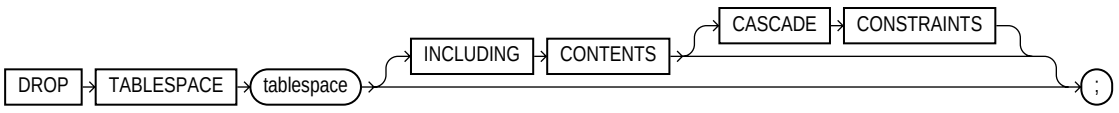
See Also:

- CREATE TABLESPACE on page 10-56 for information on creating a tablespace
- ALTER TABLESPACE on page 8-67 for information on modifying a tablespace

Prerequisites

You must have the DROP TABLESPACE system privilege. You cannot drop a tablespace if it contains any rollback segments holding active transactions.

Syntax



Keywords and parameters

tablespace

Specify the name of the tablespace to be dropped.

You can drop a tablespace regardless of whether it is online or offline. Oracle recommends that you take the tablespace offline before dropping it to ensure that no SQL statements in currently running transactions access any of the objects in the tablespace.

You may want to alert any users who have been assigned the tablespace as either a default or temporary tablespace. After the tablespace has been dropped, these users cannot allocate space for objects or sort areas in the tablespace. You can reassign users new default and temporary tablespaces with the ALTER USER statement.

Restrictions:

- You cannot drop the SYSTEM tablespace.
- You cannot drop a tablespace that contains a domain index or any objects created by a domain index.

> **See Also:** *Oracle8i Data Cartridge Developer's Guide* and *Oracle8i Concepts* for more information on domain indexes

INCLUDING CONTENTS

Specify INCLUDING CONTENTS to drop all the contents of the tablespace. You must specify this clause to drop a tablespace that contains any database objects. If you omit this clause, and the tablespace is not empty, Oracle returns an error and does not drop the tablespace.

For **partitioned tables**, DROP TABLESPACE will fail even if you specify INCLUDING CONTENTS, if the tablespace contains some, but not all,

- Partitions of a range- or hash-partitioned table, or
- Subpartitions of a composite-partitioned table.

> **Note:** If all the partitions of a partitioned table reside in *tablespace*, DROP TABLESPACE ... INCLUDING CONTENTS will drop *tablespace*, as well as any associated index segments, LOB data segments, and LOB index segments in the other tablespace(s).

For a **partitioned index-organized table**, if all the primary key index segments are in this tablespace, this clause will also drop any overflow segments that exist in other tablespaces. If some of the primary key index segments are *not* in this tablespace, the statement will fail. In that case, before you can drop the tablespace, you must use ALTER TABLE ... MOVE PARTITION to move those primary key index segments into this tablespace, drop the partitions whose overflow data segments are not in this tablespace, and drop the partitioned index-organized table.

If the tablespace contains a container table or detail table of a materialized view, Oracle invalidates the materialized view.

If the tablespace contains a materialized view / snapshot log, Oracle drops this log and any other direct-load INSERT refresh information associated with the table.

CASCADE CONSTRAINTS

Specify CASCADE CONSTRAINTS to drop all referential integrity constraints from tables outside *tablespace* that refer to primary and unique keys of tables inside *tablespace*. If you omit this clause and such referential integrity constraints exist, Oracle returns an error and does not drop the tablespace.

Example

DROP TABLESPACE Example The following statement drops the mfrg tablespace and all its contents:

```
DROP TABLESPACE mfrg
    INCLUDING CONTENTS
        CASCADE CONSTRAINTS;
```

DROP TRIGGER

Purpose

Use the DROP TRIGGER statement to remove a database trigger from the database.

See Also:

- CREATE TRIGGER on page 10-66 for information on creating triggers
- ALTER TRIGGER on page 8-76 for information on enabling, disabling, and compiling triggers

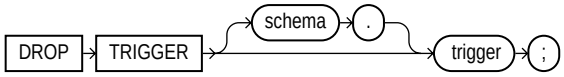
Prerequisites

The trigger must be in your own schema or you must have the DROP ANY TRIGGER system privilege.

In addition, to drop a trigger on DATABASE in another user's schema, you must have the ADMINISTER DATABASE TRIGGER system privilege.

See Also: CREATE TRIGGER on page 10-66 for information on these privileges

Syntax



Keywords and Parameters

schema

Specify the schema containing the trigger. If you omit *schema*, Oracle assumes the trigger is in your own schema.

trigger

Specify the name of the trigger to be dropped. Oracle removes it from the database and does not fire it again.

Example

DROP TRIGGER Example The following statement drops the `reorder` trigger in the schema `ruth`:

```
DROP TRIGGER ruth.reorder;
```

DROP TYPE

Purpose

Use the DROP TYPE statement to drop the specification and body of an object, a varray, or nested table type.

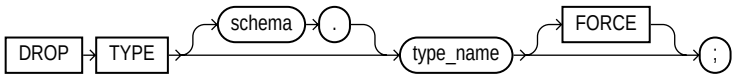
See Also:

- DROP TYPE BODY on page 11-17 for information on dropping just the body of an object type
- CREATE TYPE on page 10-80 for information on creating types.

Prerequisites

The object, varray, or nested table type must be in your own schema or you must have the DROP ANY TYPE system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the type. If you omit *schema*, Oracle assumes the type is in your own schema.

type_name

Specify the name of the object, varray, or nested table type to be dropped. You can drop only types with no type or table dependencies.

If *type_name* is a statistics type, this statement will fail unless you also specify FORCE. If you specify FORCE, Oracle first disassociates all objects that are associated with *type_name*, and then drops *type_name*.

> **See Also:** ASSOCIATE STATISTICS on page 8-110 and DISASSOCIATE STATISTICS on page 10-123 for more information on statistics types

If *type_name* is an object type that has been associated with a statistics type, Oracle first attempts to disassociate *type_name* from the statistics type and then drop *type_name*. However, if statistics have been collected using the statistics type, Oracle will be unable to disassociate *type_name* from the statistics type, and this statement will fail.

If *type_name* is an implementation type for an indextype, the indextype will be marked INVALID.

Unless you specify FORCE, you can drop only object, nested table, or varray types that are standalone schema objects with no dependencies. This is the default behavior.

> **See Also:** CREATE INDEXTYPE on page 9-76

FORCE

Specify FORCE to drop the type even if it has dependent database objects. Oracle marks UNUSED all columns dependent on the type to be dropped, and those columns become inaccessible.

Caution: Oracle does not recommend that you specify FORCE to drop types with dependencies. This operation is not recoverable and could cause the data in the dependent tables or columns to become inaccessible. For information about type dependencies, see *Oracle8i Application Developer's Guide - Fundamentals.*

Example

DROP TYPE Example The following statement removes object type person_t:

```
DROP TYPE person_t;
```

DROP TYPE BODY

Purpose

Use the DROP TYPE BODY statement to drop the body of an object, varray, or nested table type. When you drop a type body, the object type specification still exists, and you can re-create the type body. Prior to re-creating the body, you can still use the object type, although you cannot call the member functions.

See Also:

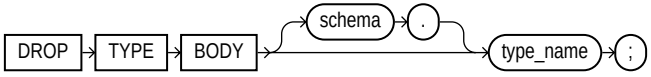
- DROP TYPE on page 11-15 for information on dropping the specification of an object along with the body
- CREATE TYPE BODY on page 10-93 for more information on type bodies

Prerequisites

The object type body must be in your own schema, and you must have

- The CREATE TYPE or CREATE ANY TYPE system privilege, or
- The DROP ANY TYPE system privilege

Syntax



Keywords and Parameters

schema

Specify the schema containing the object type. If you omit *schema*, Oracle assumes the object type is in your own schema.

type_name

Specify the name of the object type body to be dropped.

Restriction: You can drop a type body only if it has no type or table dependencies.

Example

DROP TYPE BODY Example The following statement removes object type body `rational`:

```
DROP TYPE BODY rational;
```

DROP USER

Purpose

Use the DROP USER statement to remove a database user and optionally remove the user's objects.

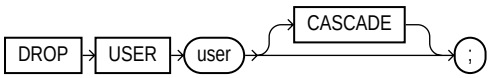
See Also:

- CREATE USER on page 10-99 for information on creating a user
- ALTER USER on page 8-88 for information on modifying the definition of a user

Prerequisites

You must have the DROP USER system privilege.

Syntax



Keywords and Parameters

user

Specify the user to be dropped. Oracle does not drop users whose schemas contain objects unless you specify CASCADE, or unless you first explicitly drop the user's objects.

CASCADE

Specify CASCADE to drop all objects in the user's schema before dropping the user. You must specify this clause to drop a user whose schema contains any objects.

- If the user's schema contains tables, Oracle drops the tables and automatically drops any referential integrity constraints on tables in other schemas that refer to primary and unique keys on these tables.

- If this clause results in tables being dropped, Oracle also drops all domain indexes created on columns of those tables, and invokes appropriate drop routines.

  See Also: *Oracle8i Data Cartridge Developer's Guide* for more information on these routines

- Oracle invalidates, but does not drop, the following objects in **other** schemas: views or synonyms for objects in the dropped user's schema; and stored procedures, functions, or packages that query objects in the dropped user's schema.
- Oracle does not drop materialized views on tables or views in the dropped user's schema, but if you specify CASCADE, the materialized views can no longer be refreshed.
- Oracle drops all triggers in the user's schema.
- Oracle does **not** drop roles created by the user.

Caution: Oracle also drops with FORCE all types owned by the user. See the FORCE keyword of DROP TYPE on page 11-16.

Examples

DROP USER Example If user Bradley's schema contains no objects, you can drop `bradley` by issuing the statement:

```
DROP USER bradley;
```

If Bradley's schema contains objects, you must use the CASCADE clause to drop `bradley` and the objects:

```
DROP USER bradley CASCADE;
```

DROP VIEW

Purpose

Use the `DROP VIEW` statement to remove a view or an object view from the database. You can change the definition of a view by dropping and re-creating it.

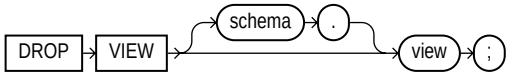
See Also:

- CREATE VIEW on page 10-105 for information on creating a view
- ALTER VIEW on page 8-94 for information on modifying a view

Prerequisites

The view must be in your own schema or you must have the `DROP ANY VIEW` system privilege.

Syntax



Keywords and Parameters

schema

Specify the schema containing the view. If you omit *schema*, Oracle assumes the view is in your own schema.

view

Specify the name of the view to be dropped.

Views, materialized views, and synonyms that refer to the view are not dropped, but become invalid. You can drop them or redefine views and synonyms, or you can define other views in such a way that the invalid views and synonyms become valid again.

See Also:

- CREATE TABLE on page 10-7 and CREATE SYNONYM on page 10-3
- ALTER MATERIALIZED VIEW on page 7-61 for information on revalidating invalid materialized views

Example

DROP VIEW Example The following statement drops the `view_data` view:

```
DROP VIEW view_data;
```

EXPLAIN PLAN

Purpose

Use the EXPLAIN PLAN statement to determine the execution plan Oracle follows to execute a specified SQL statement. This statement inserts a row describing each step of the execution plan into a specified table. If you are using cost-based optimization, this statement also determines the cost of executing the statement. If any domain indexes are defined on the table, user-defined CPU and I/O costs will also be inserted.

The definition of a sample output table PLAN_TABLE is available in a SQL script on your distribution media. Your output table must have the same column names and datatypes as this table. The common name of this script is UTLXPLAN.SQL. The exact name and location depend on your operating system.

You can also issue the EXPLAIN PLAN statement as part of the SQL trace facility.

See Also:

- *Oracle8i Performance Guide and Reference* for information on the output of EXPLAIN PLAN
- *Oracle8i Performance Guide and Reference* for information on how to use the SQL trace facility, as well as a detailed discussion of how to generate and interpret execution plans

Prerequisites

To issue an EXPLAIN PLAN statement, you must have the privileges necessary to insert rows into an existing output table that you specify to hold the execution plan.

You must also have the privileges necessary to execute the SQL statement for which you are determining the execution plan. If the SQL statement accesses a view, you must have privileges to access any tables and views on which the view is based. If the view is based on another view that is based on a table, you must have privileges to access both the other view and its underlying table.

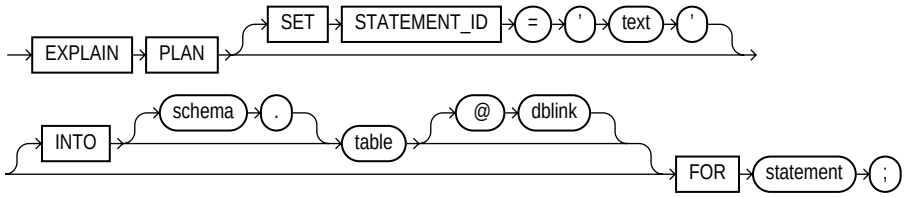
To examine the execution plan produced by an EXPLAIN PLAN statement, you must have the privileges necessary to query the output table.

The EXPLAIN PLAN statement is a data manipulation language (DML) statement, rather than a data definition language (DDL) statement. Therefore, Oracle does not implicitly commit the changes made by an EXPLAIN PLAN statement. If you want to

keep the rows generated by an EXPLAIN PLAN statement in the output table, you must commit the transaction containing the statement.

> **See Also:** INSERT on page 11-51 and SELECT and subquery on page 11-88 information on the privileges you need to populate and query the plan table

Syntax



Keywords and Parameters

SET STATEMENT_ID = '*text*'

Specify the value of the STATEMENT_ID column for the rows of the execution plan in the output table. You can then use this value to identify these rows among others in the output table. Be sure to specify a STATEMENT_ID value if your output table contains rows from many execution plans. If you omit this clause, the STATEMENT_ID value defaults to null.

INTO *table*

Specify the name of the output table, and optionally its schema and database. This table must exist before you use the EXPLAIN PLAN statement.

If you omit *schema*, Oracle assumes the table is in your own schema.

The *dblink* can be a complete or partial name of a database link to a remote Oracle database where the output table is located. You can specify a remote output table only if you are using Oracle's distributed functionality. If you omit *dblink*, Oracle assumes the table is on your local database.

> **See Also:** "Referring to Objects in Remote Databases" on page 2-90 for information on referring to database links

If you omit INTO altogether, Oracle assumes an output table named PLAN_TABLE in your own schema on your local database.

FOR *statement*

Specify a SELECT, INSERT, UPDATE, DELETE, CREATE TABLE, CREATE INDEX, or ALTER INDEX ... REBUILD statement for which the execution plan is generated.

Notes:

- If *statement* includes the *parallel_clause*, the resulting execution plan will indicate parallel execution. However, EXPLAIN PLAN actually inserts the statement into the plan table, so that the parallel DML statement you submit is no longer the first DML statement in the transaction. This violates the Oracle restriction of one parallel DML statement per transaction, and the statement will be executed serially. To maintain parallel execution of the statements, you must commit or roll back the EXPLAIN PLAN statement, and then submit the parallel DML statement.
- To determine the execution plan for an operation on a temporary table, EXPLAIN PLAN must be run from the same session, because the data in temporary tables is session specific.

Examples

EXPLAIN PLAN Examples The following statement determines the execution plan and cost for an UPDATE statement and inserts rows describing the execution plan into the specified output table with the STATEMENT_ID value of 'Raise in Chicago':

```
EXPLAIN PLAN
    SET STATEMENT_ID = 'Raise in Chicago'
    INTO output
    FOR UPDATE emp
        SET sal = sal * 1.10
        WHERE deptno =  (SELECT deptno
                  FROM dept
                  WHERE loc = 'CHICAGO');
```

The following SELECT statement queries the output table and returns the execution plan and the cost:

```
SELECT LPAD(' ',2*(LEVEL-1))||operation operation, options,
object_name, position
    FROM output
    START WITH id = 0 AND statement_id = 'Raise in Chicago'
    CONNECT BY PRIOR id = parent_id AND
    statement_id = 'Raise in Chicago';
```

The query returns this execution plan:

```
OPERATION             OPTIONS     OBJECT_NAME     POSITION
-------------------------------------------------------------
UPDATE STATEMENT                                         1
    FILTER                                               0
      TABLE ACCESS    FULL        EMP                    1
      TABLE ACCESS    FULL        DEPT                   2
```

The value in the POSITION column of the first row shows that the statement has a cost of 1.

EXPLAIN PLAN: Partitioned Example Assume that stocks is a table with eight partitions on a stock_num column, and that a local prefixed index stock_ix on column stock_num exists. The partition HIGHVALUES are 1000, 2000, 3000, 4000, 5000, 6000, 7000, and 8000.

Consider the query:

```
SELECT * FROM stocks WHERE stock_num BETWEEN 3800 AND :h;
```

(where :h represents a bind variable). EXPLAIN PLAN executes this query with PLAN_TABLE as the output table. The basic execution plan, including partitioning information, is obtained with the query:

```
SELECT id, operation, options, object_name,
  partition_start, partition_stop, partition_id FROM plan_table;
```

filespec

Purpose

Use the *filespec* syntax to specify a file as a datafile or tempfile, or to specify a group of one or more files as a redo log file group.

Prerequisites

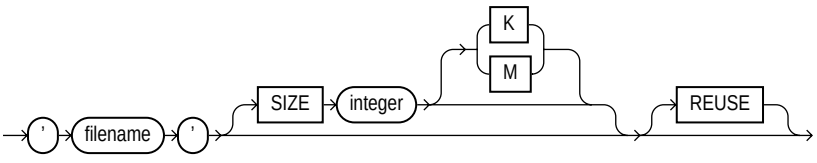
A *filespec* can appear in the statements CREATE DATABASE, ALTER DATABASE, CREATE TABLESPACE, ALTER TABLESPACE, CREATE CONTROLFILE, CREATE LIBRARY, and CREATE TEMPORARY TABLESPACE. You must have the privileges necessary to issue one of these statements.

See Also:

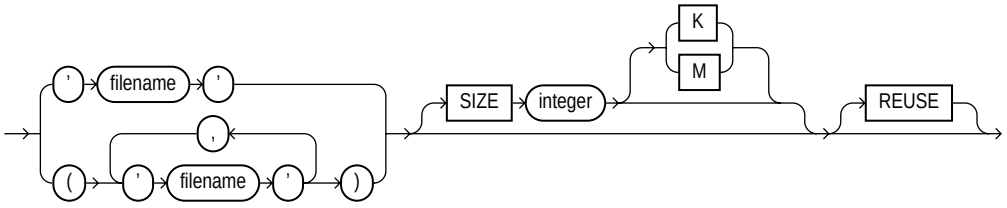
- CREATE DATABASE on page 9-21
- ALTER DATABASE on page 7-9
- CREATE TABLESPACE on page 10-56
- ALTER TABLESPACE on page 8-67
- CREATE CONTROLFILE on page 9-15
- CREATE LIBRARY on page 9-86
- CREATE TEMPORARY TABLESPACE on page 10-63

Syntax

filespec_datafiles & filespec_tempfiles::=



filespec_redo_log_file_groups::=



Keywords and Parameters

'filename'

Specify the name of either a datafile, tempfile, or a redo log file member. A '*filename*' can contain only single-byte characters from 7-bit ASCII or EBCDIC character sets. Multibyte characters are not valid.

A redo log file group can have one or more members (copies). Each '*filename*' must be fully specified according to the conventions for your operating system.

SIZE *integer*

Specify the size of the file in bytes. Use K or M to specify the size in kilobytes or megabytes.

- You can omit this parameter **only** if the file already exists.
- The size of a tablespace must be one block greater than the sum of the sizes of the objects contained in it.

REUSE

Specify REUSE to allow Oracle to reuse an existing file.

- If the file already exists, Oracle verifies that its size matches the value of the SIZE parameter (if you specify SIZE).
- If the file does not exist, Oracle ignores this clause and creates the file.
- You can omit this clause only if the file does not already exist. If you omit this clause, Oracle creates the file.

Note: Whenever Oracle uses an existing file, the file's previous contents are lost.

Examples

Specifying a Log File Example The following statement creates a database named payable that has two redo log file groups, each with two members, and one datafile:

```
CREATE DATABASE payable
   LOGFILE GROUP 1 ('diska:log1.log', 'diskb:log1.log') SIZE 50K,
           GROUP 2 ('diska:log2.log', 'diskb:log2.log') SIZE 50K
   DATAFILE 'diskc:dbone.dat' SIZE 30M;
```

The first *filespec* in the LOGFILE clause specifies a redo log file group with the GROUP value 1. This group has members named 'diska:log1.log' and 'diskb:log1.log', each 50 kilobytes in size.

The second *filespec* in the LOGFILE clause specifies a redo log file group with the GROUP value 2. This group has members named 'diska:log2.log' and 'diskb:log2.log', also 50 kilobytes in size.

The *filespec* in the DATAFILE clause specifies a datafile named 'diskc:dbone.dat', 30 megabytes in size.

Each *filespec* specifies a value for the SIZE parameter and omits the REUSE clause, so none of these files can already exist. Oracle must create them.

Adding a Log File Example The following statement adds another redo log file group with two members to the payable database:

```
ALTER DATABASE payable
   ADD LOGFILE GROUP 3 ('diska:log3.log', 'diskb:log3.log')
   SIZE 50K REUSE;
```

The *filespec* in the ADD LOGFILE clause specifies a new redo log file group with the GROUP value 3. This new group has members named 'diska:log3.log' and 'diskb:log3.log', each 50 kilobytes in size. Because the *filespec* specifies the REUSE clause, each member can already exist. If a member exists, it must have a size of 50 kilobytes. If it does not exist, Oracle creates it with that size.

Specifying a Datafile Example The following statement creates a tablespace named stocks that has three datafiles:

```
CREATE TABLESPACE stocks
   DATAFILE 'diskc:stock1.dat',
            'diskc:stock2.dat',
            'diskc:stock3.dat';
```

The *filespecs* for the datafiles specifies files named 'diskc:stock1.dat', 'diskc:stock2.dat', and 'diskc:stock3.dat'. Since each *filespec* omits the SIZE parameter, each file must already exist.

Adding a Datafile Example The following statement alters the stocks tablespace and adds a new datafile:

```
ALTER TABLESPACE stocks
   ADD DATAFILE 'diskc:stock4.dat' REUSE;
```

The *filespec* specifies a datafile named 'diskc:stock4.dat'. Since the *filespec* omits the SIZE parameter, the file must already exist and the REUSE clause is not significant.

GRANT

Purpose

Use the GRANT statement to grant:

- **System privileges** to users and roles
- **Roles** to users and roles. Both privileges and roles are either local, global, or external. Table 11–1 lists the system privileges (organized by the database object operated upon). Table 11–2 lists Oracle predefined roles.
- **Object privileges** for a particular object to users, roles, and PUBLIC. Table 11–3 summarizes the object privileges that you can grant on each type of object. Table 11–4 lists object privileges and the operations that they authorize. You can grant any of these system privileges with the GRANT statement.

Note: You can authorize database users to use roles through means other than the database and the GRANT statement. For example, some operating systems have facilities that let you grant roles to Oracle users with the initialization parameter OS_ROLES. If you choose to grant roles to users through operating system facilities, you cannot also grant roles to users with the GRANT statement, although you can use the GRANT statement to grant system privileges to users and system privileges and roles to other roles.

See Also:

- CREATE USER on page 10-99 and CREATE ROLE on page 9-146 for definitions of local, global, and external privileges
- *Oracle8i Administrator's Guide* for information about other authorization methods
- REVOKE on page 11-73 for information on revoking grants

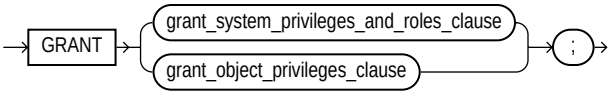
Prerequisites

To grant a **system privilege**, you must either have been granted the system privilege with the ADMIN OPTION or have been granted the GRANT ANY PRIVILEGE system privilege.

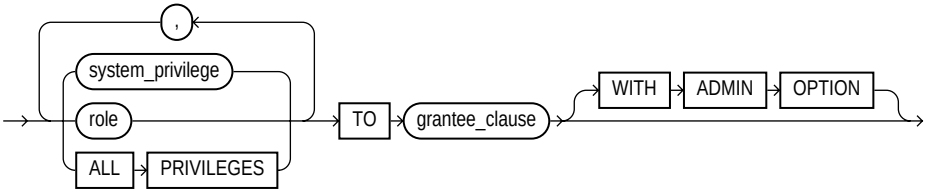
To grant a **role**, you must either have been granted the role with the ADMIN OPTION or have been granted the GRANT ANY ROLE system privilege, or you must have created the role.

To grant an **object privilege**, you must own the object or the owner of the object must have granted you the object privileges with the GRANT OPTION. This rule applies to users with the DBA role.

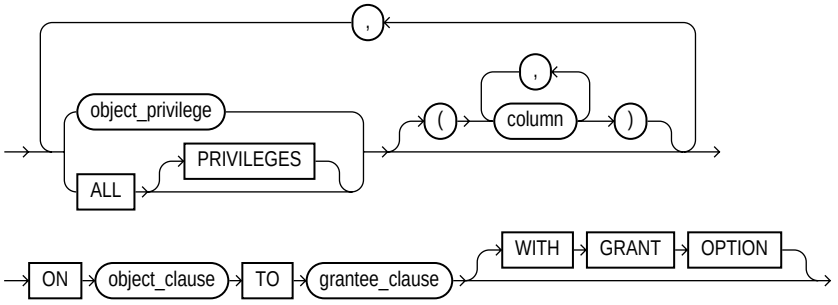
Syntax



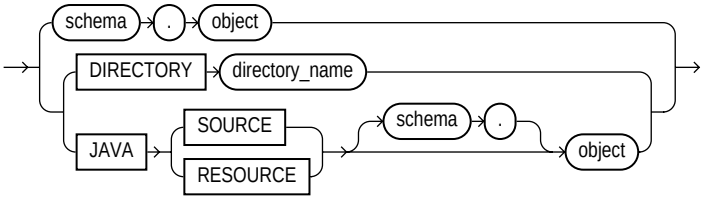
grant_system_privileges_and_roles_clause::=



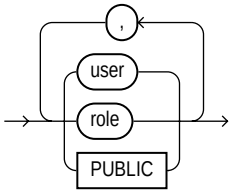
grant_object_privileges_clause::=



object_clause::=



grantee_clause::=



Keywords and Parameters

grant_system_privileges_and_roles_clause

system_privileges Specify the system privilege you want to grant. Table 11–1 lists the system privileges (organized by the database object operated upon).

- If you grant a privilege to a **user**, Oracle adds the privilege to the user's privilege domain. The user can immediately exercise the privilege.
- If you grant a privilege to a **role**, Oracle adds the privilege to the role's privilege domain. Users who have been granted and have enabled the role can immediately exercise the privilege. Other users who have been granted the role can enable the role and exercise the privilege.
- If you grant a privilege to **PUBLIC**, Oracle adds the privilege to the privilege domains of each user. All users can immediately perform operations authorized by the privilege.

Oracle provides a shortcut for specifying all system privileges at once:

- **ALL PRIVILEGES**: Specify ALL PRIVILEGES to grant all the system privileges listed in Table 11–1, " System Privileges" on page 11-37.

role

Specify the role you want to grant. You can grant an Oracle predefined role or a user-defined role. Table 11–2 lists the predefined roles.

- If you grant a role to a **user**, Oracle makes the role available to the user. The user can immediately enable the role and exercise the privileges in the role's privilege domain.
- If you grant a role to another **role**, Oracle adds the granted role's privilege domain to the grantee role's privilege domain. Users who have been granted the grantee role can enable it and exercise the privileges in the granted role's privilege domain.
- If you grant a role to **PUBLIC**, Oracle makes the role available to all users. All users can immediately enable the role and exercise the privileges in the roles privilege domain.

  See Also: CREATE ROLE on page 9-146 for information on creating a user-defined role

WITH ADMIN OPTION

Specify WITH ADMIN OPTION to enable the grantee to:

- Grant the role to another user or role, unless the role is a GLOBAL role
- Revoke the role from another user or role
- Alter the role to change the authorization needed to access it
- Drop the role

If you grant a system privilege or role to a user without specifying WITH ADMIN OPTION, and then subsequently grant the privilege or role to the user WITH ADMIN OPTION, the user has the ADMIN OPTION on the privilege or role.

To revoke the admin option on a system privilege or role from a user, you must revoke the privilege or role from the user altogether and then grant the privilege or role to the user without the admin option.

grantee_clause — TO *grantee_clause* identifies users or roles to which the system privilege, role, or object privilege is granted.

Restriction: A user, role, or PUBLIC cannot appear more than once in TO *grantee_clause*.

PUBLIC — Specify PUBLIC to grant the privileges to all users.

Restrictions on granting system privileges and roles:

- A privilege or role cannot appear more than once in the list of privileges and roles to be granted.
- You cannot grant a role to itself.
- You cannot grant a role IDENTIFIED GLOBALLY to anything.
- You cannot grant a role IDENTIFIED EXTERNALLY to a global user or global role.
- You cannot grant roles circularly. For example, if you grant the role banker to the role teller, you cannot subsequently grant teller to banker.

grant_object_privileges_clause

object_privileges — Specify the object privilege you want to grant. You can substitute any of the values shown in Table 11–3. See also Table 11–4.

Restriction: A privilege cannot appear more than once in the list of privileges to be granted.

ALL [PRIVILEGES] — Specify ALL to grant all the privileges for the object that you have been granted with the GRANT OPTION. The user who owns the schema containing an object automatically has all privileges on the object with the GRANT OPTION. (The keyword PRIVILEGES is optional.)

column — Specify the table or view column on which privileges are to be granted. You can specify columns only when granting the INSERT, REFERENCES, or UPDATE privilege. If you do not list columns, the grantee has the specified privilege on all columns in the table or view.

For information on existing column object grants, query the USER_,ALL_, and DBA_COL_PRIVS data dictionary view.

> **See Also:** *Oracle8i Reference* for information on the data dictionary views

WITH GRANT OPTION — Specify WITH GRANT OPTION to enable the grantee to grant the object privileges to other users and roles.

Restriction: You can specify WITH GRANT OPTION only when granting to a user or to PUBLIC, not when granting to a role.

object_clause — ON *object_clause* identifies the object on which the privileges are granted. Directory schema objects and Java source and resource schema objects are identified separately because they reside in separate namespaces.

object — Specify the schema object on which the privileges are to be granted. If you do not qualify *object* with *schema*, Oracle assumes the object is in your own schema. The object can be one of the following types:

- Table, view, or materialized view / snapshot
- Sequence
- Procedure, function, or package
- User-defined type
- Synonym for any of the above items
- Directory, library, operator, or indextype
- Java source, class, or resource

Note: You cannot grant privileges directly to a single partition of a partitioned table. For information on how to grant privileges to a single partition indirectly, refer to *Oracle8i Concepts*.

DIRECTORY *directory_name*	Specify a directory schema object on which privileges are to be granted. You cannot qualify *directory_name* with a schema name. **See Also:** CREATE DIRECTORY on page 9-40
JAVA SOURCE | RESOURCE	The JAVA clause lets you specify a Java source or resource schema object on which privileges are to be granted. **See Also:** CREATE JAVA on page 9-79

Table 11–1 System Privileges

System Privilege Name	Operations Authorized
CLUSTERS	
CREATE CLUSTER	Create clusters in grantee's schema
CREATE ANY CLUSTER	Create a cluster in any schema. Behaves similarly to CREATE ANY TABLE.
ALTER ANY CLUSTER	Alter clusters in any schema
DROP ANY CLUSTER	Drop clusters in any schema
CONTEXTS	
CREATE ANY CONTEXT	Create any context namespace
DROP ANY CONTEXT	Drop any context namespace
DATABASE	
ALTER DATABASE	Alter the database
ALTER SYSTEM	Issue ALTER SYSTEM statements
AUDIT SYSTEM	Issue AUDIT *sql_statements* statements
DATABASE LINKS	
CREATE DATABASE LINK	Create private database links in grantee's schema
CREATE PUBLIC DATABASE LINK	Create public database links

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 11–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
DROP PUBLIC DATABASE LINK	Drop public database links
DIMENSIONS	
CREATE DIMENSION	Create dimensions in the grantee's schema
CREATE ANY DIMENSION	Create dimensions in any schema
ALTER ANY DIMENSION	Alter dimensions in any schema
DROP ANY DIMENSION	Drop dimensions in any schema
DIRECTORIES	
CREATE ANY DIRECTORY	Create directory database objects
DROP ANY DIRECTORY	Drop directory database objects
INDEXTYPES	
CREATE INDEXTYPE	Create an indextype in the grantee's schema
CREATE ANY INDEXTYPE	Create an indextype in any schema
ALTER ANY INDEXTYPE	Modify indextypes in any schema
DROP ANY INDEXTYPE	Drop an indextype in any schema
EXECUTE ANY INDEXTYPE	Reference an indextype in any schema
INDEXES	
CREATE ANY INDEX	Create in any schema a domain index or an index on any table in any schema
ALTER ANY INDEX	Alter indexes in any schema
DROP ANY INDEX	Drop indexes in any schema
QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables and views that are in the grantee's own schema.
GLOBAL QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables or views in any schema.

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 11–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
LIBRARIES	
CREATE LIBRARY	Create external procedure / function libraries in grantee's schema
CREATE ANY LIBRARY	Create external procedure / function libraries in any schema
DROP LIBRARY	Drop external procedure / function libraries in the grantee's schema
DROP ANY LIBRARY	Drop external procedure / function libraries in any schema
MATERIALIZED VIEWS (which are identical to SNAPSHOTS)	
CREATE MATERIALIZED VIEW	Create a materialized view in the grantee's schema
CREATE ANY MATERIALIZED VIEW	Create materialized views in any schema
ALTER ANY MATERIALIZED VIEW	Alter materialized views in any schema
DROP ANY MATERIALIZED VIEW	Drop materialized views in any schema
QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables and views that are in the grantee's own schema.
GLOBAL QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables or views in any schema.
OPERATORS	
CREATE OPERATOR	Create an operator and its bindings in the grantee's schema
CREATE ANY OPERATOR	Create an operator and its bindings in any schema
DROP ANY OPERATOR	Drop an operator in any schema
EXECUTE ANY OPERATOR	Reference an operator in any schema
OUTLINES	
CREATE ANY OUTLINE	Create outlines that can be used in any schema that uses outlines
ALTER ANY OUTLINE	Modify outlines.

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 11–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
DROP ANY OUTLINE	Drop outlines
PROCEDURES	
CREATE PROCEDURE	Create stored procedures, functions, and packages in grantee's schema
CREATE ANY PROCEDURE	Create stored procedures, functions, and packages in any schema
ALTER ANY PROCEDURE	Alter stored procedures, functions, or packages in any schema
DROP ANY PROCEDURE	Drop stored procedures, functions, or packages in any schema
EXECUTE ANY PROCEDURE	Execute procedures or functions (standalone or packaged) Reference public package variables in any schema
PROFILES	
CREATE PROFILE	Create profiles
ALTER PROFILE	Alter profiles
DROP PROFILE	Drop profiles
ROLES	
CREATE ROLE	Create roles
ALTER ANY ROLE	Alter any role in the database
DROP ANY ROLE	Drop roles
GRANT ANY ROLE	Grant any role in the database
ROLLBACK SEGMENTS	
CREATE ROLLBACK SEGMENT	Create rollback segments
ALTER ROLLBACK SEGMENT	Alter rollback segments
DROP ROLLBACK SEGMENT	Drop rollback segments
SEQUENCES	
CREATE SEQUENCE	Create sequences in grantee's schema
CREATE ANY SEQUENCE	Create sequences in any schema

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 11–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
ALTER ANY SEQUENCE	Alter any sequence in the database
DROP ANY SEQUENCE	Drop sequences in any schema
SELECT ANY SEQUENCE	Reference sequences in any schema
SESSIONS	
CREATE SESSION	Connect to the database
ALTER RESOURCE COST	Set costs for session resources
ALTER SESSION	Issue ALTER SESSION statements
RESTRICTED SESSION	Logon after the instance is started using the SQL*Plus STARTUP RESTRICT statement
SNAPSHOTS (which are identical to MATERIALIZED VIEWS)	
CREATE SNAPSHOT	Create snapshots in grantee's schema
CREATE ANY SNAPSHOT	Create snapshots in any schema
ALTER ANY SNAPSHOT	Alter any snapshot in the database
DROP ANY SNAPSHOT	Drop snapshots in any schema
GLOBAL QUERY REWRITE	Enable rewrite using a snapshot, or create a function-based index, when that snapshot or index references tables or views in any schema.
QUERY REWRITE	Enable rewrite using a snapshot, or create a function-based index, when that snapshot or index references tables and views that are in the grantee's own schema.
SYNONYMS	
CREATE SYNONYM	Create synonyms in grantee's schema
CREATE ANY SYNONYM	Create private synonyms in any schema
CREATE PUBLIC SYNONYM	Create public synonyms
DROP ANY SYNONYM	Drop private synonyms in any schema
DROP PUBLIC SYNONYM	Drop public synonyms

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 11–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
TABLES	
CREATE ANY TABLE	Create tables in any schema. The owner of the schema containing the table must have space quota on the tablespace to contain the table.
ALTER ANY TABLE	Alter any table or view in the schema
BACKUP ANY TABLE	Use the Export utility to incrementally export objects from the schema of other users
DELETE ANY TABLE	Delete rows from tables, table partitions, or views in any schema
DROP ANY TABLE	Drop or truncate tables or table partitions in any schema
INSERT ANY TABLE	Insert rows into tables and views in any schema
LOCK ANY TABLE	Lock tables and views in any schema
UPDATE ANY TABLE	Update rows in tables and views in any schema
SELECT ANY TABLE	Query tables, views, or snapshots in any schema
TABLESPACES	
CREATE TABLESPACE	Create tablespaces
ALTER TABLESPACE	Alter tablespaces
DROP TABLESPACE	Drop tablespaces
MANAGE TABLESPACE	Take tablespaces offline and online and begin and end tablespace backups
UNLIMITED TABLESPACE	Use an unlimited amount of any tablespace. This privilege overrides any specific quotas assigned. If you revoke this privilege from a user, the user's schema objects remain but further tablespace allocation is denied unless authorized by specific tablespace quotas. You cannot grant this system privilege to roles.
TRIGGERS	
CREATE TRIGGER	Create a database trigger in grantee's schema
CREATE ANY TRIGGER	Create database triggers in any schema
ALTER ANY TRIGGER	Enable, disable, or compile database triggers in any schema
DROP ANY TRIGGER	Drop database triggers in any schema

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 11–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
ADMINISTER DATABASE TRIGGER	Create a trigger on DATABASE. (You must also have the CREATE TRIGGER or CREATE ANY TRIGGER privilege.)
TYPES	
CREATE TYPE	Create object types and object type bodies in grantee's schema
CREATE ANY TYPE	Create object types and object type bodies in any schema
ALTER ANY TYPE	Alter object types in any schema
DROP ANY TYPE	Drop object types and object type bodies in any schema
EXECUTE ANY TYPE	Use and reference object types and collection types in any schema, and invoke methods of an object type in any schema if you make the grant to a specific user. If you grant EXECUTE ANY TYPE to a role, users holding the enabled role will not be able to invoke methods of an object type in any schema.
USERS	
CREATE USER	Create users. This privilege also allows the creator to Assign quotas on **any** tablespace, Set default and temporary tablespaces, and Assign a profile as part of a CREATE USER statement.
ALTER USER	Alter any user. This privilege authorizes the grantee to Change another user's password or authentication method, Assign quotas on **any** tablespace, Set default and temporary tablespaces, and Assign a profile and default roles
BECOME USER	Become another user. (Required by any user performing a full database import.)
DROP USER	Drop users
VIEWS	
CREATE VIEW	Create views in grantee's schema
CREATE ANY VIEW	Create views in any schema

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 11–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
DROP ANY VIEW	Drop views in any schema
MISCELLANEOUS	
ANALYZE ANY	Analyze any table, cluster, or index in any schema
AUDIT ANY	Audit any object in any schema using AUDIT *schema_objects* statements
COMMENT ANY TABLE	Comment on any table, view, or column in any schema
FORCE ANY TRANSACTION	Force the commit or rollback of any in-doubt distributed transaction in the local database
	Induce the failure of a distributed transaction
FORCE TRANSACTION	Force the commit or rollback of grantee's in-doubt distributed transactions in the local database
GRANT ANY PRIVILEGE	Grant any system privilege.
SYSDBA	Perform STARTUP and SHUTDOWN operations
	ALTER DATABASE: open, mount, back up, or change character set
	CREATE DATABASE
	ARCHIVELOG and RECOVERY
	Includes the RESTRICTED SESSION privilege
SYSOPER	Perform STARTUP and SHUTDOWN operations
	ALTER DATABASE OPEN / MOUNT / BACKUP
	ARCHIVELOG and RECOVERY
	Includes the RESTRICTED SESSION privilege

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 11–2 Oracle Predefined Roles

Predefined Role	Purpose
CONNECT, RESOURCE, and DBA	These roles are provided for compatibility with previous versions of Oracle. You can determine the privileges encompassed by these roles by querying the DBA_SYS_PRIVILEGES data dictionary view. **See Also:** *Oracle8i Reference* for a description of this view **Note:** Oracle Corporation recommends that you design your own roles for database security rather than relying on these roles. These roles may not be created automatically by future versions of Oracle.
DELETE_CATALOG_ROLE EXECUTE_CATALOG_ROLE SELECT_CATALOG_ROLE	These roles are provided for accessing data dictionary views and packages. **See Also:** *Oracle8i Administrator's Guide* for more information on these roles
EXP_FULL_DATABASE IMP_FULL_DATABASE	These roles are provided for convenience in using the Import and Export utilities. **See Also:** *Oracle8i Utilities* for more information on these roles
AQ_USER_ROLE AQ_ADMINISTRATOR_ROLE	You need these roles to use Oracle's Advanced Queuing functionality. **See Also:** *Oracle8i Application Developer's Guide - Advanced Queuing* for more information on these roles
SNMPAGENT	This role is used by Enterprise Manager / Intelligent Agent. **See Also:** *Oracle Enterprise Manager Administrator's Guide*
RECOVERY_CATALOG_OWNER	You need this role to create a user who owns a recovery catalog. **See Also:** *Oracle8i Backup and Recovery Guide* for more information on recovery catalogs
HS_ADMIN_ROLE	A DBA using Oracle's heterogeneous services feature needs this role to access appropriate tables in the data dictionary and to manipulate them with the DBMS_HS package. **See Also:** *Oracle8i Distributed Database Systems* and *Oracle8i Supplied PL/SQL Packages Reference* for more information

Oracle also creates other roles that authorize you to administer the database. On many operating systems, these roles are called OSOPER and OSDBA. Their names may be different on your operating system.

Table 11–3 Object Privileges Available for Particular Objects

Object Privilege	Table	View	Sequence	Procedures, Functions, Packages[a]	Materialized View	Directory	Library	User-defined Type	Operator	Indextype
ALTER	X		X							
DELETE	X	X			X[b]					
EXECUTE				X			X	X	X	X
INDEX	X									
INSERT	X	X			X[b]					
READ						X				
REFERENCES	X									
SELECT	X	X	X		X					
UPDATE	X	X			X[b]					

[a]Oracle treats a Java class, source, or resource as if it were a procedure for purposes of granting object privileges.
[b]The DELETE, INSERT, and UPDATE privileges can be granted only to *updatable* materialized views.

Table 11–4 Object Privileges and the Operations They Authorize

Object Privilege	Operations Authorized
The following **table privileges** authorize operations on a table. Any one of following object privileges allows the grantee to lock the table in any lock mode with the LOCK TABLE statement.	
ALTER	Change the table definition with the ALTER TABLE statement.
DELETE	Remove rows from the table with the DELETE statement. **Note:** You must grant the SELECT privilege on the table along with the DELETE privilege.
INDEX	Create an index on the table with the CREATE INDEX statement.
INSERT	Add new rows to the table with the INSERT statement.
REFERENCES	Create a constraint that refers to the table. You cannot grant this privilege to a role.

Table 11–4 (Cont.) Object Privileges and the Operations They Authorize

Object Privilege	Operations Authorized
SELECT	Query the table with the SELECT statement.
UPDATE	Change data in the table with the UPDATE statement.

> **Note:** You must grant the SELECT privilege on the table along with the UPDATE privilege.

The following **view privileges** authorize operations on a view. Any one of the following object privileges allows the grantee to lock the view in any lock mode with the LOCK TABLE statement.

To grant a privilege on a view, you must have that privilege with the GRANT OPTION on all of the view's base tables.

DELETE	Remove rows from the view with the DELETE statement.
INSERT	Add new rows to the view with the INSERT statement.
SELECT	Query the view with the SELECT statement.
UPDATE	Change data in the view with the UPDATE statement.

The following **sequence privileges** authorize operations on a sequence.

ALTER	Change the sequence definition with the ALTER SEQUENCE statement.
SELECT	Examine and increment values of the sequence with the CURRVAL and NEXTVAL pseudocolumns.

The following **procedure, function, and package privilege** authorizes operations on procedures, functions, or packages. This privilege also applies to **Java sources, classes, and resources**, which Oracle treats as though they were procedures for purposes of granting object privileges.

EXECUTE	Compile the procedure or function or execute it directly, or access any program object declared in the specification of a package.

> **Note:** Users do not need this privilege to execute a procedure, function, or package indirectly.

> **See Also:** *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals*

The following **snapshot privilege** authorizes operations on a snapshot.

SELECT	Query the snapshot with the SELECT statement.

Table 11–4 (Cont.) Object Privileges and the Operations They Authorize

Object Privilege	Operations Authorized
Synonym privileges are the same as the privileges for the base object. Granting a privilege on a synonym is equivalent to granting the privilege on the base object. Similarly, granting a privilege on a base object is equivalent to granting the privilege on all synonyms for the object. If you grant a user a privilege on a synonym, the user can use either the synonym name or the base object name in the SQL statement that exercises the privilege.	
The following **directory privilege** provides secured access to the files stored in the operating system directory to which the directory object serves as a pointer. The directory object contains the full pathname of the operating system directory where the files reside. Because the files are actually stored outside the database, Oracle server processes also need to have appropriate file permissions on the file system server. Granting object privileges on the directory database object to individual database users, rather than on the operating system, allows Oracle to enforce security during file operations.	
READ	Read files in the directory.
The following **object type privilege** authorizes operations on an object type	
EXECUTE	Use and reference the specified object and to invoke its methods.
The following **indextype privilege** authorizes operations on indextypes.	
EXECUTE	Reference an indextype.
The following **operator privilege** authorizes operations on user-defined operators.	
EXECUTE	Reference an operator.

Examples

Granting a System Privilege to a User Example To grant the CREATE SESSION system privilege to `richard`, allowing `richard` to log on to Oracle, issue the following statement:

```
GRANT CREATE SESSION
   TO richard;
```

Granting a System Privilege to a Role Example To grant the CREATE TABLE system privilege to the role `travel_agent`, issue the following statement:

```
GRANT CREATE TABLE
   TO travel_agent;
```

`travel_agent`'s privilege domain now contains the CREATE TABLE system privilege.

Granting a Role to a Role Example The following statement grants the travel_agent role to the EXECUTIVE role:

```
GRANT travel_agent
   TO executive;
```

travel_agent is now granted to executive. executive's privilege domain contains the CREATE TABLE system privilege.

Granting a Role with the Admin Option Example To grant the executive role with the ADMIN OPTION to THOMAS, issue the following statement:

```
GRANT executive
   TO thomas
   WITH ADMIN OPTION;
```

thomas can now perform the following operations with the executive role:

- Enable the role and exercise any privileges in the role's privilege domain, including the CREATE TABLE system privilege
- Grant and revoke the role to and from other users
- Drop the role

Granting an Object Privilege on a Directory Example To grant READ on directory bfile_dir1 to user scott, with the GRANT OPTION, issue the following statement:

```
GRANT READ ON DIRECTORY bfile_dir1 TO scott
   WITH GRANT OPTION;
```

Granting Object Privileges on a Table to a User Example To grant all privileges on the table bonus to the user jones with the GRANT OPTION, issue the following statement:

```
GRANT ALL ON bonus TO jones
   WITH GRANT OPTION;
```

jones can subsequently perform the following operations:

- Exercise any privilege on the bonus table
- Grant any privilege on the bonus table to another user or role

Granting Object Privileges on a View Example To grant SELECT and UPDATE privileges on the view golf_handicap to all users, issue the following statement:

```
GRANT SELECT, UPDATE
   ON golf_handicap TO PUBLIC;
```

All users can subsequently query and update the view of golf handicaps.

Granting Object Privileges to a Sequence in Another Schema Example To grant SELECT privilege on the eseq sequence in the schema elly to the user blake, issue the following statement:

```
GRANT SELECT
   ON elly.eseq TO blake;
```

blake can subsequently generate the next value of the sequence with the following statement:

```
SELECT elly.eseq.NEXTVAL
   FROM DUAL;
```

Granting Multiple Object Privileges on Individual Columns Example To grant blake the REFERENCES privilege on the empno column and the UPDATE privilege on the empno, sal, and comm columns of the emp table in the schema scott, issue the following statement:

```
GRANT REFERENCES (empno), UPDATE (empno, sal, comm)
   ON scott.emp
   TO blake;
```

blake can subsequently update values of the empno, sal, and comm columns. blake can also define referential integrity constraints that refer to the empno column. However, because the GRANT statement lists only these columns, blake cannot perform operations on any of the other columns of the emp table.

For example, blake can create a table with a constraint:

```
CREATE TABLE dependent
   (dependno   NUMBER,
    dependname VARCHAR2(10),
    employee   NUMBER
   CONSTRAINT in_emp REFERENCES scott.emp(empno) );
```

The constraint in_emp ensures that all dependents in the dependent table correspond to an employee in the emp table in the schema scott.

INSERT

Purpose

Use the INSERT statement to add rows to a table, a view's base table, a partition of a partitioned table or a subpartition of a composite-partitioned table, or an object table or an object view's base table.

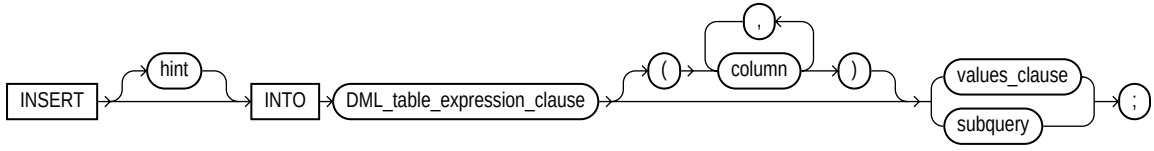
Prerequisites

For you to insert rows into a table, the table must be in your own schema or you must have INSERT privilege on the table.

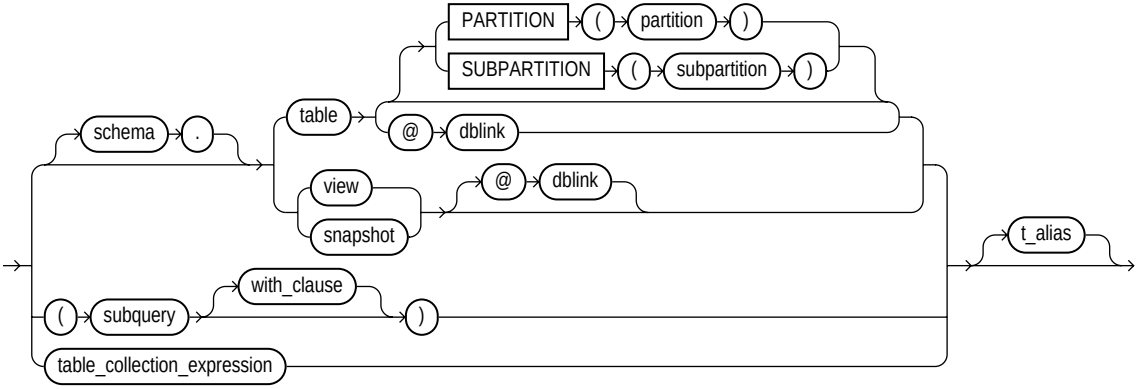
For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, you must have INSERT privilege on the view.

If you have the INSERT ANY TABLE system privilege, you can also insert rows into any table or any view's base table.

Syntax

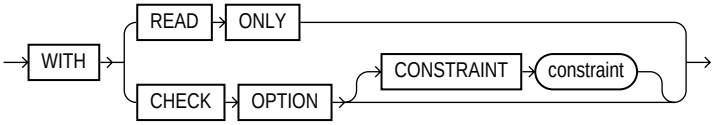


DML_table_expression_clause::=



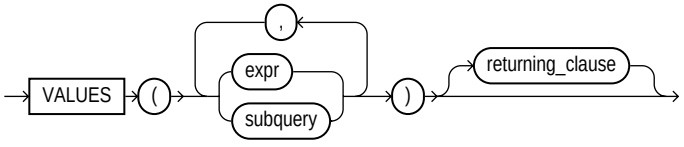
subquery: see SELECT and subquery on page 11-88.

with_clause::=

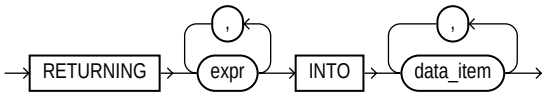


table_collection_expression::=

values_clause::=



returning_clause::=



Keywords and Parameters

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

> **See Also:** "Hints" on page 2-67 and *Oracle8i Performance Guide and Reference* for the syntax and description of hints

DML_table_expression_clause

schema — Specify the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table | *view* | *subquery* — Specify the name of the table or object table, or view or object view, or the column or columns returned by a subquery, into which rows are to be inserted. If you specify a view or object view, Oracle inserts rows into the view's base table.

If any value to be inserted is a REF to an object table, and if the object table has a primary key object identifier, then the column into which you insert the REF must be a REF column with a referential integrity or SCOPE constraint to the object table.

If *table* (or the base table of *view*) contains one or more domain index columns, this statement executes the appropriate indextype insert routine.

Issuing an INSERT statement against a table fires any INSERT triggers defined on the table.

> **See Also:** *Oracle8i Data Cartridge Developer's Guide* for more information on these routines

PARTITION *(partition_name)* | SUBPARTITION *(subpartition_name)*	Specify the name of the partition or subpartition within *table* (or the base table of *view*) targeted for inserts. If a row to be inserted does not map into a specified partition or subpartition, Oracle returns an error. **Restriction:** This clause is not valid for object tables or object views.
dblink	Specify a complete or partial name of a database link to a remote database where the table or view is located. You can insert rows into a remote table or view only if you are using Oracle's distributed functionality. If you omit *dblink*, Oracle assumes that the table or view is on the local database. **See Also:** "Syntax for Schema Objects and Parts in SQL Statements" on page 2-88for information on referring to database links

Restrictions on the *DML_table_expression_clause*:

- You cannot execute this statement if *table* (or the base table of *view*) contains any domain indexes marked LOADING or FAILED.
- With regard to the ORDER BY clause of the *subquery* in the *DML_query_expression_clause*, ordering is guaranteed only for the rows being inserted, and only within each extent of the table. Ordering of new rows with respect to existing rows is not guaranteed.
- If a view was created using the WITH CHECK OPTION, then you can insert into the view only rows that satisfy the view's defining query.
- If a view was created using a single base table, then you can insert rows into the view and then retrieve those values using the *returning_clause*.
- You cannot insert rows into a view except with INSTEAD OF triggers if the view's defining query contains one of the following constructs:
  - A set operator
  - A DISTINCT operator
  - An aggregate or analytic function
  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
  - A collection expression in a SELECT list

  - A subquery in a SELECT list
  - Joins (with some exceptions). See *Oracle8i Administrator's Guide* for details.
- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, the INSERT statement will fail unless the SKIP_UNUSABLE_INDEXES parameter has been set to TRUE.

See Also: ALTER SESSION on page 7-105

with_clause

Use the *with_clause* to restrict the subquery in one of the following ways:

- WITH READ ONLY specifies that the subquery cannot be updated.
- WITH CHECK OPTION specifies that Oracle prohibits any changes to that table that would produce rows that are not included in the subquery.

See Also: "WITH CHECK OPTION Example" on page 11-108

table_collection_expression

Use the *table_collection_expression* to inform Oracle that the collection value expression should be treated as a table.

See Also: "Table Collection Examples" on page 11-115

collection_expression Specify a subquery that selects a nested table column from *table* or *view*.

> **Note:** In earlier releases of Oracle, *table_collection_expression* was expressed as "THE *subquery*". That usage is now deprecated.

t_alias

Specify a **correlation name** (alias) for the table, view, or subquery to be referenced elsewhere in the statement.

column

Specify a column of the table or view. In the inserted row, each column in this list is assigned a value from the *values_clause* or the subquery.

If you omit one of the table's columns from this list, the column's value for the inserted row is the column's default value as specified when the table was created. If any of these columns has a NOT NULL constraint, then Oracle returns an error indicating that the constraint has been violated and rolls back the INSERT statement.

If you omit the column list altogether, the *values_clause* or query must specify values for all columns in the table.

> **See Also:** CREATE TABLE on page 10-7 for more information on default column values

values_clause

Specify a row of values to be inserted into the table or view. You must specify a value in the *values_clause* for each column in the column list. If you omit the column list, then the *values_clause* must provide values for every column in the table.

Restrictions:

- You cannot initialize an internal LOB attribute in an object with a value other than empty or null. That is, you cannot use a literal.
- You cannot insert a BFILE value until you have initialized the BFILE locator to null or to a directory alias and filename.

> **See Also:**
>
> - "Inserting into a BFILE Example" on page 11-61
> - *Oracle Call Interface Programmer's Guide* and *Oracle8i Application Developer's Guide - Fundamentals* for information on initializing BFILEs
> - "Expressions" on page 5-2 and SELECT and subquery on page 11-88 for syntax of valid expressions

---

Note: If you insert string literals into a RAW column, during subsequent queries, Oracle will perform a full table scan rather than using any index that might exist on the RAW column.

---

returning_clause

The returning clause retrieves the rows affected by a DML (INSERT, UPDATE, or DELETE) statement. You can specify this clause for tables and snapshots, and for views with a single base table.

- When operating on a single row, a DML statement with a *returning_clause* can retrieve column expressions using the affected row, rowid, and REFs to the affected row and store them in host variables or PL/SQL variables.
- When operating on multiple rows, a DML statement with the *returning_clause* stores values from expressions, rowids, and REFs involving the affected rows in bind arrays.

expr	Each item in the *expr* list must be a valid expression syntax.
INTO	The INTO clause indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item* list.
data_item	Each *data_item* is a host variable or PL/SQL variable that stores the retrieved *expr* value.

For each expression in the RETURNING list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the INTO list.

Restrictions:

- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.
- You cannot specify this clause for a view on which an INSTEAD OF trigger has been defined.

> **See Also:** *PL/SQL User's Guide and Reference* for information on using the BULK COLLECT clause to return multiple values to collection variables

subquery

Specify a subquery that returns rows that are inserted into the table. If the subquery selects no rows, Oracle inserts no rows into the table.

- When specified without VALUES, the subquery can return zero or more rows, which are then inserted.
- When specified with VALUES, the subquery must be a scalar subquery. That is, it must return exactly one row with one value.

The subquery can refer to any table, view, or snapshot, including the target table of the INSERT statement. The select list of this subquery must have the same number of columns as the column list of the INSERT statement. If you omit the column list, then the subquery must provide values for every column in the table.

You can use *subquery* in combination with the TO_LOB function to convert the values in a LONG column to LOB values in another column in the same or another table. To migrate LONGs to LOBs in a view, you must perform the migration on the base table, and then add the LOB to the view.

See Also:

- "Conversion Functions" on page 4-5
- *Oracle8i Migration* for a discussion of why and when to copy a LONG to a LOB
- Inserting with TO_LOB Example on page 11-60 for a description of how to use the TO_LOB function
- SELECT and subquery on page 11-88

Notes:

- If *subquery* returns (in part or totally) the equivalent of an existing materialized view, Oracle may use the materialized view (for query rewrite) in place of one or more tables specified in *subquery*.

  See Also: *Oracle8i Data Warehousing Guide* for more information on materialized views and query rewrite.

- If this *subquery* refers to remote objects, the INSERT operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if the *subquery* in the *DML_query_expression_clause* refers to any remote objects, the INSERT operation will run serially without notification.

  See Also: parallel_clause in CREATE TABLE on page 10-40

Examples

Inserting Values Examples The following statement inserts a row into the dept table:

```
INSERT INTO dept
   VALUES (50, 'PRODUCTION', 'SAN FRANCISCO');
```

The following statement inserts a row with six columns into the emp table. One of these columns is assigned NULL and another is assigned a number in scientific notation:

```
INSERT INTO emp (empno, ename, job, sal, comm, deptno)
   VALUES (7890, 'JINKS', 'CLERK', 1.2E3, NULL, 40);
```

The following statement has the same effect as the preceding example, but uses a subquery in the *DML_query_expression_clause*:

```
INSERT INTO (SELECT empno, ename, job, sal, comm, deptno FROM emp)
   VALUES (7890, 'JINKS', 'CLERK', 1.2E3, NULL, 40);
```

Inserting Values with a Subquery Example The following statement copies managers and presidents or employees whose commission exceeds 25% of their salary into the bonus table:

```
INSERT INTO bonus
   SELECT ename, job, sal, comm
   FROM emp
   WHERE comm > 0.25 * sal
   OR job IN ('PRESIDENT', 'MANAGER');
```

Inserting into a Remote Database Example The following statement inserts a row into the accounts table owned by the user scott on the database accessible by the database link sales:

```
INSERT INTO scott.accounts@sales (acc_no, acc_name)
   VALUES (5001, 'BOWER');
```

Assuming that the accounts table has a balance column, the newly inserted row is assigned the default value for this column (if one has been defined), because this INSERT statement does not specify a balance value.

Inserting Sequence Values Example The following statement inserts a new row containing the next value of the employee sequence into the emp table:

```
INSERT INTO emp
   VALUES  (empseq.nextval, 'LEWIS', 'CLERK',
            7902, SYSDATE, 1200, NULL, 20);
```

Inserting into a Partition Example The following example adds rows from `latest_data` into partition `oct98` of the `sales` table:

```
INSERT INTO sales PARTITION (oct98)
   SELECT * FROM latest_data;
```

Inserting Using Bind Variables Example The following example returns the values of the inserted rows into output bind variables :`bnd1` and :`bnd2`:

```
INSERT INTO emp VALUES (empseq.nextval, 'LEWIS', 'CLARK',
                        7902, SYSDATE, 1200, NULL, 20)
   RETURNING sal*12, job INTO :bnd1, :bnd2;
```

Returning Values into a Bind Array Example The following example returns the reference value for the inserted row into bind array :`1`:

```
INSERT INTO employee
   VALUES ('Kitty Mine', 'Peaches Fuzz', 'Meena Katz')
   RETURNING REF(employee) INTO :1;
```

Inserting with TO_LOB Example The following example copies `LONG` data to a LOB column in the following existing table:

```
CREATE TABLE long_tab (long_pics LONG RAW);
```

First you must create a table with a LOB.

```
CREATE TABLE lob_tab (lob_pics BLOB);
```

Next, use an `INSERT ... SELECT` statement to copy the data in all rows for the `LONG` column into the newly created LOB column:

```
INSERT INTO lob_tab (lob_pics)
   SELECT TO_LOB(long_pics) FROM long_tab;
```

Once you are confident that the migration has been successful, you can drop the `long_pics` table. Alternatively, if the table contains other columns, you can simply drop the `LONG` column from the table as follows:

```
ALTER TABLE long_tab DROP COLUMN long_pics;
```

Inserting into a BFILE Example When you INSERT or UPDATE a BFILE, you must initialize it to null or to a directory alias and filename, as shown in the next example. Assume that the emp table has a number column followed by a BFILE column:

```
INSERT INTO emp
   VALUES (1, BFILENAME ('a_dir_alias', 'a_filename'));
```

LOCK TABLE

Purpose

Use the LOCK TABLE statement to lock one or more tables (or table partitions or subpartitions) in a specified mode. This lock manually overrides automatic locking and permits or denies access to a table or view by other users for the duration of your operation.

Some forms of locks can be placed on the same table at the same time. Other locks allow only one lock per table.

A locked table remains locked until you either commit your transaction or roll it back, either entirely or to a savepoint before you locked the table.

A lock never prevents other users from querying the table. A query never places a lock on a table. Readers never block writers and writers never block readers.

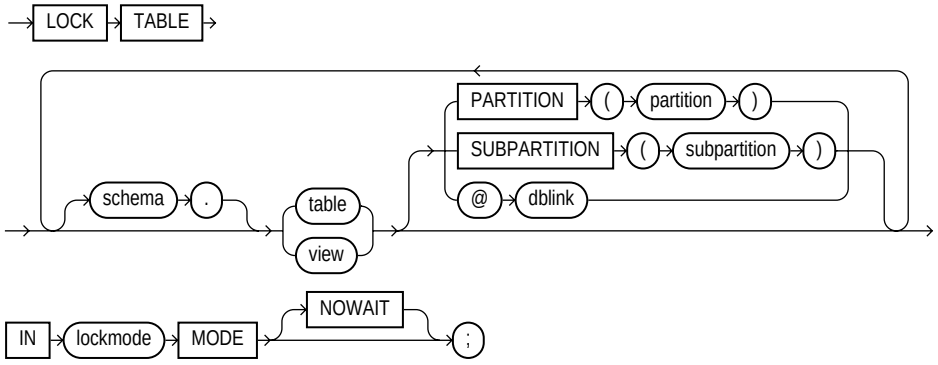
See Also:

- *Oracle8i Concepts* for a complete description of the interaction of lock modes
- COMMIT on page 8-133
- ROLLBACK on page 11-83
- SAVEPOINT on page 11-86

Prerequisites

The table or view must be in your own schema or you must have the LOCK ANY TABLE system privilege, or you must have any object privilege on the table or view.

Syntax



Keywords and Parameters

schema

Specify the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table/*view*

Specify the name of the table to be locked. If you specify *view*, Oracle locks the view's base tables.

If you specify PARTITION (*partition*) or SUBPARTITION (*subpartition*), Oracle first acquires an implicit lock on the table. The table lock is the same as the lock you specify for *partition* or *subpartition*, with two exceptions:

- If you specify a SHARE lock for the subpartition, Oracle acquires an implicit ROW SHARE lock on the table.
- If you specify an EXCLUSIVE lock for the subpartition, Oracle acquires an implicit ROW EXCLUSIVE lock on the table.

If you specify PARTITION and *table* is composite-partitioned, then Oracle acquires locks on all the subpartitions of *partition*.

dblink

Specify a database link to a remote Oracle database where the table or view is located. You can lock tables and views on a remote database only if you are using

Oracle's distributed functionality. All tables locked by a LOCK TABLE statement must be on the same database.

If you omit *dblink*, Oracle assumes the table or view is on the local database.

> **See Also:** "Referring to Objects in Remote Databases" on page 2-90 for information on specifying database links

lockmode

Specify one of the following modes:

- ROW SHARE allows concurrent access to the locked table, but prohibits users from locking the entire table for exclusive access. ROW SHARE is synonymous with SHARE UPDATE, which is included for compatibility with earlier versions of Oracle.
- ROW EXCLUSIVE is the same as ROW SHARE, but also prohibits locking in SHARE mode. Row Exclusive locks are automatically obtained when updating, inserting, or deleting.
- SHARE UPDATE—see ROW SHARE.
- SHARE allows concurrent queries but prohibits updates to the locked table.
- SHARE ROW EXCLUSIVE is used to look at a whole table and to allow others to look at rows in the table but to prohibit others from locking the table in SHARE mode or updating rows.
- EXCLUSIVE allows queries on the locked table but prohibits any other activity on it.

NOWAIT

Specify NOWAIT if you want Oracle to return control to you immediately if the specified table (or specified partition or subpartition) is already locked by another user. In this case, Oracle returns a message indicating that the table, partition, or subpartition is already locked by another user.

If you omit this clause, Oracle waits until the table is available, locks it, and returns control to you.

Examples

LOCK TABLE Example The following statement locks the emp table in exclusive mode, but does not wait if another user already has locked the table:

```
LOCK TABLE emp
   IN EXCLUSIVE MODE
   NOWAIT;
```

The following statement locks the remote `accounts` table that is accessible through the database link `boston`:

```
LOCK TABLE accounts@boston
   IN SHARE MODE;
```

NOAUDIT

Purpose

Use the NOAUDIT statement to stop auditing previously enabled by the AUDIT statement.

The NOAUDIT statement must have the same syntax as the previous AUDIT statement. Further, it reverses the effects only of that particular statement.For example, suppose one AUDIT statement (statement A) enables auditing for a specific user. A second (statement B) enables auditing for all users. A NOAUDIT statement to disable auditing for all users (statement C) reverses statement B. However, statement C leaves statement A in effect and continues to audit the user that statement A specified.

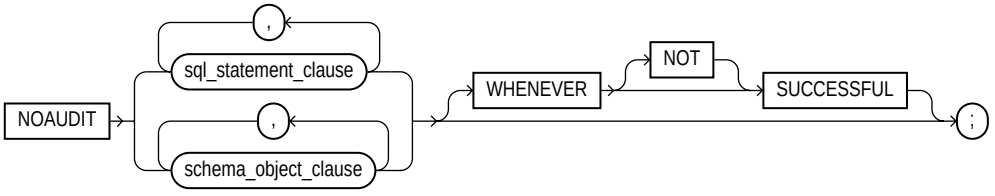
See Also: AUDIT on page 8-114 for more information on auditing

Prerequisites

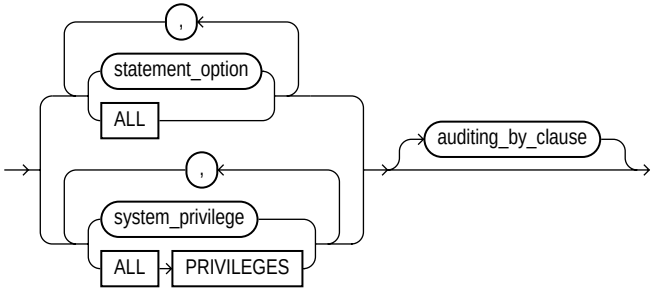
To stop auditing of SQL statements, you must have the AUDIT SYSTEM system privilege.

To stop auditing of schema objects, you must be the owner of the object on which you stop auditing or you must have the AUDIT ANY system privilege. In addition, if the object you chose for auditing is a directory, even if you created it, you must have the AUDIT ANY system privilege.

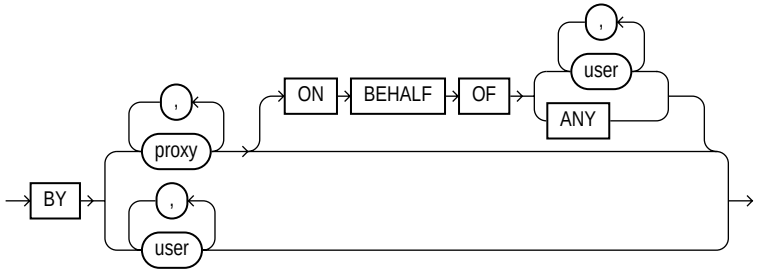
Syntax



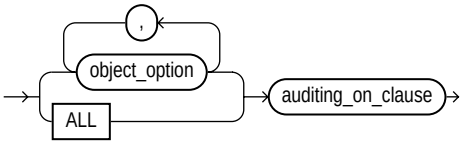
sql_statement_clause::=



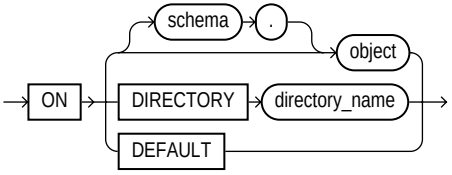
auditing_by_clause::=



schema_object_clause::=



auditing_on_clause::=



Keywords and Parameters

sql_statement_clause

statement_option — Specify the statement option for which auditing is to be stopped.

See Also: Table 8–1 on page 8-120 Table 8–2 on page 8-122 and for a list of the statement options and the SQL statements they audit

ALL — Specify ALL to stop auditing of all statement options currently being audited.

system_privilege — Specify the system privilege for which auditing is to be stopped.

See Also: Table 11–1 on page 11-37 for a list of the system privileges and the statements they authorize

ALL PRIVILEGES — Specify ALL PRIVILEGES to stop auditing of all system privileges currently being audited.

auditing_by_clause — Use the *auditing_by_clause* to stop auditing only those SQL statements issued by particular users. If you omit this clause, Oracle stops auditing all users' statements.

- BY *user* — Specify BY *user* to stop auditing only for SQL statements issued by the specified users in their subsequent sessions. If you omit this clause, Oracle stops auditing for all users' statements, except for the situation described for WHENEVER SUCCESSFUL.
- BY *proxy*' — Specify BY *proxy* to stop auditing only for the SQL statements issued by the specified proxy, on behalf of a specific user or any user.

schema_object_clause

object_option — Specify the type of operation for which auditing is to be stopped on the object specified in the ON clause.

See Also: Table 8–3 on page 8-124 for a list of these options

ALL — Specify ALL as a shortcut equivalent to specifying all object options applicable for the type of object.

auditing_on_clause The *auditing_on_clause* lets you specify the particular schema object for which auditing is to be stopped.

- *object* — Specify the object name of a table, view, sequence, stored procedure, function, or package, snapshot, or library. If you do not qualify *object* with *schema*, Oracle assumes the object is in your own schema.

  See Also: AUDIT on page 8-114 for information on auditing specific schema objects

- DIRECTORY *directory_name* — The DIRECTORY clause lets you specify the name of the directory on which auditing is to be stopped.

- DEFAULT — Specify DEFAULT to remove the specified object options as default object options for subsequently created objects.

WHENEVER [NOT] SUCCESSFUL — Specify WHENEVER SUCCESSFUL to stop auditing only for SQL statements and operations on schema objects that complete successfully.

Specify NOT to stop auditing only for statements and operations that result in Oracle errors.

If you omit this clause, Oracle stops auditing for all statements or operations, regardless of success or failure.

Examples

Stop Auditing of SQL Statements Related to Roles Example If you have chosen auditing for every SQL statement that creates or drops a role, you can stop auditing of such statements by issuing the following statement:

```
NOAUDIT ROLE;
```

Stop Auditing of Updates or Queries on Objects Owned by a Particular User Example If you have chosen auditing for any statement that queries or updates any table issued by the users scott and blake, you can stop auditing for scott's queries by issuing the following statement:

```
NOAUDIT SELECT TABLE BY scott;
```

The above statement stops auditing only scott's queries, so Oracle continues to audit blake's queries and updates as well as scott's updates.

Stop Auditing of Statements Authorized by a Particular Object Privilege Example To stop auditing on all statements that are authorized by DELETE ANY TABLE system privilege, issue the following statement:

```
NOAUDIT DELETE ANY TABLE;
```

Stop Auditing of Queries on a Particular Object Example If you have chosen auditing for every SQL statement that queries the emp table in the schema scott, you can stop auditing for such queries by issuing the following statement:

```
NOAUDIT SELECT
   ON scott.emp;
```

Stop Auditing of Queries that Complete Successfully Example You can stop auditing for queries that complete successfully by issuing the following statement:

```
NOAUDIT SELECT
   ON scott.emp
   WHENEVER SUCCESSFUL;
```

This statement stops auditing only for successful queries. Oracle continues to audit queries resulting in Oracle errors.

RENAME

Purpose

Use the RENAME statement to rename a table, view, sequence, or private synonym for a table, view, or sequence.

- Oracle automatically transfers integrity constraints, indexes, and grants on the old object to the new object.
- Oracle invalidates all objects that depend on the renamed object, such as views, synonyms, and stored procedures and functions that refer to a renamed table.

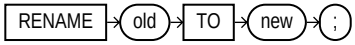
Do not use this statement to rename public synonyms. Instead, drop the public synonym and then create another public synonym with the new name.

> **See Also:** CREATE SYNONYM on page 10-3 and DROP SYNONYM on page 11-5

Prerequisites

The object must be in your own schema.

Syntax



Keywords and Parameters

old

Specify the name of an existing table, view, sequence, or private synonym.

new

Specify the new name to be given to the existing object. The new name must not already be used by another schema object in the same namespace and must follow the rules for naming schema objects.

See Also: "Schema Object Naming Rules" on page 2-83

Example

Rename a Database Object Example To change the name of table `dept` to `emp_dept`, issue the following statement:

```
RENAME dept TO emp_dept;
```

You cannot use this statement directly to rename columns. However, you can rename a column using this statement together with the CREATE TABLE statement with AS *subquery*. The following statements re-create the table `static`, renaming a column from `oldname` to newname:

```
CREATE TABLE temporary (newname, col2, col3)
    AS SELECT oldname, col2, col3 FROM static;

DROP TABLE static;

RENAME temporary TO static;
```

REVOKE

Purpose

Use the REVOKE statement to:

- Revoke system privileges from users and roles
- Revoke roles from users and roles
- Revoke object privileges for a particular object from users and roles

See Also:

- GRANT on page 11-31 for information on granting system privileges and roles
- Table 11–3 on page 11-46 for a summary of the object privileges for each type of object

Prerequisites

To revoke a **system privilege or role,** you must have been granted the privilege with the ADMIN OPTION.

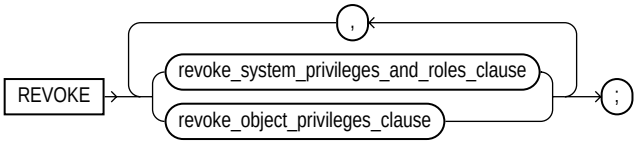
To revoke a **role**, you must have been granted the role with the ADMIN OPTION. You can revoke any role if you have the GRANT ANY ROLE system privilege.

To revoke an **object privilege**, you must have previously granted the object privileges to each user and role.

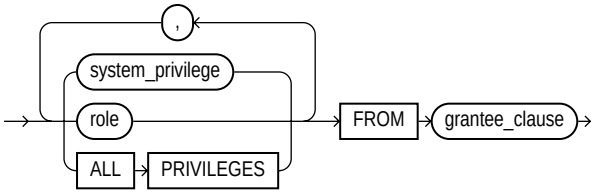
The REVOKE statement can revoke only privileges and roles that were previously granted directly with a GRANT statement. You cannot use this statement to revoke:

- Privileges or roles not granted to the revokee
- Roles or object privileges granted through the operating system
- Privileges or roles granted to the revokee through roles

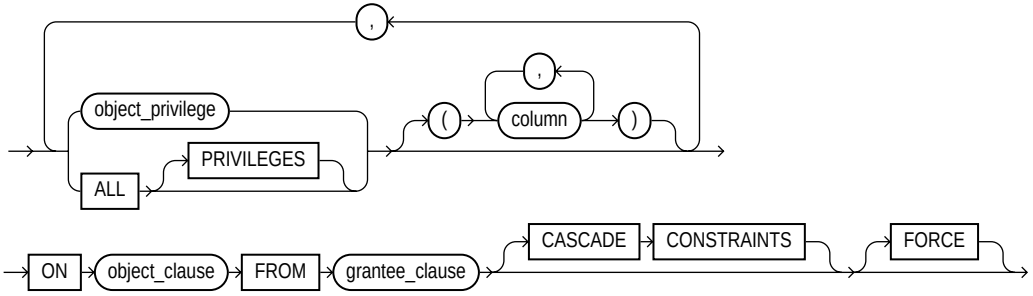
Syntax



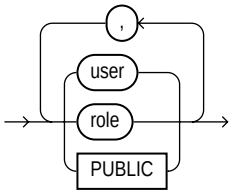
revoke_system_privileges_and_roles_clause::=



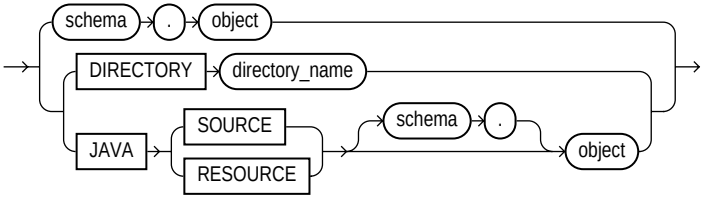
revoke_object_privileges_clause::=



grantee_clause::=



object_clause::=



Keywords and Parameters

revoke_system_privileges_and_roles_clause

system_privilege

Specify the system privilege to be revoked.

See Also: Table 11–1 on page 11-37 for a list of the system privileges

- If you revoke a privilege from a **user**, Oracle removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.
- If you revoke a privilege from a **role**, Oracle removes the privilege from the role's privilege domain. Effective immediately, users with the role enabled cannot exercise the privilege. Also, other users who have been granted the role and subsequently enable the role cannot exercise the privilege.
- If you revoke a privilege from **PUBLIC**, Oracle removes the privilege from the privilege domain of each user who has been granted the privilege through PUBLIC. Effective immediately, such users can no longer exercise the privilege. However, the privilege is not revoked from users who have been granted the privilege directly or through roles.

Restriction: A system privilege cannot appear more than once in the list of privileges to be revoked.

Oracle provides a shortcut for specifying all system privileges at once:

- ALL PRIVILEGES: Specify ALL PRIVILEGES to revoke all the system privileges listed in Table 11–1 on page 11-37.

role Specify the role to be revoked.

- If you revoke a role from a **user**, Oracle makes the role unavailable to the user. If the role is currently enabled for the user, the user can continue to exercise the privileges in the role's privilege domain as long as it remains enabled. However, the user cannot subsequently enable the role.
- If you revoke a role from another **role**, Oracle removes the revoked role's privilege domain from the revokee role's privilege domain. Users who have been granted and have enabled the revokee role can continue to exercise the privileges in the revoked role's privilege domain as long as the revokee role remains enabled. However, other users who have been granted the revokee role and subsequently enable it cannot exercise the privileges in the privilege domain of the revoked role.
- If you revoke a role from **PUBLIC**, Oracle makes the role unavailable to all users who have been granted the role through PUBLIC. Any user who has enabled the role can continue to exercise the privileges in its privilege domain as long as it remains enabled. However, users cannot subsequently enable the role. The role is not revoked from users who have been granted the role directly or through other roles.

Restriction: A system role cannot appear more than once in the list of roles to be revoked.

See Also: Table 11–2 on page 11-45 for a list of the roles predefined by Oracle

grantee_clause FROM *grantee_clause* identifies users or roles from which the system privilege, role, or object privilege is to be revoked.

PUBLIC Specify PUBLIC to revoke the privileges or roles from all users.

revoke_object_privileges_clause

object_privilege Specify the object privilege to be revoked. You can substitute any of the following values: ALTER, DELETE, EXECUTE, INDEX, INSERT, READ, REFERENCES, SELECT, UPDATE.

Note: Each privilege authorizes some operation. By revoking a privilege, you prevent the revokee from performing that operation. However, multiple users may grant the same privilege to the same user, role, or PUBLIC. To remove the privilege from the grantee's privilege domain, all grantors must revoke the privilege. If even one grantor does not revoke the privilege, the grantee can still exercise the privilege by virtue of that grant.

- If you revoke a privilege from a **user**, Oracle removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

  - If that user has granted that privilege to other users or roles, Oracle also revokes the privilege from those other users or roles.

  - If that user's schema contains a procedure, function, or package that contains SQL statements that exercise the privilege, the procedure, function, or package can no longer be executed.

  - If that user's schema contains a view on that object, Oracle invalidates the view.

  - If you revoke the REFERENCES privilege from a user who has exercised the privilege to define referential integrity constraints, you must specify the CASCADE CONSTRAINTS clause.

- If you revoke a privilege from a **role**, Oracle removes the privilege from the role's privilege domain. Effective immediately, users with the role enabled cannot exercise the privilege. Other users who have been granted the role cannot exercise the privilege after enabling the role.

- If you revoke a privilege from **PUBLIC**, Oracle removes the privilege from the privilege domain of each user who has been granted the privilege through PUBLIC. Effective immediately, all such users are restricted from exercising the privilege. However, the privilege is not revoked from users who have been granted the privilege directly or through roles.

Restriction: A privilege cannot appear more than once in the list of privileges to be revoked. A user, a role, or PUBLIC cannot appear more than once in the FROM clause.

ALL [PRIVILEGES]
Specify ALL to revoke all object privileges that you have granted to the revokee. (The keyword PRIVILEGES is optional.)

Note: If no privileges have been granted on the object, Oracle takes no action and does not return an error.

CASCADE CONSTRAINTS
This clause is relevant only if you revoke the REFERENCES privilege or ALL [PRIVILEGES]. It drops any referential integrity constraints that the revokee has defined using the REFERENCES privilege (which might have been granted either explicitly or implicitly through a grant of ALL [PRIVILEGES]).

FORCE
Specify FORCE to revoke the EXECUTE object privilege on user-defined type objects with table or type dependencies. You must use FORCE to revoke the EXECUTE object privilege on user-defined type objects with table dependencies.

If you specify FORCE, all privileges will be revoked, but all dependent objects are marked INVALID, data in dependent tables becomes inaccessible, and all dependent function-based indexes are marked UNUSABLE. (Regranting the necessary type privilege will revalidate the table.)

See Also: *Oracle8i Concepts* for detailed information about type dependencies and user-defined object privileges

object_clause
ON *object_clause* identifies the objects on which privileges are to be revoked.

object
Specify the object on which the object privileges are to be revoked. This object can be:

- A table, view, sequence, procedure, stored function, or package, materialized view / snapshot
- A synonym for a table, view, sequence, procedure, stored function, package, or materialized view / snapshot
- A library, indextype, or user-defined operator

If you do not qualify object with *schema*, Oracle assumes the object is in your own schema.

If you revoke the SELECT object privilege (with or without the GRANT OPTION) on the containing table or snapshot of a materialized view, the materialized view will be invalidated.

If you revoke the SELECT object privilege (with or without the GRANT OPTION) on any of the master tables of a materialized view, both the view and its containing table or materialized view will be invalidated.

DIRECTORY *directory_name*

Specify the directory object on which privileges are to be revoked. You cannot qualify *directory_name* with *schema*. The object must be a directory.

See Also: CREATE DIRECTORY on page 9-40

JAVA SOURCE | RESOURCE

The JAVA clause lets you specify a Java source or resource schema object on which privileges are to be revoked.

Examples

Revoke a System Privilege from Users Example The following statement revokes the DROP ANY TABLE system privilege from the users bill and mary:

```
REVOKE DROP ANY TABLE
    FROM bill, mary;
```

bill and mary can no longer drop tables in schemas other than their own.

Revoke a Role from a User Example The following statement revokes the role controller from the user hanson:

```
REVOKE controller
    FROM hanson;
```

hanson can no longer enable the controller role.

Revoke a System Privilege from a Role Example The following statement revokes the CREATE TABLESPACE system privilege from the controller role:

```
REVOKE CREATE TABLESPACE
   FROM controller;
```

Enabling the controller role no longer allows users to create tablespaces.

Revoke a Role from a Role Example To revoke the role vp from the role ceo, issue the following statement:

```
REVOKE vp
  FROM ceo;
```

VP is no longer granted to ceo.

Revoke an Object Privilege from a User Example You can grant DELETE, INSERT, SELECT, and UPDATE privileges on the table bonus to the user pedro with the following statement:

```
GRANT ALL
    ON bonus TO pedro;
```

To revoke the DELETE privilege on bonus from pedro, issue the following statement:

```
REVOKE DELETE
    ON bonus FROM pedro;
```

Revoke All Object Privileges from a User Example To revoke the remaining privileges on bonus that you granted to pedro, issue the following statement:

```
REVOKE ALL
    ON bonus FROM pedro;
```

Revoke Object Privileges from PUBLIC Example You can grant SELECT and UPDATE privileges on the view reports to all users by granting the privileges to the role PUBLIC:

```
GRANT SELECT, UPDATE
    ON reports TO public;
```

The following statement revokes UPDATE privilege on reports from all users:

```
REVOKE UPDATE
    ON reports FROM public;
```

Users can no longer update the reports view, although users can still query it. However, if you have also granted the UPDATE privilege on reports to any users, either directly or through roles, these users retain the privilege.

Revoke an Object Privilege on a Sequence from a User Example You can grant the user blake the SELECT privilege on the eseq sequence in the schema elly with the following statement:

```
GRANT SELECT
    ON elly.eseq TO blake;
```

To revoke the SELECT privilege on eseqfrom blake, issue the following statement:

```
REVOKE SELECT
    ON elly.eseq FROM blake;
```

However, if the user elly has also granted SELECT privilege on eseq to blake, blake can still use eseq by virtue of elly's grant.

Revoke an Object Privilege with CASCADE CONSTRAINTS Example You can grant blake the privileges REFERENCES and UPDATE on the emp table in the schema scott with the following statement:

```
GRANT REFERENCES, UPDATE
    ON scott.emp TO blake;
```

blake can exercise the REFERENCES privilege to define a constraint in his own dependent table that refers to the emp table in the schema scott:

```
CREATE TABLE dependent
(dependno   NUMBER,
 dependname VARCHAR2(10),
 employee   NUMBER
    CONSTRAINT in_emp REFERENCES scott.emp(ename) );
```

You can revoke the REFERENCES privilege on scott.emp from blake, by issuing the following statement that contains the CASCADE CONSTRAINTS clause:

```
REVOKE REFERENCES
    ON scott.emp
    FROM blake
    CASCADE CONSTRAINTS;
```

Revoking blake's REFERENCES privilege on scott.emp causes Oracle to drop the in_emp constraint, because blake required the privilege to define the constraint.

However, if blake has also been granted the REFERENCES privilege on scott.emp by a user other than you, Oracle does not drop the constraint. blake still has the privilege necessary for the constraint by virtue of the other user's grant.

Revoke an Object Privilege on a Directory from a User Example You can revoke READ privilege on directory bfile_dir1 from sue, by issuing the following statement:

```
REVOKE READ ON DIRECTORY bfile_dir1 FROM sue;
```

ROLLBACK

Purpose

Use the ROLLBACK statement to undo work done in the current transaction, or to manually undo the work done by an in-doubt distributed transaction.

Note: Oracle recommends that you explicitly end transactions in application programs using either a COMMIT or ROLLBACK statement. If you do not explicitly commit the transaction and the program terminates abnormally, Oracle rolls back the last uncommitted transaction.

See Also:

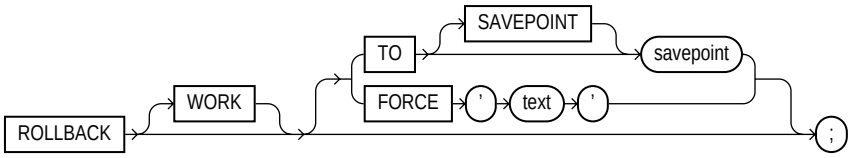
- *Oracle8i Concepts* for information on transactions
- *Oracle8i Distributed Database Systems* for information on distributed transactions
- SET TRANSACTION on page 11-125 for information on setting characteristics of the current transaction
- COMMIT on page 8-133
- SAVEPOINT on page 11-86

Prerequisites

To roll back your current transaction, no privileges are necessary.

To manually roll back an in-doubt distributed transaction that you originally committed, you must have the FORCE TRANSACTION system privilege. To manually roll back an in-doubt distributed transaction originally committed by another user, you must have the FORCE ANY TRANSACTION system privilege.

Syntax



Keywords and Parameters

WORK

The keyword WORK is optional and is provided for ANSI compatibility.

TO SAVEPOINT *savepoint*

Specify the savepoint to which you want to roll back the current transaction. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.

Using ROLLBACK **without** the TO SAVEPOINT clause performs the following operations:

- Ends the transaction
- Undoes all changes in the current transaction
- Erases all savepoints in the transaction
- Releases the transaction's locks

See Also: SAVEPOINT on page 11-86

Using ROLLBACK **with** the TO SAVEPOINT clause performs the following operations:

- Rolls back just the portion of the transaction after the savepoint.
- Erases all savepoints created after that savepoint. The named savepoint is retained, so you can roll back to the same savepoint multiple times. Prior savepoints are also retained.
- Releases all table and row locks acquired since the savepoint. Other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the rows immediately.

Restriction: You cannot manually roll back an in-doubt transaction to a savepoint.

FORCE

Specify FORCE to manually roll back an in-doubt distributed transaction. The transaction is identified by the '*text*' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING.

A ROLLBACK statement with a FORCE clause rolls back only the specified transaction. Such a statement does not affect your current transaction.

Restriction: ROLLBACK statements with the FORCE clause are not supported in PL/SQL.

> **See Also:** *Oracle8i Distributed Database Systems* for more information on distributed transactions and rolling back in-doubt transactions

Examples

The following statement rolls back your entire current transaction:

```
ROLLBACK;
```

The following statement rolls back your current transaction to savepoint sp5:

```
ROLLBACK TO SAVEPOINT sp5;
```

The following statement manually rolls back an in-doubt distributed transaction:

```
ROLLBACK WORK
    FORCE '25.32.87';
```

SAVEPOINT

Purpose

Use the SAVEPOINT statement to identify a point in a transaction to which you can later roll back.

See Also:

- *Oracle8i Concepts* for information on savepoints.
- ROLLBACK on page 11-83 for information on rolling back transactions
- SET TRANSACTION on page 11-125 for information on setting characteristics of the current transaction

Prerequisites

None.

Syntax

Keywords and Parameters

savepoint

Specify the name of the savepoint to be created.

Savepoint names must be distinct within a given transaction. If you create a second savepoint with the same identifier as an earlier savepoint, the earlier savepoint is erased. After a savepoint has been created, you can either continue processing, commit your work, roll back the entire transaction, or roll back to the savepoint.

Example

To update blake's and clark's salary, check that the total company salary does not exceed 27,000, then reenter clark's salary, enter:

```
UPDATE emp
    SET sal = 2000
```

```
    WHERE ename = 'BLAKE';
SAVEPOINT blake_sal;

UPDATE emp
    SET sal = 1500
    WHERE ename = 'CLARK';
SAVEPOINT clark_sal;

SELECT SUM(sal) FROM emp;

ROLLBACK TO SAVEPOINT blake_sal;

UPDATE emp
    SET sal = 1200
    WHERE ename = 'CLARK';

COMMIT;
```

SELECT and *subquery*

Purpose

Use a SELECT statement or subquery to retrieve data from one or more tables, object tables, views, object views, or materialized views.

Note: If the result (or part of the result) of a SELECT statement is equivalent to an existing materialized view, Oracle may use the materialized view in place of one or more tables specified in the SELECT statement. This substitution is called **query rewrite**, and takes place only if cost optimization is enabled and the QUERY_REWRITE_ENABLED parameter is set to TRUE. To determine whether query write has occurred, use the EXPLAIN PLAN statement.

See Also:

- "Queries and Subqueries" on page 5-21 for general information on queries and subqueries
- *Oracle8i Data Warehousing Guide* for more information on materialized views and query rewrite
- EXPLAIN PLAN on page 11-23

Prerequisites

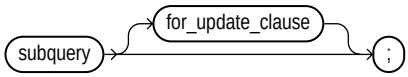
For you to select data from a table or materialized view, the table or materialized view must be in your own schema or you must have the SELECT privilege on the table or materialized view.

For you to select rows from the base tables of a view,

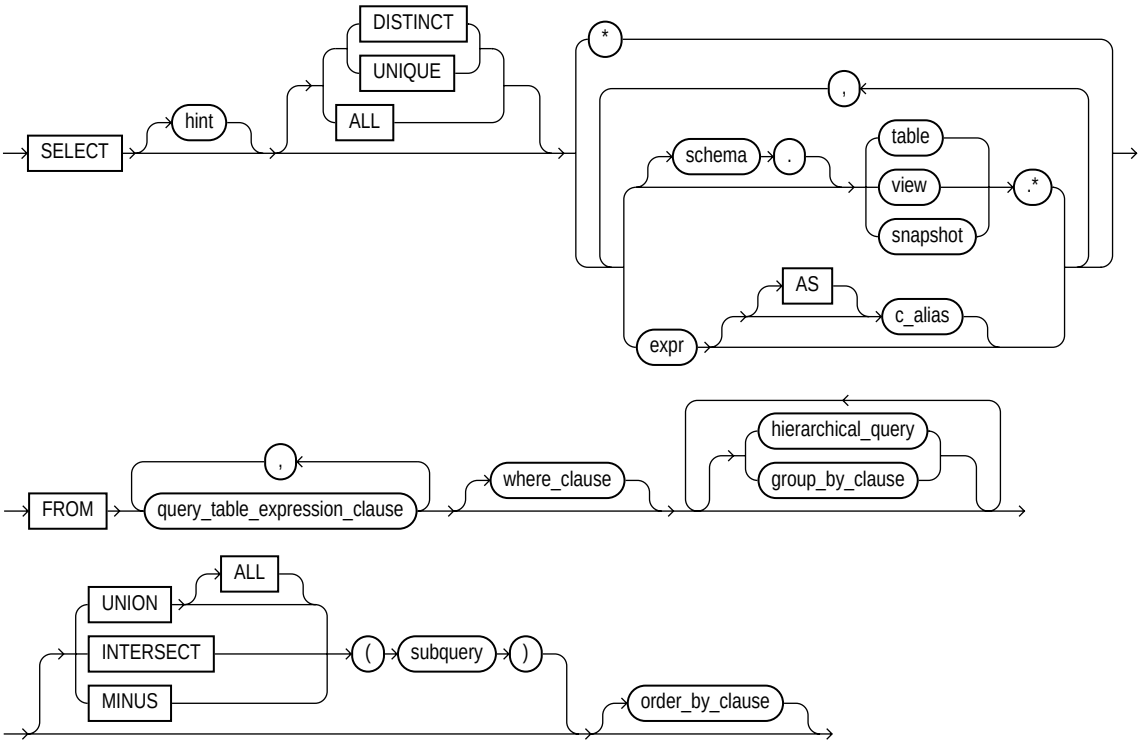
- You must have the SELECT privilege on the view, and
- Whoever owns the schema containing the view must have the SELECT privilege on the base tables.

The SELECT ANY TABLE system privilege also allows you to select data from any table or any materialized view or any view's base table.

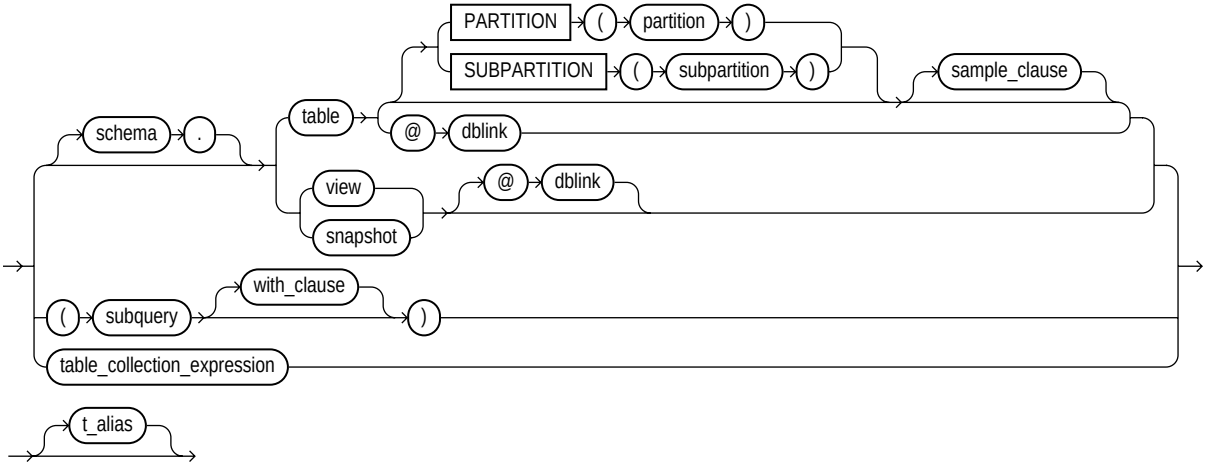
Syntax



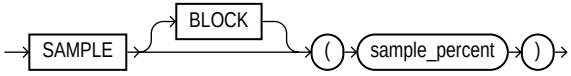
subquery::=



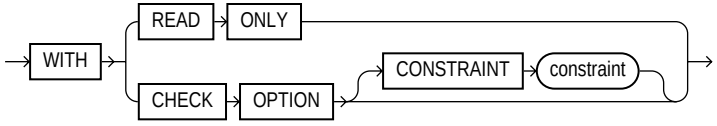
query_table_expression_clause::=



sample_clause::=

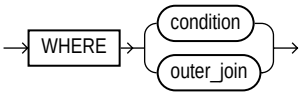


with_clause::=

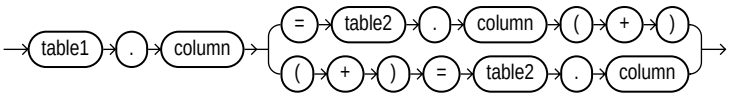


table_collection_expression::=

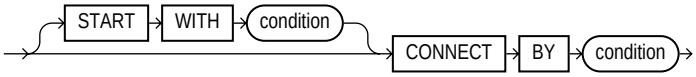
where_clause::=



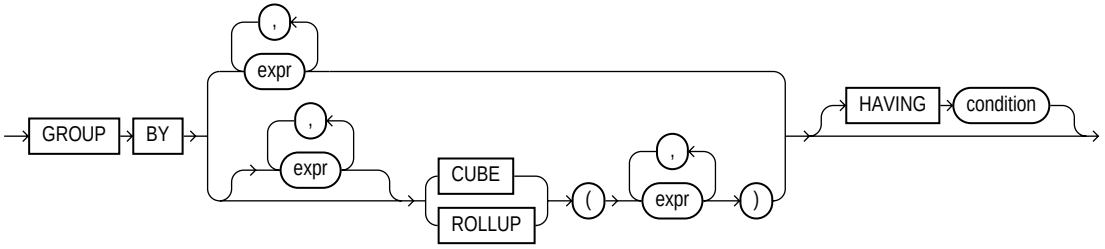
outer_join::=



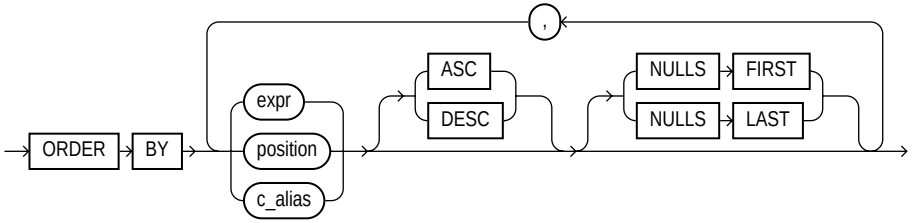
hierarchical_query_clause::=



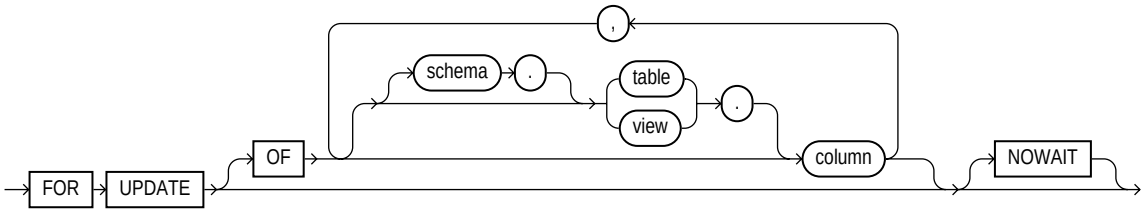
group_by_clause::=



order_by_clause::=



for_update_clause::=



Keywords and Parameters

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

> **See Also:** "Hints" on page 2-67 and *Oracle8i Performance Guide and Reference* for the syntax and description of hints

DISTINCT | UNIQUE

Specify DISTINCT or UNIQUE if you want Oracle to return only one copy of each set of duplicate rows selected (these two keywords are synonymous). Duplicate rows are those with matching values for each expression in the select list.

Restrictions:

- When you specify DISTINCT or UNIQUE, the total number of bytes in all select list expressions is limited to the size of a data block minus some overhead. This size is specified by the initialization parameter DB_BLOCK_SIZE.
- You cannot specify DISTINCT if the FROM clause contains LOB columns.

ALL

Specify ALL if you want Oracle to return all rows selected, including all copies of duplicates. The default is ALL.

*

Specify the asterisk to select all columns from all tables, views, or materialized views listed in the FROM clause.

Note: If you are selecting from a table (that is, you specify a table in the FROM clause rather than a view or a materialized view), columns that have been marked as UNUSED by the ALTER TABLE SET UNUSED statement are not selected.

See Also: ALTER TABLE on page 8-2

schema

Specify the schema containing the selected table, view, or materialized view. If you omit *schema*, Oracle assumes the table, view, or materialized view is in your own schema.

table.|view.*|snapshot.**

Specify the object name followed by a period and the asterisk to select all columns from the specified table, view, or materialized view. You can use the schema qualifier to select from a table, view, or materialized view in a schema other than your own. A query that selects rows from two or more tables, views, or materialized views is a join.

See Also: "Joins" on page 5-24

expr

Specify an expression representing the information you want to select. A column name in this list can be qualified with *schema* only if the table, view, or materialized view containing the column is qualified with *schema* in the FROM clause.

See Also: "Expressions" on page 5-2 for the syntax of *expr*

Restrictions:

- If you also specify a *group_by_clause* in this statement, this select list can contain only the following types of expressions:
  - Constants
  - Aggregate functions and the functions USER, UID, and SYSDATE
  - Expressions identical to those in the *group_by_clause*

  - Expressions involving the above expressions that evaluate to the same value for all rows in a group

- You can select a rowid from a join view only if the join has one and only one key-preserved table. The rowid of that table becomes the rowid of the view. For information on key-preserved tables, see *Oracle8i Administrator's Guide*.
- If two or more tables have some column names in common, you must qualify column names with names of tables.

c_alias Specify a different name (alias) for the column expression. Oracle will use this alias in the column heading. The AS keyword is optional. The alias effectively renames the select list item for the duration of the query. The alias can be used in the *order_by_clause*, but not other clauses in the query.

FROM Clause

query_table_expression_clause The FROM clause lets you specify the table, view, materialized view, or partition from which data is selected, or a subquery that specifies the objects from which data is selected.

PARTITION (*partition*)

SUBPARTITION (*subpartition*) Specify the partition or subpartition from which you want to retrieve data. The *partition* parameter may be the name of the partition within *table* from which to retrieve data or a more complicated predicate restricting retrieval to just one partition of the table.

dblink Specify the complete or partial name for a database link to a remote database where the table, view, or materialized view is located. This database need not be an Oracle database.

See Also:

- "Referring to Objects in Remote Databases" on page 2-90 for more information on referring to database links

- "Distributed Queries" on page 5-29for more information about distributed queries

If you omit *dblink*, Oracle assumes that the table, view, or materialized view is on the local database.

Restriction: You cannot query a user-defined type or an object REF on a remote table.

table, *view*, *snapshot* — Specify the name of a table, view, or materialized view from which data is selected. "Materialized view" is synonymous with "snapshot".

sample_clause

The sample_clause lets you instruct Oracle to select from a random sample of rows from the table, rather than from the entire table.

BLOCK — BLOCK instructs Oracle to perform random block sampling instead of random row sampling.

See Also: *Oracle8i Concepts* for a discussion of the difference

sample_percent — *sample_percent* is a number specifying the percentage of the total row or block count to be included in the sample. The value must be in the range .000001 to (but not including) 100.

Restrictions on the *sample_clause*:

- You can specify SAMPLE only in a query that selects from a single table. Joins are not supported. However, you can achieve the same results by using a CREATE TABLE ... AS SELECT query to materialize a sample of an underlying table and then rewrite the original query to refer to the newly created table sample. If you wish, you can write additional queries to materialize samples for other tables.

  See Also: "SAMPLE Examples" on page 11-104

- When you specify SAMPLE, Oracle automatically uses cost-based optimization. Rule-based optimization is not supported with this clause.

Caution: The use of statistically incorrect assumptions when using this feature can lead to incorrect or undesirable results. Refer to *Oracle8i Concepts* for more information on using the *sample_clause*.

with_clause

The *with_clause* lets you restrict the subquery in one of the following ways:

WITH READ ONLY	Specify WITH READ ONLY to indicate that the subquery cannot be updated.
WITH CHECK OPTION	Specify WITH CHECK OPTION to indicate that, if the subquery is used in place of a table in an INSERT, UPDATE, or DELETE statement, Oracle prohibits any changes to that table that would produce rows that are not included in the subquery.

See Also: WITH CHECK OPTION Example on page 11-108

table_collection_expression

The *table_collection_expression* lets you inform Oracle that the collection value expression should be treated as a table for purposes of query and DML operations. The *collection_expression* can be a subquery, a column, a CAST or DECODE expression, a function, or a collection constructor. Regardless of its form, it must return a collection value (that is, a value whose type is nested table or varray). This process of extracting the elements of a collection is called **collection unnesting**.

The *collection_expression* can reference columns of tables defined to its left in the FROM clause. This is called **left correlation**. Left correlation can occur only in *table_collection_expression*. Other subqueries cannot contains references to columns defined outside the subquery.

The optional "(+)" lets you specify that *table_collection_expression* should return a row with all fields set to NULL if the collection is null or empty. The "(+)" is valid only if *collection_expression* uses left correlation. The result is similar to that of an outer join.

Note: In earlier releases of Oracle, when *collection_expression* was a subquery, *table_collection_expr* was expressed as "THE *subquery*". That usage is now deprecated.

See Also:

- "Outer Joins" on page 5-25
- "Collection Unnesting Examples" on page 11-115

t_alias

Specify a **correlation name** (alias) for the table, view, materialized view, or subquery for evaluating the query. Correlation names are most often used in a correlated query. Other references to the table, view, or materialized view throughout the query must refer to this alias.

Note: This alias is **required** if the *query_table_expression_clause* references any object type attributes or object type methods.

---

where_clause

The *where_clause* lets you restrict the rows selected to those that satisfy one or more conditions.

- *condition* can be any valid SQL condition.

  See Also: the syntax description of condition in "Expressions" on page 5-2

- *outer_join* applies only if the *query_table_expression_clause* specifies more than one table. This special form of condition requires Oracle to return all the rows that satisfy the condition, as well as all the rows from one of the tables for which no rows of the other table satisfy the condition.

  If one of the elements in the *query_table_expression_clause* is actually a nested table or some other form of collection, you specify the outer-join syntax in the *table_collection_expression* rather than in the *where_clause*.

  See Also: "Outer Joins" on page 5-25 for more information, including rules and restrictions that apply to outer joins

If you omit this clause, Oracle returns all rows from the tables, views, or materialized views in the FROM clause.

Note: If this clause refers to a DATE column of a partitioned table or index, Oracle performs partition pruning only if (1) you created the table or index partitions by fully specifying the year using the TO_DATE function with a 4-digit format mask, **and** (2) you specify the date in the query's *where_clause* using the TO_DATE function and either a 2- or 4-digit format mask.

See Also: the "PARTITION Example" on page 11-104

hierarchical_query_clause

The *hierarchical_query_clause* lets you select rows in a hierarchical order. For a discussion of hierarchical queries, see "Hierarchical Queries" on page 5-22.

The preceding *where_clause*, if specified, restricts the rows returned by the query without affecting other rows of the hierarchy.

SELECT statements that contain hierarchical queries can contain the LEVEL pseudocolumn. LEVEL returns the value 1 for a root node, 2 for a child node of a root node, 3 for a grandchild, etc. The number of levels returned by a hierarchical query may be limited by available user memory.

See Also:

- "Pseudocolumns" on page 2-59 for more information on LEVEL
- "Hierarchical Queries" on page 5-22 for general information on hierarchical queries

Restrictions: If you specify a hierarchical query:

- The same statement cannot also perform a join.
- The same statement cannot also select data from a view whose query performs a join.
- If you also specify the *order_by_clause*, it takes precedence over any ordering specified by the hierarchical query.

START WITH *condition* — Specify a condition that identifies the row(s) to be used as the root(s) of a hierarchical query. Oracle uses as root(s) all rows that satisfy this condition. If you omit this clause, Oracle uses all rows in the table as root rows. The START WITH *condition* can contain a subquery.

CONNECT BY *condition*

Specify a condition that identifies the relationship between parent rows and child rows of the hierarchy. *condition* can be any condition as described in "Conditions" on page 5-15. However, some part of the condition must use the PRIOR operator to refer to the parent row. The part of the condition containing the PRIOR operator must have one of the following forms:

- `PRIOR expr comparison_operator expr`
- `expr comparison_operator PRIOR expr`

Restriction: The CONNECT BY condition cannot contain a subquery.

group_by_clause

Use the *group_by_clause* to group the selected rows based on the value of *expr(s)* for each row, and returns a single row of summary information for each group. If this clause contains CUBE or ROLLUP extensions, then superaggregate groupings are produced in addition to the regular groupings.

Expressions in the *group_by_clause* can contain any columns in the tables, views, and materialized views in the FROM clause, regardless of whether the columns appear in the select list.

Restrictions:

- The *group_by_clause* can contain no more than 255 expressions.
- You cannot specify LOB columns, nested tables, or varrays as part of *expr*.
- The total number of bytes in all expressions in the *group_by_clause* is limited to the size of a data block (specified by the initialization parameter DB_BLOCK_SIZE) minus some overhead.
- If the *group_by_clause* references any object columns, the query will not be parallelized.

ROLLUP

ROLLUP is an extension to the *group_by_clause* that groups the selected rows based on the values of the first n, n-1, n-2, ... 0 expressions for each row, and returns a single row of summary for each group. You can use the ROLLUP operation to produce **subtotal values**.

For example, given three expressions in the ROLLUP clause of the *group_by_clause*, the operation results in n+1 = 3+1 = 4 groupings.

Rows based on the values of the first 'n' expressions are called **regular rows**, and the others are called **superaggregate rows**.

See Also:

- GROUPING on page 4-41 for an example

- *Oracle8i Data Warehousing Guide*

CUBE

CUBE is an extension to the *group_by_clause* that groups the selected rows based on the values of all possible combinations of expressions for each row, and returns a single row of summary information for each group. You can use the CUBE operation to produce **cross-tabulation values**.

For example, given three expressions in the CUBE clause of the *group_by_clause*, the operation results in $2^n = 2^3 = 8$ groupings. Rows based on the values of 'n' expressions are called regular rows, and the rest are called **superaggregate rows**.

See Also:

- GROUPING on page 4-41

- "CUBE Example" on page 11-105 for an example

- *Oracle8i Data Warehousing Guide.*

HAVING

Use the HAVING clause to restrict the groups of rows returned to those groups for which the specified *condition* is TRUE. If you omit this clause, Oracle returns summary rows for all groups.

Specify GROUP BY and HAVING after the *where_clause* and CONNECT BY clause. If you specify both GROUP BY and HAVING, they can appear in either order.

See Also: the syntax description of *expr* in "Expressions" on page 5-2 and the syntax description of *condition* in "Conditions" on page 5-15

Set Operators

`UNION | UNION ALL | INTERSECT | MINUS`

These set operators combine the rows returned by two `SELECT` statements into a single result. The number and datatypes of the columns selected by each component query must be the same, but the column lengths can be different.

If you combine more than two queries with set operators, Oracle evaluates adjacent queries from left to right. You can use parentheses to specify a different order of evaluation.

See Also: "Set Operators: UNION [ALL], INTERSECT, MINUS" on page 3-12 for information on these operators

Restrictions:

- These set operators are not valid on columns of type `BLOB`, `CLOB`, `BFILE`, varray, or nested table.
- The `UNION`, `INTERSECT`, and `MINUS` operators are not valid on `LONG` columns.
- To reference a column, you must use an alias to name the column.
- You cannot also specify the *`for_update_clause`* with these set operators.
- You cannot specify the *`order_by_clause`* in the *`subquery`* of these operators.
- You cannot use these operators in `SELECT` statements containing `TABLE` collection expressions.
- The total number of bytes in all select list expressions of a component query is limited to the size of a data block (specified by the initialization parameter `DB_BLOCK_SIZE`) minus some overhead.

Note: To comply with emerging SQL standards, a future release of Oracle will give the `INTERSECT` operator greater precedence than the other set operators. Therefore, you should use parentheses to specify order of evaluation in queries that use the `INTERSECT` operator with other set operators.

---

order_by_clause

Use the *order_by_clause* to order rows returned by the statement. Without an *order_by_clause*, no guarantee exists that the same query executed more than once will retrieve rows in the same order.

- *expr* orders rows based on their value for *expr*. The expression is based on columns in the select list or columns in the tables, views, or materialized views in the FROM clause.
- *position* orders rows based on their value for the expression in this position of the select list; *position* must be an integer.

> **See Also:** "Sorting Query Results" on page 5-23 for a discussion of ordering query results

You can specify multiple expressions in the *order_by_clause*. Oracle first sorts rows based on their values for the first expression. Rows with the same value for the first expression are then sorted based on their values for the second expression, and so on. Oracle sorts nulls following all others in ascending order and preceding all others in descending order.

ASC | DESC	Specify whether the ordering sequence is ascending or descending. ASC is the default.
NULLS FIRST | NULLS LAST	Specify whether returned rows containing null values should appear first or last in the ordering sequence. NULLS LAST is the default for ascending order, and NULLS FIRST is the default for descending order.

Restrictions:

- If you have specified the DISTINCT operator in this statement, this clause cannot refer to columns unless they appear in the select list.
- An *order_by_clause* can contain no more than 255 expressions.
- You cannot order by a LOB column, nested table, or varray.

If you specify a *group_by_clause* in the same statement, this *order_by_clause* is restricted to the following expressions:

- Constants
- Aggregate functions
- Analytic functions

- The functions USER, UID, and SYSDATE
- Expressions identical to those in the *group_by_clause*
- Expressions involving the above expressions that evaluate to the same value for all rows in a group.

for_update_clause

The *for_update_clause* lets you lock the selected rows so that other users cannot lock or update the rows until you end your transaction. You can specify this clause only in a top-level SELECT statement (not in subqueries).

- Prior to updating a LOB value, you must lock the row containing the LOB. One way to lock the row is with a SELECT ... FOR UPDATE statement.

  See Also: "LOB Locking Example" on page 11-108

- Nested table rows are not locked as a result of locking the parent table rows. If you want the nested table rows to be locked, you must lock them explicitly.

OF — Use the OF clause to lock the select rows only for a particular table or view in a join. The columns in the OF clause only indicate which table or view rows are locked. The specific columns that you specify are not significant. However, you must specify an actual column name, not a column alias. If you omit this clause, Oracle locks the selected rows from all the tables in the query.

NOWAIT — Specify NOWAIT to return control to you if the SELECT statement attempts to lock a row that is locked by another user. If you omit this clause, Oracle waits until the row is available and then returns the results of the SELECT statement.

Restrictions:

- You cannot specify this clause with the following other constructs: the DISTINCT or CURSOR operator, set operators, *group_by_clause*, or aggregate functions.
- The tables locked by this clause must all be located on the same database, and on the same database as any LONG columns and sequences referenced in the same statement.

Examples

Simple Query Examples The following statement selects rows from the emp table with the department number of 30:

```
SELECT *
    FROM emp
    WHERE deptno = 30;
```

The following statement selects the name, job, salary and department number of all employees except sales people from department number 30:

```
SELECT ename, job, sal, deptno
    FROM emp
    WHERE NOT (job = 'SALESMAN' AND deptno = 30);
```

The following statement selects from subqueries in the FROM clause and gives departments' total employees and salaries as a decimal value of all the departments:

```
SELECT a.deptno "Department",
       a.num_emp/b.total_count "%Employees",
       a.sal_sum/b.total_sal "%Salary"
  FROM
 (SELECT deptno, COUNT(*) num_emp, SUM(SAL) sal_sum
    FROM scott.emp
    GROUP BY deptno) a,
 (SELECT COUNT(*) total_count, SUM(sal) total_sal
    FROM scott.emp) b ;
```

PARTITION Example You can select rows from a single partition of a partitioned table by specifying the keyword PARTITION in the FROM clause. This SQL statement assigns an alias for and retrieves rows from the nov98 partition of the sales table:

```
SELECT * FROM sales PARTITION (nov98) s
   WHERE s.amount_of_sale > 1000;
```

The following example selects rows from the sales table for sales earlier than a specified date:

```
SELECT * FROM sales
   WHERE sale_date < TO_DATE('1998-06-15', 'YYYY-MM-DD');
```

SAMPLE Examples The following query estimates the number of employees in the emp table:

```
SELECT COUNT(*) * 100 FROM emp SAMPLE BLOCK (1);
```

The following example creates a sampled subset of the emp table and then joins the resulting sampled table with dept. This operation circumvents the restriction that you cannot specify the *sample_clause* in join queries:

```
CREATE TABLE sample_emp AS SELECT empno, deptno FROM emp SAMPLE(10);
SELECT e.empno FROM sample_emp e, dept d
   WHERE e.deptno = d.deptno AND d.name = 'DEV';
```

GROUP BY Examples To return the minimum and maximum salaries for each department in the employee table, issue the following statement:

```
SELECT deptno, MIN(sal), MAX (sal)
     FROM emp
     GROUP BY deptno;

DEPTNO     MIN(SAL)   MAX(SAL)
---------- ---------- ----------
        10       1300       5000
        20        800       3000
        30        950       2850
```

To return the minimum and maximum salaries for the clerks in each department, issue the following statement:

```
SELECT deptno, MIN(sal), MAX (sal)
     FROM emp
     WHERE job = 'CLERK'
     GROUP BY deptno;

DEPTNO     MIN(SAL)   MAX(SAL)
---------- ---------- ----------
        10       1300      1300
        20        800      1100
        30        950       950
```

CUBE Example To return the number of employees and their average yearly salary across all possible combinations of department and job category, issue the following query:

```
SELECT DECODE(GROUPING(dname), 1, 'All Departments',
      dname) AS dname,
   DECODE(GROUPING(job), 1, 'All Jobs', job) AS job,
   COUNT(*) "Total Empl", AVG(sal) * 12 "Average Sal"
   FROM emp, dept
   WHERE dept.deptno = emp.deptno
```

```
   GROUP BY CUBE (dname, job);

DNAME          JOB       Total Empl Average Sa
-------------- --------- ---------- ----------
ACCOUNTING     CLERK              1      15600
ACCOUNTING     MANAGER            1      29400
ACCOUNTING     PRESIDENT          1      60000
ACCOUNTING     All Jobs           3      35000
RESEARCH       ANALYST            2      36000
RESEARCH       CLERK              2      11400
RESEARCH       MANAGER            1      35700
RESEARCH       All Jobs           5      26100
SALES          CLERK              1      11400
SALES          MANAGER            1      34200
SALES          SALESMAN           4      16800
SALES          All Jobs           6      18800
All Departments ANALYST           2      36000
All Departments CLERK             4      12450
All Departments MANAGER           3      33100
All Departments PRESIDENT         1      60000
All Departments SALESMAN          4      16800
All Departments All Jobs         14 24878.5714
```

Hierarchical Query Examples The following CONNECT BY clause defines a hierarchical relationship in which the empno value of the parent row is equal to the mgr value of the child row:

```
CONNECT BY PRIOR empno = mgr;
```

In the following CONNECT BY clause, the PRIOR operator applies only to the empno value. To evaluate this condition, Oracle evaluates empno values for the parent row and mgr, sal, and comm values for the child row:

```
CONNECT BY PRIOR empno = mgr AND sal > comm;
```

To qualify as a child row, a row must have a mgr value equal to the empno value of the parent row and it must have a sal value greater than its comm value.

HAVING Example To return the minimum and maximum salaries for the clerks in each department whose lowest salary is below $1,000, issue the next statement:

```
SELECT deptno, MIN(sal), MAX (sal)
     FROM emp
     WHERE job = 'CLERK'
```

```
     GROUP BY deptno
     HAVING MIN(sal) < 1000;

DEPTNO     MIN(SAL)   MAX(SAL)
---------- ---------- ----------
        20        800       1100
        30        950        950
```

ORDER BY Examples To select all salesmen's records from emp, and order the results by commission in descending order, issue the following statement:

```
SELECT *
    FROM emp
    WHERE job = 'SALESMAN'
    ORDER BY comm DESC;
```

To select the employees from emp ordered first by ascending department number and then by descending salary, issue the following statement:

```
SELECT ename, deptno, sal
    FROM emp
    ORDER BY deptno ASC, sal DESC;
```

To select the same information as the previous SELECT and use the positional ORDER BY notation, issue the following statement:

```
SELECT ename, deptno, sal
    FROM emp
    ORDER BY 2 ASC, 3 DESC;
```

FOR UPDATE Examples The following statement locks rows in the emp table with clerks located in New York and locks rows in the dept table with departments in New York that have clerks:

```
SELECT empno, sal, comm
    FROM emp, dept
    WHERE job = 'CLERK'
        AND emp.deptno = dept.deptno
        AND loc = 'NEW YORK'
    FOR UPDATE;
```

The following statement locks only those rows in the emp table with clerks located in New York. No rows are locked in the dept table:

```
SELECT empno, sal, comm
    FROM emp, dept
```

```
    WHERE job = 'CLERK'
        AND emp.deptno = dept.deptno
        AND loc = 'NEW YORK'
    FOR UPDATE OF emp.sal;
```

LOB Locking Example The following example uses a SELECT ... FOR UPDATE statement to lock a row containing a LOB prior to updating the LOB value.

```
INSERT INTO t_table VALUES (1, 'abcd');

COMMIT;
   DECLARE
     num_var      NUMBER;
     clob_var     CLOB;
     clob_locked  CLOB;
     write_amount NUMBER;
     write_offset NUMBER;
     buffer       VARCHAR2(20) := 'efg';

   BEGIN
     SELECT clob_col INTO clob_locked FROM t_table
     WHERE num_col = 1 FOR UPDATE;

  write_amount := 3;
  dbms_lob.write(clob_locked, write_amount, write_offset, buffer);
END;
```

WITH CHECK OPTION Example The following statement is legal even though the second value violates the condition of the subquery *where_clause*:

```
INSERT INTO
   (SELECT ename, deptno FROM emp WHERE deptno < 10)
    VALUES ('Taylor', 20);
```

However, the following statement is illegal because of the WITH CHECK OPTION clause:

```
INSERT INTO
   (SELECT ename, deptno FROM emp
       WHERE deptno < 10
       WITH CHECK OPTION)
    VALUES ('Taylor', 20);
```

Equijoin Examples This equijoin returns the name and job of each employee and the number and name of the department in which the employee works:

```
SELECT ename, job, dept.deptno, dname
     FROM emp, dept
     WHERE emp.deptno = dept.deptno;

ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
CLARK      MANAGER           10 ACCOUNTING
KING       PRESIDENT         10 ACCOUNTING
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
FORD       ANALYST           20 RESEARCH
SCOTT      ANALYST           20 RESEARCH
JONES      MANAGER           20 RESEARCH
ALLEN      SALESMAN          30 SALES
BLAKE      MANAGER           30 SALES
MARTIN     SALESMAN          30 SALES
JAMES      CLERK             30 SALES
TURNER     SALESMAN          30 SALES
WARD       SALESMAN          30 SALES
```

You must use a join to return this data because employee names and jobs are stored in a different table than department names. Oracle combines rows of the two tables according to this join condition:

```
emp.deptno = dept.deptno
```

The following equijoin returns the name, job, department number, and department name of all clerks:

```
SELECT ename, job, dept.deptno, dname
     FROM emp, dept
     WHERE emp.deptno = dept.deptno
     AND job = 'CLERK';

ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
JAMES      CLERK             30 SALES
```

This query is identical to the preceding example, except that it uses an additional *where_clause* condition to return only rows with a `job` value of '`CLERK`'.

Subquery Examples To determine who works in Taylor's department, issue the following statement:

```
SELECT ename, deptno
    FROM emp
    WHERE deptno =
        (SELECT deptno
            FROM emp
            WHERE ename = 'TAYLOR');
```

To give all employees in the emp table a 10% raise if they have not already been issued a bonus (if they do not appear in the bonus table), issue the following statement:

```
UPDATE emp
    SET sal = sal * 1.1
    WHERE empno NOT IN (SELECT empno FROM bonus);
```

To create a duplicate of the dept table named newdept, issue the following statement:

```
CREATE TABLE newdept (deptno, dname, loc)
    AS SELECT deptno, dname, loc FROM dept;
```

Self Join Example The following query uses a self join to return the name of each employee along with the name of the employee's manager:

```
SELECT e1.ename||' works for '||e2.ename
"Employees and their Managers"
    FROM emp e1, emp e2   WHERE e1.mgr = e2.empno;

Employees and their Managers
------------------------------
BLAKE works for KING
CLARK works for KING
JONES works for KING
FORD works for JONES
SMITH works for FORD
ALLEN works for BLAKE
WARD works for BLAKE
MARTIN works for BLAKE
SCOTT works for JONES
TURNER works for BLAKE
ADAMS works for SCOTT
JAMES works for BLAKE
MILLER works for CLARK
```

The join condition for this query uses the aliases `e1` and `e2` for the `emp` table:

```
e1.mgr = e2.empno
```

Outer Join Examples This query uses an outer join to extend the results of the Equijoin example above:

```
SELECT ename, job, dept.deptno, dname
    FROM emp, dept
    WHERE emp.deptno (+) = dept.deptno;

ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
CLARK      MANAGER           10 ACCOUNTING
KING       PRESIDENT         10 ACCOUNTING
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
FORD       ANALYST           20 RESEARCH
SCOTT      ANALYST           20 RESEARCH
JONES      MANAGER           20 RESEARCH
ALLEN      SALESMAN          30 SALES
BLAKE      MANAGER           30 SALES
MARTIN     SALESMAN          30 SALES
JAMES      CLERK             30 SALES
TURNER     SALESMAN          30 SALES
WARD       SALESMAN          30 SALES
                             40 OPERATIONS
```

In this outer join, Oracle returns a row containing the `operations` department even though no employees work in this department. Oracle returns `NULL` in the `ename` and `job` columns for this row. The join query in this example selects only departments that have employees.

The following query uses an outer join to extend the results of the preceding example:

```
SELECT ename, job, dept.deptno, dname
    FROM emp, dept
    WHERE emp.deptno (+) = dept.deptno
        AND job (+) = 'CLERK';

ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
MILLER     CLERK             10 ACCOUNTING
```

```
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
JAMES      CLERK             30 SALES
                             40 OPERATIONS
```

In this outer join, Oracle returns a row containing the `operations` department even though no clerks work in this department. The (+) operator on the `job` column ensures that rows for which the `job` column is `NULL` are also returned. If this (+) were omitted, the row containing the `operations` department would not be returned because its `job` value is not '`CLERK`'.

This example shows four outer join queries on the customers, `orders`, `lineitems`, and `parts` tables. These tables are shown here:

```
SELECT custno, custname
    FROM customers;

CUSTNO     CUSTNAME
---------- --------------------
         1 Angelic Co.
         2 Believable Co.
         3 Cables R Us

SELECT orderno, custno,
    TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE"
    FROM orders;

ORDERNO    CUSTNO     ORDERDATE
---------- ---------- -----------
      9001          1 OCT-13-1998
      9002          2 OCT-13-1998
      9003          1 OCT-20-1998
      9004          1 OCT-27-1998
      9005          2 OCT-31-1998

SELECT orderno, lineno, partno, quantity
    FROM lineitems;

ORDERNO    LINENO     PARTNO     QUANTITY
---------- ---------- ---------- ----------
      9001          1        101         15
      9001          2        102         10
      9002          1        101         25
      9002          2        103         50
      9003          1        101         15
```

```
      9004          1        102         10
      9004          2        103         20

SELECT partno, partname
    FROM parts;

PARTNO PARTNAME
------ --------
   101 X-Ray Screen
   102 Yellow Bag
   103 Zoot Suit
```

The customer Cables R Us has placed no orders, and order number 9005 has no line items.

The following outer join returns all customers and the dates they placed orders. The (+) operator ensures that customers who placed no orders are also returned:

```
SELECT custname, TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE"
    FROM customers, orders
    WHERE customers.custno = orders.custno (+);

CUSTNAME             ORDERDATE
-------------------- --------------
Angelic Co.          OCT-13-1993
Angelic Co.          OCT-20-1993
Angelic Co.          OCT-27-1993
Believable Co.       OCT-13-1993
Believable Co.       OCT-31-1993
Cables R Us
```

The following outer join builds on the result of the previous one by adding the `lineitems` table to the `FROM` clause, columns from this table to the select list, and a join condition joining this table to the `orders` table to the *`where_clause`*. This query joins the results of the previous query to the `lineitems` table and returns all customers, the dates they placed orders, and the part number and quantity of each part they ordered. The first (+) operator serves the same purpose as in the previous query. The second (+) operator ensures that orders with no line items are also returned:

```
SELECT custname,
   TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE",
   partno,
   quantity
      FROM customers, orders, lineitems
```

```
      WHERE customers.custno = orders.custno (+)
      AND orders.orderno = lineitems.orderno (+);

CUSTNAME             ORDERDATE      PARTNO     QUANTITY
-------------------- -------------- ---------- ----------
Angelic Co.          OCT-13-1993           101         15
Angelic Co.          OCT-13-1993           102         10
Angelic Co.          OCT-20-1993           101         15
Angelic Co.          OCT-27-1993           102         10
Angelic Co.          OCT-27-1993           103         20
Believable Co.       OCT-13-1993           101         25
Believable Co.       OCT-13-1993           103         50
Believable Co.       OCT-31-1993
Cables R Us
```

The following outer join builds on the result of the previous one by adding the `parts` table to the `FROM` clause, the `partname` column from this table to the select list, and a join condition joining this table to the `lineitems` table to the *`where_clause`*. This query joins the results of the previous query to the `parts` table to return all customers, the dates they placed orders, and the quantity and name of each part they ordered. The first two (+) operators serve the same purposes as in the previous query. The third (+) operator ensures that rows with `NULL` part numbers are also returned:

```
SELECT custname, TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE",
    quantity, partname
    FROM customers, orders, lineitems, parts
    WHERE customers.custno = orders.custno (+)
    AND orders.orderno = lineitems.orderno (+)
    AND lineitems.partno = parts.partno (+);

CUSTNAME             ORDERDATE        QUANTITY PARTNAME
-------------------- -------------- ---------- ------------
Angelic Co.          OCT-13-1993            15 X-Ray Screen
Angelic Co.          OCT-13-1993            10 Yellow Bag
Angelic Co.          OCT-20-1993            15 X-Ray Screen
Angelic Co.          OCT-27-1993            10 Yellow Bag
Angelic Co.          OCT-27-1993            20 Zoot Suit
Believable Co.       OCT-13-1993            25 X-Ray Screen
Believable Co.       OCT-13-1993            50 Zoot Suit
Believable Co.       OCT-31-1993
Cables R Us
```

Table Collection Examples You can perform DML operations on nested tables only if they are defined as columns of a table. Therefore, when the *query_table_expression_clause* of an INSERT, DELETE, or UPDATE statement is a *table_collection_expression*, the collection expression must be a subquery that selects the table's nested table column. The examples that follow are based on this scenario:

```
CREATE TYPE ProjectType AS OBJECT(
    pno    NUMBER,
    pname  CHAR(31),
    budget NUMBER);
CREATE TYPE ProjectSet AS TABLE OF ProjectType;

CREATE TABLE Dept (dno NUMBER, dname CHAR(31), projs ProjectSet)
    NESTED TABLE projs STORE AS
        ProjectSetTable ((Primary Key(Nested_Table_Id, pno))
ORGANIZATION
INDEX COMPRESS 1);

INSERT INTO Dept VALUES (1, 'Engineering', ProjectSet());
```

This example inserts into the 'Engineering' department's 'projs' nested table:

```
INSERT INTO TABLE(SELECT d.projs
                  FROM   Dept d
                  WHERE  d.dno = 1)
  VALUES (1, 'Collection Enhancements', 10000);
```

This example updates the 'Engineering' department's 'projs' nested table:

```
UPDATE TABLE(SELECT d.projs
             FROM   Dept d
             WHERE  d.dno = 1) p
  SET  p.budget = p.budget + 1000;
```

This example deletes from the 'Engineering' department's 'projs' nested table

```
DELETE TABLE(SELECT d.projs
             FROM   Dept d
             WHERE  d.dno = 1) p
 WHERE p.budget > 100000;
```

Collection Unnesting Examples Suppose the database contains a table hr_info with columns dept, location, and mgr, and a column of nested table type

people which has name, dept, and sal columns. You could get all the rows from hr_info and all the rows from people using the following statement:

```
SELECT t1.dept, t2.* FROM hr_info t1, TABLE(t1.people) t2
   WHERE t2.dept = t1.dept;
```

Now suppose that people is not a nested table column of hr_info, but is instead a separate table with columns name, dept, address, hiredate, and sal. You can extract the same rows as in the preceding example with this statement:

```
SELECT t1.department, t2.*
   FROM hr_info t1, TABLE(CAST(MULTISET(
      SELECT t3.name, t3.dept, t3.sal FROM people t3
      WHERE t3.dept = t1.dept)
      AS NESTED_PEOPLE)) t2;
```

Finally, suppose that people is neither a nested table column of table hr_info nor a table itself. Instead, you have created a function people_func that extracts from various sources the name, department, and salary of all employees. You can get the same information as in the preceding examples with the following query:

```
SELECT t1.dept, t2.* FROM HY_INFO t1, TABLE(CAST
   (people_func( ... ) AS NESTED_PEOPLE)) t2;
```

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* for more examples of collection unnesting.

LEVEL Examples The following statement returns all employees in hierarchical order. The root row is defined to be the employee whose job is 'PRESIDENT'. The child rows of a parent row are defined to be those who have the employee number of the parent row as their manager number.

```
SELECT LPAD(' ',2*(LEVEL-1)) || ename org_chart,
        empno, mgr, job
    FROM emp
    START WITH job = 'PRESIDENT'
    CONNECT BY PRIOR empno = mgr;

ORG_CHART    EMPNO      MGR        JOB
------------ ---------- ---------- ---------
KING               7839            PRESIDENT
  JONES            7566       7839 MANAGER
    SCOTT          7788       7566 ANALYST
      ADAMS        7876       7788 CLERK
    FORD           7902       7566 ANALYST
```

```
      SMITH        7369       7902 CLERK
  BLAKE            7698       7839 MANAGER
    ALLEN          7499       7698 SALESMAN
    WARD           7521       7698 SALESMAN
    MARTIN         7654       7698 SALESMAN
    TURNER         7844       7698 SALESMAN
      JAMES        7900       7698 CLERK
 CLARK             7782       7839 MANAGER
   MILLER          7934       7782 CLERK
```

The following statement is similar to the previous one, except that it does not select employees with the job 'ANALYST'.

```
SELECT LPAD(' ', 2*(LEVEL-1)) || ename org_chart,
     empno, mgr, job
     FROM emp
     WHERE job != 'ANALYST'
     START WITH job = 'PRESIDENT'
     CONNECT BY PRIOR empno = mgr;

ORG_CHART            EMPNO      MGR         JOB
-------------------- ---------- ---------- ---------
KING                       7839            PRESIDENT
  JONES                    7566       7839 MANAGER
      ADAMS                7876       7788 CLERK
      SMITH                7369       7902 CLERK
  BLAKE                    7698       7839 MANAGER
    ALLEN                  7499       7698 SALESMAN
    WARD                   7521       7698 SALESMAN
    MARTIN                 7654       7698 SALESMAN
    TURNER                 7844       7698 SALESMAN
    JAMES                  7900       7698 CLERK
  CLARK                    7782       7839 MANAGER
    MILLER                 7934       7782 CLERK
```

Oracle does not return the analysts `scott` and `ford`, although it does return employees who are managed by `scott` and `ford`.

The following statement is similar to the first one, except that it uses the LEVEL pseudocolumn to select only the first two levels of the management hierarchy:

```
SELECT LPAD(' ',2*(LEVEL-1)) || ename org_chart,
empno, mgr, job
    FROM emp
    START WITH job = 'PRESIDENT'
    CONNECT BY PRIOR empno = mgr AND LEVEL <= 2;
```

```
ORG_CHART    EMPNO      MGR        JOB
------------ ---------- ---------- ---------
KING               7839            PRESIDENT
  JONES            7566       7839 MANAGER
  BLAKE            7698       7839 MANAGER
  CLARK            7782       7839 MANAGER
```

Distributed Query Example This example shows a query that joins the dept table on the local database with the emp table on the houston database:

```
SELECT ename, dname
    FROM emp@houston, dept
    WHERE emp.deptno = dept.deptno;
```

Correlated Subquery Examples The following examples show the general syntax of a correlated subquery:

```
SELECT select_list
    FROM table1 t_alias1
    WHERE expr operator
        (SELECT column_list
            FROM table2 t_alias2
            WHERE t_alias1.column
               operator t_alias2.column);
UPDATE table1 t_alias1
    SET column =
        (SELECT expr
            FROM table2 t_alias2
            WHERE t_alias1.column = t_alias2.column);
DELETE FROM table1 t_alias1
    WHERE column operator
        (SELECT expr
            FROM table2 t_alias2
            WHERE t_alias1.column = t_alias2.column);
```

The following statement returns data about employees whose salaries exceed their department average. The following statement assigns an alias to emp, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT deptno, ename, sal
    FROM emp x
    WHERE sal > (SELECT AVG(sal)
          FROM emp
          WHERE x.deptno = deptno)
    ORDER BY deptno;
```

For each row of the emp table, the parent query uses the correlated subquery to compute the average salary for members of the same department. The correlated subquery performs the following steps for each row of the emp table:

1. The deptno of the row is determined.
2. The deptno is then used to evaluate the parent query.
3. If that row's salary is greater than the average salary for that row's department, then the row is returned.

The subquery is evaluated once for each row of the emp table.

DUAL Table Example The following statement returns the current date:

```
SELECT SYSDATE FROM DUAL;
```

You could select SYSDATE from the emp table, but Oracle would return 14 rows of the same SYSDATE, one for every row of the emp table. Selecting from DUAL is more convenient.

Sequence Examples The following statement increments the zseq sequence and returns the new value:

```
SELECT zseq.nextval
    FROM dual;
```

The following statement selects the current value of zseq:

```
SELECT zseq.currval
    FROM dual;
```

SET CONSTRAINT[S]

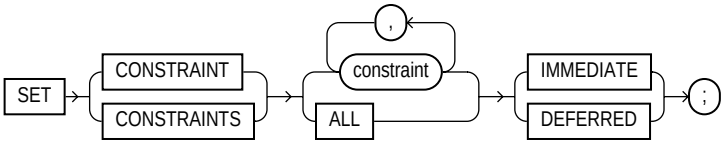
Purpose

Use the SET CONSTRAINTS statement to specify, for a particular transaction, whether a deferrable constraint is checked following each DML statement or when the transaction is committed.

Prerequisites

To specify when a deferrable constraint is checked, you must have SELECT privilege on the table to which the constraint is applied unless the table is in your schema.

Syntax



Keywords and Parameters

constraint

Specify the name of one or more integrity constraints.

ALL

Specify ALL to set all deferrable constraints for this transaction.

IMMEDIATE

Specify IMMEDIATE to indicate that the conditions specified by the deferrable constraint are checked immediately after each DML statement.

DEFERRED

Specify DEFERRED to indicate that the conditions specified by the deferrable constraint are checked when the transaction is committed.

Note: You can verify the success of deferrable constraints prior to committing them by issuing a SET CONSTRAINTS ALL IMMEDIATE statement.

Examples

Setting Constraints Examples The following statement sets all deferrable constraints in this transaction to be checked immediately following each DML statement:

```
SET CONSTRAINTS ALL IMMEDIATE;
```

The following statement checks three deferred constraints when the transaction is committed:

```
SET CONSTRAINTS unq_name, scott.nn_sal,
  adams.pk_dept@dblink DEFERRED;
```

SET ROLE

Purpose

Use the SET ROLE statement to enable and disable roles for your current session.

When a user logs on, Oracle enables all privileges granted explicitly to the user and all privileges in the user's default roles. During the session, the user or an application can use the SET ROLE statement any number of times to change the roles currently enabled for the session. The number of roles that can be concurrently enabled is limited by the initialization parameter MAX_ENABLED_ROLES.

You can see which roles are currently enabled by examining the SESSION_ROLES data dictionary view.

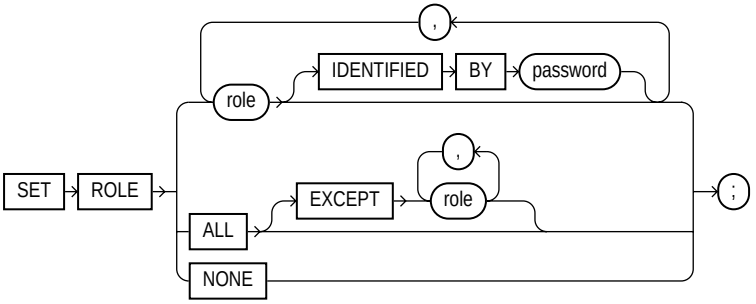
See Also:

- CREATE ROLE on page 9-146 for information on creating roles
- ALTER USER on page 8-88 for information on changing a user's default roles

Prerequisites

You must already have been granted the roles that you name in the SET ROLE statement.

Syntax



Keywords and Parameters

role

Specify a role to be enabled for the current session. Any roles not listed are disabled for the current session.

Restriction: You cannot specify a role unless it was granted to you either directly or through other roles.

IDENTIFIED BY *password*	Specify the password for a role. If the role has a password, you must specify the password to enable the role.

ALL

Specify ALL to enable all roles granted to you for the current session except those optionally listed in the EXCEPT clause.

Restriction: You cannot use this clause to enable roles with passwords that have been granted directly to you.

EXCEPT	Roles listed in the EXCEPT clause must be roles granted directly to you. They cannot be roles granted to you through other roles. If you list a role in the EXCEPT clause that has been granted to you both directly and through another role, the role remains enabled by virtue of the role to which it has been granted.

NONE

Specify NONE to disable all roles for the current session, including the DEFAULT role.

Examples

Setting Roles Examples To enable the role gardener identified by the password marigolds for your current session, issue the following statement:

```
SET ROLE gardener IDENTIFIED BY marigolds;
```

To enable all roles granted to you for the current session, issue the following statement:

```
SET ROLE ALL;
```

To enable all roles granted to you except banker, issue the following statement:

```
SET ROLE ALL EXCEPT banker;
```

To disable all roles granted to you for the current session, issue the following statement:

```
SET ROLE NONE;
```

SET TRANSACTION

Purpose

Use the SET TRANSACTION statement to establish the current transaction as read only or read write, establish its isolation level, or assign it to a specified rollback segment.

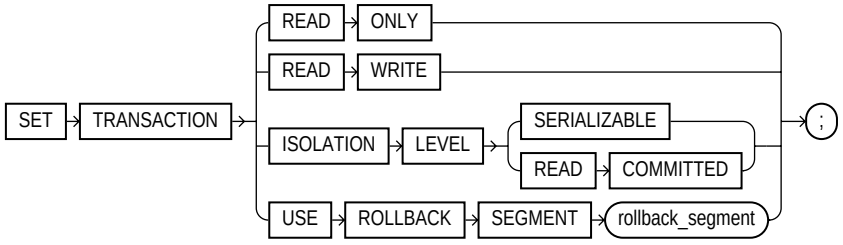
The operations performed by a SET TRANSACTION statement affect only your current transaction, not other users or other transactions. Your transaction ends whenever you issue a COMMIT or ROLLBACK statement. Oracle implicitly commits the current transaction before and after executing a data definition language (DDL) statement.

See Also: COMMIT on page 8-133 and ROLLBACK on page 11-83

Prerequisites

If you use a SET TRANSACTION statement, it must be the first statement in your transaction. However, a transaction need not have a SET TRANSACTION statement.

Syntax



Keywords and Parameters

READ ONLY

The READ ONLY clause establishes the current transaction as a read-only transaction. This clause established **transaction-level read consistency**.

All subsequent queries in that transaction only see changes committed before the transaction began. Read-only transactions are useful for reports that run multiple queries against one or more tables while other users update these same tables.

> **Note:** This clause is not supported for the user SYS. That is, queries by SYS will return changes made during the transaction even if SYS has set the transaction to be READ ONLY.

Restriction: Only the following statements are permitted in a read-only transaction:

- Subqueries (that is, SELECT statements without the *for_update_clause*)
- LOCK TABLE
- SET ROLE
- ALTER SESSION
- ALTER SYSTEM

> **See Also:** *Oracle8i Concepts*

READ WRITE

Specify READ WRITE to establish the current transaction as a read-write transaction. This clause establishes **statement-level read consistency**, which is the default.

Restriction: You cannot toggle between transaction-level and statement-level read consistency in the same transaction.

ISOLATION LEVEL

Use the ISOLATION LEVEL clause to specify how transactions containing database modifications are handled.

SERIALIZABLE The SERIALIAZBLE setting specifies serializable transaction isolation mode as defined in SQL92. If a serializable transaction contains data manipulation language (DML) that attempts to update any resource that may have been updated in a transaction uncommitted at the start of the serializable transaction, then the DML statement fails.

> **Note:** The COMPATIBLE initialization parameter must be set to 7.3.0 or higher for SERIALIZABLE mode to work.

READ COMMITTED — The READ COMMITTED setting is the default Oracle transaction behavior. If the transaction contains DML that requires row locks held by another transaction, then the DML statement waits until the row locks are released.

USE ROLLBACK SEGMENT

Specify USE ROLLBACK SEGMENT to assign the current transaction to the specified rollback segment. This clause also implicitly establishes the transaction as a read-write transaction.

This clause lets you to assign transactions of different types to rollback segments of different sizes. For example:

- If no long-running queries are concurrently reading the same tables, you can assign small transactions to small rollback segments, which are more likely to remain in memory.
- You can assign transactions that modify tables that are concurrently being read by long-running queries to large rollback segments, so that the rollback information needed for the read-consistent queries is not overwritten.
- You can assign transactions that insert, update, or delete large amounts of data to rollback segments large enough to hold the rollback information for the transaction.

You cannot use the READ ONLY clause and the USE ROLLBACK SEGMENT clause in a single SET TRANSACTION statement or in different statements in the same transaction. Read-only transactions do not generate rollback information and therefore are not assigned rollback segments.

Examples

The following statements could be run at midnight of the last day of every month to count how many ships and containers the company owns. This report would not be affected by any other user who might be adding or removing ships and/or containers.

```
COMMIT;
SET TRANSACTION READ ONLY;
SELECT COUNT(*) FROM ship;
```

```
SELECT COUNT(*) FROM container;
COMMIT;
```

The first `COMMIT` statement ensures that `SET TRANSACTION` is the first statement in the transaction. The last `COMMIT` statement does not actually make permanent any changes to the database. It simply ends the read-only transaction.

The following statement assigns your current transaction to the rollback segment `oltp_5`:

```
SET TRANSACTION USE ROLLBACK SEGMENT oltp_5;
```

storage_clause

Purpose

Use the *storage_clause* to specify storage characteristics for any of the following schema objects:

- clusters
- indexes
- rollback segments
- materialized views
- materialized view logs
- tables
- tablespaces
- partitions

Storage parameters affect both how long it takes to access data stored in the database and how efficiently space in the database is used. For a discussion of the effects of these parameters, see *Oracle8i Performance Guide and Reference*.

When you create a tablespace, you can specify values for the storage parameters. These values serve as default values for segments allocated in the tablespace.

When you alter a tablespace, you can change the values of storage parameters. The new values serve as default values only for subsequently allocated segments (or subsequently created objects).

Note: The *storage_clause* is interpreted differently for locally managed tablespaces. At creation, Oracle ignores MAXEXTENTS and uses the remaining parameter values to calculate the initial size of the segment. For more information, see CREATE TABLESPACE on page 10-56.

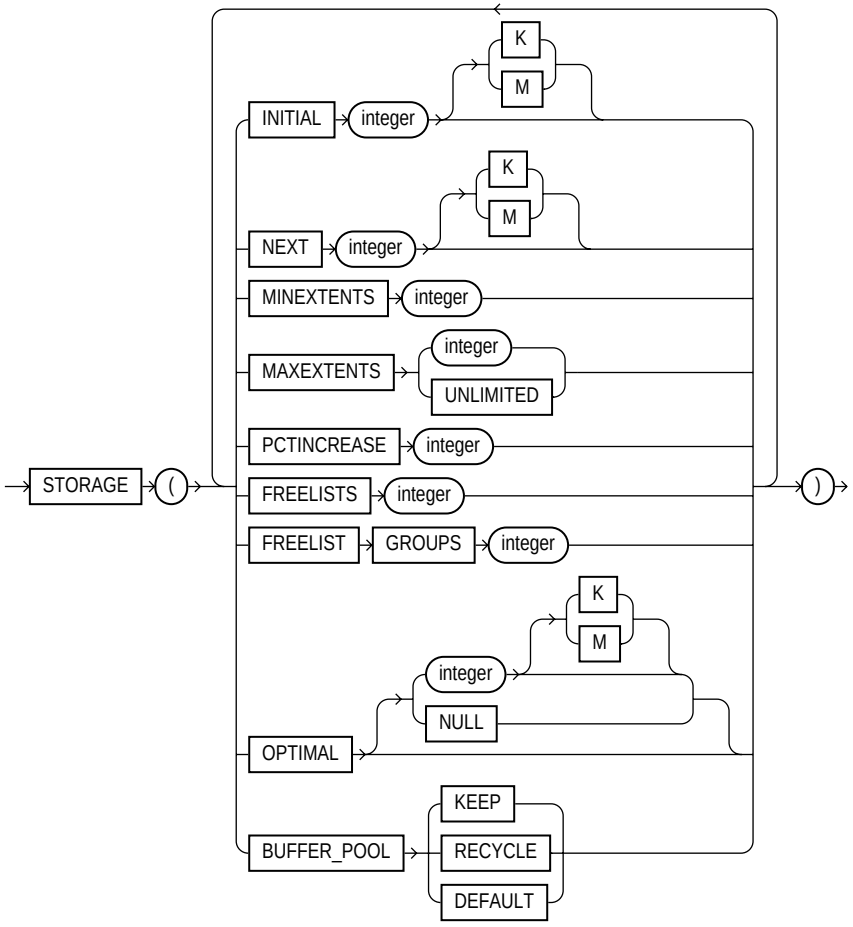
When you create a cluster, index, rollback segment, snapshot, snapshot log, table, or partition, you can specify values for the storage parameters for the segments allocated to these objects. If you omit any storage parameter, Oracle uses the value of that parameter specified for the tablespace.

When you alter a cluster, index, rollback segment, snapshot, snapshot log, table, or partition, you can change the values of storage parameters. The new values affect only future extent allocations.

Prerequisites

To change the value of a STORAGE parameter, you must have the privileges necessary to use the appropriate CREATE or ALTER statement.

Syntax



Keywords and Parameters

INITIAL

Specify in bytes the size of the object's first extent. Oracle allocates space for this extent when you create the schema object. Use K or M to specify this size in kilobytes or megabytes.

The default value is the size of 5 data blocks. The minimum value is the size of 2 data blocks for nonbitmapped segments or 3 data blocks for bitmapped segments, plus one data block for each free list group you specify. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size for values less than 5 data blocks, and rounds up to the next multiple of 5 data blocks for values greater than 5 data blocks.

Restriction: You cannot specify INITIAL in an ALTER statement.

> **See Also:** FREELIST GROUPS on page 11-133 for information on freelist groups

NEXT

Specify in bytes the size of the next extent to be allocated to the object. Use K or M to specify the size in kilobytes or megabytes. The default value is the size of 5 data blocks. The minimum value is the size of 1 data block. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size for values less than 5 data blocks. For values greater than 5 data blocks, Oracle rounds up to a value that minimizes fragmentation, as described in .

If you change the value of the NEXT parameter (that is, if you specify it in an ALTER statement), the next allocated extent will have the specified size, regardless of the size of the most recently allocated extent and the value of the PCTINCREASE parameter.

> **See Also:** *Oracle8i Concepts* for information on how Oracle minimizes fragmentation

PCTINCREASE

Specify the percent by which the third and subsequent extents grow over the preceding extent. The default value is 50, meaning that each subsequent extent is 50% larger than the preceding extent. The minimum value is 0, meaning all extents after the first are the same size. The maximum value depends on your operating system.

Oracle rounds the calculated size of each new extent to the nearest multiple of the data block size.

If you change the value of the PCTINCREASE parameter (that is, if you specify it in an ALTER statement), Oracle calculates the size of the next extent using this new value and the size of the most recently allocated extent.

> **Suggestion:** If you wish to keep all extents the same size, you can prevent SMON from coalescing extents by setting the value of PCTINCREASE to 0. In general, Oracle Corporation recommends a setting of 0 as a way to minimize fragmentation and avoid the possibility of very large temporary segments during processing.

Restriction: You cannot specify PCTINCREASE for rollback segments. Rollback segments always have a PCTINCREASE value of 0.

MINEXTENTS

Specify the total number of extents to allocate when the object is created. This parameter enables you to allocate a large amount of space when you create an object, even if the space available is not contiguous. The default and minimum value is 1, meaning that Oracle allocates only the initial extent, except for rollback segments, for which the default and minimum value is 2. The maximum value depends on your operating system.

If the MINEXTENTS value is greater than 1, then Oracle calculates the size of subsequent extents based on the values of the INITIAL, NEXT, and PCTINCREASE parameters.

Restriction: You cannot specify MINEXTENTS in an ALTER statement.

MAXEXTENTS

Specify the total number of extents, including the first, that Oracle can allocate for the object. The minimum value is 1 (except for rollback segments, which always have a minimum value of 2). The default value depends on your data block size.

UNLIMITED Specify UNLIMITED if you want extents to be allocated automatically as needed. Oracle Corporation recommends this setting as a way to minimize fragmentation.

However, do not use this clause for rollback segments. Rogue transactions containing inserts, updates, or deletes that continue for a long time will continue to create new extents until a disk is full.

Caution: A rollback segment that you create without specifying the *storage_clause* has the same storage parameters as the tablespace that the rollback segment is created in. Thus, if you create the tablespace with MAXEXTENTS UNLIMITED, then the rollback segment will also have the same default.

FREELIST GROUPS

Specify the number of groups of free lists for the database object you are creating. The default and minimum value for this parameter is 1. Oracle uses the instance number of Oracle Parallel Server instances to map each instance to one free list group.

Each free list group uses one database block. Therefore:

- If you do not specify a large enough value for INITIAL to cover the minimum value plus one data block for each free list group, Oracle increases the value of INITIAL the necessary amount.
- If you are creating an object in a uniform locally managed tabledspace, and the extent size is not large enough to acommodate the number of freelist groups, the create operation will fail.

Restriction: You can specify the FREELIST GROUPS parameter only in CREATE TABLE, CREATE CLUSTER, CREATE MATERIALIZED VIEW, CREATE MATERIALIZED VIEW LOG, and CREATE INDEX statements.

See Also: *Oracle8i Parallel Server Concepts*

FREELISTS

For objects other than tablespaces, specify the number of free lists for each of the free list groups for the table, partition, cluster, or index. The default and minimum value for this parameter is 1, meaning that each free list group contains one free list.

The maximum value of this parameter depends on the data block size. If you specify a FREELISTS value that is too large, Oracle returns an error indicating the maximum value.

Restriction: You can specify FREELISTS in the *storage_clause* of any statement except when creating or altering a tablespace or rollback segment.

OPTIMAL

The OPTIMAL keyword is relevant only to rollback segments. It specifies an optimal size in bytes for a rollback segment. Use K or M to specify this size in kilobytes or megabytes. Oracle tries to maintain this size for the rollback segment by dynamically deallocating extents when their data is no longer needed for active transactions. Oracle deallocates as many extents as possible without reducing the total size of the rollback segment below the OPTIMAL value.

NULL — Specify NULL for no optimal size for the rollback segment, meaning that Oracle never deallocates the rollback segment's extents. This is the default behavior.

The value of OPTIMAL cannot be less than the space initially allocated for the rollback segment specified by the MINEXTENTS, INITIAL, NEXT, and PCTINCREASE parameters. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size.

BUFFER_POOL

The BUFFER_POOL clause lets you specify a default buffer pool (cache) for a schema object. All blocks for the object are stored in the specified cache. If a buffer pool is defined for a partitioned table or index, then the partitions inherit the buffer pool from the table or index definition, unless overridden by a partition-level definition.

Note: BUFFER_POOL is not a valid clause for creating or altering tablespaces or rollback segments.

KEEP — Specify KEEP to retain the schema object in memory to avoid I/O operations. KEEP takes precedence over any NOCACHE clause you specify for a table, cluster, materialized view, or materialized view log.

RECYCLE — Specify RECYCLE to eliminate blocks from memory as soon as they are no longer needed, thus preventing an object from taking up unnecessary cache space.

DEFAULT — Specify DEFAULT to indicate the default buffer pool. This is the default for objects not assigned to KEEP or RECYCLE.

> **See Also:** *Oracle8i Performance Guide and Reference* for more information about using multiple buffer pools

Examples

Creating a table with storage attributes The following statement creates a table and provides storage parameter values:

```
CREATE TABLE dept
    (deptno     NUMBER(2),
     dname      VARCHAR2(14),
     loc        VARCHAR2(13) )
     STORAGE  ( INITIAL 100K  NEXT     50K
                MINEXTENTS 1  MAXEXTENTS 50  PCTINCREASE 5);
```

Oracle allocates space for the table based on the STORAGE parameter values as follows:

- The MINEXTENTS value is 1, so Oracle allocates 1 extent for the table upon creation.
- The INITIAL value is 100K, so the first extent's size is 100 kilobytes.
- If the table data grows to exceed the first extent, Oracle allocates a second extent. The NEXT value is 50K, so the second extent's size would be 50 kilobytes.
- If the table data subsequently grows to exceed the first two extents, Oracle allocates a third extent. The PCTINCREASE value is 5, so the calculated size of the third extent is 5% larger than the second extent, or 52.5 kilobytes. If the data block size is 2 kilobytes, Oracle rounds this value to 52 kilobytes.

  If the table data continues to grow, Oracle allocates more extents, each 5% larger than the previous one.
- The MAXEXTENTS value is 50, so Oracle can allocate as many as 50 extents for the table.

Creating a rollback segment with storage attributes The following statement creates a rollback segment and provides storage parameter values:

```
CREATE ROLLBACK SEGMENT rsone
    STORAGE ( INITIAL  10K  NEXT 10K
              MINEXTENTS 2  MAXEXTENTS 25
               OPTIMAL 50K );
```

Oracle allocates space for the rollback segment based on the STORAGE parameter values as follows:

- The MINEXTENTS value is 2, so Oracle allocates 2 extents for the rollback segment upon creation.
- The INITIAL value is 10K, so the first extent's size is 10 kilobytes.
- The NEXT value is 10K, so the second extent's size is 10 kilobytes.
- If the rollback data exceeds the first two extents, Oracle allocates a third extent. The PCTINCREASE value for rollback segments is always 0, so the third and subsequent extents are the same size as the second extent, 10 kilobytes.
- The MAXEXTENTS value is 25, so Oracle can allocate as many as 25 extents for the rollback segment.
- The OPTIMAL value is 50K, so Oracle deallocates extents if the rollback segment exceeds 50 kilobytes. Oracle deallocates only extents that contain data for transactions that are no longer active.

TRUNCATE

Caution: You cannot roll back a TRUNCATE statement.

Purpose

Use the TRUNCATE statement to remove all rows from a table or cluster and reset the STORAGE parameters to the values when the table or cluster was created.

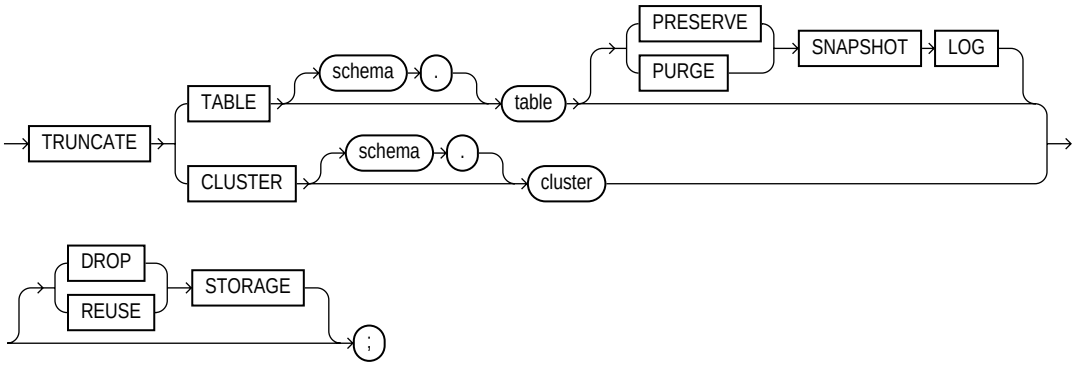
Deleting rows with the TRUNCATE statement can be more efficient than dropping and re-creating a table. Dropping and re-creating a table invalidates the table's dependent objects, requires you to regrant object privileges on the table, and requires you to re-create the table's indexes, integrity constraint, and triggers and respecify its storage parameters. Truncating has none of these effects.

> **See Also:** DELETE on page 10-115, DROP CLUSTER on page 10-126, and DROP TABLE on page 11-7

Prerequisites

To truncate a table or cluster, the table or cluster must be in your schema or you must have DROP ANY TABLE system privilege.

Syntax



Keywords and Parameters

TABLE

Specify the schema and name of the table to be truncated. This table cannot be part of a cluster. If you omit *schema*, Oracle assumes the table is in your own cluster.

- You can truncate index-organized tables and temporary tables. When you truncate a temporary table, only the rows created during the current session are truncated.
- The table's storage parameter NEXT is changed to be the size of the last extent deleted from the segment in the process of truncation.
- Oracle also automatically truncates and resets any existing UNUSABLE indicators for the following indexes on *table*: range and hash partitions of local indexes and subpartitions of local indexes.
- If *table* is not empty, Oracle marks UNUSABLE all nonpartitioned indexes and all partitions of global partitioned indexes on the table.
- For a domain index, this statement invokes the appropriate truncate routine to truncate the domain index data.

  See Also: *Oracle8i Data Cartridge Developer's Guide*

- If *table* (whether it is a regular or index-organized table) contains LOB columns, all LOB data and LOB index segments will be truncated.
- If *table* is partitioned, all partitions or subpartitions, as well as the LOB data and LOB index segments for each partition or subpartition, will be truncated.

Note: When you truncate a table, Oracle automatically deletes all data in the table's indexes and any materialized view direct-load INSERT information held in association with the table. (This information is independent of any materialized view / snapshot log.) If this direct-load INSERT information is deleted, an incremental refresh of the materialized view may lose data.

Restrictions:

- You cannot individually truncate a table that is part of a cluster. You must either truncate the cluster, delete all rows from the table, or drop and re-create the table.

- You cannot truncate the parent table of an enabled referential integrity constraint. You must disable the constraint before truncating the table. (An exception is that you may truncate the table if the integrity constraint is self-referential.)
- You cannot truncate a table if any domain indexes defined on any of its columns are marked LOADING or FAILED.

SNAPSHOT LOG

The SNAPSHOT LOG clause lets you specify whether a snapshot log defined on the table is to be preserved or purged when the table is truncated. This clause allows snapshot master tables to be reorganized through export/import without affecting the ability of primary-key snapshots defined on the master to be fast refreshed. To support continued fast refresh of primary-key snapshots, the snapshot log must record primary-key information.

PRESERVE — Specify PRESERVE if any snapshot log should be preserved when the master table is truncated. This is the default.

PURGE — Specify PURGE if any snapshot log should be purged when the master table is truncated.

> **See Also:** *Oracle8i Replication* for more information about snapshot logs and the TRUNCATE statement

CLUSTER

Specify the schema and name of the cluster to be truncated. You can truncate only an indexed cluster, not a hash cluster. If you omit *schema*, Oracle assumes the table is in your own cluster.

When you truncate a cluster, Oracle also automatically deletes all data in the cluster's tables' indexes.

STORAGE Clauses

DROP STORAGE — Specify DROP STORAGE to deallocate all space from the deleted rows from the table or cluster except the space allocated by the table's or cluster's MINEXTENTS parameter. This space can subsequently be used by other objects in the tablespace. This is the default.

REUSE STORAGE — Specify REUSE STORAGE to retain the space from the deleted rows allocated to the table or cluster. Storage values are not reset to the values when the table or cluster was created. This space can subsequently be used only by new data in the table or cluster resulting from inserts or updates.

> **Note:** If you have specified more than one free list for the object you are truncating, the REUSE STORAGE clause also removes any mapping of free lists to instances, and resets the high-water mark to the beginning of the first extent.

The DROP STORAGE clause and REUSE STORAGE clause also apply to the space freed by the data deleted from associated indexes.

Examples

Simple TRUNCATE Example The following statement deletes all rows from the emp table and returns the freed space to the tablespace containing emp:

```
TRUNCATE TABLE emp;
```

The above statement also deletes all data from all indexes on emp and returns the freed space to the tablespaces containing them.

Retaining free space after truncating The following statement deletes all rows from all tables in the cust cluster, but leaves the freed space allocated to the tables:

```
TRUNCATE CLUSTER cust REUSE STORAGE
```

The above statement also deletes all data from all indexes on the tables in cust.

Preserving materialized view logs after truncating The following statements are examples of truncate statements that preserve snapshot logs:

```
TRUNCATE TABLE emp PRESERVE SNAPSHOT LOG;
TRUNCATE TABLE stock;
```

UPDATE

Purpose

Use the UPDATE statement to change existing values in a table or in a view's base table.

Prerequisites

For you to update values in a table, the table must be in your own schema or you must have UPDATE privilege on the table.

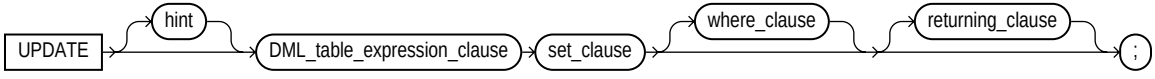
For you to update values in the base table of a view,

- You must have UPDATE privilege on the view, and
- Whoever owns the schema containing the view must have UPDATE privilege on the base table.

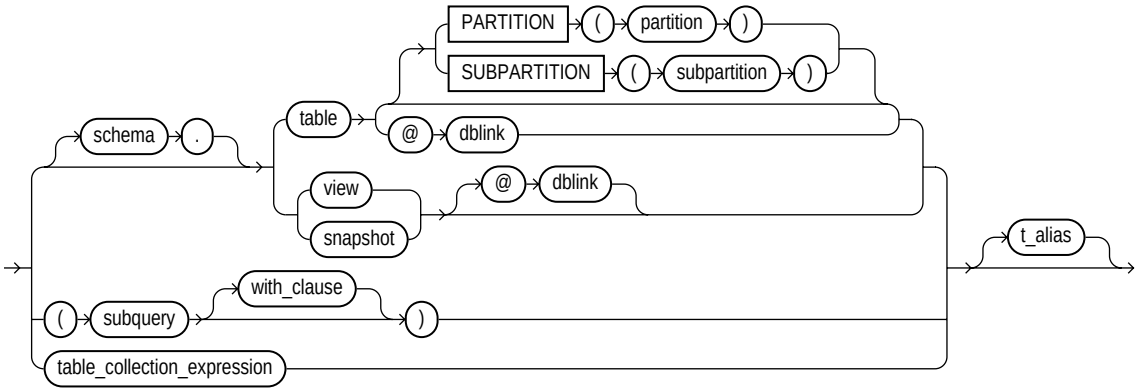
If the SQL92_SECURITY initialization parameter is set to TRUE, then you must have SELECT privilege on the table whose column values you are referencing (such as the columns in a *where_clause*) to perform an UPDATE.

The UPDATE ANY TABLE system privilege also allows you to update values in any table or any view's base table.

Syntax

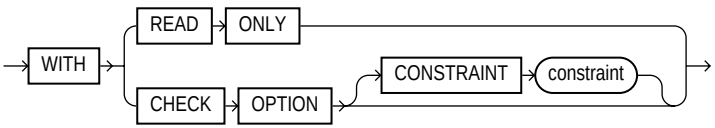


DML_table_expression_clause::=



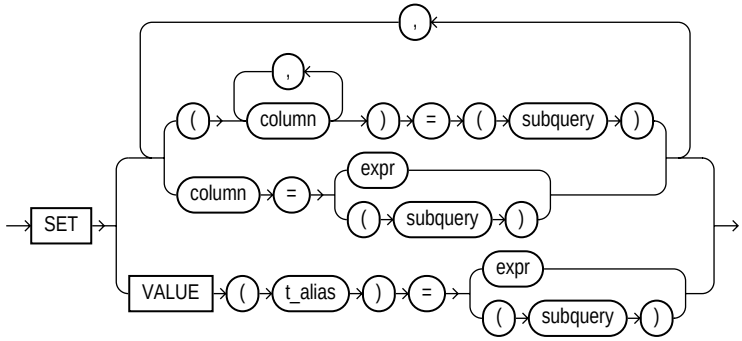
subquery: see SELECT and subquery on page 11-88.

with_clause::=

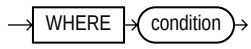


table_collection_expression::=

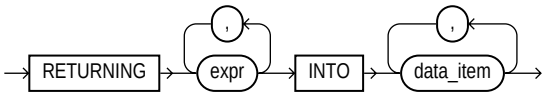
set_clause::=



where_clause::=



returning_clause::=



Keywords and Parameters

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

You can place a parallel hint immediately after the UPDATE keyword to parallelize both the underlying scan and UPDATE operations.

See Also:

- *Oracle8i Performance Guide and Reference* and "Hints" on page 2-67 for the syntax and description of hints
- *Oracle8i Performance Guide and Reference*, *Oracle8i Parallel Server Concepts*, and *Oracle8i Concepts* for detailed information about parallel DML

DML_table_expression_clause

schema — Specify the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table | *view* | *subquery* — Specify the name of the table or view, or the columns returned by a subquery, to be updated. Issuing an UPDATE statement against a table fires any UPDATE triggers associated with the table. If you specify *view*, Oracle updates the view's base table.

If *table* (or the base table of *view*) contains one or more domain index columns, this statement executes the appropriate indextype update routine.

See Also: *Oracle8i Data Cartridge Developer's Guide* for more information on these routines

PARTITION (*partition*) | SUBPARTITION (*subpartition*) — Specify the name of the partition or subpartition within *table* targeted for updates. You need not specify the partition name when updating values in a partitioned table. However in some cases specifying the partition name can be more efficient than a complicated *where_clause*.

dblink — Specify a complete or partial name of a database link to a remote database where the table or view is located. You can use a database link to update a remote table or view only if you are using Oracle's distributed functionality.

If you omit *dblink*, Oracle assumes the table or view is on the local database.

See Also: "Referring to Objects in Remote Databases" on page 2-90 for information on referring to database links

with_clause — Use the *with_clause* to restrict the subquery in one of the following ways:

- WITH READ ONLY specifies that the subquery cannot be updated.
- WITH CHECK OPTION specifies that Oracle prohibits any changes to that table that would produce rows that are not included in the subquery.

See Also: "WITH CHECK OPTION Example" on page 11-108

Restrictions on the *DML_table_expression_clause*:

- You cannot execute this statement if *table* (or the base table of *view*) contains any domain indexes marked LOADING or FAILED.
- You cannot specify the *order_by_clause* in the subquery of the *DML_query_expression_clause*.
- You cannot update a view except with INSTEAD OF triggers if the view's defining query contains one of the following constructs:
  - A set operator
  - A DISTINCT operator
  - An aggregate or analytic function
  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
  - A collection expression in a SELECT list
  - A subquery in a SELECT list
  - Joins (with some exceptions). See *Oracle8i Administrator's Guide* for details.
- If a view was created with the WITH CHECK OPTION, you can update the view only if the resulting data satisfies the view's defining query.
- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, the UPDATE statement will fail unless the SKIP_UNUSABLE_INDEXES parameter has been set to TRUE.

See Also: ALTER SESSION on page 7-105

table_collection_expression

Use the *table_collection_expression* to inform Oracle that the collection value expression should be treated as a table. You can use a *table_collection_expression* to update rows in one table based on rows from another table. For example, you could roll up four quarterly sales tables into a yearly sales table.

collection_expression Specify a subquery that selects a nested table column from *table* or *view*.

Note: In earlier releases of Oracle, *table_collection_expr* was expressed as "THE *subquery*". That usage is now deprecated.

---

t_alias

Specify a **correlation name** (alias) for the table, view, or subquery to be referenced elsewhere in the statement.

Note: This alias is **required** if the *DML_query_expression_clause* references any object type attributes or object type methods.

---

set_clause

The *set_clause* lets you set column values.

column Specify the name of a column of the table or view that is to be updated. If you omit a column of the table from the *set_clause*, that column's value remains unchanged.

Restrictions:

- If *column* refers to a LOB object attribute, you must first initialize it with a value of empty or null. You cannot update it with a literal. Also, if you are updating a LOB value using some method other than a direct UPDATE SQL statement, you must first lock the row containing the LOB.

  See Also: "LOB Locking Example" on page 11-108

- If *column* is part of the partitioning key of a partitioned table, UPDATE will fail if you change a value in the column that would move the row to a different partition or subpartition, unless you enable row movement.

  See Also: The *row_movement_clause* of CREATE TABLE on page 10-7 or ALTER TABLE on page 8-2

subquery Specify a subquery that returns exactly one row for each row updated.

- If you specify only one column in the *set_clause*, the subquery can return only one value.
- If you specify multiple columns in the *set_clause*, the subquery must return as many values as you have specified columns.

If the subquery returns no rows, then the column is assigned a null.

See Also: SELECT and subquery on page 11-88 and "Using Subqueries" on page 5-26

Note: If this *subquery* refers to remote objects, the UPDATE operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if the *subquery* in the *DML_query_expression_clause* refers to any remote objects, the UPDATE operation will run serially without notification.

See Also: *parallel_clause* in CREATE TABLE on page 10-40

expr Specify an expression that resolves to the new value assigned to the corresponding column. This expression can contain host variables and optional indicator variables.

See Also: The syntax description in "Expressions" on page 5-2

VALUE The VALUE clause lets you specify the entire row of an object table.

Restriction: You can specify this clause only for an object table.

See Also: "SET VALUE Example" on page 11-150

Note: If you insert string literals into a RAW column, during subsequent queries, Oracle will perform a full table scan rather than using any index that might exist on the RAW column.

---

where_clause

The *where_clause* lets you restrict the rows updated to those for which the specified *condition* is `true`. If you omit this clause, Oracle updates all rows in the table or view.

The *where_clause* determines the rows in which values are updated. If you do not specify the *where_clause*, all rows are updated. For each row that satisfies the *where_clause*, the columns to the left of the equals (=) operator in the *set_clause* are set to the values of the corresponding expressions on the right. The expressions are evaluated as the row is updated.

See Also: The syntax description of "Conditions" on page 5-15

returning_clause

The returning clause retrieves the rows affected by a DML (`INSERT`, `UPDATE`, or `DELETE`) statement. You can specify this clause for tables and snapshots, and for views with a single base table.

- When operating on a single row, a DML statement with a *returning_clause* can retrieve column expressions using the affected row, rowid, and `REF`s to the affected row and store them in host variables or PL/SQL variables.
- When operating on multiple rows, a DML statement with the *returning_clause* stores values from expressions, rowids, and `REF`s involving the affected rows in bind arrays.

expr	Each item in the *expr* list must be a valid expression syntax.
`INTO`	The `INTO` clause indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item* list.
data_item	Each *data_item* is a host variable or PL/SQL variable that stores the retrieved *expr* value.

For each expression in the `RETURNING` list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the `INTO` list.

Restrictions:

- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve `LONG` types with this clause.
- You cannot specify this clause for a view on which an `INSTEAD OF` trigger has been defined.

See Also: *PL/SQL User's Guide and Reference* for information on using the BULK COLLECT clause to return multiple values to collection variables

Examples

Simple Examples The following statement gives null commissions to all employees with the job `trainee`:

```
UPDATE emp
    SET comm = NULL
    WHERE job = 'TRAINEE';
```

The following statement promotes `jones` to manager of Department 20 with a $1,000 raise (assuming there is only one `jones`):

```
UPDATE emp
    SET job = 'MANAGER', sal = sal + 1000, deptno = 20
    WHERE ename = 'JONES';
```

The following statement increases the balance of bank account number 5001 in the `accounts` table on a remote database accessible through the database link `boston`:

```
UPDATE accounts@boston
    SET balance = balance + 500
    WHERE acc_no = 5001;
```

PARTITION Example The following example updates values in a single partition of the `sales` table:

```
UPDATE sales PARTITION (feb96) s
   SET s.account_name = UPPER(s.account_name);
```

Complex Example This example shows the following syntactic constructs of the UPDATE statement:

- Both forms of the *`set_clause`* together in a single statement
- A correlated subquery
- A *`where_clause`* to limit the updated rows

```
UPDATE emp a
    SET deptno =
        (SELECT deptno
            FROM dept
```

```
          WHERE loc = 'BOSTON'),
      (sal, comm) =
      (SELECT 1.1*AVG(sal), 1.5*AVG(comm)
        FROM emp b
        WHERE a.deptno = b.deptno)
  WHERE deptno IN
      (SELECT deptno
        FROM dept
        WHERE loc = 'DALLAS'
            OR loc = 'DETROIT');
```

The above UPDATE statement performs the following operations:

- Updates only those employees who work in Dallas or Detroit
- Sets deptno for these employees to the deptno of Boston
- Sets each employee's salary to 1.1 times the average salary of their department
- Sets each employee's commission to 1.5 times the average commission of their department

SET VALUE Example The following statement updates a row of object table table1 by selecting a row from another object table table2:

```
UPDATE table1 p SET VALUE(p) =
   (SELECT VALUE(q) FROM table2 q WHERE p.id = q.id)
   WHERE p.id = 10;
```

The subquery uses the value object reference function in its expression.

Correlated Update Example The following example updates particular rows of the projs nested table corresponding to the department whose department equals 123:

```
UPDATE TABLE(SELECT projs
           FROM dept d WHERE d.dno = 123)  p
  SET p.budgets = p.budgets + 1
  WHERE p.pno IN (123, 456);
```

RETURNING Clause Example The following example returns values from the updated row and stores the result in PL/SQL variables bnd1, bnd2, bnd3:

```
UPDATE emp
  SET job ='MANAGER', sal = sal + 1000, deptno = 20
  WHERE ename = 'JONES'
  RETURNING sal*0.25, ename, deptno INTO bnd1, bnd2, bnd3;
```

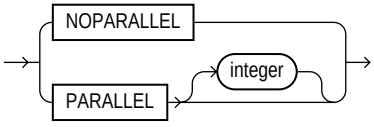
A

Syntax Diagrams

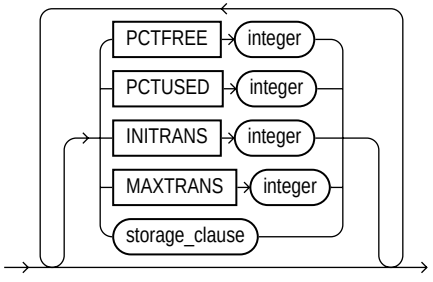
Syntax diagrams are drawings that illustrate valid SQL syntax. To read a diagram, trace it from left to right, in the direction shown by the arrows.

Commands and other keywords appear in UPPERCASE inside rectangles. Type them exactly as shown in the rectangles. Parameters appear in lowercase inside ovals. Variables are used for the parameters. Punctuation, operators, delimiters, and terminators appear inside circles.

If the syntax diagram has more than one path, you can choose any path to travel. For example, in the following syntax you can specify either NOPARALLEL or PARALLEL:



If you have the choice of more than one keyword, operator, or parameter, your options appear in a vertical list. For example, in the following syntax, you can specify one or more of the four parameters in the stack:



The following table shows parameters that appear in the syntax diagrams and provides examples of the values you might substitute for them in your statements:

Parameter	Description	Examples
table	The substitution value must be the name of an object of the type specified by the parameter. For a list of all types of objects, see the section, "Schema Objects" on page 2-79.	`emp`
c	The substitution value must be a single character from your database character set.	`T` `s`
'text'	The substitution value must be a text string in single quotes. See the syntax description of *'text'* in "Text" on page 2-33.	`'Employee records'`
char	The substitution value must be an expression of datatype CHAR or VARCHAR2 or a character literal in single quotes.	`ename` `'Smith'`
condition	The substitution value must be a condition that evaluates to TRUE or FALSE. See the syntax description of *condition* in "Conditions" on page 5-15.	`ename >'A'`

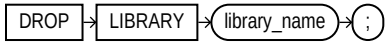
date
d
	The substitution value must be a date constant or an expression of DATE datatype.	`TO_DATE(`
		`'01-Jan-1994',`
		`'DD-MON-YYYY')`

expr	The substitution value can be an expression of any datatype as defined in the syntax description of *expr* in "Expressions" on page 5-2.	`sal + 1000`
integer	The substitution value must be an integer as defined by the syntax description of integer in "Integer" on page 2-34.	`72`
number *m* *n*	The substitution value must be an expression of NUMBER datatype or a number constant as defined in the syntax description of *number* in "Number" on page 2-35.	`AVG(sal)` `15 * 7`
raw	The substitution value must be an expression of datatype RAW.	`HEXTORAW('7D')`
subquery	The substitution value must be a SELECT statement that will be used in another SQL statement. See SELECT and subquery on page 11-88.	`SELECT ename` `FROM emp`

Parameter	Description	Examples
db_name	The substitution value must be the name of a nondefault database in an embedded SQL program.	`sales_db`
db_string	The substitution value must be the database identification string for a Net8 database connection. For details, see the user's guide for your specific Net8 protocol.	

Required Keywords and Parameters

Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the *main path*, that is, on the horizontal line you are currently traveling. In the following example, *library_name* is a required parameter:

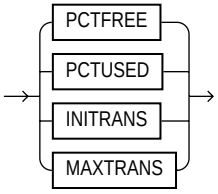


If there is a library named `HQ_LIB`, then, according to the diagram, the following statement is valid:

```
DROP LIBRARY hq_lib;
```

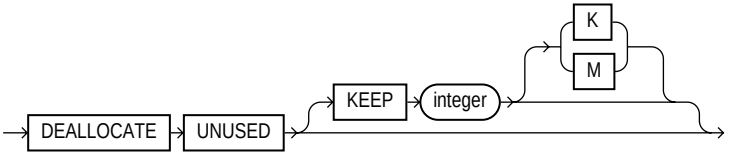
If multiple keywords or parameters appear in a vertical list that intersects the main path, one of them is required. That is, you must choose one of the keywords or

parameters, but not necessarily the one that appears on the main path. In the following example, you must choose one of the four settings:



Optional Keywords and Parameters

If keywords and parameters appear in a vertical list *above* the main path, they are optional. In the following example, instead of traveling down a vertical line, you can continue along the main path:

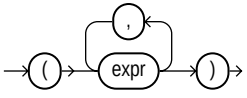


According to the diagram, all of the following statements are valid:

```
DEALLOCATE UNUSED;
DEALLOCATE UNUSED KEEP 1000;
DEALLOCATE UNUSED KEEP 10M;
```

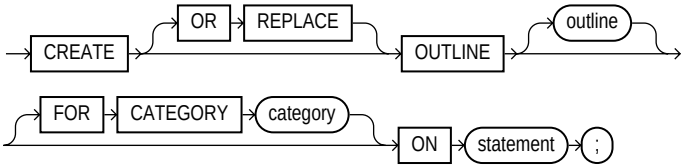
Syntax Loops

Loops let you repeat the syntax within them as many times as you like. In the following example, after choosing one expression, you can go back repeatedly to choose another, separated by commas.



Multipart Diagrams

Read a multipart diagram as if all the main paths were joined end to end. The following example is a two-part diagram:



According to the diagram, the following statement is valid:

```
CREATE OUTLINE ON UPDATE;
```

Database Objects

The names of Oracle identifiers, such as tables and columns, must not exceed 30 characters in length. The first character must be a letter, but the rest can be any combination of letters, numerals, dollar signs ($), pound signs (#), and underscores (_).

However, if an Oracle identifier is enclosed by double quotation marks ("), it can contain any combination of legal characters, including spaces but excluding quotation marks.

Oracle identifiers are not case-sensitive except when enclosed by double quotation marks.

For more information, see "Schema Object Naming Rules" on page 2-83.

B

Oracle and Standard SQL

This appendix discusses Oracle's conformance to the SQL-92 standards established by industry standards governing bodies. We are assessing the changes that will be required to this appendix in light of the new SQL-99 standards. The appendix also described how to locate extensions to standard SQL with the FIPS Flagger.

Conformance with Standard SQL

This section declares Oracle's conformance to the SQL standards established by these organizations:

- American National Standards Institute (ANSI)
- International Standards Organization (ISO)
- United States Federal Government

ANSI and ISO Compliance

Oracle8*i* complies at the Entry level as defined in the ANSI document, X3.135-1992, "Database Language SQL." You can obtain a copy of the ANSI standard from this address:

American National Standards Institute
1430 Broadway
New York, NY 10018 USA

The ANSI and ISO SQL standards require conformance claims to state the type of conformance and the implemented facilities. The Oracle server, Oracle Precompilers for C/C++ Release 8.1, Oracle Precompiler for Cobol Release 8.1, and SQL*Module for ADA Release 8.0.4 provide conformance with the ANSI X3.135-1992/ISO 9075-1992 standard:

- Compliance at Entry Level (including both SQL-DDL and SQL-DML)
- Module Language for ADA
- Embedded SQL C
- Embedded SQL COBOL

In addition to full compliance at the Entry level, Oracle complies partially at the Transitional, Intermediate, and Full levels as described in Table B–1 (including both SQL-DDL and SQL-DML).

Table B–1 Oracle Compliance at Transitional, Intermediate, and Full Levels

Level	SQL92 Feature (number and name)
Transitional	7. `TRIM` function
	8. `UNION` in views
	9. Implicit numeric casting
	10. Implicit character casting
	13. Grouped operations
	14. Qualified * in `SELECT` list
	15. Lowercase identifiers
	16. `PRIMARY KEY` enhancement
	18. Multiple module support
	21. `INSERT` expressions
Intermediate	31. Schema definition statement
	42. National character
	48. Expanded null predicate
Full	60. Trailing underscore
	62. Referential name order

FIPS Compliance

Oracle complies completely with FIPS PUB 127-2 for Entry SQL. In addition, the following information is provided for Section 16, “Special Procurement Considerations.”

Section 16.2 Programming Language Interfaces

The Oracle precompilers support the use of embedded SQL in C and COBOL. SQL*Module supports the use of Module Language in ADA.

Section 16.3 Style of Language Interface

Oracle with SQL*Module supports Module Language for Ada. Oracle with the Oracle precompilers supports C and COBOL. The specific languages supported depend on your operating system.

Section 16.5 Interactive Direct SQL

Oracle8*i* with SQL*Plus Version 3.1 (as well as other Oracle tools) supports "direct invocation" of the following SQL statements, meeting the requirements of FIPS PUB 127-2:

- `CREATE TABLE` statement
- `CREATE VIEW` statement
- `GRANT` statement
- `INSERT` statement
- `SELECT` statement, with `ORDER BY` clause but not `INTO` clause
- `UPDATE` statement: searched
- `DELETE` statement: searched
- `COMMIT WORK` statement
- `ROLLBACK WORK` statement

Most other SQL statements described in this reference are also supported interactively.

Section 16.6 Sizing for Database Constructs

Table B–2 lists requirements identified in FIPS PUB 127-1 and how they are met by Oracle8*i*.

Table B–2 Sizing for Database Constructs

Database Constructs	FIPS	Oracle8*i*
Length of an identifier (in bytes)	18	30
Length of `CHARACTER` datatype (in bytes)	240	2000

Table B–2 (Cont.) Sizing for Database Constructs

Decimal precision of `NUMERIC` datatype	15	38
Decimal precision of `DECIMAL` datatype	15	38
Decimal precision of `INTEGER` datatype	9	38
Decimal precision of `SMALLINT` datatype	4	38
Binary precision of `FLOAT` datatype	20	126
Binary precision of `REAL` datatype	20	63
Binary precision of `DOUBLE PRECISION` datatype	30	126
Columns in a table	100	1000
Values in an `INSERT` statement	100	1000
SET clauses in an `UPDATE` statement(a)	20	1000
Length of a row(b,c)	2,000	2,000,000
Columns in a `UNIQUE` constraint	6	32
Length of a `UNIQUE` constraint(b)	120	(d)
Length of foreign key column list(b)	120	(d)
Columns in a `GROUP BY` clause	6	255(e)
Length of `GROUP BY` column list	120	(e)
Sort specifications in `ORDER BY` clause	6	255(e)
Length of `ORDER BY` column list	120	(e)
Columns in a referential integrity constraint	6	32
Tables referenced in a SQL statement	15	No limit
Cursors simultaneously open	10	(f)
Items in a `SELECT` list	100	1000

Table B–2 (Cont.) Sizing for Database Constructs

(a) The number of SET clauses in an UPDATE statement refers to the number items separated by commas following the SET keyword.

(b) The FIPS PUB defines the length of a collection of columns to be the sum of: twice the number of columns, the length of each character column in bytes, decimal precision plus 1 of each exact numeric column, binary precision divided by 4 plus 1 of each approximate numeric column.

(c) The Oracle limit for the maximum row length is based on the maximum length of a row containing a LONG value of length 2 gigabytes and 999 VARCHAR2 values, each of length 4000 bytes: 2(254) + 231 + (999(4000)).

(d) The Oracle limit for a UNIQUE key is half the size of an Oracle data block (specified by the initialization parameter DB_BLOCK_SIZE) minus some overhead.

(e) Oracle places no limit on the number of columns in a GROUP BY clause or the number of sort specifications in an ORDER BY clause. However, the sum of the sizes of all the expressions in either a GROUP BY clause or an ORDER BY clause is limited to the size of an Oracle data block (specified by the initialization parameter DB_BLOCK_SIZE) minus some overhead.

(f) The Oracle limit for the number of cursors simultaneously opened is specified by the initialization parameter OPEN_CURSORS. The maximum value of this parameter depends on the memory available on your operating system and exceeds 100 in all cases.

Section 16.7 Character Set Support

Oracle supports the ASCII character set (FIPS PUB 1-2) on most computers and the EBCDIC character set on IBM mainframe computers. Oracle supports both single-byte and multibyte character sets.

Oracle Extensions to Standard SQL

Oracle supports numerous features that extend beyond standard SQL. In your Oracle applications, you can use these extensions just as you can use Entry SQL92.

If you are concerned with the portability of your applications to other implementations of SQL, use Oracle's FIPS Flagger to locate Oracle extensions to Entry SQL92 in your embedded SQL programs. The FIPS Flagger is part of the Oracle precompilers and the SQL*Module compiler.

> **See Also:** *Pro*COBOL Precompiler Programmer's Guide* and *Pro*C/C++ Precompiler Programmer's Guide* for information on how to use the FIPS Flagger.

C

Oracle Reserved Words

This appendix lists Oracle reserved words. Words followed by an asterisk (*) are also ANSI reserved words.

> **Note:** In addition to the following reserved words, Oracle uses system-generated names beginning with "SYS_" for implicitly generated schema objects and subobjects. Oracle discourages you from using this prefix in the names you explicitly provide to your schema objects and subobjects to avoid possible conflict in name resolution.

Table C–1 Oracle Reserved Words

ACCESS	CHAR *	DEFAULT *
ADD *	CHECK *	DELETE *
ALL *	CLUSTER	DESC *
ALTER *	COLUMN	DISTINCT *
AND *	COMMENT	DROP *
ANY *	COMPRESS	ELSE *
AS *	CONNECT *	EXCLUSIVE
ASC *	CREATE *	EXISTS
AUDIT	CURRENT *	FILE
BETWEEN *	DATE *	FLOAT *
BY *	DECIMAL *	FOR *
FROM *	NOT *	SHARE
GRANT *	NOWAIT	SIZE *

Table C–1 Oracle Reserved Words

GROUP *	NULL *	SMALLINT *
HAVING *	NUMBER	START
IDENTIFIED	OF *	SUCCESSFUL
IMMEDIATE *	OFFLINE	SYNONYM
IN *	ON *	SYSDATE
INCREMENT	ONLINE	TABLE *
INDEX	OPTION *	THEN *
INITIAL	OR *	TO *
INSERT *	ORDER *	TRIGGER
INTEGER *	PCTFREE	UID
INTERSECT *	PRIOR *	UNION *
INTO *	PRIVILEGES *	UNIQUE *
IS *	PUBLIC *	UPDATE *
LEVEL *	RAW	USER *
LIKE *	RENAME	VALIDATE
LOCK	RESOURCE	VALUES *
LONG	REVOKE *	VARCHAR *
MAXEXTENTS	ROW	VARCHAR2
MINUS	ROWID	VIEW *
MLSLABEL	ROWNUM	WHENEVER *
MODE	ROWS *	WHERE
MODIFY	SELECT *	WITH *
NOAUDIT	SESSION *	
NOCOMPRESS	SET *	

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y