Database Compatibility for Oracle® Developers Reference Guide

EDB Postgres™ Advanced Server 10

September 17, 2020

Database Compatibility for Oracle® Developers
Reference Guide
by EnterpriseDB® Corporation

EnterpriseDB Corporation, 34 Crosby Drive, Suite 201, Bedford, MA 01730, USA
T +1 781 357 3390 **F** +1 978 467 1307 **E** info@enterprisedb.com **www**.enterprisedb.com

Table of Contents

1 Introduction

Database Compatibility for Oracle means that an application runs in an Oracle environment as well as in the EDB Postgres Advanced Server (Advanced Server) environment with minimal or no changes to the application code.

This guide provides reference material about the compatibility features offered by Advanced Server:

- SQL Language syntax support
- Compatible Data Types
- SQL Commands
- Catalog Views
- System Catalog Tables

Developing an application that is compatible with Oracle databases in the Advanced Server requires special attention to which features are used in the construction of the application. For example, developing a compatible application means selecting:

- Data types to define the application's database tables that are compatible with Oracle databases
- SQL statements that are compatible with Oracle SQL
- System and built-in functions for use in SQL statements and procedural logic that are compatible with Oracle databases
- Stored Procedure Language (SPL) to create database server-side application logic for stored procedures, functions, triggers, and packages
- System catalog views that are compatible with Oracle's data dictionary

For detailed information about Advanced Server's compatibility features and extended functionality, please see the complete library of Advanced Server documentation, available at:

http://www.enterprisedb.com/products-services-training/products/documentation

1.1 What's New

The following database compatibility for Oracle features have been added to Advanced Server 9.6 to create Advanced Server 10:

- Advanced Server now supports usage of a composite type (created by the `CREATE TYPE AS` command) referenced by a field as its data type within a user-defined record type (declared with the `TYPE IS RECORD` statement). This record type containing a composite type can only be declared in a package specification or a package body. A composite type is not compatible with Oracle databases. However, composite types can generally be used within all SPL programs (functions, procedures, triggers, packages, etc.) as long as the composite type is not part of a record type (with the exception of packages). For more information on composite types, see Section 2.3.36.

1.2 Typographical Conventions Used in this Guide

Certain typographical conventions are used in this manual to clarify the meaning and usage of various commands, statements, programs, examples, etc. This section provides a summary of these conventions.

In the following descriptions a *term* refers to any word or group of words which may be language keywords, user-supplied values, literals, etc. A term's exact meaning depends upon the context in which it is used.

- *Italic font* introduces a new term, typically, in the sentence that defines it for the first time.
- `Fixed-width (mono-spaced) font` is used for terms that must be given literally such as SQL commands, specific table and column names used in the examples, programming language keywords, etc. For example, `SELECT * FROM emp;`
- *`Italic fixed-width font`* is used for terms for which the user must substitute values in actual usage. For example, `DELETE FROM` *`table_name`*;
- A vertical pipe | denotes a choice between the terms on either side of the pipe. A vertical pipe is used to separate two or more alternative terms within square brackets (optional choices) or braces (one mandatory choice).
- Square brackets [ ] denote that one or none of the enclosed term(s) may be substituted. For example, `[ a | b ]`, means choose one of "`a`" or "`b`" or neither of the two.
- Braces {} denote that exactly one of the enclosed alternatives must be specified. For example, `{ a | b }`, means exactly one of "`a`" or "`b`" must be specified.
- Ellipses ... denote that the proceeding term may be repeated. For example, `[ a | b ] ...` means that you may have the sequence, "`b a a b a`".

2 The SQL Language

The following sections describe the subset of the Advanced Server SQL language compatible with Oracle databases. The following SQL syntax, commands, data types, and functions work in both EDB Postgres Advanced Server and Oracle.

The Advanced Server documentation set includes syntax and commands for extended functionality (functionality that does not provide database compatibility for Oracle or support Oracle-styled applications) that is not included in this guide.

This section is organized into the following sections:

- General discussion of Advanced Server SQL syntax and language elements
- Data types
- Summary of SQL commands
- Built-in functions

2.1 SQL Syntax

This section describes the general syntax of SQL. It forms the foundation for understanding the following chapters that include detail about how the SQL commands are applied to define and modify data.

2.1.1 Lexical Structure

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

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

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

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

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

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

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a `SELECT`, an `UPDATE`, and an `INSERT` command. But for instance the `UPDATE` command always requires a `SET` token to appear in a certain position, and this particular variation of `INSERT` also requires a `VALUES` token in order to be complete. The precise syntax rules for each command are described in Section 2.3.

2.1.2 Identifiers and Key Words

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

SQL identifiers and key words must begin with a letter (`a-z` or `A-Z`). Subsequent characters in an identifier or key word can be letters, underscores, digits (`0-9`), dollar signs (`$`), or number signs (`#`).

Identifier and key word names are case insensitive. Therefore

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

can equivalently be written as:

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

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

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

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

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

Quoted identifiers can contain any character, except the character with the numeric code zero.

To include a double quote, use two double quotes. This allows you to construct table or column names that would otherwise not be possible (such as ones containing spaces or ampersands). The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers `FOO`, `foo`, and `"foo"` are considered the same by Advanced Server, but `"Foo"` and `"FOO"` are different from these three and each other. The folding of unquoted names to lower case is not compatible with Oracle databases. In Oracle syntax, unquoted names are folded to upper case: for example, `foo` is equivalent to `"FOO"` not `"foo"`. If you want to write portable applications you are advised to always quote a particular name or never quote it.

2.1.3 Constants

The kinds of implicitly-typed constants in Advanced Server are *strings* and *numbers*. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

2.1.3.1 String Constants

A *string constant* in SQL is an arbitrary sequence of characters bounded by single quotes (`'`), for example `'This is a string'`. To include a single-quote character within a string constant, write two adjacent single quotes, e.g. `'Dianne''s horse'`. Note that this is not the same as a double-quote character (").

2.1.3.2 Numeric Constants

Numeric constants are accepted in these general forms:

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

where *digits* is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (`e`), if one is present. There may not be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

```
42
3.5
4.
.001
5e2
1.925e-3
```

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type `INTEGER` if its value fits in type `INTEGER` (32 bits); otherwise it is presumed to be type `BIGINT` if its value fits in type `BIGINT` (64 bits); otherwise it is taken to be type `NUMBER`. Constants that contain decimal points and/or exponents are always initially presumed to be type `NUMBER`.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant will be automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it as described in the following section.

2.1.3.3 Constants of Other Types

A constant of an arbitrary type can be entered using the following notation:

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

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

`CAST` can also be used to specify runtime type conversions of arbitrary expressions.

2.1.4 Comments

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

```
-- This is a standard SQL comment
```

Alternatively, C-style block comments can be used:

```
/* multiline comment
 * block
 */
```

where the comment begins with `/*` and extends to the matching occurrence of `*/`.

2.2 Data Types

The following table shows the built-in general-purpose data types.

Table 2-1 Data Types

Name	Alias	Description
BLOB	LONG RAW, RAW(*n*), BYTEA	Binary data
BOOLEAN		Logical Boolean (true/false)
CHAR [(*n*)]	CHARACTER [(*n*)]	Fixed-length character string of n characters
CLOB	LONG, LONG VARCHAR	Long character string
DATE	TIMESTAMP	Date and time to the second
DOUBLE PRECISION	FLOAT, FLOAT(25) – FLOAT(53)	Double precision floating-point number
INTEGER	INT, BINARY_INTEGER, PLS_INTEGER	Signed four-byte integer
NUMBER	DEC, DECIMAL, NUMERIC	Exact numeric with optional decimal places
NUMBER(*p* [, *s*])	DEC(p [, *s*]), DECIMAL(*p* [, *s*]), NUMERIC(*p* [, *s*])	Exact numeric of maximum precision, *p*, and optional scale, *s*
REAL	FLOAT(1) – FLOAT(24)	Single precision floating-point number
TIMESTAMP [(*p*)]		Date and time with optional, fractional second precision, *p*
TIMESTAMP [(*p*)] WITH TIME ZONE		Date and time with optional, fractional second precision, *p*, and with time zone
VARCHAR2(*n*)	CHAR VARYING(*n*), CHARACTER VARYING(*n*), VARCHAR(*n*)	Variable-length character string with a maximum length of *n* characters
XMLTYPE		XML data

2.2.1 Numeric Types

Numeric types consist of four-byte integers, four-byte and eight-byte floating-point numbers, and fixed-precision decimals. The following table lists the available types.

Table 2-2 Numeric Types

Name	Storage Size	Description	Range
BINARY_INTEGER	4 bytes	Signed integer, Alias for INTEGER	-2,147,483,648 to +2,147,483,647
DOUBLE PRECISION	8 bytes	Variable-precision, inexact	15 decimal digits precision
INTEGER	4 bytes	Usual choice for integer	-2,147,483,648 to +2,147,483,647
NUMBER	Variable	User-specified precision, exact	Up to 1000 digits of precision
NUMBER(*p* [, *s*])	Variable	Exact numeric of maximum precision, *p*, and optional scale, *s*	Up to 1000 digits of precision
PLS_INTEGER	4 bytes	Signed integer, Alias for INTEGER	-2,147,483,648 to +2,147,483,647
REAL	4 bytes	Variable-precision, inexact	6 decimal digits precision
ROWID	8 bytes	Signed 8 bit integer.	-9223372036854775808 to 9223372036854775807

The following sections describe the types in detail.

2.2.1.1 Integer Types

The type, INTEGER, stores whole numbers (without fractional components) between the values of -2,147,483,648 and +2,147,483,647. Attempts to store values outside of the allowed range will result in an error.

Columns of the ROWID type holds fixed-length binary data that describes the physical address of a record. ROWID is an unsigned, four-byte INTEGER that stores whole numbers (without fractional components) between the values of 0 and 4,294,967,295. Attempts to store values outside of the allowed range will result in an error.

2.2.1.2 Arbitrary Precision Numbers

The type, NUMBER, can store practically an unlimited number of digits of precision and perform calculations exactly. It is especially recommended for storing monetary amounts and other quantities where exactness is required. However, the NUMBER type is very slow compared to the floating-point types described in the next section.

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

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

```
NUMBER(precision, scale)
```

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

```
NUMBER(precision)
```

selects a scale of 0. Specifying NUMBER without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas NUMBER columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. For maximum portability, it is best to specify the precision and scale explicitly.)

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

2.2.1.3 Floating-Point Types

The data types REAL and DOUBLE PRECISION are *inexact*, variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

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

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

If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.

Comparing two floating-point values for equality may or may not work as expected.

On most platforms, the REAL type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The DOUBLE PRECISION type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding may take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

Advanced Server also supports the SQL standard notations FLOAT and FLOAT(*p*) for specifying inexact numeric types. Here, *p* specifies the minimum acceptable precision in binary digits. Advanced Server accepts FLOAT(1) to FLOAT(24) as selecting the REAL type, while FLOAT(25) to FLOAT(53) as selecting DOUBLE PRECISION. Values of *p* outside the allowed range draw an error. FLOAT with no precision specified is taken to mean DOUBLE PRECISION.

2.2.2 Character Types

The following table lists the general-purpose character types available in Advanced Server.

Table 2-3 Character Types

Name	Description
CHAR[(*n*)]	Fixed-length character string, blank-padded to the size specified by *n*
CLOB	Large variable-length up to 1 GB
LONG	Variable unlimited length.
NVARCHAR(*n*)	Variable-length national character string, with limit.
NVARCHAR2(*n*)	Variable-length national character string, with limit.
STRING	Alias for VARCHAR2.
VARCHAR(*n*)	Variable-length character string, with limit (considered deprecated, but supported for compatibility)
VARCHAR2(*n*)	Variable-length character string, with limit

Where *n* is a positive integer; these types can store strings up to *n* characters in length. An attempt to assign a value that exceeds the length of *n* will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length.

CHAR

If you do not specify a value for *n*, *n* will default to 1. If the string to be assigned is shorter than *n*, values of type CHAR will be space-padded to the specified width (*n*), and will be stored and displayed that way.

Padding spaces are treated as semantically insignificant. That is, trailing spaces are disregarded when comparing two values of type CHAR, and they will be removed when converting a CHAR value to one of the other string types.

If you explicitly cast an over-length value to a CHAR(*n*) type, the value will be truncated to *n* characters without raising an error (as specified by the SQL standard).

VARCHAR, VARCHAR2, NVARCHAR and NVARCHAR2

If the string to be assigned is shorter than *n*, values of type VARCHAR, VARCHAR2, NVARCHAR and NVARCHAR2 will store the shorter string without padding.

Note that trailing spaces *are* semantically significant in VARCHAR values.

If you explicitly cast a value to a VARCHAR type, an over-length value will be truncated to *n* characters without raising an error (as specified by the SQL standard).

CLOB

You can store a large character string in a CLOB type. CLOB is semantically equivalent to VARCHAR2 except no length limit is specified. Generally, you should use a CLOB type if the maximum string length is not known.

The longest possible character string that can be stored in a CLOB type is about 1 GB.

The storage requirement for data of these types is the actual string plus 1 byte if the string is less than 127 bytes, or 4 bytes if the string is 127 bytes or greater. In the case of CHAR, the padding also requires storage. Long strings are compressed by the system automatically, so the physical requirement on disk may be less. Long values are stored in background tables so they do not interfere with rapid access to the shorter column values.

The database character set determines the character set used to store textual values.

2.2.3 Binary Data

The following data types allows storage of binary strings.

Table 2-4 Binary Large Object

Name	Storage Size	Description
BINARY	The length of the binary string.	Fixed-length binary string, with a length between 1 and 8300.
BLOB	The actual binary string plus 1 byte if the binary string is less than 127 bytes, or 4 bytes if the binary string is 127 bytes or greater.	Variable-length binary string, with a maximum size of 1 GB.
VARBINARY	The length of the binary string	Variable-length binary string, with a length between 1 and 8300.

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from characters strings by two characteristics: First, binary strings specifically allow storing octets of value zero and other "non-printable" octets (defined as octets outside the range 32 to 126). Second, operations on binary strings process the actual bytes, whereas the encoding and processing of character strings depends on locale settings.

2.2.4 Date/Time Types

The following discussion of the date/time types assumes that the configuration parameter, `edb_redwood_date`, has been set to `TRUE` whenever a table is created or altered.

Advanced Server supports the date/time types shown in the following table.

Table 2-5 Date/Time Types

Name	Storage Size	Description	Low Value	High Value	Resolution
`DATE`	8 bytes	Date and time	4713 BC	5874897 AD	1 second
`INTERVAL DAY TO SECOND [(p)]`	12 bytes	Period of time	-178000000 years	178000000 years	1 microsecond / 14 digits
`INTERVAL YEAR TO MONTH`	12 bytes	Period of time	-178000000 years	178000000 years	1 microsecond / 14 digits
`TIMESTAMP [(p)]`	8 bytes	Date and time	4713 BC	5874897 AD	1 microsecond
`TIMESTAMP [(p)] WITH TIME ZONE`	8 bytes	Date and time with time zone	4713 BC	5874897 AD	1 microsecond

When `DATE` appears as the data type of a column in the data definition language (DDL) commands, `CREATE TABLE` or `ALTER TABLE`, it is translated to `TIMESTAMP` at the time the table definition is stored in the database. Thus, a time component will also be stored in the column along with the date.

When `DATE` appears as a data type of a variable in an SPL declaration section, or the data type of a formal parameter in an SPL procedure or an SPL function, or the return type of an SPL function, it is always translated to `TIMESTAMP` and thus can handle a time component if present.

`TIMESTAMP` accepts an optional precision value *p* which specifies the number of fractional digits retained in the seconds field. The allowed range of *p* is from 0 to 6 with the default being 6.

When `TIMESTAMP` values are stored as double precision floating-point numbers (currently the default), the effective limit of precision may be less than 6. `TIMESTAMP` values are stored as seconds before or after midnight 2000-01-01. Microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. When `TIMESTAMP` values are stored as eight-byte integers (a compile-time option), microsecond precision is available over the full range of values. However eight-byte integer timestamps have a more limited range of dates than shown above: from 4713 BC up to 294276 AD.

`TIMESTAMP (p) WITH TIME ZONE` is similar to `TIMESTAMP (p)`, but includes the time zone as well.

2.2.4.1 INTERVAL Types

INTERVAL values specify a period of time. Values of INTERVAL type are composed of fields that describe the value of the data. The following table lists the fields allowed in an INTERVAL type:

Table 2-6 Interval Types

Field Name	INTERVAL Values Allowed
YEAR	Integer value (positive or negative)
MONTH	0 through 11
DAY	Integer value (positive or negative)
HOUR	0 through 23
MINUTE	0 through 59
SECOND	0 through 59.9(*p*) where 9(*p*) is the precision of fractional seconds

The fields must be presented in descending order – from YEARS to MONTHS, and from DAYS to HOURS, MINUTES and then SECONDS.

Advanced Server supports two INTERVAL types compatible with Oracle databases.

The first variation supported by Advanced Server is INTERVAL DAY TO SECOND [(*p*)]. INTERVAL DAY TO SECOND [(*p*)] stores a time interval in days, hours, minutes and seconds.

p specifies the precision of the second field.

Advanced Server interprets the value:

```
INTERVAL '1 2:34:5.678' DAY TO SECOND(3)
```

as 1 day, 2 hours, 34 minutes, 5 seconds and 678 thousandths of a second.

Advanced Server interprets the value:

```
INTERVAL '1 23' DAY TO HOUR
```

as 1 day and 23 hours.

Advanced Server interprets the value:

```
INTERVAL '2:34' HOUR TO MINUTE
```

as 2 hours and 34 minutes.

Advanced Server interprets the value:

```
INTERVAL '2:34:56.129' HOUR TO SECOND(2)
```

as 2 hours, 34 minutes, 56 seconds and 13 thousandths of a second. Note that the fractional second is rounded up to 13 because of the specified precision.

The second variation supported by Advanced Server that is compatible with Oracle databases is `INTERVAL YEAR TO MONTH`. This variation stores a time interval in years and months.

Advanced Server interprets the value:

```
INTERVAL '12-3' YEAR TO MONTH
```

as 12 years and 3 months.

Advanced Server interprets the value:

```
 INTERVAL '456' YEAR(2)
```

as 12 years and 3 months.

Advanced Server interprets the value:

```
INTERVAL '300' MONTH
```

as 25 years.

2.2.4.2 Date/Time Input

Date and time input is accepted in ISO 8601 SQL-compatible format, the Oracle default dd-MON-yy format, as well as a number of other formats provided that there is no ambiguity as to which component is the year, month, and day. However, use of the `TO_DATE` function is strongly recommended to avoid ambiguities.

Any date or time literal input needs to be enclosed in single quotes, like text strings. The following SQL standard syntax is also accepted:

```
type 'value'
```

`type` is either `DATE` or `TIMESTAMP`.

`value` is a date/time text string.

2.2.4.2.1 Dates

The following table shows some possible input formats for dates, all of which equate to January 8, 1999.

Table 2-7 Date Input

Example
January 8, 1999
1999-01-08
1999-Jan-08
Jan-08-1999
08-Jan-1999
08-Jan-99
Jan-08-99
19990108
990108

The date values can be assigned to a `DATE` or `TIMESTAMP` column or variable. The hour, minute, and seconds fields will be set to zero if the date value is not appended with a time value.

2.2.4.2.2 Times

Some examples of the time component of a date or time stamp are shown in the following table.

Table 2-8 Time Input

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

2.2.4.2.3 Time Stamps

Valid input for time stamps consists of a concatenation of a date and a time. The date portion of the time stamp can be formatted according to any of the examples shown in Table 2-7. The time portion of the time stamp can be formatted according to any of examples shown in Table 2-8.

The following is an example of a time stamp which follows the Oracle default format.

```
08-JAN-99 04:05:06
```

The following is an example of a time stamp which follows the ISO 8601 standard.

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

2.2.4.3 Date/Time Output

The default output format of the date/time types will be either (dd-MON-yy) referred to as the *Redwood date style*, compatible with Oracle databases, or (yyyy-mm-dd) referred to as the ISO 8601 format, depending upon the application interface to the database. Applications that use JDBC such as SQL Interactive always present the date in ISO 8601 form. Other applications such as PSQL present the date in Redwood form.

The following table shows examples of the output formats for the two styles, Redwood and ISO 8601.

Table 2-9 Date/Time Output Styles

Description	Example
Redwood style	31-DEC-05 07:37:16
ISO 8601/SQL standard	1997-12-17 07:37:16

2.2.4.4 Internals

Advanced Server uses Julian dates for all date/time calculations. Julian dates correctly predict or calculate any date after 4713 BC based on the assumption that the length of the year is 365.2425 days.

2.2.5 Boolean Type

Advanced Server provides the standard SQL type `BOOLEAN`. `BOOLEAN` can have one of only two states: `TRUE` or `FALSE`. A third state, `UNKNOWN`, is represented by the SQL `NULL` value.

Table 2-10 Boolean Type

Name	Storage Size	Description
`BOOLEAN`	1 byte	Logical Boolean (true/false)

The valid literal value for representing the true state is `TRUE`. The valid literal for representing the false state is `FALSE`.

2.2.6 XML Type

The `XMLTYPE` data type is used to store XML data. Its advantage over storing XML data in a character field is that it checks the input values for well-formedness, and there are support functions to perform type-safe operations on it.

The XML type can store well-formed "documents", as defined by the XML standard, as well as "content" fragments, which are defined by the production `XMLDecl? content` in the XML standard. Roughly, this means that content fragments can have more than one top-level element or character node.

Note: Oracle does not support the storage of content fragments in `XMLTYPE` columns.

The following example shows the creation and insertion of a row into a table with an `XMLTYPE` column.

```
CREATE TABLE books (
    content         XMLTYPE
);

INSERT INTO books VALUES (XMLPARSE (DOCUMENT '<?xml
version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>'));

SELECT * FROM books;

                          content
-----------------------------------------------------------
 <book><title>Manual</title><chapter>...</chapter></book>
(1 row)
```

2.3 SQL Commands

This section provides a summary of the SQL commands compatible with Oracle databases that are supported by Advanced Server. The SQL commands in this section will work on both an Oracle database and an Advanced Server database.

Note the following points:

- Advanced Server supports other commands that are not listed here. These commands may have no Oracle equivalent or they may provide the similar or same functionality as an Oracle SQL command, but with different syntax.
- The SQL commands in this section do not necessarily represent the full syntax, options, and functionality available for each command. In most cases, syntax, options, and functionality that are not compatible with Oracle databases have been omitted from the command description and syntax.
- The Advanced Server documentation set documents command functionality that may not be compatible with Oracle databases.

2.3.1 ALTER INDEX

Name

`ALTER INDEX` -- modify an existing index.

Synopsis

Advanced Server supports two variations of the `ALTER INDEX` command compatible with Oracle databases. Use the first variation to rename an index:

```
ALTER INDEX name RENAME TO new_name
```

Use the second variation of the `ALTER INDEX` command to rebuild an index:

```
ALTER INDEX name REBUILD
```

Description

`ALTER INDEX` changes the definition of an existing index. The `RENAME` clause changes the name of the index. The `REBUILD` clause reconstructs an index, replacing the old copy of the index with an updated version based on the index's table.

The `REBUILD` clause invokes the PostgreSQL `REINDEX` command; for more information about using the `REBUILD` clause, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/10/static/sql-reindex.html

`ALTER INDEX` has no effect on stored data.

Parameters

name

The name (possibly schema-qualified) of an existing index.

new_name

New name for the index.

Examples

To change the name of an index from `name_idx` to `empname_idx`:

```
ALTER INDEX name_idx RENAME TO empname_idx;
```

To rebuild an index named `empname_idx`:

```
ALTER INDEX empname_idx REBUILD;
```

See Also

CREATE INDEX, DROP INDEX

2.3.2 ALTER PROCEDURE

Name

ALTER PROCEDURE

Synopsis

ALTER PROCEDURE *procedure_name options* [RESTRICT]

Description

Use the ALTER PROCEDURE statement to specify that a procedure is a SECURITY INVOKER or SECURITY DEFINER.

Parameters

procedure_name

procedure_name specifies the (possibly schema-qualified) name of a stored procedure.

options may be:

[EXTERNAL] SECURITY DEFINER

Specify SECURITY DEFINER to instruct the server to execute the procedure with the privileges of the user that created the procedure. The EXTERNAL keyword is accepted for compatibility, but ignored.

[EXTERNAL] SECURITY INVOKER

Specify SECURITY INVOKER to instruct the server to execute the procedure with the privileges of the user that is invoking the procedure. The EXTERNAL keyword is accepted for compatibility, but ignored.

The RESTRICT keyword is accepted for compatibility, but ignored.

Examples

The following command specifies that the update_balance procedure should execute with the privileges of the user invoking the procedure:

```
ALTER PROCEDURE update_balance SECURITY INVOKER;
```

2.3.3 ALTER PROFILE

Name

`ALTER PROFILE` – alter an existing profile

Synopsis

```
ALTER PROFILE profile_name RENAME TO new_name;

ALTER PROFILE profile_name
      LIMIT {parameter value}[...];
```

Description

Use the `ALTER PROFILE` command to modify a user-defined profile; Advanced Server supports two forms of the command:

- Use `ALTER PROFILE…RENAME TO` to change the name of a profile.
- Use `ALTER PROFILE…LIMIT` to modify the limits associated with a profile.

Include the `LIMIT` clause and one or more space-delimited *`parameter`*/*`value`* pairs to specify the rules enforced by Advanced Server, or use `ALTER PROFILE…RENAME TO` to change the name of a profile.

Parameters

`profile_name`

The name of the profile.

`new_name`

`new_name` specifies the new name of the profile.

`parameter`

`parameter` specifies the attribute limited by the profile.

`value`

`value` specifies the parameter limit.

Advanced Server supports the *`value`* shown below for each *`parameter`*:

FAILED_LOGIN_ATTEMPTS specifies the number of failed login attempts that a user may make before the server locks the user out of their account for the length of time specified by PASSWORD_LOCK_TIME. Supported values are:

- An INTEGER value greater than 0.
- DEFAULT - the value of FAILED_LOGIN_ATTEMPTS specified in the DEFAULT profile.
- UNLIMITED – the connecting user may make an unlimited number of failed login attempts.

PASSWORD_LOCK_TIME specifies the length of time that must pass before the server unlocks an account that has been locked because of FAILED_LOGIN_ATTEMPTS. Supported values are:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- DEFAULT - the value of PASSWORD_LOCK_TIME specified in the DEFAULT profile.
- UNLIMITED – the account is locked until it is manually unlocked by a database superuser.

PASSWORD_LIFE_TIME specifies the number of days that the current password may be used before the user is prompted to provide a new password. Include the PASSWORD_GRACE_TIME clause when using the PASSWORD_LIFE_TIME clause to specify the number of days that will pass after the password expires before connections by the role are rejected. If PASSWORD_GRACE_TIME is not specified, the password will expire on the day specified by the default value of PASSWORD_GRACE_TIME, and the user will not be allowed to execute any command until a new password is provided. Supported values are:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- DEFAULT - the value of PASSWORD_LIFE_TIME specified in the DEFAULT profile.
- UNLIMITED – The password does not have an expiration date.

PASSWORD_GRACE_TIME specifies the length of the grace period after a password expires until the user is forced to change their password. When the grace period expires, a user will be allowed to connect, but will not be allowed to execute any command until they update their expired password. Supported values are:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- DEFAULT - the value of PASSWORD_GRACE_TIME specified in the DEFAULT profile.
- UNLIMITED – The grace period is infinite.

PASSWORD_REUSE_TIME specifies the number of days a user must wait before re-using a password. The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse. Supported values are:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- DEFAULT - the value of PASSWORD_REUSE_TIME specified in the DEFAULT profile.
- UNLIMITED – The password can be re-used without restrictions.

PASSWORD_REUSE_MAX specifies the number of password changes that must occur before a password can be reused. The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse. Supported values are:

- An INTEGER value greater than or equal to 0.
- DEFAULT - the value of PASSWORD_REUSE_MAX specified in the DEFAULT profile.
- UNLIMITED – The password can be re-used without restrictions.

PASSWORD_VERIFY_FUNCTION specifies password complexity. Supported values are:

- The name of a PL/SQL function.
- DEFAULT - the value of PASSWORD_VERIFY_FUNCTION specified in the DEFAULT profile.
- NULL

Examples

The following example modifies a profile named `acctg_profile`:

```
ALTER PROFILE acctg_profile
       LIMIT FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 1;
```

`acctg_profile` will count failed connection attempts when a login role attempts to connect to the server. The profile specifies that if a user has not authenticated with the correct password in three attempts, the account will be locked for one day.

The following example changes the name of `acctg_profile` to `payables_profile`:

```
ALTER PROFILE acctg_profile RENAME TO payables_profile;
```

2.3.4 ALTER QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the `ALTER QUEUE SQL` command. This syntax can be used in association with the `DBMS_AQADM` package.

Name

`ALTER QUEUE` -- allows a superuser or a user with the `aq_administrator_role` privilege to modify the attributes of a queue.

Synopsis

This command is available in four forms. The first form of this command changes the name of a queue.

```
ALTER QUEUE queue_name RENAME TO new_name
```

Parameters

queue_name

The name (optionally schema-qualified) of an existing queue.

RENAME TO

Include the `RENAME TO` clause and a new name for the queue to rename the queue.

new_name

New name for the queue.

The second form of the `ALTER QUEUE` command modifies the attributes of the queue:

```
ALTER QUEUE queue_name SET [ ( { option_name option_value  }
[,SET option_name
```

Parameters

queue_name

The name (optionally schema-qualified) of an existing queue.

Include the SET clause and option_name/option_value pairs to modify the attributes of the queue:

option_name option_value

The name of the option or options to be associated with the new queue and the corresponding value of the option. If you provide duplicate option names, the server will return an error.

- If *option_name* is retries, provide an integer that represents the number of times a dequeue may be attempted.
- If *option_name* is retrydelay, provide a double-precision value that represents the delay in seconds.
- If *option_name* is retention, provide a double-precision value that represents the retention time in seconds.

Use the third form of the ALTER QUEUE command to enable or disable enqueuing and/or dequeuing on a particular queue:

```
ALTER QUEUE queue_name ACCESS { START | STOP } [ FOR {
enqueue | dequeue } ] [ NOWAIT ]
```

Parameters

queue_name

The name (optionally schema-qualified) of an existing queue.

ACCESS

Include the ACCESS keyword to enable or disable enqueuing and/or dequeuing on a particular queue.

START | STOP

Use the START and STOP keywords to indicate the desired state of the queue.

FOR enqueue|dequeue

Use the FOR clause to indicate if you are specifying the state of enqueueing or dequeueing activity on the specified queue.

NOWAIT

Include the NOWAIT keyword to specify that the server should not wait for the completion of outstanding transactions before changing the state of the queue. The NOWAIT keyword can only be used when specifying an ACCESS value of STOP. The server will return an error if NOWAIT is specified with an ACCESS value of START.

Use the fourth form to ADD or DROP callback details for a particular queue.

```
ALTER QUEUE queue_name { ADD | DROP } CALL TO location_name
[ WITH callback_option ]
```

Parameters

queue_name

The name (optionally schema-qualified) of an existing queue.

ADD | DROP

Include the ADD or DROP keywords to enable add or remove callback details for a queue.

location_name

location_name specifies the name of the callback procedure.

callback_option

callback_option can be context; specify a RAW value when including this clause.

Example

The following example changes the name of a queue from work_queue_east to work_order:

```
ALTER QUEUE work_queue_east RENAME TO work_order;
```

The following example modifies a queue named work_order, setting the number of retries to 100, the delay between retries to 2 seconds, and the length of time that the queue will retain dequeued messages to 10 seconds:

```
ALTER QUEUE work_order SET (retries 100, retrydelay 2, retention 10);
```

The following commands enable enqueueing and dequeueing in a queue named work_order:

```
ALTER QUEUE work_order ACCESS START;
ALTER QUEUE work_order ACCESS START FOR enqueue;
ALTER QUEUE work_order ACCESS START FOR dequeue;
```

The following commands disable enqueueing and dequeueing in a queue named `work_order`:

```
ALTER QUEUE work_order ACCESS STOP NOWAIT;
ALTER QUEUE work_order ACCESS STOP FOR enqueue;
ALTER QUEUE work_order ACCESS STOP FOR dequeue;
```

See Also

CREATE QUEUE, DROP QUEUE

2.3.5 ALTER QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the `ALTER QUEUE SQL` command. This syntax can be used in association with the `DBMS_AQADM` package.

Name

`ALTER QUEUE TABLE`-- modify an existing queue table.

Synopsis

Use `ALTER QUEUE TABLE` to change the name of an existing queue table:

```
ALTER QUEUE TABLE name RENAME TO new_name
```

Description

`ALTER QUEUE TABLE` allows a superuser or a user with the `aq_administrator_role` privilege to change the name of an existing queue table.

Parameters

name

The name (optionally schema-qualified) of an existing queue table.

new_name

New name for the queue table.

Example

To change the name of a queue table from `wo_table_east` to `work_order_table`:

```
ALTER QUEUE TABLE wo_queue_east RENAME TO work_order_table;
```

See Also

CREATE QUEUE TABLE, DROP QUEUE TABLE

2.3.6 ALTER ROLE… IDENTIFIED BY

Name

ALTER ROLE - change the password associated with a database role

Synopsis

```
ALTER ROLE role_name IDENTIFIED BY password
          [REPLACE prev_password]
```

Description

A role without the CREATEROLE privilege may use this command to change their own password. An unprivileged role must include the REPLACE clause and their previous password if PASSWORD_VERIFY_FUNCTION is not NULL in their profile. When the REPLACE clause is used by a non-superuser, the server will compare the password provided to the existing password and raise an error if the passwords do not match.

A database superuser can use this command to change the password associated with any role. If a superuser includes the REPLACE clause, the clause is ignored; a non-matching value for the previous password will not throw an error.

If the role for which the password is being changed has the SUPERUSER attribute, then a superuser must issue this command. A role with the CREATEROLE attribute can use this command to change the password associated with a role that is not a superuser.

Parameters

role_name

The name of the role whose password is to be altered.

password

The role's new password.

prev_password

The role's previous password.

Examples

To change a role's password:

```
ALTER ROLE john IDENTIFIED BY xyRP35z REPLACE 23PJ74a;
```

2.3.7 ALTER ROLE - Managing Database Link and DBMS_RLS Privileges

Advanced Server includes extra syntax (not offered by Oracle) for the ALTER ROLE command. This syntax can be useful when assigning privileges related to creating and dropping database links compatible with Oracle databases, and fine-grained access control (using DBMS_RLS).

CREATE DATABASE LINK

A user who holds the CREATE DATABASE LINK privilege may create a private database link. The following ALTER ROLE command grants privileges to an Advanced Server role that allow the specified role to create a private database link:

```
ALTER ROLE role_name
    WITH [CREATEDBLINK | CREATE DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT CREATE DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
  WITH [NOCREATEDBLINK | NO CREATE DATABASE LINK]
```

Please note: the CREATEDBLINK and NOCREATEDBLINK keywords should be considered deprecated syntax; we recommend using the CREATE DATABASE LINK and NO CREATE DATABASE LINK syntax options.

CREATE PUBLIC DATABASE LINK

A user who holds the CREATE PUBLIC DATABASE LINK privilege may create a public database link. The following ALTER ROLE command grants privileges to an Advanced Server role that allow the specified role to create a public database link:

```
ALTER ROLE role_name
  WITH [CREATEPUBLICDBLINK | CREATE PUBLIC DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT CREATE PUBLIC DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
  WITH [NOCREATEPUBLICDBLINK | NO CREATE PUBLIC DATABASE LINK]
```

Please note: the CREATEPUBLICDBLINK and NOCREATEPUBLICDBLINK keywords should be considered deprecated syntax; we recommend using the CREATE PUBLIC DATABASE LINK and NO CREATE PUBLIC DATABASE LINK syntax options.

DROP PUBLIC DATABASE LINK

A user who holds the DROP PUBLIC DATABASE LINK privilege may drop a public database link. The following ALTER ROLE command grants privileges to an Advanced Server role that allow the specified role to drop a public database link:

```
ALTER ROLE role_name
  WITH [DROPPUBLICDBLINK | DROP PUBLIC DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT DROP PUBLIC DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
  WITH [NODROPPUBLICDBLINK | NO DROP PUBLIC DATABASE LINK]
```

Please note: the DROPPUBLICDBLINK and NODROPPUBLICDBLINK keywords should be considered deprecated syntax; we recommend using the DROP PUBLIC DATABASE LINK and NO DROP PUBLIC DATABASE LINK syntax options.

EXEMPT ACCESS POLICY

A user who holds the EXEMPT ACCESS POLICY privilege is exempt from fine-grained access control (DBMS_RLS) policies. A user who holds these privileges will be able to view or modify any row in a table constrained by a DBMS_RLS policy. The following ALTER ROLE command grants privileges to an Advanced Server role that exempt the specified role from any defined DBMS_RLS policies:

```
ALTER ROLE role_name
    WITH [POLICYEXEMPT | EXEMPT ACCESS POLICY]
```

This command is the functional equivalent of:

```
GRANT EXEMPT ACCESS POLICY TO role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
  WITH [NOPOLICYEXEMPT | NO EXEMPT ACCESS POLICY]
```

Please note: the `POLICYEXEMPT` and `NOPOLICYEXEMPT` keywords should be considered deprecated syntax; we recommend using the `EXEMPT ACCESS POLICY` and `NO EXEMPT ACCESS POLICY` syntax options.

See Also

CREATE ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE

2.3.8 ALTER SEQUENCE

Name

`ALTER SEQUENCE` -- change the definition of a sequence generator

Synopsis

```
ALTER SEQUENCE name [ INCREMENT BY increment ]
  [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
  [ CACHE cache | NOCACHE ] [ CYCLE ]
```

Description

`ALTER SEQUENCE` changes the parameters of an existing sequence generator. Any parameter not specifically set in the `ALTER SEQUENCE` command retains its prior setting.

Parameters

name

The name (optionally schema-qualified) of a sequence to be altered.

increment

The clause `INCREMENT BY` *increment* is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value will be maintained.

minvalue

The optional clause `MINVALUE` *minvalue* determines the minimum value a sequence can generate. If not specified, the current minimum value will be maintained. Note that the key words, `NO MINVALUE`, may be used to set this behavior back to the defaults of 1 and $-2^{63}-1$ for ascending and descending sequences, respectively, however, this term is not compatible with Oracle databases.

maxvalue

The optional clause `MAXVALUE` *maxvalue* determines the maximum value for the sequence. If not specified, the current maximum value will be maintained. Note that the key words, `NO MAXVALUE`, may be used to set this behavior back to

the defaults of 2^{63}-1 and -1 for ascending and descending sequences, respectively, however, this term is not compatible with Oracle databases.

cache

The optional clause CACHE *cache* specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., NOCACHE). If unspecified, the old cache value will be maintained.

CYCLE

The CYCLE option allows the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively. If not specified, the old cycle behavior will be maintained. Note that the key words, NO CYCLE, may be used to alter the sequence so that it does not recycle, however, this term is not compatible with Oracle databases.

Notes

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, ALTER SEQUENCE is never rolled back; the changes take effect immediately and are not reversible.

ALTER SEQUENCE will not immediately affect NEXTVAL results in backends, other than the current one, that have pre-allocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence parameters. The current backend will be affected immediately.

Examples

Change the increment and cache value of sequence, serial.

```
ALTER SEQUENCE serial INCREMENT BY 2 CACHE 5;
```

See Also

CREATE SEQUENCE, DROP SEQUENCE

2.3.9 ALTER SESSION

Name

`ALTER SESSION` -- change a runtime parameter

Synopsis

`ALTER SESSION SET` *`name = value`*

Description

The `ALTER SESSION` command changes runtime configuration parameters. `ALTER SESSION` only affects the value used by the current session. Some of these parameters are provided solely for compatibility with Oracle syntax and have no effect whatsoever on the runtime behavior of Advanced Server. Others will alter a corresponding Advanced Server database server runtime configuration parameter.

Parameters

`name`

Name of a settable runtime parameter. Available parameters are listed below.

`value`

New value of parameter.

Configuration Parameters

The following configuration parameters can be modified using the `ALTER SESSION` command:

`NLS_DATE_FORMAT (string)`

Sets the display format for date and time values as well as the rules for interpreting ambiguous date input values. Has the same effect as setting the Advanced Server `datestyle` runtime configuration parameter.

`NLS_LANGUAGE (string)`

Sets the language in which messages are displayed. Has the same effect as setting the Advanced Server `lc_messages` runtime configuration parameter.

`NLS_LENGTH_SEMANTICS (string)`

Valid values are `BYTE` and `CHAR`. The default is `BYTE`. This parameter is provided for syntax compatibility only and has no effect in the Advanced Server.

`OPTIMIZER_MODE (string)`

Sets the default optimization mode for queries. Valid values are `ALL_ROWS`, `CHOOSE`, `FIRST_ROWS`, `FIRST_ROWS_10`, `FIRST_ROWS_100`, and `FIRST_ROWS_1000`. The default is `CHOOSE`. This parameter is implemented in Advanced Server.

`QUERY_REWRITE_ENABLED (string)`

Valid values are `TRUE`, `FALSE`, and `FORCE`. The default is `FALSE`. This parameter is provided for syntax compatibility only and has no effect in Advanced Server.

`QUERY_REWRITE_INTEGRITY (string)`

Valid values are `ENFORCED`, `TRUSTED`, and `STALE_TOLERATED`. The default is `ENFORCED`. This parameter is provided for syntax compatibility only and has no effect in Advanced Server.

Examples

Set the language to U.S. English in UTF-8 encoding. Note that in this example, the value, `en_US.UTF-8`, is in the format that must be specified for Advanced Server. This form is not compatible with Oracle databases.

```
ALTER SESSION SET NLS_LANGUAGE = 'en_US.UTF-8';
```

Set the date display format.

```
ALTER SESSION SET NLS_DATE_FORMAT = 'dd/mm/yyyy';
```

2.3.10 ALTER TABLE

Name

`ALTER TABLE` -- change the definition of a table

Synopsis

```
ALTER TABLE name
  action [, ...]
ALTER TABLE name
  RENAME COLUMN column TO new_column
ALTER TABLE name
  RENAME TO new_name
```

where *action* is one of:

```
  ADD column type [ column_constraint [ ... ] ]
  DROP COLUMN column
  ADD table_constraint
  DROP CONSTRAINT constraint_name [ CASCADE ]
```

Description

`ALTER TABLE` changes the definition of an existing table. There are several subforms:

`ADD` *`column type`*

> This form adds a new column to the table using the same syntax as `CREATE TABLE`.

`DROP COLUMN`

> This form drops a column from a table. Indexes and table constraints involving the column will be automatically dropped as well.

`ADD` *`table_constraint`*

> This form adds a new constraint to a table using the same syntax as `CREATE TABLE`.

`DROP CONSTRAINT`

This form drops constraints on a table. Currently, constraints on tables are not required to have unique names, so there may be more than one constraint matching the specified name. All matching constraints will be dropped.

`RENAME`

The `RENAME` forms change the name of a table (or an index, sequence, or view) or the name of an individual column in a table. There is no effect on the stored data.

You must own the table to use `ALTER TABLE`.

Parameters

`name`

The name (possibly schema-qualified) of an existing table to alter.

`column`

Name of a new or existing column.

`new_column`

New name for an existing column.

`new_name`

New name for the table.

`type`

Data type of the new column.

`table_constraint`

New table constraint for the table.

`constraint_name`

Name of an existing constraint to drop.

`CASCADE`

Automatically drop objects that depend on the dropped constraint.

Notes

When you invoke `ADD COLUMN`, all existing rows in the table are initialized with the column's default value (null if no `DEFAULT` clause is specified). Adding a column with a non-null default will require the entire table to be rewritten. This may take a significant amount of time for a large table; and it will temporarily require double the disk space. Adding a `CHECK` or `NOT NULL` constraint requires scanning the table to verify that existing rows meet the constraint.

The `DROP COLUMN` form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated.

Changing any part of a system catalog table is not permitted. Refer to CREATE TABLE for a further description of valid parameters.

Examples

To add a column of type `VARCHAR2` to a table:

```
ALTER TABLE emp ADD address VARCHAR2(30);
```

To drop a column from a table:

```
ALTER TABLE emp DROP COLUMN address;
```

To rename an existing column:

```
ALTER TABLE emp RENAME COLUMN address TO city;
```

To rename an existing table:

```
ALTER TABLE emp RENAME TO employee;
```

To add a check constraint to a table:

```
ALTER TABLE emp ADD CONSTRAINT sal_chk CHECK (sal > 500);
```

To remove a check constraint from a table:

```
ALTER TABLE emp DROP CONSTRAINT sal_chk;
```

See Also

CREATE TABLE, DROP TABLE

2.3.11 ALTER TABLESPACE

Name

`ALTER TABLESPACE` -- change the definition of a tablespace

Synopsis

`ALTER TABLESPACE` *`name`* `RENAME TO` *`newname`*

Description

`ALTER TABLESPACE` changes the definition of a tablespace.

Parameters

`name`

The name of an existing tablespace.

`newname`

The new name of the tablespace. The new name cannot begin with `pg_`, as such names are reserved for system tablespaces.

Examples

Rename tablespace `empspace` to `employee_space`:

```
ALTER TABLESPACE empspace RENAME TO employee_space;
```

See Also

DROP TABLESPACE

2.3.12 ALTER USER… IDENTIFIED BY

Name

`ALTER USER` -- change a database user account

Synopsis

`ALTER USER` *`role_name`* `IDENTIFIED BY` *`password`* `REPLACE` *`prev_password`*

Description

A role without the `CREATEROLE` privilege may use this command to change their own password. An unprivileged role must include the `REPLACE` clause and their previous password if `PASSWORD_VERIFY_FUNCTION` is not `NULL` in their profile. When the `REPLACE` clause is used by a non-superuser, the server will compare the password provided to the existing password and raise an error if the passwords do not match.

A database superuser can use this command to change the password associated with any role. If a superuser includes the `REPLACE` clause, the clause is ignored; a non-matching value for the previous password will not throw an error.

If the role for which the password is being changed has the `SUPERUSER` attribute, then a superuser must issue this command. A role with the `CREATEROLE` attribute can use this command to change the password associated with a role that is not a superuser.

Parameters

`role_name`

The name of the role whose password is to be altered.

`password`

The role's new password.

`prev_password`

The role's previous password.

Examples

Change a user password:

```
ALTER USER john IDENTIFIED BY xyRP35z REPLACE 23PJ74a;
```

See Also

CREATE USER, DROP USER

2.3.13 ALTER USER|ROLE… PROFILE MANAGEMENT CLAUSES

Name

ALTER USER|ROLE

Synopsis

```
ALTER USER|ROLE name [[WITH] option[…]
```

where option can be the following compatible clauses:

```
  PROFILE profile_name
| ACCOUNT {LOCK|UNLOCK}
| PASSWORD EXPIRE [AT 'timestamp']
```

or option can be the following non-compatible clauses:

```
| PASSWORD SET AT 'timestamp'
| LOCK TIME 'timestamp'
| STORE PRIOR PASSWORD {'password' 'timestamp} [, ...]
```

For information about the administrative clauses of the ALTER USER or ALTER ROLE command that are supported by Advanced Server, please see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/sql-commands.html

Only a database superuser can use the ALTER USER|ROLE clauses that enforce profile management. The clauses enforce the following behaviors:

Include the PROFILE clause and a *profile_name* to associate a pre-defined profile with a role, or to change which pre-defined profile is associated with a user.

Include the ACCOUNT clause and the LOCK or UNLOCK keyword to specify that the user account should be placed in a locked or unlocked state.

Include the LOCK TIME '*timestamp*' clause and a date/time value to lock the role at the specified time, and unlock the role at the time indicated by the PASSWORD_LOCK_TIME parameter of the profile assigned to this role. If LOCK TIME is used with the ACCOUNT LOCK clause, the role can only be unlocked by a database superuser with the ACCOUNT UNLOCK clause.

Include the PASSWORD EXPIRE clause with the AT '*timestamp*' keywords to specify a date/time when the password associated with the role will expire. If you omit the AT '*timestamp*' keywords, the password will expire immediately.

Include the PASSWORD SET AT '*timestamp*' keywords to set the password modification date to the time specified.

Include the STORE PRIOR PASSWORD {'*password*' '*timestamp*} [, ...] clause to modify the password history, adding the new password and the time the password was set.

Each login role may only have one profile. To discover the profile that is currently associated with a login role, query the profile column of the DBA_USERS view.

Parameters

name

The name of the role with which the specified profile will be associated.

password

The password associated with the role.

profile_name

The name of the profile that will be associated with the role.

timestamp

The date and time at which the clause will be enforced. When specifying a value for *timestamp*, enclose the value in single-quotes.

Notes

For information about the Postgres-compatible clauses of the ALTER USER or ALTER ROLE command, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/sql-alterrole.html

Examples

The following command uses the ALTER USER… PROFILE command to associate a profile named acctg with a user named john:

```
ALTER USER john PROFILE acctg_profile;
```


The following command uses the `ALTER ROLE… PROFILE` command to associate a profile named `acctg` with a user named `john`:

```
ALTER ROLE john PROFILE acctg_profile;
```

2.3.14 CALL

Name

CALL

Synopsis

CALL *procedure_name* '('[*argument_list*]')'

Description

Use the CALL statement to invoke a procedure. To use the CALL statement, you must have EXECUTE privileges on the procedure that the CALL statement is invoking.

Parameters

procedure_name

procedure_name is the (optionally schema-qualified) procedure name.

argument_list

argument_list specifies a comma-separated list of arguments required by the procedure. Note that each member of *argument_list* corresponds to a formal argument expected by the procedure. Each formal argument may be an IN parameter, an OUT parameter, or an INOUT parameter.

Examples

The CALL statement may take one of several forms, depending on the arguments required by the procedure:

```
CALL update_balance();
CALL update_balance(1,2,3);
```

2.3.15 COMMENT

Name

COMMENT -- define or change the comment of an object

Synopsis

```
COMMENT ON
{
  TABLE table_name |
  COLUMN table_name.column_name
} IS 'text'
```

Description

COMMENT stores a comment about a database object. To modify a comment, issue a new COMMENT command for the same object. Only one comment string is stored for each object. To remove a comment, specify the empty string (two consecutive single quotes with no intervening space) for *text*. Comments are automatically dropped when the object is dropped.

Parameters

table_name

The name of the table to be commented. The table name may be schema-qualified.

table_name.column_name

The name of a column within *table_name* to be commented. The table name may be schema-qualified.

text

The new comment.

Notes

There is presently no security mechanism for comments: any user connected to a database can see all the comments for objects in that database (although only superusers can change comments for objects that they don't own). *Do not put security-critical information in comments.*

Examples

Attach a comment to the table `emp`:

```
COMMENT ON TABLE emp IS 'Current employee information';
```

Attach a comment to the `empno` column of the `emp` table:

```
COMMENT ON COLUMN emp.empno IS 'Employee identification number';
```

Remove these comments:

```
COMMENT ON TABLE emp IS '';
COMMENT ON COLUMN emp.empno IS '';
```

2.3.16 COMMIT

Name

`COMMIT` -- commit the current transaction

Synopsis

```
COMMIT [ WORK ]
```

Description

`COMMIT` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Parameters

`WORK`

Optional key word - has no effect.

Notes

Use ROLLBACK to abort a transaction. Issuing `COMMIT` when not inside a transaction does no harm.

Examples

To commit the current transaction and make all changes permanent:

```
COMMIT;
```

See Also

ROLLBACK, ROLLBACK TO SAVEPOINT

2.3.17 CREATE DATABASE

Name

`CREATE DATABASE` -- create a new database

Synopsis

`CREATE DATABASE` *`name`*

Description

`CREATE DATABASE` creates a new database.

To create a database, you must be a superuser or have the special `CREATEDB` privilege. Normally, the creator becomes the owner of the new database. Non-superusers with `CREATEDB` privilege can only create databases owned by them.

The new database will be created by cloning the standard system database `template1`.

Parameters

`name`

The name of the database to be created.

Notes

`CREATE DATABASE` cannot be executed inside a transaction block.

Errors along the line of "could not initialize database directory" are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems.

Examples

To create a new database:

```
CREATE DATABASE employees;
```


2.3.18 CREATE [PUBLIC] DATABASE LINK

Name

`CREATE [PUBLIC] DATABASE LINK` -- create a new database link.

Synopsis

```
CREATE [ PUBLIC ] DATABASE LINK name
  CONNECT TO { CURRENT_USER |
               username IDENTIFIED BY 'password'}
  USING { postgres_fdw 'fdw_connection_string' |
        [ oci ] 'oracle_connection_string' }
```

Description

`CREATE DATABASE LINK` creates a new database link. A database link is an object that allows a reference to a table or view in a remote database within a `DELETE`, `INSERT`, `SELECT` or `UPDATE` command. A database link is referenced by appending `@dblink` to the table or view name referenced in the SQL command where *`dblink`* is the name of the database link.

Database links can be public or private. A *public database link* is one that can be used by any user. A *private database link* can be used only by the database link's owner. Specification of the `PUBLIC` option creates a public database link. If omitted, a private database link is created.

When the `CREATE DATABASE LINK` command is given, the database link name and the given connection attributes are stored in the Advanced Server system table named, `pg_catalog.edb_dblink`. When using a given database link, the database containing the `edb_dblink` entry defining this database link is called the *local database*. The server and database whose connection attributes are defined within the `edb_dblink` entry is called the *remote database*

A SQL command containing a reference to a database link must be issued while connected to the local database. When the SQL command is executed, the appropriate authentication and connection is made to the remote database to access the table or view to which the `@dblink` reference is appended.

Note: A database link cannot be used to access a remote database within a standby database server. Standby database servers are used for high availability, load balancing, and replication.

For information about high availability, load balancing, and replication for Postgres database servers, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/high-availability.html

Note: For Advanced Server 10, the `CREATE DATABASE LINK` command is tested against and certified for use with Oracle version 10g Release 2 (10.2), Oracle version 11g Release 2 (11.2), and Oracle version 12c Release 1 (12.1).

Note: The `edb_dblink_oci.rescans` GUC can be set to `SCROLL` or `SERIALIZABLE` at the server level in `postgresql.conf` file. It can also be set at session level using the `SET` command, but the setting will not be applied to existing dblink connections due to dblink connection caching.

The `edb_dblink_oci` supports both types of rescans: `SCROLL` and `SERIALIZABLE`. By default it is set to `SERIALIZABLE`. When set to `SERIALIZABLE`, `edb_dblink_oci` uses the `SERIALIZABLE` transaction isolation level on the Oracle side, which corresponds to PostgreSQL's `REPEATABLE READ`:

- This is necessary as a single PostgreSQL statement can lead to multiple Oracle queries and thereby uses a serializable isolation level to provide consistent results.
- A serialization failure may occur due to a table modification concurrent with long-running DML transactions (for example `ADD`, `UPDATE`, or `DELETE` statements). If such a failure occurs, the OCI reports `ORA-08177: can't serialize access for this transaction`, and the application must retry the transaction.
- A `SCROLL` rescan will be quick, but with each iteration will reset the current row position to `1`. `SERIALIZABLE` rescan has performance benefits over a `SCROLL` rescan.

Parameters

`PUBLIC`

Create a public database link that can be used by any user. If omitted, then the database link is private and can only be used by the database link's owner.

`name`

The name of the database link.

`username`

The username to be used for connecting to the remote database.

`CURRENT_USER`

Include `CURRENT_USER` to specify that Advanced Server should use the user mapping associated with the role that is using the link when establishing a connection to the remote server.

`password`

The password for *`username`*.

`postgres_fdw`

Specifies foreign data wrapper `postgres_fdw` as the connection to a remote Advanced Server database. If `postgres_fdw` has not been installed on the database, use the `CREATE EXTENSION` command to install `postgres_fdw`. For more information, please see the `CREATE EXTENSION` command in the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/sql-createextension.html

`fdw_connection_string`

Specify the connection information for the `postgres_fdw` foreign data wrapper.

`oci`

Specifies a connection to a remote Oracle database. This is Advanced Server's default behavior.

`oracle_connection_string`

Specify the connection information for an oci connection.

Notes

To create a non-public database link you must have the `CREATE DATABASE LINK` privilege. To create a public database link you must have the `CREATE PUBLIC DATABASE LINK` privilege.

Setting up an Oracle Instant Client for oci-dblink

In order to use oci-dblink, an Oracle instant client must be downloaded and installed on the host running the Advanced Server database in which the database link is to be created.

An instant client can be downloaded from the following site:

http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html

Oracle Instant Client for Linux

The following instructions apply to Linux hosts running Advanced Server.

Be sure the `libaio` library (the Linux-native asynchronous I/O facility) has already been installed on the Linux host running Advanced Server.

The `libaio` library can be installed with the following command:

```
yum install libaio
```

If the Oracle instant client that you've downloaded does not include the file specifically named `libclntsh.so` without a version number suffix, you must create a symbolic link named `libclntsh.so` that points to the downloaded version of the library file. Navigate to the instant client directory and execute the following command:

```
ln -s libclntsh.so.version libclntsh.so
```

Where *`version`* is the version number of the `libclntsh.so` library. For example:

```
ln -s libclntsh.so.12.1 libclntsh.so
```

When you are executing a SQL command that references a database link to a remote Oracle database, Advanced Server must know where the Oracle instant client library resides on the Advanced Server host.

The `LD_LIBRARY_PATH` environment variable must include the path to the Oracle client installation directory containing the `libclntsh.so` file. For example, assuming the installation directory containing `libclntsh.so` is `/tmp/instantclient`:

```
export LD_LIBRARY_PATH=/tmp/instantclient:$LD_LIBRARY_PATH
```

Note: This `LD_LIBRARY_PATH` environment variable setting must be in effect when the `pg_ctl` utility is executed to start or restart Advanced Server.

If you are running the current session as the user account (for example, `enterprisedb`) that will directly invoke `pg_ctl` to start or restart Advanced Server, then be sure to set `LD_LIBRARY_PATH` before invoking `pg_ctl`.

You can set `LD_LIBRARY_PATH` within the `.bash_profile` file under the home directory of the `enterprisedb` user account (that is, set `LD_LIBRARY_PATH` within file

`~enterprisedb/.bash_profile`). In this manner, `LD_LIBRARY_PATH` will be set when you log in as `enterprisedb`.

If however, you are using a Linux service script with the `systemctl` or `service` command to start or restart Advanced Server, `LD_LIBRARY_PATH` must be set within the service script so it is in effect when the script invokes the `pg_ctl` utility.

The particular script file that needs to be modified to include the `LD_LIBRARY_PATH` setting depends upon the Advanced Server version, the Linux system on which it was installed, and whether it was installed with the graphical installer or an RPM package.

See the appropriate version of the *EDB Postgres Advanced Server Installation Guide* to determine the service script that affects the startup environment. The installation guides can be found at the following location:

https://www.enterprisedb.com/resources/product-documentation

Oracle Instant Client for Windows

The following instructions apply to Windows hosts running Advanced Server.

When you are executing a SQL command that references a database link to a remote Oracle database, Advanced Server must know where the Oracle instant client library resides on the Advanced Server host.

Set the Windows `PATH` system environment variable to include the Oracle client installation directory that contains the `oci.dll` file.

As an alternative you, can set the value of the `oracle_home` configuration parameter in the `postgresql.conf` file. The value specified in the `oracle_home` configuration parameter will override the Windows `PATH` environment variable.

To set the `oracle_home` configuration parameter in the `postgresql.conf` file, edit the file, adding the following line:

```
oracle_home = 'lib_directory'
```

Substitute the name of the Windows directory that contains `oci.dll` for *`lib_directory`*. For example:

```
oracle_home = 'C:/tmp/instantclient_10_2'
```

After setting the `PATH` environment variable or the `oracle_home` configuration parameter, you must restart the server for the changes to take effect. Restart the server from the Windows Services console.

Examples

Creating an oci-dblink Database Link

The following example demonstrates using the `CREATE DATABASE LINK` command to create a database link (named `chicago`) that connects an instance of Advanced Server to an Oracle server via an oci-dblink connection. The connection information tells Advanced Server to log in to Oracle as user `admin`, whose password is `mypassword`. Including the `oci` option tells Advanced Server that this is an oci-dblink connection; the connection string, `'//127.0.0.1/acctg'` specifies the server address and name of the database.

```
CREATE DATABASE LINK chicago
  CONNECT TO admin IDENTIFIED BY 'mypassword'
  USING oci '//127.0.0.1/acctg';
```

Note: You can specify a hostname in the connection string (in place of an IP address).

Creating a postgres_fdw Database Link

The following example demonstrates using the `CREATE DATABASE LINK` command to create a database link (named `bedford`) that connects an instance of Advanced Server to another Advanced Server instance via a `postgres_fdw` foreign data wrapper connection. The connection information tells Advanced Server to log in as user `admin`, whose password is `mypassword`. Including the `postgres_fdw` option tells Advanced Server that this is a postgres_fdw connection; the connection string, `'host=127.0.0.1 port=5444 dbname=marketing'` specifies the server address and name of the database.

```
CREATE DATABASE LINK bedford
  CONNECT TO admin IDENTIFIED BY 'mypassword'
  USING postgres_fdw 'host=127.0.0.1 port=5444 dbname=marketing';
```

Note: You can specify a hostname in the connection string (in place of an IP address).

Using a Database Link

The following examples demonstrate using a database link with Advanced Server to connect to an Oracle database. The examples assume that a copy of the Advanced Server sample application's `emp` table has been created in an Oracle database and a second Advanced Server database cluster with the sample application is accepting connections at port 5443.

Create a public database link named, `oralink`, to an Oracle database named, `xe`, located at 127.0.0.1 on port 1521. Connect to the Oracle database with username, `edb`, and password, `password`.

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password'
USING '//127.0.0.1:1521/xe';
```

Issue a `SELECT` command on the `emp` table in the Oracle database using database link, `oralink`.

```
SELECT * FROM emp@oralink;

 empno | ename  |    job    | mgr  |      hiredate       | sal  | comm | deptno
-------+--------+-----------+------+---------------------+------+------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800 |      |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600 |  300 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250 |  500 |     30
  7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975 |      |     20
  7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250 | 1400 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850 |      |     30
  7782 | CLARK  | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450 |      |     10
  7788 | SCOTT  | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000 |      |     20
  7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000 |      |     10
  7844 | TURNER | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500 |    0 |     30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100 |      |     20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950 |      |     30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000 |      |     20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300 |      |     10
(14 rows)
```

Create a private database link named, `fdwlink`, to the Advanced Server database named, `edb`, located on host 192.168.2.22 running on port 5444. Connect to the Advanced Server database with username, `enterprisedb`, and password, `password`.

```
CREATE DATABASE LINK fdwlink CONNECT TO enterprisedb IDENTIFIED BY 'password'
USING postgres_fdw 'host=192.168.2.22 port=5444 dbname=edb';
```

Display attributes of database links, `oralink` and `fdwlink`, from the local `edb_dblink` system table:

```
SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;

 lnkname |   lnkuser    |               lnkconnstr
---------+--------------+----------------------------------------
 oralink | edb          | //127.0.0.1:1521/xe
 fdwlink | enterprisedb |
(2 rows)
```

Perform a join of the `emp` table from the Oracle database with the `dept` table from the Advanced Server database:

```
SELECT d.deptno, d.dname, e.empno, e.ename, e.job, e.sal, e.comm FROM
emp@oralink e, dept@fdwlink d WHERE e.deptno = d.deptno ORDER BY 1, 3;

 deptno |   dname    | empno | ename  |    job    | sal  | comm
--------+------------+-------+--------+-----------+------+------
     10 | ACCOUNTING |  7782 | CLARK  | MANAGER   | 2450 |
     10 | ACCOUNTING |  7839 | KING   | PRESIDENT | 5000 |
     10 | ACCOUNTING |  7934 | MILLER | CLERK     | 1300 |
     20 | RESEARCH   |  7369 | SMITH  | CLERK     |  800 |
     20 | RESEARCH   |  7566 | JONES  | MANAGER   | 2975 |
     20 | RESEARCH   |  7788 | SCOTT  | ANALYST   | 3000 |
```

```
     20 | RESEARCH   |  7876 | ADAMS  | CLERK     | 1100 |
     20 | RESEARCH   |  7902 | FORD   | ANALYST   | 3000 |
     30 | SALES      |  7499 | ALLEN  | SALESMAN  | 1600 |  300
     30 | SALES      |  7521 | WARD   | SALESMAN  | 1250 |  500
     30 | SALES      |  7654 | MARTIN | SALESMAN  | 1250 | 1400
     30 | SALES      |  7698 | BLAKE  | MANAGER   | 2850 |
     30 | SALES      |  7844 | TURNER | SALESMAN  | 1500 |    0
     30 | SALES      |  7900 | JAMES  | CLERK     |  950 |
(14 rows)
```

Pushdown for an oci Database Link

When the oci-dblink is used to execute SQL statements on a remote Oracle database, there are certain circumstances where pushdown of the processing occurs on the foreign server.

Pushdown refers to the occurrence of processing on the foreign (that is, the remote) server instead of the local client where the SQL statement was issued. Pushdown can result in performance improvement since the data is processed on the remote server before being returned to the local client.

Pushdown applies to statements with the standard SQL join operations (inner join, left outer join, right outer join, and full outer join). Pushdown still occurs even when a sort is specified on the resulting data set.

In order for pushdown to occur, certain basic conditions must be met. The tables involved in the join operation must belong to the same foreign server and use the identical connection information to the foreign server (that is, the same database link defined with the `CREATE DATABASE LINK` command).

In order to determine if pushdown is to be used for a SQL statement, display the execution plan by using the `EXPLAIN` command.

For information about the `EXPLAIN` command, please see the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/sql-explain.html

The following examples use the database link created as shown by the following:

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password'
USING '//192.168.2.23:1521/xe';
```

The following example shows the execution plan of an inner join:

```
EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM
dept@oralink d, emp@oralink e WHERE d.deptno = e.deptno ORDER BY 1, 3;

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Foreign Scan
```

```
   Output: d.deptno, d.dname, e.empno, e.ename
   Relations: (_dblink_dept_1 d) INNER JOIN (_dblink_emp_2 e)
   Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM (dept r1 INNER
JOIN emp r2 ON ((r1.deptno = r2.deptno))) ORDER BY r1.deptno ASC NULLS LAST, r2.empno
ASC NULLS LAST
(4 rows)
```

Note that the `INNER JOIN` operation occurs under the Foreign Scan section. The output of this join is the following:

```
 deptno |   dname    | empno | ename
--------+------------+-------+--------
     10 | ACCOUNTING |  7782 | CLARK
     10 | ACCOUNTING |  7839 | KING
     10 | ACCOUNTING |  7934 | MILLER
     20 | RESEARCH   |  7369 | SMITH
     20 | RESEARCH   |  7566 | JONES
     20 | RESEARCH   |  7788 | SCOTT
     20 | RESEARCH   |  7876 | ADAMS
     20 | RESEARCH   |  7902 | FORD
     30 | SALES      |  7499 | ALLEN
     30 | SALES      |  7521 | WARD
     30 | SALES      |  7654 | MARTIN
     30 | SALES      |  7698 | BLAKE
     30 | SALES      |  7844 | TURNER
     30 | SALES      |  7900 | JAMES
(14 rows)
```

The following shows the execution plan of a left outer join:

```
EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM
dept@oralink d LEFT OUTER JOIN emp@oralink e ON d.deptno = e.deptno ORDER BY 1, 3;

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Foreign Scan
   Output: d.deptno, d.dname, e.empno, e.ename
   Relations: (_dblink_dept_1 d) LEFT JOIN (_dblink_emp_2 e)
   Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM (dept r1 LEFT JOIN
emp r2 ON ((r1.deptno = r2.deptno))) ORDER BY r1.deptno ASC NULLS LAST, r2.empno ASC
NULLS LAST
(4 rows)
```

The output of this join is the following:

```
 deptno |   dname    | empno | ename
--------+------------+-------+--------
     10 | ACCOUNTING |  7782 | CLARK
     10 | ACCOUNTING |  7839 | KING
     10 | ACCOUNTING |  7934 | MILLER
     20 | RESEARCH   |  7369 | SMITH
     20 | RESEARCH   |  7566 | JONES
     20 | RESEARCH   |  7788 | SCOTT
     20 | RESEARCH   |  7876 | ADAMS
     20 | RESEARCH   |  7902 | FORD
     30 | SALES      |  7499 | ALLEN
     30 | SALES      |  7521 | WARD
     30 | SALES      |  7654 | MARTIN
     30 | SALES      |  7698 | BLAKE
     30 | SALES      |  7844 | TURNER
     30 | SALES      |  7900 | JAMES
     40 | OPERATIONS |       |
(15 rows)
```

The following example shows a case where the entire processing is not pushed down because the `emp` joined table resides locally instead of on the same foreign server.

```
EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM
dept@oralink d LEFT OUTER JOIN emp e ON d.deptno = e.deptno ORDER BY 1, 3;

                           QUERY PLAN
-----------------------------------------------------------------
 Sort
   Output: d.deptno, d.dname, e.empno, e.ename
   Sort Key: d.deptno, e.empno
   ->  Hash Left Join
         Output: d.deptno, d.dname, e.empno, e.ename
         Hash Cond: (d.deptno = e.deptno)
         ->  Foreign Scan on _dblink_dept_1 d
               Output: d.deptno, d.dname, d.loc
               Remote Query: SELECT deptno, dname, NULL FROM dept
         ->  Hash
               Output: e.empno, e.ename, e.deptno
               ->  Seq Scan on public.emp e
                     Output: e.empno, e.ename, e.deptno
(13 rows)
```

The output of this join is the same as the previous left outer join example.

Creating a Foreign Table from a Database Link

Note: The procedure described in this section is not compatible with Oracle databases.

After you have created a database link, you can create a foreign table based upon this database link. The foreign table can then be used to access the remote table referencing it with the foreign table name instead of using the database link syntax. Using the database link requires appending `@`*`dblink`* to the table or view name referenced in the SQL command where *`dblink`* is the name of the database link.

This technique can be used for either an oci-dblink connection for remote Oracle access, or a postgres_fdw connection for remote Postgres access.

The following example shows the creation of a foreign table to access a remote Oracle table.

First, create a database link as previously described. The following is the creation of a database link named `oralink` for connecting to the Oracle database.

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password'
USING '//127.0.0.1:1521/xe';
```

The following query shows the database link:

```
SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;

 lnkname | lnkuser |     lnkconnstr
---------+---------+---------------------
 oralink | edb     | //127.0.0.1:1521/xe
```

```
(1 row)
```

When you create the database link, Advanced Server creates a corresponding foreign server. The following query displays the foreign server:

```
SELECT srvname, srvowner, srvfdw, srvtype, srvoptions FROM pg_foreign_server;

 srvname | srvowner | srvfdw | srvtype |          srvoptions
---------+----------+--------+---------+------------------------------
 oralink |       10 |  14005 |         | {connstr=//127.0.0.1:1521/xe}
(1 row)
```

For more information about foreign servers, please see the `CREATE SERVER` command in the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/sql-createserver.html

Create the foreign table as shown by the following:

```
CREATE FOREIGN TABLE emp_ora (
    empno           NUMERIC(4),
    ename           VARCHAR(10),
    job             VARCHAR(9),
    mgr             NUMERIC(4),
    hiredate        TIMESTAMP WITHOUT TIME ZONE,
    sal             NUMERIC(7,2),
    comm            NUMERIC(7,2),
    deptno          NUMERIC(2)
)
  SERVER oralink
  OPTIONS (table_name 'emp', schema_name 'edb'
);
```

Note the following in the `CREATE FOREIGN TABLE` command:

- The name specified in the `SERVER` clause at the end of the `CREATE FOREIGN TABLE` command is the name of the foreign server, which is `oralink` in this example as displayed in the `srvname` column from the query on `pg_foreign_server`.
- The table name and schema name are specified in the `OPTIONS` clause by the `table` and `schema` options.
- The column names specified in the `CREATE FOREIGN TABLE` command must match the column names in the remote table.
- Generally, `CONSTRAINT` clauses may not be accepted or enforced on the foreign table as they are assumed to have been defined on the remote table.

For more information about the `CREATE FOREIGN TABLE` command, please see the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/sql-createforeigntable.html

The following is a query on the foreign table:

```
SELECT * FROM emp_ora;

 empno | ename  |    job    | mgr  |      hiredate       |   sal   |  comm   | deptno
-------+--------+-----------+------+---------------------+---------+---------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800.00 |         |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600.00 |  300.00 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250.00 |  500.00 |     30
  7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975.00 |         |     20
  7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850.00 |         |     30
  7782 | CLARK  | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450.00 |         |     10
  7788 | SCOTT  | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000.00 |         |     20
  7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000.00 |         |     10
  7844 | TURNER | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500.00 |    0.00 |     30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100.00 |         |     20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950.00 |         |     30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000.00 |         |     20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |         |     10
(14 rows)
```

In contrast, the following is a query on the same remote table, but using the database link instead of the foreign table:

```
SELECT * FROM emp@oralink;

 empno | ename  |    job    | mgr  |      hiredate       | sal  | comm | deptno
-------+--------+-----------+------+---------------------+------+------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800 |      |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600 |  300 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250 |  500 |     30
  7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975 |      |     20
  7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250 | 1400 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850 |      |     30
  7782 | CLARK  | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450 |      |     10
  7788 | SCOTT  | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000 |      |     20
  7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000 |      |     10
  7844 | TURNER | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500 |    0 |     30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100 |      |     20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950 |      |     30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000 |      |     20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300 |      |     10
(14 rows)
```

Note: For backward compatibility reasons, it is still possible to write `USING libpq` rather than `USING postgres_fdw`. However, the `libpq` connector is missing many important optimizations which are present in the `postgres_fdw` connector. Therefore, the `postgres_fdw` connector should be used whenever possible. The `libpq` option is deprecated and may be removed entirely in a future Advanced Server release.

See Also

DROP DATABASE LINK

2.3.19 CREATE DIRECTORY

Name

`CREATE DIRECTORY` -- create an alias for a file system directory path

Synopsis

```
CREATE DIRECTORY name AS 'pathname'
```

Description

The `CREATE DIRECTORY` command creates an alias for a file system directory pathname. You must be a database superuser to use this command.

When the alias is specified as the appropriate parameter to the programs of the `UTL_FILE` package, the operating system files are created in, or accessed from the directory corresponding to the given alias.

Parameters

name

The directory alias name.

pathname

The fully-qualified directory path represented by the alias name. The `CREATE DIRECTORY` command does not create the operating system directory. The physical directory must be created independently using the appropriate operating system commands.

Notes

The operating system user id, `enterprisedb`, must have the appropriate read and/or write privileges on the directory if the `UTL_FILE` package is to be used to create and/or read files using the directory.

The directory alias is stored in the `pg_catalog.edb_dir` system catalog table. Note that `edb_dir` is not a table compatible with Oracle databases.

The directory alias can also be viewed from the Oracle catalog views `SYS.ALL_DIRECTORIES` and `SYS.DBA_DIRECTORIES`, which are compatible with Oracle databases.

Use the `DROP DIRECTORY` command to delete the directory alias. When a directory alias is deleted, the corresponding physical file system directory is not affected. The file system directory must be deleted using the appropriate operating system commands.

In a Linux system, the directory name separator is a forward slash (`/`).

In a Windows system, the directory name separator can be specified as a forward slash (`/`) or two consecutive backslashes (`\\`).

Examples

Create an alias named `empdir` for directory `/tmp/empdir` on Linux:

```
CREATE DIRECTORY empdir AS '/tmp/empdir';
```

Create an alias named `empdir` for directory `C:\TEMP\EMPDIR` on Windows:

```
CREATE DIRECTORY empdir AS 'C:/TEMP/EMPDIR';
```

View all of the directory aliases:

```
SELECT * FROM pg_catalog.edb_dir;

dirname | dirowner |    dirpath     | diracl
---------+----------+----------------+--------
 empdir  |       10 | C:/TEMP/EMPDIR |
(1 row)
```

View the directory aliases using a view compatible with Oracle databases:

```
SELECT * FROM SYS.ALL_DIRECTORIES;

    owner     | directory_name | directory_path
--------------+----------------+----------------
 ENTERPRISEDB | EMPDIR         | C:/TEMP/EMPDIR
(1 row)
```

See Also

DROP DIRECTORY

2.3.20 CREATE FUNCTION

Name

`CREATE FUNCTION` -- define a new function

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION name [ (parameters) ]
  RETURN data_type
   [
          IMMUTABLE
        | STABLE
        | VOLATILE
        | DETERMINISTIC
        | [ NOT ] LEAKPROOF
        | CALLED ON NULL INPUT
        | RETURNS NULL ON NULL INPUT
        | STRICT
        | [ EXTERNAL ] SECURITY INVOKER
        | [ EXTERNAL ] SECURITY DEFINER
        | AUTHID DEFINER
        | AUTHID CURRENT_USER
        | PARALLEL { UNSAFE | RESTRICTED | SAFE }
        | COST execution_cost
        | ROWS result_rows
        | SET configuration_parameter
          { TO value | = value | FROM CURRENT }
   ...]
{ IS | AS }
    [ declarations ]
  BEGIN
    statements
  END [ name ];
```

Description

`CREATE FUNCTION` defines a new function. `CREATE OR REPLACE FUNCTION` will either create a new function, or replace an existing definition.

If a schema name is included, then the function is created in the specified schema. Otherwise it is created in the current schema. The name of the new function must not match any existing function with the same argument types in the same schema. However, functions of different input argument types may share a name (this is called overloading). (Overloading of functions is an Advanced Server feature - overloading of stored functions is not compatible with Oracle databases.)

To update the definition of an existing function, use `CREATE OR REPLACE FUNCTION`. It is not possible to change the name or argument types of a function this way (if you tried, you would actually be creating a new, distinct function). Also, `CREATE OR REPLACE FUNCTION` will not let you change the return type of an existing function. To do that, you must drop and recreate the function.

The user that creates the function becomes the owner of the function.

Parameters

name

> *name* is the identifier of the function. If you specify the `[OR REPLACE]` clause and a function with the same name already exists in the schema, the new function will replace the existing one. If you do not specify `[OR REPLACE]`, the new function will not replace the existing function with the same name in the same schema.

parameters

> *parameters* is a list of formal parameters.

data_type

> *data_type* is the data type of the value returned by the function's `RETURN` statement.

declarations

> *declarations* are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations.

statements

> *statements* are SPL program statements (the `BEGIN` - `END` block may contain an `EXCEPTION` section).

`IMMUTABLE`
`STABLE`
`VOLATILE`

> These attributes inform the query optimizer about the behavior of the function; you can specify only one choice. `VOLATILE` is the default behavior.

IMMUTABLE indicates that the function cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the function with all-constant arguments can be immediately replaced with the function value.

STABLE indicates that the function cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for function that depend on database lookups, parameter variables (such as the current time zone), etc.

VOLATILE indicates that the function value can change even within a single table scan, so no optimizations can be made. Please note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away.

```
DETERMINISTIC
```

DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC function cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the function with all-constant arguments can be immediately replaced with the function value.

```
 [ NOT ] LEAKPROOF
```

A LEAKPROOF function has no side effects, and reveals no information about the values used to call the function.

```
CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT
```

CALLED ON NULL INPUT (the default) indicates that the procedure will be called normally when some of its arguments are NULL. It is the author's responsibility to check for NULL values if necessary and respond appropriately.

RETURNS NULL ON NULL INPUT or STRICT indicates that the procedure always returns NULL whenever any of its arguments are NULL. If these clauses are specified, the procedure is not executed when there are NULL arguments; instead a NULL result is assumed automatically.

```
[ EXTERNAL ] SECURITY DEFINER
```


SECURITY DEFINER specifies that the function will execute with the privileges of the user that created it; this is the default. The key word EXTERNAL is allowed for SQL conformance, but is optional.

```
[ EXTERNAL ] SECURITY INVOKER
```

The SECURITY INVOKER clause indicates that the function will execute with the privileges of the user that calls it. The key word EXTERNAL is allowed for SQL conformance, but is optional.

```
AUTHID DEFINER
AUTHID CURRENT_USER
```

The AUTHID DEFINER clause is a synonym for [EXTERNAL] SECURITY DEFINER. If the AUTHID clause is omitted or if AUTHID DEFINER is specified, the rights of the function owner are used to determine access privileges to database objects.

The AUTHID CURRENT_USER clause is a synonym for [EXTERNAL] SECURITY INVOKER. If AUTHID CURRENT_USER is specified, the rights of the current user executing the function are used to determine access privileges.

```
PARALLEL { UNSAFE | RESTRICTED | SAFE }
```

The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

When set to UNSAFE, the function cannot be executed in parallel mode. The presence of such a function in a SQL statement forces a serial execution plan. This is the default setting if the PARALLEL clause is omitted.

When set to RESTRICTED, the function can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.

When set to SAFE, the function can be executed in parallel mode with no restriction.

```
COST execution_cost
```

execution_cost is a positive number giving the estimated execution cost for the function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

ROWS *result_rows*

> *result_rows* is a positive number giving the estimated number of rows that the planner should expect the function to return. This is only allowed when the function is declared to return a set. The default assumption is 1000 rows.

SET *configuration_parameter* { TO *value* | = *value* | FROM CURRENT }

> The SET clause causes the specified configuration parameter to be set to the specified value when the function is entered, and then restored to its prior value when the function exits. SET FROM CURRENT saves the session's current value of the parameter as the value to be applied when the function is entered.
>
> If a SET clause is attached to a function, then the effects of a SET LOCAL command executed inside the function for the same variable are restricted to the function; the configuration parameter's prior value is restored at function exit. An ordinary SET command (without LOCAL) overrides the SET clause, much as it would do for a previous SET LOCAL command, with the effects of such a command persisting after procedure exit, unless the current transaction is rolled back.

Please Note: The STRICT, LEAKPROOF, PARALLEL, COST, ROWS and SET keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Notes

Advanced Server allows function overloading; that is, the same name can be used for several different functions so long as they have distinct input (IN, IN OUT) argument data types.

Examples

The function emp_comp takes two numbers as input and returns a computed value. The SELECT command illustrates use of the function.

```
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal           NUMBER,
    p_comm          NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;

SELECT ename "Name", sal "Salary", comm "Commission", emp_comp(sal, comm)
    "Total Compensation"  FROM emp;

  Name  | Salary  | Commission | Total Compensation
--------+---------+------------+--------------------
 SMITH  |  800.00 |            |           19200.00
```

```
 ALLEN  | 1600.00 |     300.00 |           45600.00
 WARD   | 1250.00 |     500.00 |           42000.00
 JONES  | 2975.00 |            |           71400.00
 MARTIN | 1250.00 |    1400.00 |           63600.00
 BLAKE  | 2850.00 |            |           68400.00
 CLARK  | 2450.00 |            |           58800.00
 SCOTT  | 3000.00 |            |           72000.00
 KING   | 5000.00 |            |          120000.00
 TURNER | 1500.00 |       0.00 |           36000.00
 ADAMS  | 1100.00 |            |           26400.00
 JAMES  |  950.00 |            |           22800.00
 FORD   | 3000.00 |            |           72000.00
 MILLER | 1300.00 |            |           31200.00
(14 rows)
```

Function `sal_range` returns a count of the number of employees whose salary falls in the specified range. The following anonymous block calls the function a number of times using the arguments' default values for the first two calls.

```
CREATE OR REPLACE FUNCTION sal_range (
    p_sal_min       NUMBER DEFAULT 0,
    p_sal_max       NUMBER DEFAULT 10000
) RETURN INTEGER
IS
    v_count         INTEGER;
BEGIN
    SELECT COUNT(*) INTO v_count FROM emp
        WHERE sal BETWEEN p_sal_min AND p_sal_max;
    RETURN v_count;
END;

BEGIN
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary: ' ||
        sal_range);
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary of at least '
        || '$2000.00: ' || sal_range(2000.00));
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary between '
        || '$2000.00 and $3000.00: ' || sal_range(2000.00, 3000.00));

END;

Number of employees with a salary: 14
Number of employees with a salary of at least $2000.00: 6
Number of employees with a salary between $2000.00 and $3000.00: 5
```

The following example demonstrates using the `AUTHID CURRENT_USER` clause and `STRICT` keyword in a function declaration:

```
CREATE OR REPLACE FUNCTION dept_salaries(dept_id int) RETURN NUMBER
  STRICT
  AUTHID CURRENT_USER
BEGIN
  RETURN QUERY (SELECT sum(salary) FROM emp WHERE deptno = id);
END;
```

Include the `STRICT` keyword to instruct the server to return `NULL` if any input parameter passed is `NULL`; if a `NULL` value is passed, the function will not execute.

The `dept_salaries` function executes with the privileges of the role that is calling the function. If the current user does not have sufficient privileges to perform the `SELECT` statement querying the `emp` table (to display employee salaries), the function will report an error. To instruct the server to use the privileges associated with the role that defined the function, replace the `AUTHID CURRENT_USER` clause with the `AUTHID DEFINER` clause.

Pragmas

`PRAGMA RESTRICT_REFERENCE`

> Advanced Server accepts but ignores syntax referencing `PRAGMA RESTRICT_REFERENCE`.

See Also DROP FUNCTION

2.3.21 CREATE INDEX

Name

`CREATE INDEX` -- define a new index

Synopsis

```
CREATE [ UNIQUE ] INDEX name ON table
  ( { column | ( expression ) } )
  [ TABLESPACE tablespace ]
```

Description

`CREATE INDEX` constructs an index, *`name`*, on the specified table. Indexes are primarily used to enhance database performance (though inappropriate use will result in slower performance).

Note: An index cannot be created on a partitioned table.

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified to create multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on `UPPER(col)` would allow the clause `WHERE UPPER(col) = 'JIM'` to use an index.

Advanced Server provides the B-tree index method. The B-tree index method is an implementation of Lehman-Yao high-concurrency B-trees.

Indexes are not used for `IS NULL` clauses by default.

All functions and operators used in an index definition must be "immutable", that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression remember to mark the function immutable when you create it.

Parameters

`UNIQUE`

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

name

The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table.

table

The name (possibly schema-qualified) of the table to be indexed.

column

The name of a column in the table.

expression

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses may be omitted if the expression has the form of a function call.

tablespace

The tablespace in which to create the index. If not specified, `default_tablespace` is used, or the database's default tablespace if `default_tablespace` is an empty string.

Notes

Up to 32 fields may be specified in a multicolumn index.

Examples

To create a B-tree index on the column, `ename`, in the table, `emp`:

```
CREATE INDEX name_idx ON emp (ename);
```

To create the same index as above, but have it reside in the `index_tblspc` tablespace:

```
CREATE INDEX name_idx ON emp (ename) TABLESPACE index_tblspc;
```

See Also

DROP INDEX, ALTER INDEX

2.3.22 CREATE MATERIALIZED VIEW

Name

`CREATE MATERIALIZED VIEW` -- define a new materialized view

Synopsis

```
CREATE MATERIALIZED VIEW name
   [build_clause][create_mv_refresh] AS subquery
```

Where *build_clause* is:

```
BUILD {IMMEDIATE | DEFERRED}
```

Where *create_mv_refresh* is:

```
REFRESH [COMPLETE] [ON DEMAND]
```

Description

`CREATE MATERIALIZED VIEW` defines a view of a query that is not updated each time the view is referenced in a query. By default, the view is populated when the view is created; you can include the `BUILD DEFERRED` keywords to delay the population of the view.

A materialized view may be schema-qualified; if you specify a schema name when invoking the `CREATE MATERIALIZED VIEW` command, the view will be created in the specified schema. The view name must be distinct from the name of any other view, table, sequence, or index in the same schema.

Parameters

name

The name (optionally schema-qualified) of a view to be created.

subquery

A `SELECT` statement that specifies the contents of the view. Refer to `SELECT` for more information about valid queries.

build_clause

Include a *build_clause* to specify when the view should be populated. Specify BUILD IMMEDIATE, or BUILD DEFERRED:

- BUILD IMMEDIATE instructs the server to populate the view immediately. This is the default behavior.
- BUILD DEFERRED instructs the server to populate the view at a later time (during a REFRESH operation).

create_mv_refresh

Include the *create_mv_refresh* clause to specify when the contents of a materialized view should be updated. The clause contains the REFRESH keyword followed by COMPLETE and/or ON DEMAND, where:

- COMPLETE instructs the server to discard the current content and reload the materialized view by executing the view's defining query when the materialized view is refreshed.
- ON DEMAND instructs the server to refresh the materialized view on demand by calling the DBMS_MVIEW package or by calling the Postgres REFRESH MATERIALIZED VIEW statement. This is the default behavior.

Notes

Materialized views are read only - the server will not allow an INSERT, UPDATE, or DELETE on a view.

Access to tables referenced in the view is determined by permissions of the view owner; the user of a view must have permissions to call all functions used by the view.

For more information about the Postgres REFRESH MATERIALIZED VIEW command, please see the PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/10/static/sql-refreshmaterializedview.html

Examples

The following statement creates a materialized view named dept_30:

```
CREATE MATERIALIZED VIEW dept_30 BUILD IMMEDIATE AS SELECT * FROM emp WHERE
deptno = 30;
```

The view contains information retrieved from the `emp` table about any employee that works in department `30`.

2.3.23 CREATE PACKAGE

Name

`CREATE PACKAGE` -- define a new package specification

Synopsis

```
CREATE [ OR REPLACE ] PACKAGE name
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
  [ declaration; ] [, ...]
  [ { PROCEDURE proc_name
      [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
        [, ...]) ];
      [ PRAGMA RESTRICT_REFERENCES(name,
        { RNDS | RNPS | TRUST | WNDS | WNPS } [, ... ] ); ]
    |
      FUNCTION func_name
      [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
        [, ...]) ]
      RETURN rettype [ DETERMINISTIC ];
      [ PRAGMA RESTRICT_REFERENCES(name,
        { RNDS | RNPS | TRUST | WNDS | WNPS } [, ... ] ); ]
    }
  ] [, ...]
  END [ name ]
```

Description

`CREATE PACKAGE` defines a new package specification. `CREATE OR REPLACE PACKAGE` will either create a new package specification, or replace an existing specification.

If a schema name is included, then the package is created in the specified schema. Otherwise it is created in the current schema. The name of the new package must not match any existing package in the same schema unless the intent is to update the definition of an existing package, in which case use `CREATE OR REPLACE PACKAGE`.

The user that creates the procedure becomes the owner of the package.

Parameters

name

The name (optionally schema-qualified) of the package to create.

`DEFINER | CURRENT_USER`

Specifies whether the privileges of the package owner (`DEFINER`) or the privileges of the current user executing a program in the package (`CURRENT_USER`) are to be used to determine whether or not access is allowed to database objects referenced in the package. `DEFINER` is the default.

`declaration`

A public variable, type, cursor, or `REF CURSOR` declaration.

`proc_name`

The name of a public procedure.

`argname`

The name of an argument.

`IN | IN OUT | OUT`

The argument mode.

`argtype`

The data type(s) of the program's arguments.

`DEFAULT` *`value`*

Default value of an input argument.

`func_name`

The name of a public function.

`rettype`

The return data type.

`DETERMINISTIC`

`DETERMINISTIC` is a synonym for `IMMUTABLE`. A `DETERMINISTIC` procedure cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.

```
RNDS | RNPS | TRUST | WNDS | WNPS
```

The keywords are accepted for compatibility and ignored.

Examples

The package specification, `empinfo`, contains three public components - a public variable, a public procedure, and a public function.

```
CREATE OR REPLACE PACKAGE empinfo
IS
    emp_name        VARCHAR2(10);
    PROCEDURE get_name (
        p_empno     NUMBER
    );
    FUNCTION display_counter
    RETURN INTEGER;
END;
```

See Also

DROP PACKAGE

2.3.24 CREATE PACKAGE BODY

Name

`CREATE BODY PACKAGE` -- define a new package body

Synopsis

```
CREATE [ OR REPLACE ] PACKAGE BODY name
{ IS | AS }
  [ declaration; ] [, ...]
  [ { PROCEDURE proc_name
      [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
        [, ...]) ]
      [ STRICT ]
      [ LEAKPROOF ]
      [ PARALLEL { UNSAFE | RESTRICTED | SAFE } ]
      [ COST execution_cost ]
      [ ROWS result_rows ]
      [ SET config_param { TO value | = value | FROM CURRENT } ]
    { IS | AS }
        program_body
      END [ proc_name ];
    |
      FUNCTION func_name
      [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
        [, ...]) ]
      RETURN rettype [ DETERMINISTIC ]
      [ STRICT ]
      [ LEAKPROOF ]
      [ PARALLEL { UNSAFE | RESTRICTED | SAFE } ]
      [ COST execution_cost ]
      [ ROWS result_rows ]
      [ SET config_param { TO value | = value | FROM CURRENT } ]
    { IS | AS }
        program_body
      END [ func_name ];
    }
  ] [, ...]
  [ BEGIN
      statement; [, ...] ]
  END [ name ]
```

Description

`CREATE PACKAGE BODY` defines a new package body. `CREATE OR REPLACE PACKAGE BODY` will either create a new package body, or replace an existing body.

If a schema name is included, then the package body is created in the specified schema. Otherwise it is created in the current schema. The name of the new package body must match an existing package specification in the same schema. The new package body name must not match any existing package body in the same schema unless the intent is to update the definition of an existing package body, in which case use `CREATE OR REPLACE PACKAGE BODY`.

Parameters

name

The name (optionally schema-qualified) of the package body to create.

declaration

A private variable, type, cursor, or `REF CURSOR` declaration.

proc_name

The name of a public or private procedure. If *proc_name* exists in the package specification with an identical signature, then it is public, otherwise it is private.

argname

The name of an argument.

`IN | IN OUT | OUT`

The argument mode.

argtype

The data type(s) of the program's arguments.

`DEFAULT` *value*

Default value of an input argument.

`STRICT`

The `STRICT` keyword specifies that the function will not be executed if called with a `NULL` argument; instead the function will return `NULL`.

`LEAKPROOF`

The LEAKPROOF keyword specifies that the function will not reveal any information about arguments, other than through a return value.

PARALLEL { UNSAFE | RESTRICTED | SAFE }

The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

When set to UNSAFE, the procedure or function cannot be executed in parallel mode. The presence of such a procedure or function forces a serial execution plan. This is the default setting if the PARALLEL clause is omitted.

When set to RESTRICTED, the procedure or function can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.

When set to SAFE, the procedure or function can be executed in parallel mode with no restriction.

execution_cost

execution_cost specifies a positive number giving the estimated execution cost for the function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row. The default is 0.0025.

result_rows

result_rows is the estimated number of rows that the query planner should expect the function to return. The default is 1000.

SET

Use the SET clause to specify a parameter value for the duration of the function:

config_param specifies the parameter name.

value specifies the parameter value.

FROM CURRENT guarantees that the parameter value is restored when the function ends.

program_body

The declarations and SPL statements that comprise the body of the function or procedure.

The declarations may include variable, type, `REF CURSOR`, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, type, and `REF CURSOR` declarations.

`func_name`

The name of a public or private function. If *`func_name`* exists in the package specification with an identical signature, then it is public, otherwise it is private.

`rettype`

The return data type.

`DETERMINISTIC`

Include `DETERMINISTIC` to specify that the function will always return the same result when given the same argument values. A `DETERMINISTIC` function must not modify the database.

Note: The `DETERMINISTIC` keyword is equivalent to the PostgreSQL `IMMUTABLE` option.

Note: If `DETERMINISTIC` is specified for a public function in the package body, it must also be specified for the function declaration in the package specification. For private functions, there is no function declaration in the package specification.

`statement`

An SPL program statement. Statements in the package initialization section are executed once per session the first time the package is referenced.

Please Note: The `STRICT`, `LEAKPROOF`, `PARALLEL`, `COST`, `ROWS` and `SET` keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Examples

The following is the package body for the `empinfo` package.

```
CREATE OR REPLACE PACKAGE BODY empinfo
IS
    v_counter       INTEGER;
    PROCEDURE get_name (
        p_empno     NUMBER
    )
    IS
```

```
    BEGIN
        SELECT ename INTO emp_name FROM emp WHERE empno = p_empno;
        v_counter := v_counter + 1;
    END;
    FUNCTION display_counter
    RETURN INTEGER
    IS
    BEGIN
        RETURN v_counter;
    END;
BEGIN
    v_counter := 0;
    DBMS_OUTPUT.PUT_LINE('Initialized counter');
END;
```

The following two anonymous blocks execute the procedure and function in the `empinfo` package and display the public variable.

```
BEGIN
    empinfo.get_name(7369);
    DBMS_OUTPUT.PUT_LINE('Employee Name    : ' || empinfo.emp_name);
    DBMS_OUTPUT.PUT_LINE('Number of queries: ' || empinfo.display_counter);
END;

Initialized counter
Employee Name    : SMITH
Number of queries: 1

BEGIN
    empinfo.get_name(7900);
    DBMS_OUTPUT.PUT_LINE('Employee Name    : ' || empinfo.emp_name);
    DBMS_OUTPUT.PUT_LINE('Number of queries: ' || empinfo.display_counter);
END;

Employee Name    : JAMES
Number of queries: 2
```

See Also

CREATE PACKAGE, DROP PACKAGE

2.3.25 CREATE PROCEDURE

Name

`CREATE PROCEDURE` -- define a new stored procedure

Synopsis

```
CREATE [OR REPLACE] PROCEDURE name [ (parameters) ]
   [
           IMMUTABLE
         | STABLE
         | VOLATILE
         | DETERMINISTIC
         | [ NOT ] LEAKPROOF
         | CALLED ON NULL INPUT
         | RETURNS NULL ON NULL INPUT
         | STRICT
         | [ EXTERNAL ] SECURITY INVOKER
         | [ EXTERNAL ] SECURITY DEFINER
         | AUTHID DEFINER
         | AUTHID CURRENT_USER
         | PARALLEL { UNSAFE | RESTRICTED | SAFE }
         | COST execution_cost
         | ROWS result_rows
         | SET configuration_parameter
           { TO value | = value | FROM CURRENT }
   ...]
{ IS | AS }
    [ declarations ]
  BEGIN
    statements
  END [ name ];
```

Description

`CREATE PROCEDURE` defines a new stored procedure. `CREATE OR REPLACE PROCEDURE` will either create a new procedure, or replace an existing definition.

If a schema name is included, then the procedure is created in the specified schema. Otherwise it is created in the current schema. The name of the new procedure must not match any existing procedure in the same schema unless the intent is to update the definition of an existing procedure, in which case use `CREATE OR REPLACE PROCEDURE`.

Parameters

name

name is the identifier of the procedure. If you specify the [OR REPLACE] clause and a procedure with the same name already exists in the schema, the new procedure will replace the existing one. If you do not specify [OR REPLACE], the new procedure will not replace the existing procedure with the same name in the same schema.

parameters

parameters is a list of formal parameters.

declarations

declarations are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations.

statements

statements are SPL program statements (the BEGIN - END block may contain an EXCEPTION section).

IMMUTABLE
STABLE
VOLATILE

These attributes inform the query optimizer about the behavior of the procedure; you can specify only one choice. VOLATILE is the default behavior.

IMMUTABLE indicates that the procedure cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.

STABLE indicates that the procedure cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for procedures that depend on database lookups, parameter variables (such as the current time zone), etc.

VOLATILE indicates that the procedure value can change even within a single table scan, so no optimizations can be made. Please note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away.

```
DETERMINISTIC
```

DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC procedure cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.

```
 [ NOT ] LEAKPROOF
```

A LEAKPROOF procedure has no side effects, and reveals no information about the values used to call the procedure.

```
CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT
```

CALLED ON NULL INPUT (the default) indicates that the procedure will be called normally when some of its arguments are NULL. It is the author's responsibility to check for NULL values if necessary and respond appropriately.

RETURNS NULL ON NULL INPUT or STRICT indicates that the procedure always returns NULL whenever any of its arguments are NULL. If these clauses are specified, the procedure is not executed when there are NULL arguments; instead a NULL result is assumed automatically.

```
[ EXTERNAL ] SECURITY DEFINER
```

SECURITY DEFINER specifies that the procedure will execute with the privileges of the user that created it; this is the default. The key word EXTERNAL is allowed for SQL conformance, but is optional.

```
[ EXTERNAL ] SECURITY INVOKER
```

The SECURITY INVOKER clause indicates that the procedure will execute with the privileges of the user that calls it. The key word EXTERNAL is allowed for SQL conformance, but is optional.

```
AUTHID DEFINER
AUTHID CURRENT_USER
```

The AUTHID DEFINER clause is a synonym for [EXTERNAL] SECURITY DEFINER. If the AUTHID clause is omitted or if AUTHID DEFINER is specified, the rights of the procedure owner are used to determine access privileges to database objects.

The AUTHID CURRENT_USER clause is a synonym for [EXTERNAL] SECURITY INVOKER. If AUTHID CURRENT_USER is specified, the rights of the current user executing the procedure are used to determine access privileges.

PARALLEL { UNSAFE | RESTRICTED | SAFE }

The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

When set to UNSAFE, the procedure cannot be executed in parallel mode. The presence of such a procedure forces a serial execution plan. This is the default setting if the PARALLEL clause is omitted.

When set to RESTRICTED, the procedure can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.

When set to SAFE, the procedure can be executed in parallel mode with no restriction.

COST *execution_cost*

execution_cost is a positive number giving the estimated execution cost for the procedure, in units of cpu_operator_cost. If the procedure returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

ROWS *result_rows*

result_rows is a positive number giving the estimated number of rows that the planner should expect the procedure to return. This is only allowed when the procedure is declared to return a set. The default assumption is 1000 rows.

SET *configuration_parameter* { TO *value* | = *value* | FROM CURRENT }

The SET clause causes the specified configuration parameter to be set to the specified value when the procedure is entered, and then restored to its prior value when the procedure exits. SET FROM CURRENT saves the session's current value of the parameter as the value to be applied when the procedure is entered.

If a SET clause is attached to a procedure, then the effects of a SET LOCAL command executed inside the procedure for the same variable are restricted to the procedure; the configuration parameter's prior value is restored at procedure exit. An ordinary SET command (without LOCAL) overrides the SET clause, much as it

would do for a previous `SET LOCAL` command, with the effects of such a command persisting after procedure exit, unless the current transaction is rolled back.

Please Note: The `STRICT`, `LEAKPROOF`, `PARALLEL`, `COST`, `ROWS` and `SET` keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Examples

The following procedure lists the employees in the `emp` table:

```
CREATE OR REPLACE PROCEDURE list_emp
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur IS
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;

EXEC list_emp;

EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
```

The following procedure uses `IN OUT` and `OUT` arguments to return an employee's number, name, and job based upon a search using first, the given employee number, and if that is not found, then using the given name. An anonymous block calls the procedure.

```
CREATE OR REPLACE PROCEDURE emp_job (
    p_empno         IN OUT emp.empno%TYPE,
    p_ename         IN OUT emp.ename%TYPE,
    p_job           OUT    emp.job%TYPE
)
```

```
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
BEGIN
    SELECT ename, job INTO v_ename, v_job FROM emp WHERE empno = p_empno;
    p_ename := v_ename;
    p_job   := v_job;
    DBMS_OUTPUT.PUT_LINE('Found employee # ' || p_empno);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        BEGIN
            SELECT empno, job INTO v_empno, v_job FROM emp
                WHERE ename = p_ename;
            p_empno := v_empno;
            p_job   := v_job;
            DBMS_OUTPUT.PUT_LINE('Found employee ' || p_ename);
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE('Could not find an employee with ' ||
                    'number, ' || p_empno || ' nor name, '  || p_ename);
                p_empno := NULL;
                p_ename := NULL;
                p_job   := NULL;
        END;
END;

DECLARE
    v_empno      emp.empno%TYPE;
    v_ename      emp.ename%TYPE;
    v_job        emp.job%TYPE;
BEGIN
    v_empno := 0;
    v_ename := 'CLARK';
    emp_job(v_empno, v_ename, v_job);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
END;

Found employee CLARK
Employee No: 7782
Name       : CLARK
Job        : MANAGER
```

The following example demonstrates using the `AUTHID DEFINER` and `SET` clauses in a procedure declaration. The `update_salary` procedure conveys the privileges of the role that defined the procedure to the role that is calling the procedure (while the procedure executes):

```
CREATE OR REPLACE PROCEDURE update_salary(id INT, new_salary NUMBER)
  SET SEARCH_PATH = 'public' SET WORK_MEM = '1MB'
  AUTHID DEFINER IS
BEGIN
  UPDATE emp SET salary = new_salary WHERE emp_id = id;
END;
```

Include the `SET` clause to set the procedure's search path to `public` and the work memory to `1MB`. Other procedures, functions and objects will not be affected by these settings.

In this example, the `AUTHID DEFINER` clause temporarily grants privileges to a role that might otherwise not be allowed to execute the statements within the procedure. To instruct the server to use the privileges associated with the role invoking the procedure, replace the `AUTHID DEFINER` clause with the `AUTHID CURRENT_USER` clause.

See Also

DROP PROCEDURE

2.3.26 CREATE PROFILE

Name

`CREATE PROFILE` – create a new profile

Synopsis

```
CREATE PROFILE profile_name
       [LIMIT {parameter value} ... ];
```

Description

`CREATE PROFILE` creates a new profile. Include the `LIMIT` clause and one or more space-delimited *`parameter`*/*`value`* pairs to specify the rules enforced by Advanced Server.

Advanced Server creates a default profile named `DEFAULT`. When you use the `CREATE ROLE` command to create a new role, the new role is automatically associated with the `DEFAULT` profile. If you upgrade from a previous version of Advanced Server to Advanced Server 10, the upgrade process will automatically create the roles in the upgraded version to the `DEFAULT` profile.

You must be a superuser to use `CREATE PROFILE`.

Include the `LIMIT` clause and one or more space-delimited *`parameter`*/*`value`* pairs to specify the rules enforced by Advanced Server.

Parameters

`profile_name`

The name of the profile.

`parameter`

The password attribute that will be monitored by the rule.

`value`

The value the *`parameter`* must reach before an action is taken by the server.

Advanced Server supports the *`value`* shown below for each *`parameter`*:

FAILED_LOGIN_ATTEMPTS specifies the number of failed login attempts that a user may make before the server locks the user out of their account for the length of time specified by PASSWORD_LOCK_TIME. Supported values are:

- An INTEGER value greater than 0.
- DEFAULT - the value of FAILED_LOGIN_ATTEMPTS specified in the DEFAULT profile.
- UNLIMITED – the connecting user may make an unlimited number of failed login attempts.

PASSWORD_LOCK_TIME specifies the length of time that must pass before the server unlocks an account that has been locked because of FAILED_LOGIN_ATTEMPTS. Supported values are:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- DEFAULT - the value of PASSWORD_LOCK_TIME specified in the DEFAULT profile.
- UNLIMITED – the account is locked until it is manually unlocked by a database superuser.

PASSWORD_LIFE_TIME specifies the number of days that the current password may be used before the user is prompted to provide a new password. Include the PASSWORD_GRACE_TIME clause when using the PASSWORD_LIFE_TIME clause to specify the number of days that will pass after the password expires before connections by the role are rejected. If PASSWORD_GRACE_TIME is not specified, the password will expire on the day specified by the default value of PASSWORD_GRACE_TIME, and the user will not be allowed to execute any command until a new password is provided. Supported values are:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- DEFAULT - the value of PASSWORD_LIFE_TIME specified in the DEFAULT profile.
- UNLIMITED – The password does not have an expiration date.

PASSWORD_GRACE_TIME specifies the length of the grace period after a password expires until the user is forced to change their password. When the grace period expires, a user will be allowed to connect, but will not be allowed to execute any command until they update their expired password. Supported values are:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- DEFAULT - the value of PASSWORD_GRACE_TIME specified in the DEFAULT profile.
- UNLIMITED – The grace period is infinite.

PASSWORD_REUSE_TIME specifies the number of days a user must wait before re-using a password. The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse. Supported values are:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- DEFAULT - the value of PASSWORD_REUSE_TIME specified in the DEFAULT profile.
- UNLIMITED – The password can be re-used without restrictions.

PASSWORD_REUSE_MAX specifies the number of password changes that must occur before a password can be reused. The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse. Supported values are:

- An INTEGER value greater than or equal to 0.
- DEFAULT - the value of PASSWORD_REUSE_MAX specified in the DEFAULT profile.
- UNLIMITED – The password can be re-used without restrictions.

PASSWORD_VERIFY_FUNCTION specifies password complexity. Supported values are:

- The name of a PL/SQL function.
- DEFAULT - the value of PASSWORD_VERIFY_FUNCTION specified in the DEFAULT profile.
- NULL

Notes

Use DROP PROFILE command to remove the profile.

Examples

The following command creates a profile named `acctg`. The profile specifies that if a user has not authenticated with the correct password in five attempts, the account will be locked for one day:

```
CREATE PROFILE acctg LIMIT
       FAILED_LOGIN_ATTEMPTS 5
       PASSWORD_LOCK_TIME 1;
```

The following command creates a profile named `sales`. The profile specifies that a user must change their password every 90 days:

```
CREATE PROFILE sales LIMIT
       PASSWORD_LIFE_TIME 90
       PASSWORD_GRACE_TIME 3;
```

If the user has not changed their password before the 90 days specified in the profile has passed, they will be issued a warning at login. After a grace period of 3 days, their account will not be allowed to invoke any commands until they change their password.

The following command creates a profile named `accts`. The profile specifies that a user cannot re-use a password within 180 days of the last use of the password, and must change their password at least 5 times before re-using the password:

```
CREATE PROFILE accts LIMIT
       PASSWORD_REUSE_TIME 180
       PASSWORD_REUSE_MAX 5;
```

The following command creates a profile named `resources`; the profile calls a user-defined function named `password_rules` that will verify that the password provided meets their standards for complexity:

```
CREATE PROFILE resources LIMIT
       PASSWORD_VERIFY_FUNCTION password_rules;
```

2.3.27 CREATE QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the CREATE QUEUE SQL command. This syntax can be used in association with DBMS_AQADM.

Name

CREATE QUEUE – create a queue.

Synopsis

Use CREATE QUEUE to define a new queue:

```
CREATE QUEUE name QUEUE TABLE queue_table_name [ ( {
option_name option_value} [, ... ] ) ]
```

where *option_name* and the corresponding *option_value* can be:

```
TYPE [normal_queue | exception_queue]

RETRIES [INTEGER]

RETRYDELAY [DOUBLE PRECISION]

RETENTION [DOUBLE PRECISION]
```

Description

The CREATE QUEUE command allows a database superuser or any user with the system-defined aq_administrator_role privilege to create a new queue in the current database.

If the name of the queue is schema-qualified, the queue is created in the specified schema. If a schema is not included in the CREATE QUEUE command, the queue is created in the current schema. A queue may only be created in the schema in which the queue table resides. The name of the queue must be unique from the name of any other queue in the same schema.

Use DROP QUEUE to remove a queue.

Parameters

name

The name (optionally schema-qualified) of the queue to be created.

queue_table_name

The name of the queue table with which this queue is associated.

option_name option_value

The name of any options that will be associated with the new queue, and the corresponding value for the option. If the call to CREATE QUEUE includes duplicate option names, the server will return an error. The following values are supported:

TYPE	Specify normal_queue to indicate that the queue is a normal queue, or exception_queue to indicate that the queue is an exception queue. An exception queue will only accept dequeue operations.
RETRIES	An INTEGER value that specifies the maximum number of attempts to remove a message from a queue.
RETRYDELAY	A DOUBLE PRECISION value that specifies the number of seconds after a ROLLBACK that the server will wait before retrying a message.
RETENTION	A DOUBLE PRECISION value that specifies the number of seconds that a message will be saved in the queue table after dequeueing.

Example

The following command creates a queue named work_order that is associated with a queue table named work_order_table:

```
CREATE QUEUE work_order QUEUE TABLE work_order_table (RETRIES 5, RETRYDELAY
2);
```

The server will allow 5 attempts to remove a message from the queue, and enforce a retry delay of 2 seconds between attempts.

See Also

ALTER QUEUE, DROP QUEUE

2.3.28 CREATE QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the CREATE QUEUE TABLE SQL command. This syntax can be used in association with DBMS_AQADM.

Name

CREATE QUEUE TABLE-- create a new queue table.

Synopsis

Use CREATE QUEUE TABLE to define a new queue table:

```
CREATE QUEUE TABLE name OF type_name [ ( { option_name
option_value } [, ... ] ) ]
```

where *option_name* and the corresponding option_value can be:

option_name	***option_value***
SORT_LIST	priority, enq_time
MULTIPLE_CONSUMERS	FALSE, TRUE
MESSAGE_GROUPING	NONE, TRANSACTIONAL
STORAGE_CLAUSE	TABLESPACE *tablespace_name*, PCTFREE integer, PCTUSED integer, INITRANS integer, MAXTRANS integer, STORAGE *storage_option* Where *storage_option* is one or more of the following: MINEXTENTS integer, MAXEXTENTS integer, PCTINCREASE integer, INITIAL *size_clause*, NEXT, FREELISTS integer, OPTIMAL *size_clause*, BUFFER_POOL {KEEP|RECYCLE|DEFAULT}. Please note that only the TABLESPACE option is enforced; all others are accepted for compatibility and ignored. Use the TABLESPACE clause to specify the name of a tablespace in which the table will be created.

Description

CREATE QUEUE TABLE allows a superuser or a user with the aq_administrator_role privilege to create a new queue table.

If the call to CREATE QUEUE TABLE includes a schema name, the queue table is created in the specified schema. If no schema name is provided, the new queue table is created in the current schema.

The name of the queue table must be unique from the name of any other queue table in the same schema.

Parameters

name

The name (optionally schema-qualified) of the new queue table.

type_name

The name of an existing type that describes the payload of each entry in the queue table. For information about defining a type, see CREATE TYPE.

option_name option_value

The name of any options that will be associated with the new queue table, and the corresponding value for the option. If the call to CREATE QUEUE TABLE includes duplicate option names, the server will return an error. The following values are accepted:

SORT_LIST	Use the SORT_LIST option to control the dequeueing order of the queue; specify the names of the column(s) that will be used to sort the queue (in ascending order). The currently accepted values are the following combinations of enq_time and priority: enq_time, priority priority, enq_time priority enq_time Any other value will return an ERROR.
MULTIPLE_CONSUMERS	A BOOLEAN value that indicates if a message can have more than one consumer (TRUE), or are limited to one consumer per message (FALSE).
MESSAGE_GROUPING	Specify none to indicate that each message should be dequeued individually, or transactional to indicate that messages that are added to the queue as a result of one transaction should be dequeued as a group.
STORAGE_CLAUSE	Use STORAGE_CLAUSE to specify table attributes. STORAGE_CLAUSE may be TABLESPACE *tablespace_name*, PCTFREE integer, PCTUSED integer, INITRANS integer, MAXTRANS integer, STORAGE *storage_option* Where *storage_option* is one or more of the following: MINEXTENTS integer, MAXEXTENTS integer, PCTINCREASE integer, INITIAL *size_clause*, NEXT, FREELISTS integer, OPTIMAL *size_clause*, BUFFER_POOL {KEEP|RECYCLE|DEFAULT}. Please note that only the TABLESPACE option is enforced; all others are accepted for compatibility and ignored. Use the TABLESPACE clause to specify the name of a tablespace in which the table will be created.

Example

You must create a user-defined type before creating a queue table; the type describes the columns and data types within the table. The following command creates a type named `work_order`:

```
CREATE TYPE work_order AS (name VARCHAR2, project TEXT, completed BOOLEAN);
```

The following command uses the `work_order` type to create a queue table named `work_order_table`:

```
CREATE QUEUE TABLE work_order_table OF work_order (sort_list (enq_time,
priority));
```

See Also

ALTER QUEUE TABLE, DROP QUEUE TABLE

2.3.29 CREATE ROLE

Name

CREATE ROLE -- define a new database role

Synopsis

CREATE ROLE *name* [IDENTIFIED BY *password* [REPLACE *old_password*]]

Description

CREATE ROLE adds a new role to the Advanced Server database cluster. A role is an entity that can own database objects and have database privileges; a role can be considered a “user”, a “group”, or both depending on how it is used. The newly created role does not have the LOGIN attribute, so it cannot be used to start a session. Use the ALTER ROLE command to give the role LOGIN rights. You must have CREATEROLE privilege or be a database superuser to use the CREATE ROLE command.

If the IDENTIFIED BY clause is specified, the CREATE ROLE command also creates a schema owned by, and with the same name as the newly created role.

Note that roles are defined at the database cluster level, and so are valid in all databases in the cluster.

Parameters

name

The name of the new role.

IDENTIFIED BY *password*

Sets the role’s password. (A password is only of use for roles having the LOGIN attribute, but you can nonetheless define one for roles without it.) If you do not plan to use password authentication you can omit this option.

Notes

Use ALTER ROLE to change the attributes of a role, and DROP ROLE to remove a role. The attributes specified by CREATE ROLE can be modified by later ALTER ROLE commands.

Use `GRANT` and `REVOKE` to add and remove members of roles that are being used as groups.

The maximum length limit for role name and password is 63 characters.

Examples

Create a role (and a schema) named, `admins`, with a password:

```
CREATE ROLE admins IDENTIFIED BY Rt498zb;
```

See Also

ALTER ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE

2.3.30 CREATE SCHEMA

Name

CREATE SCHEMA -- define a new schema

Synopsis

```
CREATE SCHEMA AUTHORIZATION username schema_element [ ... ]
```

Description

This variation of the CREATE SCHEMA command creates a new schema owned by *username* and populated with one or more objects. The creation of the schema and objects occur within a single transaction so either all objects are created or none of them including the schema. (Please note: if you are using an Oracle database, no new schema is created – *username*, and therefore the schema, must pre-exist.)

A schema is essentially a namespace: it contains named objects (tables, views, etc.) whose names may duplicate those of other objects existing in other schemas. Named objects are accessed either by "qualifying" their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). Unqualified objects are created in the current schema (the one at the front of the search path, which can be determined with the function CURRENT_SCHEMA). (The search path concept and the CURRENT_SCHEMA function are not compatible with Oracle databases.)

CREATE SCHEMA includes subcommands to create objects within the schema. The subcommands are treated essentially the same as separate commands issued after creating the schema. All the created objects will be owned by the specified user.

Parameters

username

> The name of the user who will own the new schema. The schema will be named the same as *username*. Only superusers may create schemas owned by users other than themselves. (Please note: In Advanced Server the role, *username*, must already exist, but the schema must not exist. In Oracle, the user (equivalently, the schema) must exist.)

schema_element

> An SQL statement defining an object to be created within the schema. CREATE TABLE, CREATE VIEW, and GRANT are accepted as clauses within CREATE

SCHEMA. Other kinds of objects may be created in separate commands after the schema is created.

Notes

To create a schema, the invoking user must have the CREATE privilege for the current database. (Of course, superusers bypass this check.)

In Advanced Server, there are other forms of the CREATE SCHEMA command that are not compatible with Oracle databases.

Examples

```
CREATE SCHEMA AUTHORIZATION enterprisedb
    CREATE TABLE empjobs (ename VARCHAR2(10), job VARCHAR2(9))
    CREATE VIEW managers AS SELECT ename FROM empjobs WHERE job = 'MANAGER'
    GRANT SELECT ON managers TO PUBLIC;
```

2.3.31 CREATE SEQUENCE

Name

`CREATE SEQUENCE` -- define a new sequence generator

Synopsis

```
CREATE SEQUENCE name [ INCREMENT BY increment ]
  [ { NOMINVALUE | MINVALUE minvalue } ]
  [ { NOMAXVALUE | MAXVALUE maxvalue } ]
  [ START WITH start ] [ CACHE cache | NOCACHE ] [ CYCLE ]
```

Description

`CREATE SEQUENCE` creates a new sequence number generator. This involves creating and initializing a new special single-row table with the name, *`name`*. The generator will be owned by the user issuing the command.

If a schema name is given then the sequence is created in the specified schema, otherwise it is created in the current schema. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema.

After a sequence is created, use the functions `NEXTVAL` and `CURRVAL` to operate on the sequence. These functions are documented in Section 2.4.9.

Parameters

`name`

The name (optionally schema-qualified) of the sequence to be created.

`increment`

The optional clause `INCREMENT BY` *`increment`* specifies the value to add to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

`NOMINVALUE | MINVALUE` *`minvalue`*

The optional clause `MINVALUE` *`minvalue`* determines the minimum value a sequence can generate. If this clause is not supplied, then defaults will be used. The defaults are 1 and $-2^{63}-1$ for ascending and descending sequences,

respectively. Note that the key words, NOMINVALUE, may be used to set this behavior to the default.

NOMAXVALUE | MAXVALUE *maxvalue*

The optional clause MAXVALUE *maxvalue* determines the maximum value for the sequence. If this clause is not supplied, then default values will be used. The defaults are 2^{63}-1 and -1 for ascending and descending sequences, respectively. Note that the key words, NOMAXVALUE, may be used to set this behavior to the default.

start

The optional clause START WITH *start* allows the sequence to begin anywhere. The default starting value is *minvalue* for ascending sequences and *maxvalue* for descending ones.

cache

The optional clause CACHE *cache* specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., NOCACHE), and this is also the default.

CYCLE

The CYCLE option allows the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively.

If CYCLE is omitted (the default), any calls to NEXTVAL after the sequence has reached its maximum value will return an error. Note that the key words, NO CYCLE, may be used to obtain the default behavior, however, this term is not compatible with Oracle databases.

Notes

Sequences are based on big integer arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807). On some older platforms, there may be no compiler support for eight-byte integers, in which case sequences use regular INTEGER arithmetic (range -2147483648 to +2147483647).

Unexpected results may be obtained if a *cache* setting greater than one is used for a sequence object that will be used concurrently by multiple sessions. Each session will allocate and cache successive sequence values during one access to the sequence object

and increase the sequence object's last value accordingly. Then, the next *cache*-1 uses of NEXTVAL within that session simply return the preallocated values without touching the sequence object. So, any numbers allocated but not used within a session will be lost when that session ends, resulting in "holes" in the sequence.

Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the sessions are considered. For example, with a *cache* setting of 10, session A might reserve values 1..10 and return NEXTVAL=1, then session B might reserve values 11..20 and return NEXTVAL=11 before session A has generated NEXTVAL=2. Thus, with a *cache* setting of one it is safe to assume that NEXTVAL values are generated sequentially; with a *cache* setting greater than one you should only assume that the NEXTVAL values are all distinct, not that they are generated purely sequentially. Also, the last value will reflect the latest value reserved by any session, whether or not it has yet been returned by NEXTVAL.

Examples

Create an ascending sequence called serial, starting at 101:

```
CREATE SEQUENCE serial START WITH 101;
```

Select the next number from this sequence:

```
SELECT serial.NEXTVAL FROM DUAL;

 nextval
---------
     101
(1 row)
```

Create a sequence called supplier_seq with the NOCACHE option:

```
CREATE SEQUENCE supplier_seq
    MINVALUE 1
    START WITH 1
    INCREMENT BY 1
    NOCACHE;
```

Select the next number from this sequence:

```
SELECT supplier_seq.NEXTVAL FROM DUAL;

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

See Also

ALTER SEQUENCE, DROP SEQUENCE

2.3.32 CREATE SYNONYM

Name

`CREATE SYNONYM` -- define a new synonym

Synopsis

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema.]syn_name
        FOR object_schema.object_name[@dblink_name];
```

Description

`CREATE SYNONYM` defines a synonym for certain types of database objects. Advanced Server supports synonyms for:

- tables
- views
- materialized views
- sequences
- stored procedures
- stored functions
- types
- objects that are accessible through a database link
- other synonyms

Parameters:

`syn_name`

`syn_name` is the name of the synonym. A synonym name must be unique within a schema.

`schema`

`schema` specifies the name of the schema that the synonym resides in. If you do not specify a schema name, the synonym is created in the first existing schema in your search path.

`object_name`

`object_name` specifies the name of the object.

`object_schema`

object_schema specifies the name of the schema that the referenced object resides in.

dblink_name

dblink_name specifies the name of the database link through which an object is accessed.

Include the REPLACE clause to replace an existing synonym definition with a new synonym definition.

Include the PUBLIC clause to create the synonym in the public schema. The CREATE PUBLIC SYNONYM command, compatible with Oracle databases, creates a synonym that resides in the public schema:

```
CREATE [OR REPLACE] PUBLIC SYNONYM syn_name FOR
object_schema.object_name;
```

This just a shorthand way to write:

```
CREATE [OR REPLACE] SYNONYM public.syn_name FOR
object_schema.object_name;
```

Notes

Access to the object referenced by the synonym is determined by the permissions of the current user of the synonym; the synonym user must have the appropriate permissions on the underlying database object.

Examples

Create a synonym for the emp table in a schema named, enterprisedb:

```
CREATE SYNONYM personnel FOR enterprisedb.emp;
```

See Also

DROP SYNONYM

2.3.33 CREATE TABLE

Name

`CREATE TABLE` -- define a new table

Synopsis

```
CREATE [ GLOBAL TEMPORARY ] TABLE table_name (
  { column_name data_type [ DEFAULT default_expr ]
  [ column_constraint [ ... ] ] | table_constraint } [, ...]
  )
  [ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]
  [ TABLESPACE tablespace ]
```

where *column_constraint* is:

```
  [ CONSTRAINT constraint_name ]
  { NOT NULL |
    NULL |
    UNIQUE [ USING INDEX TABLESPACE tablespace ] |
    PRIMARY KEY [ USING INDEX TABLESPACE tablespace ] |
    CHECK (expression) |
    REFERENCES reftable [ ( refcolumn ) ]
      [ ON DELETE action ] }
  [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED |
    INITIALLY IMMEDIATE ]
```

and *table_constraint* is:

```
  [ CONSTRAINT constraint_name ]
  { UNIQUE ( column_name [, ...] )
      [ USING INDEX TABLESPACE tablespace ] |
    PRIMARY KEY ( column_name [, ...] )
      [ USING INDEX TABLESPACE tablespace ] |
    CHECK ( expression ) |
    FOREIGN KEY ( column_name [, ...] )
        REFERENCES reftable [ ( refcolumn [, ...] ) ]
      [ ON DELETE action ] }
  [ DEFERRABLE | NOT DEFERRABLE ]
  [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Description

`CREATE TABLE` will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.

If a schema name is given (for example, `CREATE TABLE` *`myschema`*.*`mytable`* ...) then the table is created in the specified schema. Otherwise it is created in the current schema. Temporary tables exist in a special schema, so a schema name may not be given when creating a temporary table. The table name must be distinct from the name of any other table, sequence, index, or view in the same schema.

`CREATE TABLE` also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

A table cannot have more than 1600 columns. (In practice, the effective limit is lower because of tuple-length constraints).

The optional constraint clauses specify constraints (or tests) that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience if the constraint only affects one column.

Parameters

`GLOBAL TEMPORARY`

> If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see `ON COMMIT` below). Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. In addition, temporary tables are not visible outside the session in which it was created. (This aspect of global temporary tables is not compatible with Oracle databases.) Any indexes created on a temporary table are automatically temporary as well.

`table_name`

> The name (optionally schema-qualified) of the table to be created.

`column_name`

> The name of a column to be created in the new table.

data_type

The data type of the column. This may include array specifiers. For more information on the data types included with Advanced Server, refer to Section 2.2.

DEFAULT *default_expr*

The DEFAULT clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

CONSTRAINT *constraint_name*

An optional name for a column or table constraint. If not specified, the system generates a name.

NOT NULL

The column is not allowed to contain null values.

NULL

The column is allowed to contain null values. This is the default.

This clause is only available for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

UNIQUE - column constraint
UNIQUE (*column_name* [, ...]) - table constraint

The UNIQUE constraint specifies that a group of one or more distinct columns of a table may contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

For the purpose of a unique constraint, null values are not considered equal.

Each unique table constraint must name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table. (Otherwise it would just be the same constraint listed twice.)

`PRIMARY KEY` - column constraint
`PRIMARY KEY` (*`column_name`* [, ...]) - table constraint

The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-null values. Technically, `PRIMARY KEY` is merely a combination of `UNIQUE` and `NOT NULL`, but identifying a set of columns as primary key also provides metadata about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows.

Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

The primary key constraint should name a set of columns that is different from other sets of columns named by any unique constraint defined for the same table.

`CHECK (`*`expression`*`)`

The `CHECK` clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or "unknown" succeed. Should any row of an insert or update operation produce a FALSE result an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column's value only, while an expression appearing in a table constraint may reference multiple columns.

Currently, `CHECK` expressions cannot contain subqueries nor refer to variables other than columns of the current row.

`REFERENCES` reftable [(*`refcolumn`*)] [`ON DELETE` *`action`*] - column constraint
`FOREIGN KEY` (*`column`* [, ...]) `REFERENCES` *`reftable`* [(*`refcolumn`* [, ...])] [`ON DELETE` *`action`*] - table constraint

These clauses specify a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table. If *`refcolumn`* is omitted, the primary key of the *`reftable`* is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table.

In addition, when the data in the referenced columns is changed, certain actions are performed on the data in this table's columns. The `ON DELETE` clause specifies the action to perform when a referenced row in the referenced table is

being deleted. Referential actions cannot be deferred even if the constraint is deferrable. Here are the following possible actions for each clause:

`CASCADE`

Delete any rows referencing the deleted row, or update the value of the referencing column to the new value of the referenced column, respectively.

`SET NULL`

Set the referencing column(s) to `NULL`.

If the referenced column(s) are changed frequently, it may be wise to add an index to the foreign key column so that referential actions associated with the foreign key column can be performed more efficiently.

`DEFERRABLE`
`NOT DEFERRABLE`

This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable may be postponed until the end of the transaction (using the `SET CONSTRAINTS` command). `NOT DEFERRABLE` is the default. Only foreign key constraints currently accept this clause. All other constraint types are not deferrable.

`INITIALLY IMMEDIATE`
`INITIALLY DEFERRED`

If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is `INITIALLY IMMEDIATE`, it is checked after each statement. This is the default. If the constraint is `INITIALLY DEFERRED`, it is checked only at the end of the transaction. The constraint check time can be altered with the `SET CONSTRAINTS` command.

`ON COMMIT`

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The two options are:

`PRESERVE ROWS`

No special action is taken at the ends of transactions. This is the default behavior. (Note that this aspect is not compatible with Oracle databases. The Oracle default is `DELETE ROWS`.)

`DELETE ROWS`

All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic `TRUNCATE` is done at each commit.

`TABLESPACE` *`tablespace`*

The *`tablespace`* is the name of the tablespace in which the new table is to be created. If not specified, `default tablespace` is used, or the database's default tablespace if `default_tablespace` is an empty string.

`USING INDEX TABLESPACE` *`tablespace`*

This clause allows selection of the tablespace in which the index associated with a `UNIQUE` or `PRIMARY KEY` constraint will be created. If not specified, `default tablespace` is used, or the database's default tablespace if `default_tablespace` is an empty string.

Notes

Advanced Server automatically creates an index for each unique constraint and primary key constraint to enforce the uniqueness. Thus, it is not necessary to create an explicit index for primary key columns. (See `CREATE INDEX` for more information.)

Examples

Create table `dept` and table `emp`:

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14),
    loc             VARCHAR2(13)
);
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk 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 emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
```

Define a unique table constraint for the table `dept`. Unique table constraints can be defined on one or more columns of the table.

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR2(13)
);
```

Define a check column constraint:

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
```

Define a check table constraint:

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk 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 emp_ref_dept_fk
                        REFERENCES dept(deptno),
    CONSTRAINT new_emp_ck CHECK (ename IS NOT NULL AND empno > 7000)
);
```

Define a primary key table constraint for the table `jobhist`. Primary key table constraints can be defined on one or more columns of the table.

```
CREATE TABLE jobhist (
    empno           NUMBER(4) NOT NULL,
    startdate       DATE NOT NULL,
    enddate         DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2),
    chgdesc         VARCHAR2(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate)
);
```

This assigns a literal constant default value for the column, `job` and makes the default value of `hiredate` be the date at which the row is inserted.

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9) DEFAULT 'SALESMAN',
    mgr             NUMBER(4),
    hiredate        DATE DEFAULT SYSDATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
```

Create table `dept` in tablespace `diskvol1`:

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14),
    loc             VARCHAR2(13)
) TABLESPACE diskvol1;
```

See Also

ALTER TABLE, DROP TABLE

2.3.34 CREATE TABLE AS

Name

`CREATE TABLE AS` -- define a new table from the results of a query

Synopsis

```
CREATE [ GLOBAL TEMPORARY ] TABLE table_name
  [ (column_name [, ...] ) ]
  [ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]
  [ TABLESPACE tablespace ]
  AS query
```

Description

`CREATE TABLE AS` creates a table and fills it with data computed by a `SELECT` command. The table columns have the names and data types associated with the output columns of the `SELECT` (except that you can override the column names by giving an explicit list of new column names).

`CREATE TABLE AS` bears some resemblance to creating a view, but it is really quite different: it creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query. In contrast, a view re-evaluates its defining `SELECT` statement whenever it is queried.

Parameters

`GLOBAL TEMPORARY`

If specified, the table is created as a temporary table. Refer to `CREATE TABLE` for details.

`table_name`

The name (optionally schema-qualified) of the table to be created.

`column_name`

The name of a column in the new table. If column names are not provided, they are taken from the output column names of the query.

query

A query statement (a `SELECT` command). Refer to `SELECT` for a description of the allowed syntax.

2.3.35 CREATE TRIGGER

Name

`CREATE TRIGGER` -- define a new trigger

Synopsis

```
CREATE [ OR REPLACE ] TRIGGER name
  { BEFORE | AFTER | INSTEAD OF }
  { INSERT | UPDATE | DELETE }
      [ OR { INSERT | UPDATE | DELETE } ] [, ...]
    ON table
  [ REFERENCING { OLD AS old | NEW AS new } ...]
  [ FOR EACH ROW ]
  [ WHEN condition ]
  [ DECLARE
      declaration; [, ...] ]
    BEGIN
      statement; [, ...]
  [ EXCEPTION
    { WHEN exception [ OR exception ] [...] THEN
        statement; [, ...] } [, ...]
  ]
    END
```

Description

`CREATE TRIGGER` defines a new trigger. `CREATE OR REPLACE TRIGGER` will either create a new trigger, or replace an existing definition.

If you are using the `CREATE TRIGGER` keywords to create a new trigger, the name of the new trigger must not match any existing trigger defined on the same table. New triggers will be created in the same schema as the table on which the triggering event is defined.

If you are updating the definition of an existing trigger, use the `CREATE OR REPLACE TRIGGER` keywords.

When you use syntax that is compatible with Oracle to create a trigger, the trigger runs as a `SECURITY DEFINER` function.

Parameters

name

The name of the trigger to create.

BEFORE | AFTER

Determines whether the trigger is fired before or after the triggering event.

INSERT | UPDATE | DELETE

Defines the triggering event.

table

The name of the table on which the triggering event occurs.

condition

condition is a Boolean expression that determines if the trigger will actually be executed; if *condition* evaluates to TRUE, the trigger will fire.

If the trigger definition includes the FOR EACH ROW keywords, the WHEN clause can refer to columns of the old and/or new row values by writing OLD.*column_name* or NEW.*column_name* respectively. INSERT triggers cannot refer to OLD and DELETE triggers cannot refer to NEW.

If the trigger includes the INSTEAD OF keywords, it may not include a WHEN clause. A WHEN clause cannot contain subqueries.

REFERENCING { OLD AS *old* | NEW AS *new* } ...

REFERENCING clause to reference old rows and new rows, but restricted in that *old* may only be replaced by an identifier named old or any equivalent that is saved in all lowercase (for example, REFERENCING OLD AS old, REFERENCING OLD AS OLD, or REFERENCING OLD AS "old"). Also, *new* may only be replaced by an identifier named new or any equivalent that is saved in all lowercase (for example, REFERENCING NEW AS new, REFERENCING NEW AS NEW, or REFERENCING NEW AS "new").

Either one, or both phrases OLD AS *old* and NEW AS *new* may be specified in the REFERENCING clause (for example, REFERENCING NEW AS New OLD AS Old).

This clause is not compatible with Oracle databases in that identifiers other than old or new may not be used.

FOR EACH ROW

Determines whether the trigger should be fired once for every row affected by the triggering event, or just once per SQL statement. If specified, the trigger is fired

once for every affected row (row-level trigger), otherwise the trigger is a statement-level trigger.

declaration

A variable, type, `REF CURSOR`, or subprogram declaration. If subprogram declarations are included, they must be declared after all other variable, type, and `REF CURSOR` declarations.

statement

An SPL program statement. Note that a `DECLARE - BEGIN - END` block is considered an SPL statement unto itself. Thus, the trigger body may contain nested blocks.

exception

An exception condition name such as `NO_DATA_FOUND`, `OTHERS`, etc.

Examples

The following is a statement-level trigger that fires after the triggering statement (insert, update, or delete on table `emp`) is executed.

```
CREATE OR REPLACE TRIGGER user_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    IF INSERTING THEN
        v_action := ' added employee(s) on ';
    ELSIF UPDATING THEN
        v_action := ' updated employee(s) on ';
    ELSIF DELETING THEN
        v_action := ' deleted employee(s) on ';
    END IF;
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
        TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
```

The following is a row-level trigger that fires before each row is either inserted, updated, or deleted on table `emp`.

```
CREATE OR REPLACE TRIGGER emp_sal_trig
    BEFORE DELETE OR INSERT OR UPDATE ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
        DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
    END IF;
```

```
    IF UPDATING THEN
        sal_diff := :NEW.sal - :OLD.sal;
        DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
        DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
        DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
        DBMS_OUTPUT.PUT_LINE('..Raise     : ' || sal_diff);
    END IF;
    IF DELETING THEN
        DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
        DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
    END IF;
END;
```

See Also

DROP TRIGGER

2.3.36 CREATE TYPE

Name

`CREATE TYPE` -- define a new user-defined type, which can be an object type, a collection type (a nested table type or a varray type), or a composite type.

Synopsis

Object Type

```
CREATE [ OR REPLACE ] TYPE name
  [ AUTHID { DEFINER | CURRENT_USER } ]
  { IS | AS } OBJECT
( { attribute { datatype | objtype | collecttype } }
    [, ...]
  [ method_spec ] [, ...]
) [ [ NOT ] { FINAL | INSTANTIABLE } ] ...
```

where *method_spec* is:

```
  [ [ NOT ] { FINAL | INSTANTIABLE } ] ...
  [ OVERRIDING ]
    subprogram_spec
```

and *subprogram_spec* is:

```
  { MEMBER | STATIC }
  { PROCEDURE proc_name
      [ ( [ SELF [ IN | IN OUT ] name ]
          [, argname [ IN | IN OUT | OUT ] argtype
                     [ DEFAULT value ]
          ] ...)
      ]
  |
    FUNCTION func_name
      [ ( [ SELF [ IN | IN OUT ] name ]
          [, argname [ IN | IN OUT | OUT ] argtype
                     [ DEFAULT value ]
          ] ...)
      ]
    RETURN rettype
  }
```

Nested Table Type


```
CREATE [ OR REPLACE ] TYPE name { IS | AS } TABLE OF
  { datatype | objtype | collecttype }
```

Varray Type

```
CREATE [ OR REPLACE ] TYPE name { IS | AS }
  { VARRAY | VARYING ARRAY } (maxsize) OF { datatype | objtype }
```

Composite Type

```
CREATE [ OR REPLACE ] TYPE name { IS | AS }
( [ attribute datatype ][, ...]
)
```

Description

`CREATE TYPE` defines a new, user-defined data type. The types that can be created are an object type, a nested table type, a varray type, or a composite type. Nested table and varray types belong to the category of types known as *collections*.

Composite types are not compatible with Oracle databases. However, composite types can be accessed by SPL programs as with other types described in this section.

Note: For packages only, a composite type can be included in a user-defined record type declared with the `TYPE IS RECORD` statement within the package specification or package body. Such nested structure is not permitted in other SPL programs such as functions, procedures, triggers, etc.

In the `CREATE TYPE` command, if a schema name is included, then the type is created in the specified schema, otherwise it is created in the current schema. The name of the new type must not match any existing type in the same schema unless the intent is to update the definition of an existing type, in which case use `CREATE OR REPLACE TYPE`.

Note: The `OR REPLACE` option cannot be currently used to add, delete, or modify the attributes of an existing object type. Use the `DROP TYPE` command to first delete the existing object type. The `OR REPLACE` option can be used to add, delete, or modify the methods in an existing object type.

Note: The PostgreSQL form of the `ALTER TYPE ALTER ATTRIBUTE` command can be used to change the data type of an attribute in an existing object type. However, the `ALTER TYPE` command cannot add or delete attributes in the object type.

The user that creates the type becomes the owner of the type.

Parameters

name

The name (optionally schema-qualified) of the type to create.

DEFINER | CURRENT_USER

Specifies whether the privileges of the object type owner (DEFINER) or the privileges of the current user executing a method in the object type (CURRENT_USER) are to be used to determine whether or not access is allowed to database objects referenced in the object type. DEFINER is the default.

attribute

The name of an attribute in the object type or composite type.

datatype

The data type that defines an attribute of the object type or composite type, or the elements of the collection type that is being created.

objtype

The name of an object type that defines an attribute of the object type or the elements of the collection type that is being created.

collecttype

The name of a collection type that defines an attribute of the object type or the elements of the collection type that is being created.

FINAL
NOT FINAL

For an object type, specifies whether or not a subtype can be derived from the object type. FINAL (subtype cannot be derived from the object type) is the default.

For *method_spec*, specifies whether or not the method may be overridden in a subtype. NOT FINAL (method may be overridden in a subtype) is the default.

INSTANTIABLE
NOT INSTANTIABLE

For an object type, specifies whether or not an object instance can be created of this object type. INSTANTIABLE (an instance of this object type can be created) is

the default. If `NOT INSTANTIABLE` is specified, then `NOT FINAL` must be specified as well. If *`method_spec`* for any method in the object type contains the `NOT INSTANTIABLE` qualifier, then the object type, itself, must be defined with `NOT INSTANTIABLE` and `NOT FINAL` following the closing parenthesis of the object type specification.

For *`method_spec`*, specifies whether or not the object type definition provides an implementation for the method. `INSTANTIABLE` (the `CREATE TYPE BODY` command for the object type provides the implementation of the method) is the default. If `NOT INSTANTIABLE` is specified, then the `CREATE TYPE BODY` command for the object type must not contain the implementation of the method.

`OVERRIDING`

If `OVERRIDING` is specified, *`method_spec`* overrides an identically named method with the same number of identically named method arguments with the same data types, in the same order, and the same return type (if the method is a function) as defined in a supertype.

`MEMBER`
`STATIC`

Specify `MEMBER` if the subprogram operates on an object instance. Specify `STATIC` if the subprogram operates independently of any particular object instance.

`proc_name`

The name of the procedure to create.

`SELF [ IN | IN OUT ]` *`name`*

For a member method there is an implicit, built-in parameter named `SELF` whose data type is that of the object type being defined. `SELF` refers to the object instance that is currently invoking the method. `SELF` can be explicitly declared as an `IN` or `IN OUT` parameter in the parameter list. If explicitly declared, `SELF` must be the first parameter in the parameter list. If `SELF` is not explicitly declared, its parameter mode defaults to `IN OUT` for member procedures and `IN` for member functions.

`argname`

The name of an argument. The argument is referenced by this name within the method body.

`argtype`

The data type(s) of the method's arguments. The argument types may be a base data type or a user-defined type such as a nested table or an object type. A length must not be specified for any base type - for example, specify `VARCHAR2`, not `VARCHAR2(10)`.

`DEFAULT` *`value`*

Supplies a default value for an input argument if one is not supplied in the method call. `DEFAULT` may not be specified for arguments with modes `IN OUT` or `OUT`.

`func_name`

The name of the function to create.

`rettype`

The return data type, which may be any of the types listed for *`argtype`*. As for *`argtype`*, a length must not be specified for *`rettype`*.

`maxsize`

The maximum number of elements permitted in the varray.

Examples

Creating an Object Type

Create object type `addr_obj_typ`.

```
CREATE OR REPLACE TYPE addr_obj_typ AS OBJECT (
    street          VARCHAR2(30),
    city            VARCHAR2(20),
    state           CHAR(2),
    zip             NUMBER(5)
);
```

Create object type `emp_obj_typ` that includes a member method `display_emp`.

```
CREATE OR REPLACE TYPE emp_obj_typ AS OBJECT (
    empno           NUMBER(4),
    ename           VARCHAR2(20),
    addr            ADDR_OBJ_TYP,
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
);
```

Create object type `dept_obj_typ` that includes a static method `get_dname`.

```
CREATE OR REPLACE TYPE dept_obj_typ AS OBJECT (
    deptno          NUMBER(2),
    STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2,
```

```
    MEMBER PROCEDURE display_dept
);
```

Creating a Collection Type

Create a nested table type, `budget_tbl_typ`, of data type, `NUMBER(8,2)`.

```
CREATE OR REPLACE TYPE budget_tbl_typ IS TABLE OF NUMBER(8,2);
```

Creating and Using a Composite Type

The following example shows the usage of a composite type accessed from an anonymous block.

The composite type is created by the following:

```
CREATE OR REPLACE TYPE emphist_typ AS (
    empno           NUMBER(4),
    ename           VARCHAR2(10),
    hiredate        DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2)
);
```

The following is the anonymous block that accesses the composite type:

```
DECLARE
    v_emphist       EMPHIST_TYP;
BEGIN
    v_emphist.empno    := 9001;
    v_emphist.ename    := 'SMITH';
    v_emphist.hiredate := '01-AUG-17';
    v_emphist.job      := 'SALESMAN';
    v_emphist.sal      := 8000.00;
    DBMS_OUTPUT.PUT_LINE('   EMPNO: ' || v_emphist.empno);
    DBMS_OUTPUT.PUT_LINE('   ENAME: ' || v_emphist.ename);
    DBMS_OUTPUT.PUT_LINE('HIREDATE: ' || v_emphist.hiredate);
    DBMS_OUTPUT.PUT_LINE('     JOB: ' || v_emphist.job);
    DBMS_OUTPUT.PUT_LINE('     SAL: ' || v_emphist.sal);
END;

   EMPNO: 9001
   ENAME: SMITH
HIREDATE: 01-AUG-17 00:00:00
     JOB: SALESMAN
     SAL: 8000.00
```

The following example shows the usage of a composite type accessed from a user-defined record type, declared within a package body.

The composite type is created by the following:

```
CREATE OR REPLACE TYPE salhist_typ AS (
    startdate       DATE,
    job             VARCHAR2(9),
```

```
    sal             NUMBER(7,2)
);
```

The package specification is defined by the following:

```
CREATE OR REPLACE PACKAGE emp_salhist
IS
    PROCEDURE fetch_emp (
        p_empno     IN NUMBER
    );
END;
```

The package body is defined by the following:

```
CREATE OR REPLACE PACKAGE BODY emp_salhist
IS
    TYPE emprec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR(10),
        salhist     SALHIST_TYP
    );
    TYPE emp_arr_typ IS TABLE OF emprec_typ INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;

    PROCEDURE fetch_emp (
        p_empno     IN NUMBER
    )
    IS
        CURSOR emp_cur IS SELECT e.empno, e.ename, h.startdate, h.job, h.sal
            FROM emp e, jobhist h
            WHERE e.empno = p_empno
              AND e.empno = h.empno;

        i           INTEGER := 0;
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    STARTDATE  JOB         ' ||
        'SAL        ');
        DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '---------');

        FOR r_emp IN emp_cur LOOP
            i := i + 1;
            emp_arr(i) := (r_emp.empno, r_emp.ename,
                (r_emp.startdate, r_emp.job, r_emp.sal));
        END LOOP;

        FOR i IN 1 .. emp_arr.COUNT LOOP
            DBMS_OUTPUT.PUT_LINE(emp_arr(i).empno || '   ' ||
                RPAD(emp_arr(i).ename,8) || ' ' ||
                TO_CHAR(emp_arr(i).salhist.startdate,'DD-MON-YY') || '  ' ||
                RPAD(emp_arr(i).salhist.job,10) || ' ' ||
                TO_CHAR(emp_arr(i).salhist.sal,'99,999.99'));
        END LOOP;
    END;
END;
```

Note that in the declaration of the `TYPE emprec_typ IS RECORD` data structure in the package body, the `salhist` field is defined with the `SALHIST_TYP` composite type as created by the `CREATE TYPE salhist_typ` statement.

The associative array definition `TYPE emp_arr_typ IS TABLE OF emprec_typ` references the record type data structure `emprec_typ` that includes the field `salhist` that is defined with the `SALHIST_TYP` composite type.

Invocation of the package procedure that loads the array from a join of the `emp` and `jobhist` tables, then displays the array content is shown by the following:

```
EXEC emp_salhist.fetch_emp(7788);

EMPNO  ENAME    STARTDATE  JOB         SAL
-----  -------  ---------  ---------   ---------
7788   SCOTT    19-APR-87  CLERK        1,000.00
7788   SCOTT    13-APR-88  CLERK        1,040.00
7788   SCOTT    05-MAY-90  ANALYST      3,000.00

EDB-SPL Procedure successfully completed
```

See Also

CREATE TYPE BODY, DROP TYPE

2.3.37 CREATE TYPE BODY

Name

`CREATE TYPE BODY` -- define a new object type body

Synopsis

```
CREATE [ OR REPLACE ] TYPE BODY name
  { IS | AS }
  method_spec [...]
END
```

where *method_spec* is:

```
    subprogram_spec
```

and *subprogram_spec* is:

```
  { MEMBER | STATIC }
  { PROCEDURE proc_name
      [ ( [ SELF [ IN | IN OUT ] name ]
          [, argname [ IN | IN OUT | OUT ] argtype
                     [ DEFAULT value ]
          ] ...)
      ]
  { IS | AS }
      program_body
    END;
  |
    FUNCTION func_name
      [ ( [ SELF [ IN | IN OUT ] name ]
          [, argname [ IN | IN OUT | OUT ] argtype
                     [ DEFAULT value ]
          ] ...)
      ]
    RETURN rettype
  { IS |AS }
      program_body
    END;
  }
```

Description

`CREATE TYPE BODY` defines a new object type body. `CREATE OR REPLACE TYPE BODY` will either create a new object type body, or replace an existing body.

If a schema name is included, then the object type body is created in the specified schema. Otherwise it is created in the current schema. The name of the new object type body must match an existing object type specification in the same schema. The new object type body name must not match any existing object type body in the same schema unless the intent is to update the definition of an existing object type body, in which case use `CREATE OR REPLACE TYPE BODY`.

Parameters

name

The name (optionally schema-qualified) of the object type for which a body is to be created.

`MEMBER`
`STATIC`

Specify `MEMBER` if the subprogram operates on an object instance. Specify `STATIC` if the subprogram operates independently of any particular object instance.

proc_name

The name of the procedure to create.

`SELF [ IN | IN OUT ]` *name*

For a member method there is an implicit, built-in parameter named `SELF` whose data type is that of the object type being defined. `SELF` refers to the object instance that is currently invoking the method. `SELF` can be explicitly declared as an `IN` or `IN OUT` parameter in the parameter list. If explicitly declared, `SELF` must be the first parameter in the parameter list. If `SELF` is not explicitly declared, its parameter mode defaults to `IN OUT` for member procedures and `IN` for member functions.

argname

The name of an argument. The argument is referenced by this name within the method body.

argtype

The data type(s) of the method's arguments. The argument types may be a base data type or a user-defined type such as a nested table or an object type. A length must not be specified for any base type - for example, specify `VARCHAR2`, not `VARCHAR2(10)`.

`DEFAULT` *value*

Supplies a default value for an input argument if one is not supplied in the method call. `DEFAULT` may not be specified for arguments with modes `IN OUT` or `OUT`.

program_body

The declarations and SPL statements that comprise the body of the function or procedure.

func_name

The name of the function to create.

rettype

The return data type, which may be any of the types listed for *argtype*. As for *argtype*, a length must not be specified for *rettype*.

Examples

Create the object type body for object type `emp_obj_typ` given in the example for the `CREATE TYPE` command.

```
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Employee No   : ' || empno);
        DBMS_OUTPUT.PUT_LINE('Name          : ' || ename);
        DBMS_OUTPUT.PUT_LINE('Street        : ' || addr.street);
        DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || addr.city || ', ' ||
            addr.state || ' ' || LPAD(addr.zip,5,'0'));
    END;
END;
```

Create the object type body for object type `dept_obj_typ` given in the example for the `CREATE TYPE` command.

```
CREATE OR REPLACE TYPE BODY dept_obj_typ AS
    STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2
    IS
        v_dname     VARCHAR2(14);
    BEGIN
        CASE p_deptno
            WHEN 10 THEN v_dname := 'ACCOUNING';
            WHEN 20 THEN v_dname := 'RESEARCH';
            WHEN 30 THEN v_dname := 'SALES';
            WHEN 40 THEN v_dname := 'OPERATIONS';
            ELSE v_dname := 'UNKNOWN';
        END CASE;
        RETURN v_dname;
    END;
```

```
    MEMBER PROCEDURE display_dept
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Dept No    : ' || SELF.deptno);
        DBMS_OUTPUT.PUT_LINE('Dept Name  : ' ||
            dept_obj_typ.get_dname(SELF.deptno));
    END;
END;
```

See Also

CREATE TYPE, DROP TYPE

2.3.38 CREATE USER

Name

`CREATE USER` -- define a new database user account

Synopsis

`CREATE USER` *`name`* `[IDENTIFIED BY` *`password`*`]`

Description

`CREATE USER` adds a new user to an Advanced Server database cluster. You must be a database superuser to use this command.

When the `CREATE USER` command is given, a schema will also be created with the same name as the new user and owned by the new user. Objects with unqualified names created by this user will be created in this schema.

Parameters

`name`

The name of the user.

`password`

The user's password. The password can be changed later using `ALTER USER`.

Notes

The maximum length allowed for the user name and password is 63 characters.

Examples

Create a user named, `john`.

```
CREATE USER john IDENTIFIED BY abc;
```

See Also

DROP USER

2.3.39 CREATE USER|ROLE… PROFILE MANAGEMENT CLAUSES

Name

`CREATE USER|ROLE`

Synopsis

`CREATE USER|ROLE` *`name`* `[[WITH]` *`option`* `[…]]`

where `option` can be the following compatible clauses:

```
   PROFILE profile_name
|  ACCOUNT {LOCK|UNLOCK}
|  PASSWORD EXPIRE [AT 'timestamp']
```

or `option` can be the following non-compatible clauses:

```
|  LOCK TIME 'timestamp'
```

For information about the administrative clauses of the `CREATE USER` or `CREATE ROLE` command that are supported by Advanced Server, please see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/sql-commands.html

Description

`CREATE ROLE|USER… PROFILE` adds a new role with an associated profile to an Advanced Server database cluster.

Roles created with the `CREATE USER` command are (by default) login roles. Roles created with the `CREATE ROLE` command are (by default) not login roles. To create a login account with the `CREATE ROLE` command, you must include the `LOGIN` keyword.

Only a database superuser can use the `CREATE USER|ROLE` clauses that enforce profile management; these clauses enforce the following behaviors:

Include the `PROFILE` clause and a *`profile_name`* to associate a pre-defined profile with a role, or to change which pre-defined profile is associated with a user.

Include the ACCOUNT clause and the LOCK or UNLOCK keyword to specify that the user account should be placed in a locked or unlocked state.

Include the LOCK TIME '*timestamp*' clause and a date/time value to lock the role at the specified time, and unlock the role at the time indicated by the PASSWORD_LOCK_TIME parameter of the profile assigned to this role. If LOCK TIME is used with the ACCOUNT LOCK clause, the role can only be unlocked by a database superuser with the ACCOUNT UNLOCK clause.

Include the PASSWORD EXPIRE clause with the optional AT '*timestamp*' keywords to specify a date/time when the password associated with the role will expire. If you omit the AT '*timestamp*' keywords, the password will expire immediately.

Each login role may only have one profile. To discover the profile that is currently associated with a login role, query the profile column of the DBA_USERS view.

Parameters

name

The name of the role.

profile_name

The name of the profile associated with the role.

timestamp

The date and time at which the clause will be enforced. When specifying a value for *timestamp*, enclose the value in single-quotes.

Examples

The following example uses CREATE USER to create a login role named john who is associated with the acctg_profile profile:

```
CREATE USER john PROFILE acctg_profile IDENTIFIED BY "1safepwd";
```

john can log in to the server, using the password 1safepwd.

The following example uses CREATE ROLE to create a login role named john who is associated with the acctg_profile profile:

```
CREATE ROLE john PROFILE acctg_profile LOGIN PASSWORD "1safepwd";
```

`john` can log in to the server, using the password `1safepwd`.

2.3.40 CREATE VIEW

Name

`CREATE VIEW` -- define a new view

Synopsis

```
CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ]
  AS query
```

Description

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

`CREATE OR REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced.

If a schema name is given (for example, `CREATE VIEW myschema.myview` ...) then the view is created in the specified schema. Otherwise it is created in the current schema. The view name must be distinct from the name of any other view, table, sequence, or index in the same schema.

Parameters

name

The name (optionally schema-qualified) of a view to be created.

column_name

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

query

A query (that is, a `SELECT` statement) which will provide the columns and rows of the view.

Refer to `SELECT` for more information about valid queries.

Notes

Currently, views are read only - the system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts, etc. on the view into appropriate actions on other tables.

Access to tables referenced in the view is determined by permissions of the view owner. However, functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore the user of a view must have permissions to call all functions used by the view.

Examples

Create a view consisting of all employees in department 30:

```
CREATE VIEW dept_30 AS SELECT * FROM emp WHERE deptno = 30;
```

See Also

DROP VIEW

2.3.41 DELETE

Name

DELETE -- delete rows of a table

Synopsis

```
DELETE [ optimizer_hint ] FROM table[@dblink ]
  [ WHERE condition ]
  [ RETURNING return_expression [, ...]
      { INTO { record | variable [, ...] }
      | BULK COLLECT INTO collection [, ...] } ]
```

Description

DELETE deletes rows that satisfy the WHERE clause from the specified table. If the WHERE clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

Note: The TRUNCATE command provides a faster mechanism to remove all rows from a table.

The RETURNING INTO { *record* | *variable* [, ...] } clause may only be specified if the DELETE command is used within an SPL program. In addition the result set of the DELETE command must not include more than one row, otherwise an exception is thrown. If the result set is empty, then the contents of the target record or variables are set to null.

The RETURNING BULK COLLECT INTO *collection* [, ...] clause may only be specified if the DELETE command is used within an SPL program. If more than one *collection* is specified as the target of the BULK COLLECT INTO clause, then each *collection* must consist of a single, scalar field – i.e., *collection* must not be a record. The result set of the DELETE command may contain none, one, or more rows. *return_expression* evaluated for each row of the result set, becomes an element in *collection* starting with the first element. Any existing rows in *collection* are deleted. If the result set is empty, then *collection* will be empty.

You must have the DELETE privilege on the table to delete from it, as well as the SELECT privilege for any table whose values are read in the condition.

Parameters

optimizer_hint

Comment-embedded hints to the optimizer for selection of an execution plan.

table

The name (optionally schema-qualified) of an existing table.

dblink

Database link name identifying a remote database. See the CREATE DATABASE LINK command for information on database links.

condition

A value expression that returns a value of type BOOLEAN that determines the rows which are to be deleted.

return_expression

An expression that may include one or more columns from *table*. If a column name from *table* is specified in *return_expression*, the value substituted for the column when *return_expression* is evaluated is the value from the deleted row.

record

A record whose field the evaluated *return_expression* is to be assigned. The first *return_expression* is assigned to the first field in *record*, the second *return_expression* is assigned to the second field in *record*, etc. The number of fields in *record* must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

variable

A variable to which the evaluated *return_expression* is to be assigned. If more than one *return_expression* and *variable* are specified, the *first return_expression* is assigned to the first *variable*, the second *return_expression* is assigned to the second *variable*, etc. The number of variables specified following the INTO keyword must exactly match the number of expressions following the RETURNING keyword and the variables must be type-compatible with their assigned expressions.

collection

A collection in which an element is created from the evaluated *return_expression*. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding *return_expression* and *collection* field must be type-compatible.

Examples

Delete all rows for employee 7900 from the `jobhist` table:

```
DELETE FROM jobhist WHERE empno = 7900;
```

Clear the table `jobhist`:

```
DELETE FROM jobhist;
```

See Also

TRUNCATE

2.3.42 DROP DATABASE LINK

Name

`DROP DATABASE LINK` -- remove a database link

Synopsis

`DROP [ PUBLIC ] DATABASE LINK` *`name`*

Description

`DROP DATABASE LINK` drops existing database links. To execute this command you must be a superuser or the owner of the database link.

Parameters

`name`

The name of a database link to be removed.

`PUBLIC`

Indicates that *`name`* is a public database link.

Examples

Remove the public database link named, `oralink`:

```
DROP PUBLIC DATABASE LINK oralink;
```

Remove the private database link named, `edblink`:

```
DROP DATABASE LINK edblink;
```

See Also

CREATE DATABASE LINK

2.3.43 DROP DIRECTORY

Name

`DROP DIRECTORY` -- remove a directory alias for a file system directory path

Synopsis

```
DROP DIRECTORY name
```

Description

`DROP DIRECTORY` drops an existing alias for a file system directory path that was created with the `CREATE DIRECTORY` command. To execute this command you must be a superuser.

When a directory alias is deleted, the corresponding physical file system directory is not affected. The file system directory must be deleted using the appropriate operating system commands.

Parameters

name

The name of a directory alias to be removed.

Examples

Remove the directory alias named `empdir`:

```
DROP DIRECTORY empdir;
```

See Also

CREATE DIRECTORY

2.3.44 DROP FUNCTION

Name

`DROP FUNCTION` -- remove a function

Synopsis

```
DROP FUNCTION name
  [ ([ [ argmode ] [ argname ] argtype ] [, ...]) ]
```

Description

`DROP FUNCTION` removes the definition of an existing function. To execute this command you must be a superuser or the owner of the function. All input (`IN`, `IN OUT`) argument data types to the function must be specified if there is at least one input argument. (This requirement is not compatible with Oracle databases. In Oracle, only the function name is specified. Advanced Server allows overloading of function names, so the function signature given by the input argument data types is required in the Advanced Server `DROP FUNCTION` command.)

Parameters

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: `IN`, `IN OUT`, or `OUT`. If omitted, the default is `IN`. Note that `DROP FUNCTION` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list only the `IN` and `IN OUT` arguments. (Specification of *argmode* is not compatible with Oracle databases and applies only to Advanced Server.)

argname

The name of an argument. Note that `DROP FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity. (Specification of *argname* is not compatible with Oracle databases and applies only to Advanced Server.)

argtype

The data type of an argument of the function. (Specification of *argtype* is not compatible with Oracle databases and applies only to Advanced Server.)

Examples

The following command removes the emp_comp function.

```
DROP FUNCTION emp_comp(NUMBER, NUMBER);
```

See Also

CREATE FUNCTION

2.3.45 DROP INDEX

Name

`DROP INDEX` -- remove an index

Synopsis

`DROP INDEX` *`name`*

Description

`DROP INDEX` drops an existing index from the database system. To execute this command you must be a superuser or the owner of the index. If any objects depend on the index, an error will be given and the index will not be dropped.

Parameters

`name`

The name (optionally schema-qualified) of an index to remove.

Examples

This command will remove the index, `name_idx`:

```
DROP INDEX name_idx;
```

See Also

ALTER INDEX, CREATE INDEX

2.3.46 DROP PACKAGE

Name

`DROP PACKAGE` -- remove a package

Synopsis

```
DROP PACKAGE [ BODY ] name
```

Description

`DROP PACKAGE` drops an existing package. To execute this command you must be a superuser or the owner of the package. If `BODY` is specified, only the package body is removed – the package specification is not dropped. If `BODY` is omitted, both the package specification and body are removed.

Parameters

name

The name (optionally schema-qualified) of a package to remove.

Examples

This command will remove the `emp_admin` package:

```
DROP PACKAGE emp_admin;
```

See Also

CREATE PACKAGE, CREATE PACKAGE BODY

2.3.47 DROP PROCEDURE

Name

`DROP PROCEDURE` -- remove a procedure

Synopsis

`DROP PROCEDURE` *`name`*

Description

`DROP PROCEDURE` removes the definition of an existing procedure. To execute this command you must be a superuser or the owner of the procedure.

Parameters

`name`

The name (optionally schema-qualified) of an existing procedure.

Examples

The following command removes the `select_emp` procedure.

```
DROP PROCEDURE select_emp;
```

See Also

CREATE PROCEDURE

2.3.48 DROP PROFILE

Name

`DROP PROFILE` – drop a user-defined profile

Synopsis

```
DROP PROFILE [IF EXISTS] profile_name [CASCADE | RESTRICT];
```

Description

Include the `IF EXISTS` clause to instruct the server to not throw an error if the specified profile does not exist. The server will issue a notice if the profile does not exist.

Include the optional `CASCADE` clause to reassign any users that are currently associated with the profile to the `default` profile, and then drop the profile. Include the optional `RESTRICT` clause to instruct the server to not drop any profile that is associated with a role. This is the default behavior.

Parameters

profile_name

The name of the profile being dropped.

Example

The following example drops a profile named `acctg_profile`:

```
DROP PROFILE acctg_profile CASCADE;
```

The command first re-associates any roles associated with the `acctg_profile` profile with the `default` profile, and then drops the `acctg_profile` profile.

The following example drops a profile named `acctg_profile`:

```
DROP PROFILE acctg_profile RESTRICT;
```

The `RESTRICT` clause in the command instructs the server to not drop `acctg_profile` if there are any roles associated with the profile.

2.3.49 DROP QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the `DROP QUEUE SQL` command. This syntax can be used in association with `DBMS_AQADM`.

Name

`DROP QUEUE` -- drop an existing queue.

Synopsis

Use `DROP QUEUE` to drop an existing queue:

```
DROP QUEUE [IF EXISTS] name
```

Description

`DROP QUEUE` allows a superuser or a user with the `aq_administrator_role` privilege to drop an existing queue.

Parameters

name

The name (possibly schema-qualified) of the queue that is being dropped.

`IF EXISTS`

Include the `IF EXISTS` clause to instruct the server to not return an error if the queue does not exist. The server will issue a notice.

Examples

The following example drops a queue named `work_order`:

```
DROP QUEUE work_order;
```

See Also

CREATE QUEUE, ALTER QUEUE

2.3.50 DROP QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the DROP QUEUE TABLE SQL command. This syntax can be used in association with DBMS_AQADM.

Name

DROP QUEUE TABLE-- drop a queue table.

Synopsis

Use DROP QUEUE TABLE to delete a queue table:

```
DROP QUEUE TABLE [ IF EXISTS ] name [, ...]
[CASCADE | RESTRICT]
```

Description

DROP QUEUE TABLE allows a superuser or a user with the aq_administrator_role privilege to delete a queue table.

Parameters

name

The name (possibly schema-qualified) of the queue table that will be deleted.

IF EXISTS

Include the IF EXISTS clause to instruct the server to not return an error if the queue table does not exist. The server will issue a notice.

CASCADE

Include the CASCADE keyword to automatically delete any objects that depend on the queue table.

RESTRICT

Include the RESTRICT keyword to instruct the server to refuse to delete the queue table if any objects depend on it. This is the default.

Example

The following example deletes a queue table named `work_order_table` and any objects that depend on it:

```
DROP QUEUE TABLE work_order_table CASCADE;
```

See Also

CREATE QUEUE TABLE, ALTER QUEUE TABLE

2.3.51 DROP SYNONYM

Name

`DROP SYNONYM` -- remove a synonym

Synopsis

```
DROP [PUBLIC] SYNONYM [schema.]syn_name
```

Description

`DROP SYNONYM` deletes existing synonyms. To execute this command you must be a superuser or the owner of the synonym, and have `USAGE` privileges on the schema in which the synonym resides.

Parameters:

syn_name

> *syn_name* is the name of the synonym. A synonym name must be unique within a schema.

schema

> *schema* specifies the name of the schema that the synonym resides in.

Like any other object that can be schema-qualified, you may have two synonyms with the same name in your search path. To disambiguate the name of the synonym that you are dropping, include a schema name. Unless a synonym is schema qualified in the `DROP SYNONYM` command, Advanced Server deletes the first instance of the synonym it finds in your search path.

You can optionally include the `PUBLIC` clause to drop a synonym that resides in the `public` schema. The `DROP PUBLIC SYNONYM` command, compatible with Oracle databases, drops a synonym that resides in the `public` schema:

```
DROP PUBLIC SYNONYM syn_name;
```

The following example drops the synonym, `personnel`:

```
DROP SYNONYM personnel;
```

2.3.52 DROP ROLE

Name

`DROP ROLE` -- remove a database role

Synopsis

```
DROP ROLE name [ CASCADE ]
```

Description

`DROP ROLE` removes the specified role. To drop a superuser role, you must be a superuser yourself; to drop non-superuser roles, you must have `CREATEROLE` privilege.

A role cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the role, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the role has been granted.

It is not necessary to remove role memberships involving the role; `DROP ROLE` automatically revokes any memberships of the target role in other roles, and of other roles in the target role. The other roles are not dropped nor otherwise affected.

Alternatively, if the only objects owned by the role belong within a schema that is owned by the role and has the same name as the role, the `CASCADE` option can be specified. In this case the issuer of the `DROP ROLE name CASCADE` command must be a superuser and the named role, the schema, and all objects within the schema will be deleted.

Parameters

name

The name of the role to remove.

`CASCADE`

If specified, also drops the schema owned by, and with the same name as the role (and all objects owned by the role belonging to the schema) as long as no other dependencies on the role or the schema exist.

Examples

To drop a role:

```
DROP ROLE admins;
```

See Also

CREATE ROLE, SET ROLE, GRANT, REVOKE

2.3.53 DROP SEQUENCE

Name

`DROP SEQUENCE` -- remove a sequence

Synopsis

```
DROP SEQUENCE name [, ...]
```

Description

`DROP SEQUENCE` removes sequence number generators. To execute this command you must be a superuser or the owner of the sequence.

Parameters

name

The name (optionally schema-qualified) of a sequence.

Examples

To remove the sequence, `serial`:

```
DROP SEQUENCE serial;
```

See Also

ALTER SEQUENCE, CREATE SEQUENCE

2.3.54 DROP TABLE

Name

DROP TABLE -- remove a table

Synopsis

```
DROP TABLE name [CASCADE | RESTRICT | CASCADE CONSTRAINTS]
```

Description

DROP TABLE removes tables from the database. Only its owner may destroy a table. To empty a table of rows, without destroying the table, use DELETE. DROP TABLE always removes any indexes, rules, triggers, and constraints that exist for the target table.

Parameters

name

The name (optionally schema-qualified) of the table to drop.

Include the RESTRICT keyword to specify that the server should refuse to drop the table if any objects depend on it. This is the default behavior; the DROP TABLE command will report an error if any objects depend on the table.

Include the CASCADE clause to drop any objects that depend on the table.

Include the CASCADE CONSTRAINTS clause to specify that Advanced Server should drop any dependent constraints (excluding other object types) on the specified table.

Examples

The following command drops a table named emp that has no dependencies:

```
DROP TABLE emp;
```

The outcome of a DROP TABLE command will vary depending on whether the table has any dependencies - you can control the outcome by specifying a *drop behavior*. For example, if you create two tables, orders and items, where the items table is dependent on the orders table:

```
CREATE TABLE orders
  (order_id int PRIMARY KEY, order_date date, …);
```

```
CREATE TABLE items
  (order_id REFERENCES orders, quantity int, …);
```

Advanced Server will perform one of the following actions when dropping the `orders` table, depending on the drop behavior that you specify:

- If you specify `DROP TABLE orders RESTRICT`, Advanced Server will report an error.
- If you specify `DROP TABLE orders CASCADE`, Advanced Server will drop the `orders` table *and* the `items` table.
- If you specify `DROP TABLE orders CASCADE CONSTRAINTS`, Advanced Server will drop the `orders` table and remove the foreign key specification from the `items` table, but not drop the `items` table.

See Also

ALTER TABLE, CREATE TABLE

2.3.55 DROP TABLESPACE

Name

`DROP TABLESPACE` -- remove a tablespace

Synopsis

`DROP TABLESPACE` *`tablespacename`*

Description

`DROP TABLESPACE` removes a tablespace from the system.

A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all database objects before it can be dropped. It is possible that objects in other databases may still reside in the tablespace even if no objects in the current database are using the tablespace.

Parameters

`tablespacename`

The name of a tablespace.

Examples

To remove tablespace `employee_space` from the system:

```
DROP TABLESPACE employee_space;
```

See Also

ALTER TABLESPACE

2.3.56 DROP TRIGGER

Name

`DROP TRIGGER` -- remove a trigger

Synopsis

```
DROP TRIGGER name
```

Description

`DROP TRIGGER` removes a trigger from its associated table. The command must be run by a superuser or the owner of the table on which the trigger is defined.

Parameters

name

The name of a trigger to remove.

Examples

Remove trigger `emp_sal_trig`:

```
DROP TRIGGER emp_sal_trig;
```

See Also

CREATE TRIGGER

2.3.57 DROP TYPE

Name

`DROP TYPE` -- remove a type definition

Synopsis

```
DROP TYPE [ BODY ] name
```

Description

`DROP TYPE` removes the type definition. To execute this command you must be a superuser or the owner of the type.

The optional `BODY` qualifier applies only to object type definitions, not to collection types nor to composite types. If `BODY` is specified, only the object type body is removed – the object type specification is not dropped. If `BODY` is omitted, both the object type specification and body are removed.

The type will not be deleted if there are other database objects dependent upon the named type.

Parameters

name

The name of a type definition to remove.

Examples

Drop object type `addr_obj_typ`.

```
DROP TYPE addr_obj_typ;
```

Drop nested table type `budget_tbl_typ`.

```
DROP TYPE budget_tbl_typ;
```

See Also

CREATE TYPE, CREATE TYPE BODY

2.3.58 DROP USER

Name

`DROP USER` -- remove a database user account

Synopsis

```
DROP USER name [ CASCADE ]
```

Description

`DROP USER` removes the specified user. To drop a superuser, you must be a superuser yourself; to drop non-superusers, you must have `CREATEROLE` privilege.

A user cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the user, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the user has been granted.

However, it is not necessary to remove role memberships involving the user; `DROP USER` automatically revokes any memberships of the target user in other roles, and of other roles in the target user. The other roles are not dropped nor otherwise affected.

Alternatively, if the only objects owned by the user belong within a schema that is owned by the user and has the same name as the user, the `CASCADE` option can be specified. In this case the issuer of the `DROP USER name CASCADE` command must be a superuser and the named user, the schema, and all objects within the schema will be deleted.

Parameters

name

The name of the user to remove.

`CASCADE`

If specified, also drops the schema owned by, and with the same name as the user (and all objects owned by the user belonging to the schema) as long as no other dependencies on the user or the schema exist.

Examples

To drop a user account who owns no objects nor has been granted any privileges on other objects:

```
DROP USER john;
```

To drop user account, `john`, who has not been granted any privileges on any objects, and does not own any objects outside of a schema named, `john`, that is owned by user, `john`:

```
DROP USER john CASCADE;
```

See Also

CREATE USER, ALTER USER

2.3.59 DROP VIEW

Name

`DROP VIEW` -- remove a view

Synopsis

```
DROP VIEW name
```

Description

`DROP VIEW` drops an existing view. To execute this command you must be a database superuser or the owner of the view. The named view will not be deleted if other objects are dependent upon this view (such as a view of a view).

The form of the `DROP VIEW` command compatible with Oracle does not support a `CASCADE` clause; to drop a view and it's dependencies, use the PostgreSQL-compatible form of the `DROP VIEW` command. For more information, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/10/static/sql-dropview.html

Parameters

name

The name (optionally schema-qualified) of the view to remove.

Examples

This command will remove the view called `dept_30`:

```
DROP VIEW dept_30;
```

See Also

CREATE VIEW

2.3.60 EXEC

Name

EXEC

Synopsis

EXEC *function_name* ['('[*argument_list*]')']

Description

EXECUTE.

Parameters

procedure_name

procedure_name is the (optionally schema-qualified) function name.

argument_list

argument_list specifies a comma-separated list of arguments required by the function. Note that each member of *argument_list* corresponds to a formal argument expected by the function. Each formal argument may be an IN parameter, an OUT parameter, or an INOUT parameter.

Examples

The EXEC statement may take one of several forms, depending on the arguments required by the function:

```
EXEC update_balance;
EXEC update_balance();
EXEC update_balance(1,2,3);
```

2.3.61 GRANT

Name

GRANT -- define access privileges

Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES }
  [,...] | ALL [ PRIVILEGES ] }
  ON tablename
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { { INSERT | UPDATE | REFERENCES } (column [, ...]) }
  [, ...]
  ON tablename
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { SELECT | ALL [ PRIVILEGES ] }
  ON sequencename
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON FUNCTION progname
    ( [ [ argmode ] [ argname ] argtype ] [, ...] )
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON PROCEDURE progname
    [ ( [ [ argmode ] [ argname ] argtype ] [, ...] ) ]
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON PACKAGE packagename
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT role [, ...]
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH ADMIN OPTION ]

GRANT { CONNECT | RESOURCE | DBA } [, ...]
  TO { username | groupname } [, ...]
```

```
  [ WITH ADMIN OPTION ]

GRANT CREATE [ PUBLIC ] DATABASE LINK
  TO { username | groupname }

GRANT DROP PUBLIC DATABASE LINK
  TO { username | groupname }

GRANT EXEMPT ACCESS POLICY
  TO { username | groupname }
```

Description

The GRANT command has three basic variants: one that grants privileges on a database object (table, view, sequence, or program), one that grants membership in a role, and one that grants system privileges. These variants are similar in many ways, but they are different enough to be described separately.

In Advanced Server, the concept of users and groups has been unified into a single type of entity called a *role*. In this context, a *user* is a role that has the LOGIN attribute – the role may be used to create a session and connect to an application. A *group* is a role that does not have the LOGIN attribute – the role may not be used to create a session or connect to an application.

A role may be a member of one or more other roles, so the traditional concept of users belonging to groups is still valid. However, with the generalization of users and groups, users may "belong" to users, groups may "belong" to groups, and groups may "belong" to users, forming a general multi-level hierarchy of roles. User names and group names share the same namespace therefore it is not necessary to distinguish whether a grantee is a user or a group in the GRANT command.

2.3.62 GRANT on Database Objects

This variant of the GRANT command gives specific privileges on a database object to a role. These privileges are added to those already granted, if any.

The key word PUBLIC indicates that the privileges are to be granted to all roles, including those that may be created later. PUBLIC may be thought of as an implicitly defined group that always includes all roles. Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC.

If the WITH GRANT OPTION is specified, the recipient of the privilege may in turn grant it to others. Without a grant option, the recipient cannot do that. Grant options cannot be granted to PUBLIC.

There is no need to grant privileges to the owner of an object (usually the user that created it), as the owner has all privileges by default. (The owner could, however, choose to revoke some of his own privileges for safety.) The right to drop an object or to alter its definition in any way is not described by a grantable privilege; it is inherent in the owner, and cannot be granted or revoked. The owner implicitly has all grant options for the object as well.

Depending on the type of object, the initial default privileges may include granting some privileges to PUBLIC. The default is no public access for tables and EXECUTE privilege for functions, procedures, and packages. The object owner may of course revoke these privileges. (For maximum security, issue the REVOKE in the same transaction that creates the object; then there is no window in which another user may use the object.)

The possible privileges are:

SELECT

> Allows SELECT from any column of the specified table, view, or sequence. For sequences, this privilege also allows the use of the currval function.

INSERT

> Allows INSERT of a new row into the specified table.

UPDATE

> Allows UPDATE of a column of the specified table. SELECT ... FOR UPDATE also requires this privilege (besides the SELECT privilege).

`DELETE`

Allows `DELETE` of a row from the specified table.

`REFERENCES`

To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.

`EXECUTE`

Allows the use of the specified package, procedure, or function. When applied to a package, allows the use of all of the package's public procedures, public functions, public variables, records, cursors and other public objects and object types. This is the only type of privilege that is applicable to functions, procedures, and packages.

The Advanced Server syntax for granting the `EXECUTE` privilege is not fully compatible with Oracle databases. Advanced Server requires qualification of the program name by one of the keywords, `FUNCTION`, `PROCEDURE`, or `PACKAGE` whereas these keywords must be omitted in Oracle. For functions, Advanced Server requires all input (`IN`, `IN OUT`) argument data types after the function name (including an empty parenthesis if there are no function arguments). For procedures, all input argument data types must be specified if the procedure has one or more input arguments. In Oracle, function and procedure signatures must be omitted. This is due to the fact that all programs share the same namespace in Oracle, whereas functions, procedures, and packages have their own individual namespace in Advanced Server to allow program name overloading to a certain extent.

`ALL PRIVILEGES`

Grant all of the available privileges at once.

The privileges required by other commands are listed on the reference page of the respective command.

2.3.63 GRANT on Roles

This variant of the GRANT command grants membership in a role to one or more other roles. Membership in a role is significant because it conveys the privileges granted to a role to each of its members.

If the WITH ADMIN OPTION is specified, the member may in turn grant membership in the role to others, and revoke membership in the role as well. Without the admin option, ordinary users cannot do that.

Database superusers can grant or revoke membership in any role to anyone. Roles having the CREATEROLE privilege can grant or revoke membership in any role that is not a superuser.

There are three pre-defined roles that have the following meanings:

CONNECT

> Granting the CONNECT role is equivalent to giving the grantee the LOGIN privilege. The grantor must have the CREATEROLE privilege.

RESOURCE

> Granting the RESOURCE role is equivalent to granting the CREATE and USAGE privileges on a schema that has the same name as the grantee. This schema must exist before the grant is given. The grantor must have the privilege to grant CREATE or USAGE privileges on this schema to the grantee.

DBA

> Granting the DBA role is equivalent to making the grantee a superuser. The grantor must be a superuser.

Notes

The REVOKE command is used to revoke access privileges.

When a non-owner of an object attempts to GRANT privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as a privilege is available, the command will proceed, but it will grant only those privileges for which the user has grant options. The GRANT ALL PRIVILEGES forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held.

(In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

It should be noted that database superusers can access all objects regardless of object privilege settings. This is comparable to the rights of `root` in a Unix system. As with `root`, it's unwise to operate as a superuser except when absolutely necessary.

If a superuser chooses to issue a `GRANT` or `REVOKE` command, the command is performed as though it were issued by the owner of the affected object. In particular, privileges granted via such a command will appear to have been granted by the object owner. (For role membership, the membership appears to have been granted by the containing role itself.)

`GRANT` and `REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the privileges will be recorded as having been granted by the role that actually owns the object or holds the privileges `WITH GRANT OPTION`.

> For example, if table `t1` is owned by role `g1`, of which role `u1` is a member, then `u1` can grant privileges on `t1` to `u2`, but those privileges will appear to have been granted directly by `g1`. Any other member of role `g1` could revoke them later.

If the role executing `GRANT` holds the required privileges indirectly via more than one role membership path, it is unspecified which containing role will be recorded as having done the grant. In such cases it is best practice to use `SET ROLE` to become the specific role you want to do the `GRANT` as.

Currently, Advanced Server does not support granting or revoking privileges for individual columns of a table. One possible workaround is to create a view having just the desired columns and then grant privileges to that view.

Examples

Grant insert privilege to all users on table `emp`:

```
GRANT INSERT ON emp TO PUBLIC;
```

Grant all available privileges to user `mary` on view `salesemp`:

```
GRANT ALL PRIVILEGES ON salesemp TO mary;
```

Note that while the above will indeed grant all privileges if executed by a superuser or the owner of `emp`, when executed by someone else it will only grant those permissions for which the someone else has grant options.

Grant membership in role `admins` to user `joe`:

```
GRANT admins TO joe;
```

Grant `CONNECT` privilege to user `joe`:

```
GRANT CONNECT TO joe;
```

See Also

REVOKE, SET ROLE

2.3.64 GRANT on System Privileges

This variant of the GRANT command gives a role the ability to perform certain *system* operations within a database. System privileges relate to the ability to create or delete certain database objects that are not necessarily within the confines of one schema. Only database superusers can grant system privileges.

```
CREATE [PUBLIC] DATABASE LINK
```

The CREATE [PUBLIC] DATABASE LINK privilege allows the specified role to create a database link. Include the PUBLIC keyword to allow the role to create public database links; omit the PUBLIC keyword to allow the specified role to create private database links.

```
DROP PUBLIC DATABASE LINK
```

The DROP PUBLIC DATABASE LINK privilege allows a role to drop a public database link. System privileges are not required to drop a private database link. A private database link may be dropped by the link owner or a database superuser.

```
EXEMPT ACCESS POLICY
```

The EXEMPT ACCESS POLICY privilege allows a role to execute a SQL command without invoking any policy function that may be associated with the target database object. That is, the role is exempt from all security policies in the database.

The EXEMPT ACCESS POLICY privilege is not inheritable by membership to a role that has the EXEMPT ACCESS POLICY privilege. For example, the following sequence of GRANT commands does not result in user joe obtaining the EXEMPT ACCESS POLICY privilege even though joe is granted membership to the enterprisedb role, which has been granted the EXEMPT ACCESS POLICY privilege:

```
GRANT EXEMPT ACCESS POLICY TO enterprisedb;
GRANT enterprisedb TO joe;
```

The rolpolicyexempt column of the system catalog table pg_authid is set to true if a role has the EXEMPT ACCESS POLICY privilege.

Examples

Grant `CREATE PUBLIC DATABASE LINK` privilege to user `joe`:

```
GRANT CREATE PUBLIC DATABASE LINK TO joe;
```

Grant `DROP PUBLIC DATABASE LINK` privilege to user `joe`:

```
GRANT DROP PUBLIC DATABASE LINK TO joe;
```

Grant the `EXEMPT ACCESS POLICY` privilege to user `joe`:

```
GRANT EXEMPT ACCESS POLICY TO joe;
```

Using the ALTER ROLE Command to Assign System Privileges

The Advanced Server `ALTER ROLE` command also supports syntax that you can use to assign:

- the privilege required to create a public or private database link.
- the privilege required to drop a public database link.
- the `EXEMPT ACCESS POLICY` privilege.

The `ALTER ROLE` syntax is functionally equivalent to the respective commands compatible with Oracle databases.

See Also

REVOKE, ALTER ROLE

2.3.65 INSERT

Name

INSERT -- create new rows in a table

Synopsis

```
INSERT INTO table[@dblink ] [ ( column [, ...] ) ]
  { VALUES ( { expression | DEFAULT } [, ...] )
    [ RETURNING return_expression [, ...]
        { INTO { record | variable [, ...] }
        | BULK COLLECT INTO collection [, ...] } ]
  | query }
```

Description

INSERT allows you to insert new rows into a table. You can insert a single row at a time or several rows as a result of a query.

The columns in the target list may be listed in any order. Each column not present in the target list will be inserted using a default value, either its declared default value or null.

If the expression for each column is not of the correct data type, automatic type conversion will be attempted.

The RETURNING INTO { *record* | *variable* [, ...] } clause may only be specified when the INSERT command is used within an SPL program and only when the VALUES clause is used.

The RETURNING BULK COLLECT INTO *collection* [, ...] clause may only be specified if the INSERT command is used within an SPL program. If more than one *collection* is specified as the target of the BULK COLLECT INTO clause, then each *collection* must consist of a single, scalar field – i.e., *collection* must not be a record. *return_expression* evaluated for each inserted row, becomes an element in *collection* starting with the first element. Any existing rows in *collection* are deleted. If the result set is empty, then *collection* will be empty.

You must have INSERT privilege to a table in order to insert into it. If you use the *query* clause to insert rows from a query, you also need to have SELECT privilege on any table used in the query.

Parameters

table

The name (optionally schema-qualified) of an existing table.

dblink

Database link name identifying a remote database. See the CREATE DATABASE LINK command for information on database links.

column

The name of a column in *table*.

expression

An expression or value to assign to *column*.

DEFAULT

This column will be filled with its default value.

query

A query (SELECT statement) that supplies the rows to be inserted. Refer to the SELECT command for a description of the syntax.

return_expression

An expression that may include one or more columns from *table*. If a column name from *table* is specified in *return_expression*, the value substituted for the column when *return_expression* is evaluated is determined as follows:

If the column specified in *return_expression* is assigned a value in the INSERT command, then the assigned value is used in the evaluation of *return_expression*.

If the column specified in *return_expression* is not assigned a value in the INSERT command and there is no default value for the column in the table's column definition, then null is used in the evaluation of *return_expression*.

If the column specified in *return_expression* is not assigned a value in the INSERT command and there is a default value for the column in the

table's column definition, then the default value is used in the evaluation of *return_expression*.

record

A record whose field the evaluated *return_expression* is to be assigned. The first *return_expression* is assigned to the first field in *record*, the second *return_expression* is assigned to the second field in *record*, etc. The number of fields in *record* must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

variable

A variable to which the evaluated *return_expression* is to be assigned. If more than one *return_expression* and *variable* are specified, the first *return_expression* is assigned to the first *variable*, the second *return_expression* is assigned to the second *variable*, etc. The number of variables specified following the INTO keyword must exactly match the number of expressions following the RETURNING keyword and the variables must be type-compatible with their assigned expressions.

collection

A collection in which an element is created from the evaluated *return_expression*. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding *return_expression* and *collection* field must be type-compatible.

Examples

Insert a single row into table emp:

```
INSERT INTO emp VALUES (8021,'JOHN','SALESMAN',7698,'22-FEB-07',1250,500,30);
```

In this second example, the column, comm, is omitted and therefore it will have the default value of null:

```
INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, deptno)
    VALUES (8022,'PETERS','CLERK',7698,'03-DEC-06',950,30);
```

The third example uses the DEFAULT clause for the hiredate and comm columns rather than specifying a value:

```
INSERT INTO emp VALUES (8023,'FORD','ANALYST',7566,NULL,3000,NULL,20);
```

This example creates a table for the department names and then inserts into the table by selecting from the `dname` column of the `dept` table:

```
CREATE TABLE deptnames (
    deptname        VARCHAR2(14)
);
INSERT INTO deptnames SELECT dname FROM dept;
```

2.3.66 LOCK

Name

LOCK -- lock a table

Synopsis

```
LOCK TABLE name [, ...] IN lockmode MODE [ NOWAIT ]
```

where *lockmode* is one of:

```
ROW SHARE | ROW EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE |
EXCLUSIVE
```

Description

LOCK TABLE obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If NOWAIT is specified, LOCK TABLE does not wait to acquire the desired lock: if it cannot be acquired immediately, the command is aborted and an error is emitted. Once obtained, the lock is held for the remainder of the current transaction. (There is no UNLOCK TABLE command; locks are always released at transaction end.)

When acquiring locks automatically for commands that reference tables, Advanced Server always uses the least restrictive lock mode possible. LOCK TABLE provides for cases when you might need more restrictive locking. For example, suppose an application runs a transaction at the isolation level read committed and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain SHARE lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because SHARE lock mode conflicts with the ROW EXCLUSIVE lock acquired by writers, and your LOCK TABLE name IN SHARE MODE statement will wait until any concurrent holders of ROW EXCLUSIVE mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

To achieve a similar effect when running a transaction at the isolation level serializable, you have to execute the LOCK TABLE statement before executing any data modification statement. A serializable transaction’s view of data will be frozen when its first data modification statement begins. A later LOCK TABLE will still prevent concurrent writes - but it won’t ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort is going to change the data in the table, then it should use SHARE ROW EXCLUSIVE lock mode instead of SHARE mode.

This ensures that only one transaction of this type runs at a time. Without this, a deadlock is possible: two transactions might both acquire SHARE mode, and then be unable to also acquire ROW EXCLUSIVE mode to actually perform their updates. (Note that a transaction's own locks never conflict, so a transaction can acquire ROW EXCLUSIVE mode when it holds SHARE mode - but not if anyone else holds SHARE mode.) To avoid deadlocks, make sure all transactions acquire locks on the same objects in the same order, and if multiple lock modes are involved for a single object, then transactions should always acquire the most restrictive mode first.

Parameters

name

The name (optionally schema-qualified) of an existing table to lock.

The command LOCK TABLE a, b; is equivalent to LOCK TABLE a; LOCK TABLE b. The tables are locked one-by-one in the order specified in the LOCK TABLE command.

lockmode

The lock mode specifies which locks this lock conflicts with.

If no lock mode is specified, then the server uses the most restrictive mode, ACCESS EXCLUSIVE. (ACCESS EXCLUSIVE is not compatible with Oracle databases. In Advanced Server, this configuration mode ensures that no other transaction can access the locked table in any manner.)

NOWAIT

Specifies that LOCK TABLE should not wait for any conflicting locks to be released: if the specified lock cannot be immediately acquired without waiting, the transaction is aborted.

Notes

All forms of LOCK require UPDATE and/or DELETE privileges.

LOCK TABLE is useful only inside a transaction block since the lock is dropped as soon as the transaction ends. A LOCK TABLE command appearing outside any transaction block forms a self-contained transaction, so the lock will be dropped as soon as it is obtained.

`LOCK TABLE` only deals with table-level locks, and so the mode names involving `ROW` are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, `ROW EXCLUSIVE` mode is a sharable table lock. Keep in mind that all the lock modes have identical semantics so far as `LOCK TABLE` is concerned, differing only in the rules about which modes conflict with which.

2.3.67 REVOKE

Name

REVOKE -- remove access privileges

Synopsis

```
REVOKE { { SELECT | INSERT | UPDATE | DELETE | REFERENCES }
  [,...] | ALL [ PRIVILEGES ] }
  ON tablename
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { SELECT | ALL [ PRIVILEGES ] }
  ON sequencename
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
  ON FUNCTION progname
    ( [ [ argmode ] [ argname ] argtype ] [, ...] )
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
  ON PROCEDURE progname
    [ ( [ [ argmode ] [ argname ] argtype ] [, ...] ) ]
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
  ON PACKAGE packagename
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE role [, ...] FROM { username | groupname | PUBLIC }
  [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { CONNECT | RESOURCE | DBA } [, ...]
  FROM { username | groupname } [, ...]

REVOKE CREATE [ PUBLIC ] DATABASE LINK
  FROM { username | groupname }

REVOKE DROP PUBLIC DATABASE LINK
  FROM { username | groupname }
```

```
REVOKE EXEMPT ACCESS POLICY
  FROM { username | groupname }
```

Description

The REVOKE command revokes previously granted privileges from one or more roles. The key word PUBLIC refers to the implicitly defined group of all roles.

See the description of the GRANT command for the meaning of the privilege types.

Note that any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC. Thus, for example, revoking SELECT privilege from PUBLIC does not necessarily mean that all roles have lost SELECT privilege on the object: those who have it granted directly or via another role will still have it.

If the privilege had been granted with the grant option, the grant option for the privilege is revoked as well as the privilege, itself.

If a user holds a privilege with grant option and has granted it to other users then the privileges held by those other users are called dependent privileges. If the privilege or the grant option held by the first user is being revoked and dependent privileges exist, those dependent privileges are also revoked if CASCADE is specified, else the revoke action will fail. This recursive revocation only affects privileges that were granted through a chain of users that is traceable to the user that is the subject of this REVOKE command. Thus, the affected users may effectively keep the privilege if it was also granted through other users.

Note: CASCADE is not an option compatible with Oracle databases. By default Oracle always cascades dependent privileges, but Advanced Server requires the CASCADE keyword to be explicitly given, otherwise the REVOKE command will fail.

When revoking membership in a role, GRANT OPTION is instead called ADMIN OPTION, but the behavior is similar.

Notes

A user can only revoke privileges that were granted directly by that user. If, for example, user A has granted a privilege with grant option to user B, and user B has in turned granted it to user C, then user A cannot revoke the privilege directly from C. Instead, user A could revoke the grant option from user B and use the CASCADE option so that the privilege is in turn revoked from user C. For another example, if both A and B have granted the same privilege to C, A can revoke his own grant but not B's grant, so C will still effectively have the privilege.

When a non-owner of an object attempts to REVOKE privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command will proceed, but it will revoke only those privileges for which the user has grant options. The REVOKE ALL PRIVILEGES forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

If a superuser chooses to issue a GRANT or REVOKE command, the command is performed as though it were issued by the owner of the affected object. Since all privileges ultimately come from the object owner (possibly indirectly via chains of grant options), it is possible for a superuser to revoke all privileges, but this may require use of CASCADE as stated above.

REVOKE can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges WITH GRANT OPTION on the object. In this case the command is performed as though it were issued by the containing role that actually owns the object or holds the privileges WITH GRANT OPTION. For example, if table t1 is owned by role g1, of which role u1 is a member, then u1 can revoke privileges on t1 that are recorded as being granted by g1. This would include grants made by u1 as well as by other members of role g1.

If the role executing REVOKE holds privileges indirectly via more than one role membership path, it is unspecified which containing role will be used to perform the command. In such cases it is best practice to use SET ROLE to become the specific role you want to do the REVOKE as. Failure to do so may lead to revoking privileges other than the ones you intended, or not revoking anything at all.

Please Note: The Advanced Server ALTER ROLE command also supports syntax that revokes the system privileges required to create a public or private database link, or exemptions from fine-grained access control policies (DBMS_RLS). The ALTER ROLE syntax is functionally equivalent to the respective REVOKE command, compatible with Oracle databases.

Examples

Revoke insert privilege for the public on table emp:

```
REVOKE INSERT ON emp FROM PUBLIC;
```

Revoke all privileges from user mary on view salesemp:

```
REVOKE ALL PRIVILEGES ON salesemp FROM mary;
```

Note that this actually means “revoke all privileges that I granted”.

Revoke membership in role `admins` from user `joe`:

```
REVOKE admins FROM joe;
```

Revoke `CONNECT` privilege from user `joe`:

```
REVOKE CONNECT FROM joe;
```

Revoke `CREATE DATABASE LINK` privilege from user `joe`:

```
REVOKE CREATE DATABASE LINK FROM joe;
```

Revoke the `EXEMPT ACCESS POLICY` privilege from user `joe`:

```
REVOKE EXEMPT ACCESS POLICY FROM joe;
```

See Also

GRANT, SET ROLE

2.3.68 ROLLBACK

Name

ROLLBACK -- abort the current transaction

Synopsis

```
ROLLBACK [ WORK ]
```

Description

ROLLBACK rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters

WORK

Optional key word - has no effect.

Notes

Use COMMIT to successfully terminate a transaction.

Issuing ROLLBACK when not inside a transaction does no harm.

Examples

To abort all changes:

```
ROLLBACK;
```

See Also

COMMIT, ROLLBACK TO SAVEPOINT, SAVEPOINT

2.3.69 ROLLBACK TO SAVEPOINT

Name

`ROLLBACK TO SAVEPOINT` -- roll back to a savepoint

Synopsis

`ROLLBACK [ WORK ] TO [ SAVEPOINT ] savepoint_name`

Description

Roll back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

`ROLLBACK TO SAVEPOINT` implicitly destroys all savepoints that were established after the named savepoint.

Parameters

savepoint_name

The savepoint to which to roll back.

Notes

Specifying a savepoint name that has not been established is an error.

`ROLLBACK TO SAVEPOINT` is not supported within SPL programs.

Examples

To undo the effects of the commands executed savepoint `depts` was established:

```
\set AUTOCOMMIT off
INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');
SAVEPOINT depts;
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
ROLLBACK TO SAVEPOINT depts;
```

See Also

COMMIT, ROLLBACK, SAVEPOINT

2.3.70 SAVEPOINT

Name

SAVEPOINT -- define a new savepoint within the current transaction

Synopsis

```
SAVEPOINT savepoint_name
```

Description

SAVEPOINT establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

Parameters

savepoint_name

The name to be given to the savepoint.

Notes

Use ROLLBACK TO SAVEPOINT to roll back to a savepoint.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

When another savepoint is established with the same name as a previous savepoint, the old savepoint is kept, though only the more recent one will be used when rolling back.

SAVEPOINT is not supported within SPL programs.

Examples

To establish a savepoint and later undo the effects of all commands executed after it was established:

```
\set AUTOCOMMIT off
INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');
SAVEPOINT depts;
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
```

```
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
SAVEPOINT emps;
INSERT INTO jobhist VALUES (9001,'17-SEP-07',NULL,'CLERK',800,NULL,50,'New
Hire');
INSERT INTO jobhist VALUES (9002,'20-SEP-07',NULL,'CLERK',700,NULL,50,'New
Hire');
ROLLBACK TO depts;
COMMIT;
```

The above transaction will commit a row into the `dept` table, but the inserts into the `emp` and `jobhist` tables are rolled back.

See Also

COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT

2.3.71 SELECT

Name

SELECT -- retrieve rows from a table or view

Synopsis

```
SELECT [ optimizer_hint ] [ ALL | DISTINCT ]
  * | expression [ AS output_name ] [, ...]
  FROM from_item [, ...]
  [ WHERE condition ]
  [ [ START WITH start_expression ]
      CONNECT BY { PRIOR parent_expr = child_expr |
        child_expr = PRIOR parent_expr }
    [ ORDER SIBLINGS BY expression [ ASC | DESC ] [, ...] ] ]
  [ GROUP BY { expression | ROLLUP ( expr_list ) |
      CUBE ( expr_list ) | GROUPING SETS ( expr_list ) } [, ...]
      [ LEVEL ] ]
  [ HAVING condition [, ...] ]
  [ { UNION [ ALL ] | INTERSECT | MINUS } select ]
  [ ORDER BY expression [ ASC | DESC ] [, ...] ]
  [ FOR UPDATE [WAIT n|NOWAIT|SKIP LOCKED]]
```

where *from_item* can be one of:

```
  table_name[@dblink ] [ alias ]
  ( select ) alias
  from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column [, ...] ) ]
```

Description

SELECT retrieves rows from one or more tables. The general processing of SELECT is as follows:

1. All elements in the FROM list are computed. (Each element in the FROM list is a real or virtual table.) If more than one element is specified in the FROM list, they are cross-joined together. (See FROM clause, below.)
2. If the WHERE clause is specified, all rows that do not satisfy the condition are eliminated from the output. (See WHERE clause, below.)
3. If the GROUP BY clause is specified, the output is divided into groups of rows that match on one or more values. If the HAVING clause is present, it eliminates groups that do not satisfy the given condition. (See GROUP BY clause and HAVING clause below.)

4. Using the operators UNION, INTERSECT, and MINUS, the output of more than one SELECT statement can be combined to form a single result set. The UNION operator returns all rows that are in one or both of the result sets. The INTERSECT operator returns all rows that are strictly in both result sets. The MINUS operator returns the rows that are in the first result set but not in the second. In all three cases, duplicate rows are eliminated. In the case of the UNION operator, if ALL is specified then duplicates are not eliminated. (See UNION clause, INTERSECT clause, and MINUS clause below.)
5. The actual output rows are computed using the SELECT output expressions for each selected row. (See SELECT list below.)
6. The CONNECT BY clause is used to select data that has a hierarchical relationship. Such data has a parent-child relationship between rows. (See CONNECT BY clause.)
7. If the ORDER BY clause is specified, the returned rows are sorted in the specified order. If ORDER BY is not given, the rows are returned in whatever order the system finds fastest to produce. (See ORDER BY clause below.)
8. DISTINCT eliminates duplicate rows from the result. ALL (the default) will return all candidate rows, including duplicates. (See DISTINCT clause below.)
9. The FOR UPDATE clause causes the SELECT statement to lock the selected rows against concurrent updates. (See FOR UPDATE clause below.)

You must have SELECT privilege on a table to read its values. The use of FOR UPDATE requires UPDATE privilege as well.

Parameters

optimizer_hint

Comment-embedded hints to the optimizer for selection of an execution plan. See Section 3.4 for information about optimizer hints.

2.3.71.1 FROM Clause

The FROM clause specifies one or more source tables for a SELECT statement. The syntax is:

```
FROM source [, ...]
```

Where *source* can be one of following elements:

table_name[@*dblink*]

The name (optionally schema-qualified) of an existing table or view. *dblink* is a database link name identifying a remote database. See the CREATE DATABASE LINK command for information on database links.

alias

A substitute name for the FROM item containing the alias. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). When an alias is provided, it completely hides the actual name of the table or function; for example given FROM foo AS f, the remainder of the SELECT must refer to this FROM item as f not foo.

select

A sub-SELECT can appear in the FROM clause. This acts as though its output were created as a temporary table for the duration of this single SELECT command. Note that the sub-SELECT must be surrounded by parentheses, and an alias must be provided for it.

join_type

One of the following:

```
[ INNNER ] JOIN
LEFT [ OUTER ] JOIN
RIGHT [ OUTER ] JOIN
FULL [ OUTER ] JOIN
CROSS JOIN
```

For the INNER and OUTER join types, a join condition must be specified, namely exactly one of NATURAL, ON *join_condition*, or USING (join_column [, ...]). See below for the meaning. For CROSS JOIN, none of these clauses may appear.

A JOIN clause combines two FROM items. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, JOINs nest left-to-right. In any case JOIN binds more tightly than the commas separating FROM items.

CROSS JOIN and INNER JOIN produce a simple Cartesian product, the same result as you get from listing the two items at the top level of FROM, but restricted by the join condition (if any). CROSS JOIN is equivalent to INNER JOIN ON (TRUE), that is, no rows are removed by qualification. These join types are just a notational convenience, since they do nothing you couldn't do with plain FROM and WHERE.

`LEFT OUTER JOIN` returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its join condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns. Note that only the `JOIN` clause's own condition is considered while deciding which rows have matches. Outer conditions are applied afterwards.

Conversely, `RIGHT OUTER JOIN` returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a `LEFT OUTER JOIN` by switching the left and right inputs.

`FULL OUTER JOIN` returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

`ON` *`join_condition`*

`join_condition` is an expression resulting in a value of type `BOOLEAN` (similar to a `WHERE` clause) that specifies which rows in a join are considered to match.

`USING (`*`join_column`* `[, ...] )`

A clause of the form `USING (a, b, ... )` is shorthand for `ON left_table.a = right_table.a AND left_table.b = right_table.b` .... Also, `USING` implies that only one of each pair of equivalent columns will be included in the join output, not both.

`NATURAL`

`NATURAL` is shorthand for a `USING` list that mentions all columns in the two tables that have the same names.

If multiple sources are specified, the result is the Cartesian product (cross join) of all the sources. Usually qualification conditions are added to restrict the returned rows to a small subset of the Cartesian product.

Example

The following example selects all of the entries from the `dept` table:

```
SELECT * FROM dept;
deptno |  dname      |  loc
-------+-------------+-----------
    10 |  ACCOUNTING |  NEW YORK
    20 |  RESEARCH   |  DALLAS
```

```
    30 |  SALES      |  CHICAGO
    40 | OPERATIONS  |  BOSTON
 (4 rows)
```

2.3.71.2 WHERE Clause

The optional `WHERE` clause has the form:

```
WHERE condition
```

where *condition* is any expression that evaluates to a result of type `BOOLEAN`. Any row that does not satisfy this condition will be eliminated from the output. A row satisfies the condition if it returns `TRUE` when the actual row values are substituted for any variable references.

Example

The following example joins the contents of the `emp` and `dept` tables, `WHERE` the value of the `deptno` column in the `emp` table is equal to the value of the `deptno` column in the `deptno` table:

```
SELECT d.deptno, d.dname, e.empno, e.ename, e.mgr, e.hiredate
    FROM emp e, dept d
    WHERE d.deptno = e.deptno;

 deptno |   dname    | empno | ename  | mgr  |      hiredate
--------+------------+-------+--------+------+---------------------
     10 | ACCOUNTING |  7934 | MILLER | 7782 | 23-JAN-82 00:00:00
     10 | ACCOUNTING |  7782 | CLARK  | 7839 | 09-JUN-81 00:00:00
     10 | ACCOUNTING |  7839 | KING   |      | 17-NOV-81 00:00:00
     20 | RESEARCH   |  7788 | SCOTT  | 7566 | 19-APR-87 00:00:00
     20 | RESEARCH   |  7566 | JONES  | 7839 | 02-APR-81 00:00:00
     20 | RESEARCH   |  7369 | SMITH  | 7902 | 17-DEC-80 00:00:00
     20 | RESEARCH   |  7876 | ADAMS  | 7788 | 23-MAY-87 00:00:00
     20 | RESEARCH   |  7902 | FORD   | 7566 | 03-DEC-81 00:00:00
     30 | SALES      |  7521 | WARD   | 7698 | 22-FEB-81 00:00:00
     30 | SALES      |  7844 | TURNER | 7698 | 08-SEP-81 00:00:00
     30 | SALES      |  7499 | ALLEN  | 7698 | 20-FEB-81 00:00:00
     30 | SALES      |  7698 | BLAKE  | 7839 | 01-MAY-81 00:00:00
     30 | SALES      |  7654 | MARTIN | 7698 | 28-SEP-81 00:00:00
     30 | SALES      |  7900 | JAMES  | 7698 | 03-DEC-81 00:00:00
(14 rows)
```

2.3.71.3 GROUP BY Clause

The optional `GROUP BY` clause has the form:

```
GROUP BY { expression | ROLLUP ( expr_list ) |
```

```
    CUBE ( expr_list ) | GROUPING SETS ( expr_list ) } [, ...]
```

GROUP BY will condense into a single row all selected rows that share the same values for the grouped expressions. *expression* can be an input column name, or the name or ordinal number of an output column (SELECT list item), or an arbitrary expression formed from input-column values. In case of ambiguity, a GROUP BY name will be interpreted as an input-column name rather than an output column name.

ROLLUP, CUBE, and GROUPING SETS are extensions to the GROUP BY clause for supporting multidimensional analysis. See Section 2.3.71.3 for information on using these extensions.

Aggregate functions, if any are used, are computed across all rows making up each group, producing a separate value for each group (whereas without GROUP BY, an aggregate produces a single value computed across all the selected rows). When GROUP BY is present, it is not valid for the SELECT list expressions to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

Example

The following example computes the sum of the sal column in the emp table, grouping the results by department number:

```
SELECT deptno, SUM(sal) AS total
    FROM emp
    GROUP BY deptno;

 deptno |  total
--------+----------
     10 |  8750.00
     20 | 10875.00
     30 |  9400.00
(3 rows)
```

2.3.71.4 HAVING Clause

The optional HAVING clause has the form:

```
    HAVING condition
```

where *condition* is the same as specified for the WHERE clause.

HAVING eliminates group rows that do not satisfy the specified condition. HAVING is different from WHERE; WHERE filters individual rows before the application of GROUP BY, while HAVING filters group rows created by GROUP BY. Each column referenced in

condition must unambiguously reference a grouping column, unless the reference appears within an aggregate function.

Example

To sum the column, `sal` of all employees, group the results by department number and show those group totals that are less than 10000:

```
SELECT deptno, SUM(sal) AS total
    FROM emp
    GROUP BY deptno
    HAVING SUM(sal) < 10000;

 deptno |  total
--------+---------
     10 | 8750.00
     30 | 9400.00
(2 rows)
```

2.3.71.5 SELECT List

The `SELECT` list (between the key words `SELECT` and `FROM`) specifies expressions that form the output rows of the `SELECT` statement. The expressions can (and usually do) refer to columns computed in the `FROM` clause. Using the clause `AS` *`output_name`*, another name can be specified for an output column. This name is primarily used to label the column for display. It can also be used to refer to the column's value in `ORDER BY` and `GROUP BY` clauses, but not in the `WHERE` or `HAVING` clauses; there you must write out the expression instead.

Instead of an expression, `*` can be written in the output list as a shorthand for all the columns of the selected rows.

Example

The `SELECT` list in the following example specifies that the result set should include the `empno` column, the `ename` column, the `mgr` column and the `hiredate` column:

```
SELECT empno, ename, mgr, hiredate FROM emp;

 empno | ename  | mgr  |      hiredate
-------+--------+------+--------------------
  7934 | MILLER | 7782 | 23-JAN-82 00:00:00
  7782 | CLARK  | 7839 | 09-JUN-81 00:00:00
  7839 | KING   |      | 17-NOV-81 00:00:00
  7788 | SCOTT  | 7566 | 19-APR-87 00:00:00
  7566 | JONES  | 7839 | 02-APR-81 00:00:00
  7369 | SMITH  | 7902 | 17-DEC-80 00:00:00
  7876 | ADAMS  | 7788 | 23-MAY-87 00:00:00
  7902 | FORD   | 7566 | 03-DEC-81 00:00:00
  7521 | WARD   | 7698 | 22-FEB-81 00:00:00
  7844 | TURNER | 7698 | 08-SEP-81 00:00:00
  7499 | ALLEN  | 7698 | 20-FEB-81 00:00:00
```

```
  7698 | BLAKE  | 7839 | 01-MAY-81 00:00:00
  7654 | MARTIN | 7698 | 28-SEP-81 00:00:00
  7900 | JAMES  | 7698 | 03-DEC-81 00:00:00
(14 rows)
```

2.3.71.6 UNION Clause

The UNION clause has the form:

```
    select_statement UNION [ ALL ] select_statement
```

select_statement is any SELECT statement without an ORDER BY or FOR UPDATE clause. (ORDER BY can be attached to a sub-expression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the UNION, not to its right-hand input expression.)

The UNION operator computes the set union of the rows returned by the involved SELECT statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two SELECT statements that represent the direct operands of the UNION must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of UNION does not contain any duplicate rows unless the ALL option is specified. ALL prevents elimination of duplicates.

Multiple UNION operators in the same SELECT statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, FOR UPDATE may not be specified either for a UNION result or for any input of a UNION.

2.3.71.7 INTERSECT Clause

The INTERSECT clause has the form:

```
    select_statement INTERSECT select_statement
```

select_statement is any SELECT statement without an ORDER BY or FOR UPDATE clause.

The INTERSECT operator computes the set intersection of the rows returned by the involved SELECT statements. A row is in the intersection of two result sets if it appears in both result sets.

The result of INTERSECT does not contain any duplicate rows.

Multiple INTERSECT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. INTERSECT binds more tightly than UNION. That is, A UNION B INTERSECT C will be read as A UNION (B INTERSECT C).

2.3.71.8 MINUS Clause

The MINUS clause has this general form:

```
select_statement MINUS select_statement
```

select_statement is any SELECT statement without an ORDER BY or FOR UPDATE clause.

The MINUS operator computes the set of rows that are in the result of the left SELECT statement but not in the result of the right one.

The result of MINUS does not contain any duplicate rows.

Multiple MINUS operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. MINUS binds at the same level as UNION.

2.3.71.9 CONNECT BY Clause

The CONNECT BY clause determines the parent-child relationship of rows when performing a hierarchical query. It has the general form:

```
CONNECT BY { PRIOR parent_expr = child_expr |
  child_expr = PRIOR parent_expr }
```

parent_expr is evaluated on a candidate parent row. If *parent_expr = child_expr* results in TRUE for a row returned by the FROM clause, then this row is considered a child of the parent.

The following optional clauses may be specified in conjunction with the CONNECT BY clause:

```
START WITH start_expression
```

The rows returned by the FROM clause on which *start_expression* evaluates to TRUE become the root nodes of the hierarchy.

```
ORDER SIBLINGS BY expression [ ASC | DESC ] [, ...]
```

Sibling rows of the hierarchy are ordered by *expression* in the result set.

Note: Advanced Server does not support the use of AND (or other operators) in the CONNECT BY clause.

2.3.71.10 ORDER BY Clause

The optional ORDER BY clause has the form:

```
ORDER BY expression [ ASC | DESC ] [, ...]
```

expression can be the name or ordinal number of an output column (SELECT list item), or it can be an arbitrary expression formed from input-column values.

The ORDER BY clause causes the result rows to be sorted according to the specified expressions. If two rows are equal according to the leftmost expression, they are compared according to the next expression and so on. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to a result column using the AS clause.

It is also possible to use arbitrary expressions in the ORDER BY clause, including columns that do not appear in the SELECT result list. Thus the following statement is valid:

```
SELECT ename FROM emp ORDER BY empno;
```

A limitation of this feature is that an ORDER BY clause applying to the result of a UNION, INTERSECT, or MINUS clause may only specify an output column name or number, not an expression.

If an ORDER BY expression is a simple name that matches both a result column name and an input column name, ORDER BY will interpret it as the result column name. This is the

opposite of the choice that `GROUP BY` will make in the same situation. This inconsistency is made to be compatible with the SQL standard.

Optionally one may add the key word `ASC` (ascending) or `DESC` (descending) after any expression in the `ORDER BY` clause. If not specified, `ASC` is assumed by default.

The null value sorts higher than any other value. In other words, with ascending sort order, null values sort at the end, and with descending sort order, null values sort at the beginning.

Character-string data is sorted according to the locale-specific collation order that was established when the database cluster was initialized.

Note: If `SELECT DISTINCT` is specified or if a `SELECT` statement includes the `SELECT DISTINCT …ORDER BY` clause then all the expressions in `ORDER BY` must be present in the select list of the `SELECT DISTINCT` query.

Examples

The following two examples are identical ways of sorting the individual results according to the contents of the second column (`dname`):

```
SELECT * FROM dept ORDER BY dname;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     40 | OPERATIONS | BOSTON
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
(4 rows)

SELECT * FROM dept ORDER BY 2;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     40 | OPERATIONS | BOSTON
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
(4 rows)
```

The following example uses the `SELECT DISTINCT …ORDER BY` clause to fetch the `job` and `deptno` from table `emp`:

```
CREATE TABLE EMP(EMPNO NUMBER(4) NOT NULL,
ENAME VARCHAR2(10),
JOB VARCHAR2(9),
DEPTNO NUMBER(2));

INSERT INTO EMP VALUES (7369, 'SMITH', 'CLERK', 20);
INSERT 0 1
```

```
INSERT INTO EMP VALUES (7499, 'ALLEN', 'SALESMAN', 30);
INSERT 0 1
INSERT INTO EMP VALUES (7521, 'WARD', 'SALESMAN', 30);
INSERT 0 1
INSERT INTO EMP VALUES (7566, 'JONES', 'MANAGER', 20);
INSERT 0 1

SELECT DISTINCT e.job, e.deptno FROM emp e ORDER BY e.job, e.deptno;
 job      | deptno
----------+--------
 CLERK    |    20
 MANAGER  |    20
 SALESMAN |    30
(3 rows)
```

2.3.71.11 DISTINCT Clause

If a `SELECT` statement specifies `DISTINCT`, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). The `ALL` keyword specifies the opposite: all rows are kept; that is the default.

2.3.71.12 FOR UPDATE Clause

The `FOR UPDATE` clause takes the form:

```
FOR UPDATE [WAIT n|NOWAIT|SKIP LOCKED]
```

`FOR UPDATE` causes the rows retrieved by the `SELECT` statement to be locked as though for update. This prevents a row from being modified or deleted by other transactions until the current transaction ends; any transaction that attempts to `UPDATE`, `DELETE`, or `SELECT FOR UPDATE` a selected row will be blocked until the current transaction ends. If an `UPDATE`, `DELETE`, or `SELECT FOR UPDATE` from another transaction has already locked a selected row or rows, `SELECT FOR UPDATE` will wait for the first transaction to complete, and will then lock and return the updated row (or no row, if the row was deleted).

`FOR UPDATE` cannot be used in contexts where returned rows cannot be clearly identified with individual table rows (for example, with aggregation).

Use `FOR UPDATE` options to specify locking preferences:

- Include the `WAIT` *n* keywords to specify the number of seconds (or fractional seconds) that the `SELECT` statement will wait for a row locked by another session. Use a decimal form to specify fractional seconds; for example, `WAIT 1.5`

instructs the server to wait one and a half seconds. Specify up to 4 digits to the right of the decimal.

- Include the `NOWAIT` keyword to report an error immediately if a row cannot be locked by the current session.

- Include `SKIP LOCKED` to instruct the server to lock rows if possible, and skip rows that are already locked by another session.

2.3.72 SET CONSTRAINTS

Name

`SET CONSTRAINTS` -- set constraint checking modes for the current transaction

Synopsis

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

Description

`SET CONSTRAINTS` sets the behavior of constraint checking within the current transaction. `IMMEDIATE` constraints are checked at the end of each statement. `DEFERRED` constraints are not checked until transaction commit. Each constraint has its own `IMMEDIATE` or `DEFERRED` mode.

Upon creation, a constraint is given one of three characteristics: `DEFERRABLE INITIALLY DEFERRED`, `DEFERRABLE INITIALLY IMMEDIATE`, or `NOT DEFERRABLE`. The third class is always `IMMEDIATE` and is not affected by the `SET CONSTRAINTS` command. The first two classes start every transaction in the indicated mode, but their behavior can be changed within a transaction by `SET CONSTRAINTS`.

`SET CONSTRAINTS` with a list of constraint names changes the mode of just those constraints (which must all be deferrable). If there are multiple constraints matching any given name, all are affected. `SET CONSTRAINTS ALL` changes the mode of all deferrable constraints.

When `SET CONSTRAINTS` changes the mode of a constraint from `DEFERRED` to `IMMEDIATE`, the new mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction are instead checked during the execution of the `SET CONSTRAINTS` command. If any such constraint is violated, the `SET CONSTRAINTS` fails (and does not change the constraint mode). Thus, `SET CONSTRAINTS` can be used to force checking of constraints to occur at a specific point in a transaction.

Currently, only foreign key constraints are affected by this setting. Check and unique constraints are always effectively not deferrable.

Notes

This command only alters the behavior of constraints within the current transaction. Thus, if you execute this command outside of a transaction block it will not appear to have any effect.

2.3.73 SET ROLE

Name

`SET ROLE` -- set the current user identifier of the current session

Synopsis

```
SET ROLE { rolename | NONE }
```

Description

This command sets the current user identifier of the current SQL session context to be *`rolename`*. After `SET ROLE`, permissions checking for SQL commands is carried out as though the named role were the one that had logged in originally.

The specified *`rolename`* must be a role that the current session user is a member of. (If the session user is a superuser, any role can be selected.)

`NONE` resets the current user identifier to be the current session user identifier. These forms may be executed by any user.

Notes

Using this command, it is possible to either add privileges or restrict one's privileges. If the session user role has the `INHERITS` attribute, then it automatically has all the privileges of every role that it could `SET ROLE` to; in this case `SET ROLE` effectively drops all the privileges assigned directly to the session user and to the other roles it is a member of, leaving only the privileges available to the named role. On the other hand, if the session user role has the `NOINHERITS` attribute, `SET ROLE` drops the privileges assigned directly to the session user and instead acquires the privileges available to the named role. In particular, when a superuser chooses to `SET ROLE` to a non-superuser role, she loses her superuser privileges.

Examples

User `mary` takes on the identity of role `admins`:

```
SET ROLE admins;
```

User `mary` reverts back to her own identity:

```
SET ROLE NONE;
```

2.3.74 SET TRANSACTION

Name

`SET TRANSACTION` -- set the characteristics of the current transaction

Synopsis

```
SET TRANSACTION transaction_mode
```

where *`transaction_mode`* is one of:

```
  ISOLATION LEVEL { SERIALIZABLE | READ COMMITTED }
  READ WRITE | READ ONLY
```

Description

The `SET TRANSACTION` command sets the characteristics of the current transaction. It has no effect on any subsequent transactions. The available transaction characteristics are the transaction isolation level and the transaction access mode (read/write or read-only). The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently:

`READ COMMITTED`

A statement can only see rows committed before it began. This is the default.

`SERIALIZABLE`

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

The transaction isolation level cannot be changed after the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, or `FETCH`) of a transaction has been executed. The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default.

When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, and `DELETE` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `COMMENT`, `GRANT`, `REVOKE`, `TRUNCATE`; and `EXECUTE` if the command it would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.

2.3.75 TRUNCATE

Name

`TRUNCATE` -- empty a table

Synopsis

```
TRUNCATE TABLE name [DROP STORAGE]
```

Description

`TRUNCATE` quickly removes all rows from a table. It has the same effect as an unqualified `DELETE` but since it does not actually scan the table, it is faster. This is most useful on large tables.

The `DROP STORAGE` clause is accepted for compatibility, but is ignored.

Parameters

name

The name (optionally schema-qualified) of the table to be truncated.

Notes

`TRUNCATE` cannot be used if there are foreign-key references to the table from other tables. Checking validity in such cases would require table scans, and the whole point is not to do one.

`TRUNCATE` will not run any user-defined `ON DELETE` triggers that might exist for the table.

Examples

Truncate the table `bigtable`:

```
TRUNCATE TABLE bigtable;
```

See Also

DROP VIEW, DELETE

2.3.76 UPDATE

Name

UPDATE -- update rows of a table

Synopsis

```
UPDATE [ optimizer_hint ] table[@dblink ]
    SET column = { expression | DEFAULT } [, ...]
  [ WHERE condition ]
  [ RETURNING return_expression [, ...]
      { INTO { record | variable [, ...] }
      | BULK COLLECT INTO collection [, ...] } ]
```

Description

UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

The RETURNING INTO { *record* | *variable* [, ...] } clause may only be specified within an SPL program. In addition the result set of the UPDATE command must not return more than one row, otherwise an exception is thrown. If the result set is empty, then the contents of the target record or variables are set to null.

The RETURNING BULK COLLECT INTO *collection* [, ...] clause may only be specified if the UPDATE command is used within an SPL program. If more than one *collection* is specified as the target of the BULK COLLECT INTO clause, then each *collection* must consist of a single, scalar field – i.e., *collection* must not be a record. The result set of the UPDATE command may contain none, one, or more rows. *return_expression* evaluated for each row of the result set, becomes an element in *collection* starting with the first element. Any existing rows in *collection* are deleted. If the result set is empty, then *collection* will be empty.

You must have the UPDATE privilege on the table to update it, as well as the SELECT privilege to any table whose values are read in expression or *condition*.

Parameters

optimizer_hint

Comment-embedded hints to the optimizer for selection of an execution plan.

table

The name (optionally schema-qualified) of the table to update.

dblink

Database link name identifying a remote database. See the CREATE DATABASE LINK command for information on database links.

column

The name of a column in table.

expression

An expression to assign to the column. The expression may use the old values of this and other columns in the table.

DEFAULT

Set the column to its default value (which will be null if no specific default expression has been assigned to it).

condition

An expression that returns a value of type BOOLEAN. Only rows for which this expression returns true will be updated.

return_expression

An expression that may include one or more columns from table. If a column name from table is specified in *return_expression*, the value substituted for the column when *return_expression* is evaluated is determined as follows:

If the column specified in *return_expression* is assigned a value in the UPDATE command, then the assigned value is used in the evaluation of *return_expression*.

If the column specified in *return_expression* is not assigned a value in the UPDATE command, then the column's current value in the affected row is used in the evaluation of *return_expression*.

record

A record whose field the evaluated *return_expression* is to be assigned. The first *return_expression* is assigned to the first field in *record*, the second

return_expression is assigned to the second field in *record*, etc. The number of fields in *record* must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

variable

A variable to which the evaluated *return_expression* is to be assigned. If more than one *return_expression* and *variable* are specified, the first *return_expression* is assigned to the first *variable*, the second *return_expression* is assigned to the second *variable*, etc. The number of variables specified following the `INTO` keyword must exactly match the number of expressions following the `RETURNING` keyword and the variables must be type-compatible with their assigned expressions.

collection

A collection in which an element is created from the evaluated *return_expression*. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding *return_expression* and *collection* field must be type-compatible.

Examples

Change the location to `AUSTIN` for department `20` in the `dept` table:

```
UPDATE dept SET loc = 'AUSTIN' WHERE deptno = 20;
```

For all employees with `job` = `SALESMAN` in the `emp` table, update the salary by 10% and increase the commission by 500.

```
UPDATE emp SET sal = sal * 1.1, comm = comm + 500 WHERE job = 'SALESMAN';
```

2.4 Functions and Operators

Advanced Server provides a large number of functions and operators for the built-in data types.

2.4.1 Logical Operators

The usual logical operators are available: `AND`, `OR`, `NOT`

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

Table 2-11 AND/OR Truth Table

a	b	a AND b	a OR b
True	True	True	True
True	False	False	True
True	Null	Null	True
False	False	False	False
False	Null	False	Null
Null	Null	Null	Null

Table 2-12 NOT Truth Table

a	NOT a
True	False
False	True
Null	Null

The operators `AND` and `OR` are commutative, that is, you can switch the left and right operand without affecting the result.

2.4.2 Comparison Operators

The usual comparison operators are shown in the following table.

Table 2-13 Comparison Operators

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal
<>	Not equal
!=	Not equal

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

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

```
a BETWEEN x AND y
```

is equivalent to

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

Similarly,

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

is equivalent to

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

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

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

```
expression IS NULL
expression IS NOT NULL
```

Do not write *expression* = NULL because NULL is not "equal to" NULL. (The null value represents an unknown value, and it is not known whether two unknown values are equal.) This behavior conforms to the SQL standard.

Some applications may expect that *expression* = NULL returns true if *expression* evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard.

2.4.3 Mathematical Functions and Operators

Mathematical operators are provided for many Advanced Server types. For types without common mathematical conventions for all possible permutations (e.g., date/time types) the actual behavior is described in subsequent sections.

The following table shows the available mathematical operators.

Table 2-14 Mathematical Operators

Operator	Description	Example	Result
+	Addition	2 + 3	5
-	Subtraction	2 - 3	-1
*	Multiplication	2 * 3	6
/	Division (See the following note.)	4 / 2	2
**	Exponentiation Operator	2 ** 3	8

Note: If the `db_dialect` configuration parameter in the `postgresql.conf` file is set to `redwood`, then division of a pair of `INTEGER` data types does not result in a truncated value. Any fractional result is retained as shown by the following example:

```
edb=# SET db_dialect TO redwood;
SET
edb=# SHOW db dialect;
 db_dialect
------------
 redwood
(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS INTEGER) FROM dual;
      ?column?
--------------------
 3.3333333333333333
(1 row)
```

This behavior is compatible with Oracle databases where there is no native `INTEGER` data type, and any `INTEGER` data type specification is internally converted to `NUMBER(38)`, which results in retaining any fractional result.

If the `db_dialect` configuration parameter is set to `postgres`, then division of a pair of `INTEGER` data types results in a truncated value as shown by the following example:

```
edb=# SET db dialect TO postgres;
SET
edb=# SHOW db dialect;
 db_dialect
------------
 postgres
(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS INTEGER) FROM dual;
 ?column?
----------
```

```
        3
(1 row)
```

This behavior is compatible with PostgreSQL databases where division involving any pair of `INTEGER`, `SMALLINT`, or `BIGINT` data types results in truncation of the result. The same truncated result is returned by Advanced Server when `db_dialect` is set to `postgres` as shown in the previous example.

Note however, that even when `db_dialect` is set to `redwood`, only division with a pair of `INTEGER` data types results in no truncation of the result. Division that includes only `SMALLINT` or `BIGINT` data types, with or without an `INTEGER` data type, does result in truncation in the PostgreSQL fashion without retaining the fractional portion as shown by the following where `INTEGER` and `SMALLINT` are involved in the division:

```
edb=# SHOW db_dialect;
 db dialect
------------
 redwood
(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS SMALLINT) FROM dual;
 ?column?
----------
        3
(1 row)
```

The following table shows the available mathematical functions. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with `DOUBLE PRECISION` data are mostly implemented on top of the host system's C library; accuracy and behavior in boundary cases may therefore vary depending on the host system.

Table 2-15 Mathematical Functions

Function	Return Type	Description	Example	Result
`ABS(x)`	Same as *x*	Absolute value	`ABS(-17.4)`	`17.4`
`CEIL(DOUBLE PRECISION or NUMBER)`	Same as input	Smallest integer not less than argument	`CEIL(-42.8)`	`-42`
`EXP(DOUBLE PRECISION or NUMBER)`	Same as input	Exponential	`EXP(1.0)`	`2.71828182845904 52`
`FLOOR(DOUBLE PRECISION or NUMBER)`	Same as input	Largest integer not greater than argument	`FLOOR(-42.8)`	`43`
`LN(DOUBLE PRECISION or NUMBER)`	Same as input	Natural logarithm	`LN(2.0)`	`0.69314718055994 53`
`LOG(b NUMBER, x NUMBER)`	`NUMBER`	Logarithm to base *b*	`LOG(2.0, 64.0)`	`6.00000000000000 00`
`MOD(y, x)`	Same as argument types	Remainder of *y*/*x*	`MOD(9, 4)`	`1`
`NVL(x, y)`	Same as argument types; where both arguments are of	If *x* is null, then NVL returns *y*	`NVL(9, 0)`	`9`

Function	Return Type	Description	Example	Result
	the same data type			
POWER(*a* DOUBLE PRECISION, *b* DOUBLE PRECISION)	DOUBLE PRECISION	*a* raised to the power of *b*	POWER(9.0, 3.0)	729.000000000000000
POWER(*a* NUMBER, *b* NUMBER)	NUMBER	*a* raised to the power of *b*	POWER(9.0, 3.0)	729.000000000000000
ROUND(DOUBLE PRECISION or NUMBER)	Same as input	Round to nearest integer	ROUND(42.4)	42
ROUND(*v* NUMBER, *s* INTEGER)	NUMBER	Round to *s* decimal places	ROUND(42.4382, 2)	42.44
SIGN(DOUBLE PRECISION or NUMBER)	Same as input	Sign of the argument (-1, 0, +1)	SIGN(-8.4)	-1
SQRT(DOUBLE PRECISION or NUMBER)	Same as input	Square root	SQRT(2.0)	1.414213562373095
TRUNC(DOUBLE PRECISION or NUMBER)	Same as input	Truncate toward zero	TRUNC(42.8)	42
TRUNC(*v* NUMBER, *s* INTEGER)	NUMBER	Truncate to s decimal places	TRUNC(42.4382, 2)	42.43
WIDTH_BUCKET(*op* NUMBER, *b1* NUMBER, *b2* NUMBER, *count* INTEGER)	INTEGER	Return the bucket to which *op* would be assigned in an equidepth histogram with *count* buckets, in the range *b1* to *b2*	WIDTH_BUCKET(5.35, 0.024, 10.06, 5)	3

The following table shows the available trigonometric functions. All trigonometric functions take arguments and return values of type DOUBLE PRECISION.

Table 2-16 Trigonometric Functions

Function	Description
ACOS(*x*)	Inverse cosine
ASIN(*x*)	Inverse sine
ATAN(*x*)	Inverse tangent
ATAN2(*x*, *y*)	Inverse tangent of *x*/*y*
COS(*x*)	Cosine
SIN(*x*)	Sine
TAN(*x*)	Tangent

2.4.4 String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types CHAR, VARCHAR2, and CLOB. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of automatic padding when using the CHAR type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first.

Table 2-17 SQL String Functions and Operators

Function	Return Type	Description	Example	Result
string || *string*	CLOB	String concatenation	'Enterprise' || 'DB'	EnterpriseDB
CONCAT(*string*, *string*)	CLOB	String concatenation	'a' || 'b'	ab
HEXTORAW(*varchar2*)	RAW	Converts a VARCHAR2 value to a RAW value	HEXTORAW('303132')	'012'
RAWTOHEX(*raw*)	VARCHAR2	Converts a RAW value to a HEXADECIMAL value	RAWTOHEX('012')	'303132'
INSTR(*string*, *set*, [*start* [, *occurrence*]])	INTEGER	Finds the location of a set of characters in a string, starting at position *start* in the string, *string*, and looking for the first, second, third and so on occurrences of the set. Returns 0 if the set is not found.	INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI',1,3)	30
INSTRB(*string*, *set*)	INTEGER	Returns the position of the *set* within the *string*. Returns 0 if *set* is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PICK')	13
INSTRB(*string*, *set*, *start*)	INTEGER	Returns the position of the *set* within the *string*, beginning at *start*. Returns 0 if *set* is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 14)	30
INSTRB(*string*, *set*, *start*, *occurrence*)	INTEGER	Returns the position of the specified *occurrence* of *set* within the *string*, beginning at *start*. Returns 0 if *set* is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 1, 2)	30
LOWER(*string*)	CLOB	Convert *string* to lower case	LOWER('TOM')	tom
SUBSTR(*string*, *start* [, *count*])	CLOB	Extract substring starting from *start* and going for *count* characters. If *count* is not specified, the string is clipped from the start till the end.	SUBSTR('This is a test',6,2)	is
SUBSTRB(*string*, *start* [, *count*])	CLOB	Same as SUBSTR except *start* and *count* are in	SUBSTRB('abc',3) (assuming a double-byte	c

Function	Return Type	Description	Example	Result
		number of bytes.	character set)	
SUBSTR2(*string*, *start* [, *count*])	CLOB	Alias for SUBSTR.	SUBSTR2('This is a test',6,2)	is
SUBSTR2(*string*, *start* [, *count*])	CLOB	Alias for SUBSTRB.	SUBSTR2('abc',3) (assuming a double-byte character set)	c
SUBSTR4(*string*, *start* [, *count*])	CLOB	Alias for SUBSTR.	SUBSTR4('This is a test',6,2)	is
SUBSTR4(*string*, *start* [, *count*])	CLOB	Alias for SUBSTRB.	SUBSTR4('abc',3) (assuming a double-byte character set)	c
SUBSTRC(*string*, *start* [, *count*])	CLOB	Alias for SUBSTR.	SUBSTRC('This is a test',6,2)	is
SUBSTRC(*string*, *start* [, *count*])	CLOB	Alias for SUBSTRB.	SUBSTRC('abc',3) (assuming a double-byte character set)	c
TRIM([LEADING | TRAILING | BOTH] [*characters*] FROM *string*)	CLOB	Remove the longest string containing only the characters (a space by default) from the start/end/both ends of the string.	TRIM(BOTH 'x' FROM 'xTomxx')	Tom
LTRIM(*string* [, *set*])	CLOB	Removes all the characters specified in *set* from the left of a given *string*. If *set* is not specified, a blank space is used as default.	LTRIM('abcdefghi', 'abc')	defghi
RTRIM(*string* [, *set*])	CLOB	Removes all the characters specified in *set* from the right of a given *string*. If *set* is not specified, a blank space is used as default.	RTRIM('abcdefghi', 'ghi')	abcdef
UPPER(*string*)	CLOB	Convert *string* to upper case	UPPER('tom')	TOM

Additional string manipulation functions are available and are listed in the following table. Some of them are used internally to implement the SQL-standard string functions listed in Table 2-17.

Table 2-18 Other String Functions

Function	Return Type	Description	Example	Result
ASCII(*string*)	INTEGER	ASCII code of the first byte of the argument	ASCII('x')	120
CHR(INTEGER)	CLOB	Character with the given ASCII code	CHR(65)	A
DECODE(*expr*, *expr1a*, *expr1b* [, *expr2a*, *expr2b*]... [, *default*])	Same as argument types of *expr1b*, *expr2b*,..., *default*	Finds first match of *expr* with *expr1a*, *expr2a*, etc. When match found, returns corresponding parameter pair, *expr1b*, *expr2b*, etc. If no match found, returns *default*. If no match found	DECODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found')	Three

Function	Return Type	Description	Example	Result
		and *default* not specified, returns null.		
INITCAP(*string*)	CLOB	Convert the first letter of each word to uppercase and the rest to lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.	INITCAP('hi THOMAS')	Hi Thomas
LENGTH	INTEGER	Returns the number of characters in a string value.	LENGTH('Côte d''Azur')	11
LENGTHC	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTHC('Côte d''Azur')	11
LENGTH2	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTH2('Côte d''Azur')	11
LENGTH4	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTH4('Côte d''Azur')	11
LENGTHB	INTEGER	Returns the number of bytes required to hold the given value.	LENGTHB('Côte d''Azur')	12
LPAD(*string*, *length* INTEGER [, *fill*])	CLOB	Fill up *string* to size, *length* by prepending the characters, *fill* (a space by default). If *string* is already longer than *length* then it is truncated (on the right).	LPAD('hi', 5, 'xy')	xyxhi
REPLACE(*string*, *search_string* [, *replace_string*]	CLOB	Replaces one value in a string with another. If you do not specify a value for *replace_string*, the *search_string* value when found, is removed.	REPLACE('GEORGE', 'GE', 'EG')	EGOREG
RPAD(*string*, *length* INTEGER [, *fill*])	CLOB	Fill up *string* to size, *length* by appending the characters, *fill* (a space by default). If *string* is already longer than *length* then it is truncated.	RPAD('hi', 5, 'xy')	hixyx
TRANSLATE(*string*, *from*, *to*)	CLOB	Any character in *string* that matches a character in the *from* set is replaced by the corresponding character in the *to* set.	TRANSLATE('12345', '14', 'ax')	a23x5

2.4.5 Pattern Matching String Functions

Advanced Server offers support for the REGEXP_COUNT, REGEXP_INSTR and REGEXP_SUBSTR functions. These functions search a string for a pattern specified by a regular expression, and return information about occurrences of the pattern within the string. The pattern should be a POSIX-style regular expression; for more information about forming a POSIX-style regular expression, please refer to the core documentation at:

https://www.postgresql.org/docs/10/static/functions-matching.html

2.4.5.1 REGEXP_COUNT

REGEXP_COUNT searches a string for a regular expression, and returns a count of the times that the regular expression occurs. The signature is:

```
INTEGER REGEXP_COUNT
(
  srcstr    TEXT,
  pattern   TEXT,
  position  DEFAULT 1
  modifier  DEFAULT NULL
)
```

Parameters

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which REGEXP_COUNT will search.

position

position is an integer value that indicates the position in the source string at which REGEXP_COUNT will begin searching. The default value is 1.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/functions-matching.html

Example

In the following simple example, REGEXP_COUNT returns a count of the number of times the letter i is used in the character string 'reinitializing':

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 1) FROM DUAL;
 regexp_count
--------------
            5
(1 row)
```

In the first example, the command instructs REGEXP_COUNT begins counting in the first position; if we modify the command to start the count on the 6th position:

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 6) FROM DUAL;
 regexp_count
--------------
            3
(1 row)
```

REGEXP_COUNT returns 3; the count now excludes any occurrences of the letter i that occur before the 6th position.

2.4.5.2 REGEXP_INSTR

REGEXP_INSTR searches a string for a POSIX-style regular expression. This function returns the position within the string where the match was located. The signature is:

```
INTEGER REGEXP_INSTR
(
  srcstr        TEXT,
  pattern       TEXT,
  position      INT  DEFAULT 1,
  occurrence    INT  DEFAULT 1,
  returnparam   INT  DEFAULT 0,
  modifier      TEXT DEFAULT NULL,
  subexpression INT  DEFAULT 0,
)
```

Parameters:

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which REGEXP_INSTR will search.

position

position specifies an integer value that indicates the start position in a source string. The default value is 1.

occurrence

occurrence specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1.

returnparam

returnparam is an integer value that specifies the location within the string that REGEXP_INSTR should return. The default value is 0. Specify:

0 to return the location within the string of the first character that matches *pattern*.

A value greater than 0 to return the position of the first character following the end of the *pattern*.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/functions-matching.html

subexpression

subexpression is an integer value that identifies the portion of the *pattern* that will be returned by REGEXP_INSTR. The default value of *subexpression* is 0.

If you specify a value for *subexpression*, you must include one (or more) set of parentheses in the *pattern* that isolate a portion of the value being searched for. The value specified by *subexpression* indicates which set of parentheses should be returned; for example, if *subexpression* is 2, REGEXP_INSTR will return the position of the second set of parentheses.

Example

In the following simple example, REGEXP_INSTR searches a string that contains the a phone number for the first occurrence of a pattern that contains three consecutive digits:

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;
 regexp_instr
--------------
            1
(1 row)
```

The command instructs REGEXP_INSTR to return the position of the first occurrence. If we modify the command to return the start of the second occurrence of three consecutive digits:

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;
 regexp_instr
--------------
            5
(1 row)
```

REGEXP_INSTR returns 5; the second occurrence of three consecutive digits begins in the 5th position.

2.4.5.3 REGEXP_SUBSTR

The REGEXP_SUBSTR function searches a string for a pattern specified by a POSIX compliant regular expression. REGEXP_SUBSTR returns the string that matches the pattern specified in the call to the function. The signature of the function is:

```
TEXT REGEXP_SUBSTR
(
  srcstr        TEXT,
  pattern       TEXT,
  position      INT  DEFAULT 1,
  occurrence    INT  DEFAULT 1,
  modifier      TEXT DEFAULT NULL,
  subexpression INT  DEFAULT 0
)
```

Parameters:

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which REGEXP_SUBSTR will search.

position

position specifies an integer value that indicates the start position in a source string. The default value is 1.

occurrence

occurrence specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/functions-matching.html

subexpression

subexpression is an integer value that identifies the portion of the *pattern* that will be returned by REGEXP_SUBSTR. The default value of *subexpression* is 0.

If you specify a value for *subexpression*, you must include one (or more) set of parentheses in the *pattern* that isolate a portion of the value being searched for. The value specified by *subexpression* indicates which set of parentheses should be returned; for example, if *subexpression* is 2, REGEXP_SUBSTR will return the value contained within the second set of parentheses.

Example

In the following simple example, REGEXP_SUBSTR searches a string that contains a phone number for the first set of three consecutive digits:

```
edb=# SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM
DUAL;
 regexp_substr
---------------
 800
(1 row)
```

It locates the first occurrence of three digits and returns the string (`800`); if we modify the command to check for the second occurrence of three consecutive digits:

```
edb=# SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM
DUAL;
 regexp_substr
---------------
 555
(1 row)
```

`REGEXP_SUBSTR` returns `555`, the contents of the second substring.

2.4.6 Pattern Matching Using the LIKE Operator

Advanced Server provides pattern matching using the traditional SQL `LIKE` operator. The syntax for the `LIKE` operator is as follows.

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

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

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

Some examples:

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

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

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

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

It’s also possible to select no escape character by writing `ESCAPE ''`. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

2.4.7 Data Type Formatting Functions

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

Table 2-19 Formatting Functions

Function	Return Type	Description	Example	Result
TO_CHAR(DATE [, *format*])	VARCHAR2	Convert a date/time to a string with output, *format*. If omitted default format is DD-MON-YY.	TO_CHAR(SYSDATE, 'MM/DD/YYYY HH12:MI:SS AM')	07/25/2007 09:43:02 AM
TO_CHAR(TIMESTAMP [, *format*])	VARCHAR2	Convert a timestamp to a string with output, *format*. If omitted default format is DD-MON-YY.	TO_CHAR(CURRENT_TIMESTAMP, 'MM/DD/YYYY HH12:MI:SS AM')	08/13/2015 08:55:22 PM
TO_CHAR(INTEGER [, *format*])	VARCHAR2	Convert an integer to a string with output, *format*	TO_CHAR(2412, '999,999S')	2,412+
TO_CHAR(NUMBER [, *format*])	VARCHAR2	Convert a decimal number to a string with output, *format*	TO_CHAR(10125.35, '999,999.99')	10,125.35
TO_CHAR(DOUBLE PRECISION, *format*)	VARCHAR2	Convert a floating-point number to a string with output, *format*	TO_CHAR(CAST(123.5282 AS REAL), '999.99')	123.53
TO_DATE(*string* [, *format*])	DATE	Convert a date formatted string to a DATE data type	TO_DATE('2007-07-04 13:39:10', 'YYYY-MM-DD HH24:MI:SS')	04-JUL-07 13:39:10
TO_NUMBER(*string* [, *format*])	NUMBER	Convert a number formatted string to a NUMBER data type	TO_NUMBER('2,412-', '999,999S')	-2412
TO_TIMESTAMP(*string*, *format*)	TIMESTAMP	Convert a timestamp formatted string to a TIMESTAMP data type	TO_TIMESTAMP('05 Dec 2000 08:30:25 pm', 'DD Mon YYYY hh12:mi:ss pm')	05-DEC-00 20:30:25

In an output template string (for TO_CHAR), there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template

string (for anything but TO_CHAR), template patterns identify the parts of the input data string to be looked at and the values to be found there.

The following table shows the template patterns available for formatting date values using the TO_CHAR and TO_DATE functions.

Table 2-20 Template Date/Time Format Patterns

Pattern	Description
HH	Hour of day (01-12)
HH12	Hour of day (01-12)
HH24	Hour of day (00-23)
MI	Minute (00-59)
SS	Second (00-59)
SSSSS	Seconds past midnight (0-86399)
FF*n*	Fractional seconds where *n* is an optional integer from 1 to 9 for the number of digits to return. If omitted, the default is 6.
AM or A.M. or PM or P.M.	Meridian indicator (uppercase)
am or a.m. or pm or p.m.	Meridian indicator (lowercase)
Y,YYY	Year (4 and more digits) with comma
YEAR	Year (spelled out)
SYEAR	Year (spelled out) (BC dates prefixed by a minus sign)
YYYY	Year (4 and more digits)
SYYYY	Year (4 and more digits) (BC dates prefixed by a minus sign)
YYY	Last 3 digits of year
YY	Last 2 digits of year
Y	Last digit of year
IYYY	ISO year (4 and more digits)
IYY	Last 3 digits of ISO year
IY	Last 2 digits of ISO year
I	Last 1 digit of ISO year
BC or B.C. or AD or A.D.	Era indicator (uppercase)
bc or b.c. or ad or a.d.	Era indicator (lowercase)
MONTH	Full uppercase month name
Month	Full mixed-case month name
month	Full lowercase month name
MON	Abbreviated uppercase month name (3 chars in English, localized lengths vary)
Mon	Abbreviated mixed-case month name (3 chars in English, localized lengths vary)
mon	Abbreviated lowercase month name (3 chars in English, localized lengths vary)
MM	Month number (01-12)
DAY	Full uppercase day name
Day	Full mixed-case day name
day	Full lowercase day name
DY	Abbreviated uppercase day name (3 chars in English, localized lengths vary)
Dy	Abbreviated mixed-case day name (3 chars in English, localized lengths vary)
dy	Abbreviated lowercase day name (3 chars in English, localized lengths vary)

Pattern	Description
DDD	Day of year (001-366)
DD	Day of month (01-31)
D	Day of week (1-7; Sunday is 1)
W	Week of month (1-5) (The first week starts on the first day of the month)
WW	Week number of year (1-53) (The first week starts on the first day of the year)
IW	ISO week number of year; the first Thursday of the new year is in week 1
CC	Century (2 digits); the 21st century starts on 2001-01-01
SCC	Same as CC except BC dates are prefixed by a minus sign
J	Julian Day (days since January 1, 4712 BC)
Q	Quarter
RM	Month in Roman numerals (I-XII; I=January) (uppercase)
rm	Month in Roman numerals (i-xii; i=January) (lowercase)
RR	First 2 digits of the year when given only the last 2 digits of the year. Result is based upon an algorithm using the current year and the given 2-digit year. The first 2 digits of the given 2-digit year will be the same as the first 2 digits of the current year with the following exceptions: If the given 2-digit year is < 50 and the last 2 digits of the current year is >= 50, then the first 2 digits for the given year is 1 greater than the first 2 digits of the current year. If the given 2-digit year is >= 50 and the last 2 digits of the current year is < 50, then the first 2 digits for the given year is 1 less than the first 2 digits of the current year.
RRRR	Only affects TO_DATE function. Allows specification of 2-digit or 4-digit year. If 2-digit year given, then returns first 2 digits of year like RR format. If 4-digit year given, returns the given 4-digit year.

Certain modifiers may be applied to any template pattern to alter its behavior. For example, `FMMonth` is the `Month` pattern with the `FM` modifier. The following table shows the modifier patterns for date/time formatting.

Table 2-21 Template Pattern Modifiers for Date/Time Formatting

Modifier	Description	Example
FM prefix	Fill mode (suppress padding blanks and zeros)	FMMonth
TH suffix	Uppercase ordinal number suffix	DDTH
th suffix	Lowercase ordinal number suffix	DDth
FX prefix	Fixed format global option (see usage notes)	FX Month DD Day
SP suffix	Spell mode	DDSP

Usage notes for date/time formatting:

- `FM` suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern fixed-width.
- `TO_TIMESTAMP` and `TO_DATE` skip multiple blank spaces in the input string if the `FX` option is not used. `FX` must be specified as the first item in the template. For example `TO_TIMESTAMP('2000 JUN', 'YYYY MON')` is correct, but

`TO_TIMESTAMP('2000 JUN', 'FXYYYY MON')` returns an error, because `TO_TIMESTAMP` expects one space only.

- Ordinary text is allowed in `TO_CHAR` templates and will be output literally.
- In conversions from string to `timestamp` or `date`, the `CC` field is ignored if there is a `YYY`, `YYYY` or `Y,YYY` field. If `CC` is used with `YY` or `Y` then the year is computed as `(CC-1)*100+YY`.

The following table shows the template patterns available for formatting numeric values.

Table 2-22 Template Patterns for Numeric Formatting

Pattern	Description
`9`	Value with the specified number of digits
`0`	Value with leading zeroes
`.` (period)	Decimal point
`,` (comma)	Group (thousand) separator
`$`	Dollar sign
`PR`	Negative value in angle brackets
`S`	Sign anchored to number (uses locale)
`L`	Currency symbol (uses locale)
`D`	Decimal point (uses locale)
`G`	Group separator (uses locale)
`MI`	Minus sign specified in right-most position (if number < 0)
`RN` or `rn`	Roman numeral (input between 1 and 3999)
`V`	Shift specified number of digits (see notes)

Usage notes for numeric formatting:

- `9` results in a value with the same number of digits as there are `9s`. If a digit is not available it outputs a space.
- `TH` does not convert values less than zero and does not convert fractional numbers.

`V` effectively multiplies the input values by 10^n, where n is the number of digits following `V`. `TO_CHAR` does not support the use of `V` combined with a decimal point. (E.g., `99.9V99` is not allowed.)

The following table shows some examples of the use of the `TO_CHAR` and `TO_DATE` functions.

Table 2-23 TO_CHAR Examples

Expression	Result
`TO_CHAR(CURRENT_TIMESTAMP, 'Day, DD HH12:MI:SS')`	`'Tuesday , 06 05:39:18'`
`TO_CHAR(CURRENT_TIMESTAMP, 'FMDay, FMDD HH12:MI:SS')`	`'Tuesday, 6 05:39:18'`
`TO_CHAR(-0.1, '99.99')`	`' -.10'`
`TO_CHAR(-0.1, 'FM9.99')`	`'-.1'`

Expression	Result
TO_CHAR(0.1, '0.9')	' 0.1'
TO_CHAR(12, '9990999.9')	' 0012.0'
TO_CHAR(12, 'FM9990999.9')	'0012.'
TO_CHAR(485, '999')	' 485'
TO_CHAR(-485, '999')	'-485'
TO_CHAR(1485, '9,999')	' 1,485'
TO_CHAR(1485, '9G999')	' 1,485'
TO_CHAR(148.5, '999.999')	' 148.500'
TO_CHAR(148.5, 'FM999.999')	'148.5'
TO_CHAR(148.5, 'FM999.990')	'148.500'
TO_CHAR(148.5, '999D999')	' 148.500'
TO_CHAR(3148.5, '9G999D999')	' 3,148.500'
TO_CHAR(-485, '999S')	'485-'
TO_CHAR(-485, '999MI')	'485-'
TO_CHAR(485, '999MI')	'485 '
TO_CHAR(485, 'FM999MI')	'485'
TO_CHAR(-485, '999PR')	'<485>'
TO_CHAR(485, 'L999')	'$ 485'
TO_CHAR(485, 'RN')	' CDLXXXV'
TO_CHAR(485, 'FMRN')	'CDLXXXV'
TO_CHAR(5.2, 'FMRN')	'V'
TO_CHAR(12, '99V999')	' 12000'
TO_CHAR(12.4, '99V999')	' 12400'
TO_CHAR(12.45, '99V9')	' 125'

2.4.7.1 IMMUTABLE TO_CHAR(TIMESTAMP, format) Function

There are certain cases of the TO_CHAR function that can result in usage of an IMMUTABLE form of the function. Basically, a function is IMMUTABLE if the function does not modify the database, and the function returns the same, consistent value dependent upon only its input parameters. That is, the settings of configuration parameters, the locale, the content of the database, etc. do not affect the results returned by the function.

For more information about function volatility categories VOLATILE, STABLE, and IMMUTABLE, please see the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/xfunc-volatility.html

A particular advantage of an IMMUTABLE function is that it can be used in the CREATE INDEX command to create an index based on that function.

In order for the TO_CHAR function to use the IMMUTABLE form the following conditions must be satisfied:

- The first parameter of the TO_CHAR function must be of data type TIMESTAMP.

- The format specified in the second parameter of the TO_CHAR function must not affect the return value of the function based on factors such as language, locale, etc. For example a format of 'YYYY-MM-DD HH24:MI:SS' can be used for an IMMUTABLE form of the function since, regardless of locale settings, the result of the function is the date and time expressed solely in numeric form. However, a format of 'DD-MON-YYYY' cannot be used for an IMMUTABLE form of the function because the 3-character abbreviation of the month may return different results depending upon the locale setting.

Format patterns that result in a non-immutable function include any variations of spelled out or abbreviated months (MONTH, MON), days (DAY, DY), median indicators (AM, PM), or era indicators (BC, AD).

For the following example, a table with a TIMESTAMP column is created.

```
CREATE TABLE ts_tbl (ts_col TIMESTAMP);
```

The following shows the successful creation of an index with the IMMUTABLE form of the TO_CHAR function.

```
edb=# CREATE INDEX ts_idx ON ts_tbl (TO_CHAR(ts_col,'YYYY-MM-DD HH24:MI:SS'));
CREATE INDEX
edb=# \dS ts_idx
                                  Index "public.ts_idx"
 Column  |       Type        |                          Definition
---------+-------------------+-------------------------------------------------------------
----
 to_char | character varying | to_char(ts_col, 'YYYY-MM-DD HH24:MI:SS'::character
varying)
btree, for table "public.ts_tbl"
```

The following results in an error because the format specified in the TO_CHAR function prevents the use of the IMMUTABLE form since the 3-character month abbreviation, MON, may result in different return values based on the locale setting.

```
edb=# CREATE INDEX ts_idx_2 ON ts_tbl (TO_CHAR(ts_col, 'DD-MON-YYYY'));
ERROR:  functions in index expression must be marked IMMUTABLE
```

2.4.8 Date/Time Functions and Operators

Table 2-25 shows the available functions for date/time value processing, with details appearing in the following subsections. The following table illustrates the behaviors of the basic arithmetic operators (+, -). For formatting functions, refer to Section 2.4.7. You should be familiar with the background information on date/time data types from Section 2.2.4.

Table 2-24 Date/Time Operators

Operator	Example	Result
+	DATE '2001-09-28' + 7	05-OCT-01 00:00:00
+	TIMESTAMP '2001-09-28 13:30:00' + 3	01-OCT-01 13:30:00
-	DATE '2001-10-01' - 7	24-SEP-01 00:00:00
-	TIMESTAMP '2001-09-28 13:30:00' - 3	25-SEP-01 13:30:00
-	TIMESTAMP '2001-09-29 03:00:00' - TIMESTAMP '2001-09-27 12:00:00'	@ 1 day 15 hours

In the date/time functions of the following table the use of the DATE and TIMESTAMP data types are interchangeable.

Table 2-25 Date/Time Functions

Function	Return Type	Description	Example	Result
ADD_MONTHS(DATE, NUMBER)	DATE	Add months to a date; see Section 2.4.8.1	ADD_MONTHS('28-FEB-97', 3.8)	31-MAY-97 00:00:00
CURRENT_DATE	DATE	Current date; see Section 2.4.8.8	CURRENT_DATE	04-JUL-07
CURRENT_TIMESTAMP	TIMESTAMP	Returns the current date and time; see Section 2.4.8.8	CURRENT_TIMESTAMP	04-JUL-07 15:33:23.484
EXTRACT(*field* FROM TIMESTAMP)	DOUBLE PRECISION	Get subfield; see Section 2.4.8.2	EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40')	20
LAST_DAY(DATE)	DATE	Returns the last day of the month represented by the given date. If the given date contains a time portion, it is carried forward to the result unchanged.	LAST_DAY('14-APR-98')	30-APR-98 00:00:00
LOCALTIMESTAMP [(*precision*)]	TIMESTAMP	Current date and time (start of current transaction); see Section 2.4.8.8	LOCALTIMESTAMP	04-JUL-07 15:33:23.484
MONTHS_BETWEEN(DATE, DATE)	NUMBER	Number of months between two dates; see Section 2.4.8.3	MONTHS_BETWEEN('28-FEB-07', '30-NOV-06')	3
NEXT_DAY(DATE,	DATE	Date falling on	NEXT_DAY('16-APR-	20-APR-07

Function	Return Type	Description	Example	Result
dayofweek)		*dayofweek* following specified date; see Section 2.4.8.4	07','FRI')	00:00:00
NEW_TIME(DATE, VARCHAR, VARCHAR)	DATE	Converts a date and time to an alternate time zone	NEW_TIME(TO_DATE '2005/05/29 01:45', 'AST', 'PST')	2005/05/29 21:45:00
NUMTODSINTERVAL(NUMBER, INTERVAL)	INTERVAL	Converts a number to a specified day or second interval; see Section 2.4.8.9.	SELECT numtodsinterval(100, 'hour');	4 days 04:00:00
NUMTOYMINTERVAL(NUMBER, INTERVAL)	INTERVAL	Converts a number to a specified year or month interval; see Section 2.4.8.10.	SELECT numtoyminterval(100, 'month');	8 years 4 mons
ROUND(DATE [, *format*])	DATE	Date rounded according to *format*; see Section 2.4.8.6	ROUND(TO_DATE('29-MAY-05'),'MON')	01-JUN-05 00:00:00
SYS_EXTRACT_UTC(TIMESTAMP WITH TIME ZONE)	TIMESTAMP	Returns Coordinated Universal Time	SYS_EXTRACT_UTC(CAST('24-MAR-11 12:30:00PM -04:00' AS TIMESTAMP WITH TIME ZONE))	24-MAR-11 16:30:00
SYSDATE	DATE	Returns current date and time	SYSDATE	01-AUG-12 11:12:34
SYSTIMESTAMP()	TIMESTAMP	Returns current date and time	SYSTIMESTAMP	01-AUG-12 11:11:23.665229 -07:00
TRUNC(DATE [*format*])	DATE	Truncate according to format; see Section 2.4.8.7	TRUNC(TO_DATE('29-MAY-05'), 'MON')	01-MAY-05 00:00:00

2.4.8.1 ADD_MONTHS

The ADD_MONTHS functions adds (or subtracts if the second parameter is negative) the specified number of months to the given date. The resulting day of the month is the same as the day of the month of the given date except when the day is the last day of the month in which case the resulting date always falls on the last day of the month.

Any fractional portion of the number of months parameter is truncated before performing the calculation.

If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the ADD_MONTHS function.

```
SELECT ADD_MONTHS('13-JUN-07',4) FROM DUAL;

     add_months
--------------------
 13-OCT-07 00:00:00
```

```
(1 row)

SELECT ADD_MONTHS('31-DEC-06',2) FROM DUAL;

     add_months
--------------------
 28-FEB-07 00:00:00
(1 row)

SELECT ADD_MONTHS('31-MAY-04',-3) FROM DUAL;

     add_months
--------------------
 29-FEB-04 00:00:00
(1 row)
```

2.4.8.2 EXTRACT

The `EXTRACT` function retrieves subfields such as year or hour from date/time values. The `EXTRACT` function returns values of type `DOUBLE PRECISION`. The following are valid field names:

`YEAR`

The year field

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

 date_part
-----------
      2001
(1 row)
```

`MONTH`

The number of the month within the year (1 - 12)

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

 date_part
-----------
         2
(1 row)
```

`DAY`

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

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

 date_part
-----------
        16
(1 row)
```

HOUR

The hour field (0 - 23)

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

 date_part
-----------
        20
(1 row)
```

MINUTE

The minutes field (0 - 59)

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

 date_part
-----------
        38
(1 row)
```

SECOND

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

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

 date_part
-----------
        40
(1 row)
```

2.4.8.3 MONTHS_BETWEEN

The MONTHS_BETWEEN function returns the number of months between two dates. The result is a numeric value which is positive if the first date is greater than the second date or negative if the first date is less than the second date.

The result is always a whole number of months if the day of the month of both date parameters is the same, or both date parameters fall on the last day of their respective months.

The following are some examples of the MONTHS_BETWEEN function.

```
SELECT MONTHS_BETWEEN('15-DEC-06','15-OCT-06') FROM DUAL;

 months_between
----------------
              2
(1 row)
```

```
SELECT MONTHS_BETWEEN('15-OCT-06','15-DEC-06') FROM DUAL;

 months_between
----------------
             -2
(1 row)

SELECT MONTHS_BETWEEN('31-JUL-00','01-JUL-00') FROM DUAL;

 months_between
----------------
    0.967741935
(1 row)

SELECT MONTHS_BETWEEN('01-JAN-07','01-JAN-06') FROM DUAL;

 months_between
----------------
             12
(1 row)
```

2.4.8.4 NEXT_DAY

The NEXT_DAY function returns the first occurrence of the given weekday strictly greater than the given date. At least the first three letters of the weekday must be specified - e.g., SAT. If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the NEXT_DAY function.

```
SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'SUNDAY') FROM DUAL;

      next_day
--------------------
 19-AUG-07 00:00:00
(1 row)

SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'MON') FROM DUAL;

      next_day
--------------------
 20-AUG-07 00:00:00
(1 row)
```

2.4.8.5 NEW_TIME

The NEW_TIME function converts a date and time from one time zone to another. NEW_TIME returns a value of type DATE. The syntax is:

```
NEW_TIME(DATE, time_zone1, time_zone2)
```

time_zone1 and *time_zone2* must be string values from the Time Zone column of the following table:

Table 2-26 Time Zones

Time Zone	Offset from UTC	Description
AST	UTC+4	Atlantic Standard Time
ADT	UTC+3	Atlantic Daylight Time
BST	UTC+11	Bering Standard Time
BDT	UTC+10	Bering Daylight Time
CST	UTC+6	Central Standard Time
CDT	UTC+5	Central Daylight Time
EST	UTC+5	Eastern Standard Time
EDT	UTC+4	Eastern Daylight Time
GMT	UTC	Greenwich Mean Time
HST	UTC+10	Alaska-Hawaii Standard Time
HDT	UTC+9	Alaska-Hawaii Daylight Time
MST	UTC+7	Mountain Standard Time
MDT	UTC+6	Mountain Daylight Time
NST	UTC+3:30	Newfoundland Standard Time
PST	UTC+8	Pacific Standard Time
PDT	UTC+7	Pacific Daylight Time
YST	UTC+9	Yukon Standard Time
YDT	UTC+8	Yukon Daylight Time

Following is an example of the NEW_TIME function.

```
SELECT NEW_TIME(TO_DATE('08-13-07 10:35:15','MM-DD-YY HH24:MI:SS'),'AST',
'PST') "Pacific Standard Time" FROM DUAL;

Pacific Standard Time
---------------------
 13-AUG-07 06:35:15
(1 row)
```

2.4.8.6 ROUND

The ROUND function returns a date rounded according to a specified template pattern. If the template pattern is omitted, the date is rounded to the nearest day. The following table shows the template patterns for the ROUND function.

Table 2-27 Template Date Patterns for the ROUND Function

Pattern	Description
CC, SCC	Returns January 1, *cc*01 where *cc* is first 2 digits of the given year if last 2 digits <= 50, or 1 greater than the first 2 digits of the given year if last 2 digits > 50; (for AD years)
SYYY, YYYY, YEAR, SYEAR,	Returns January 1, *yyyy* where *yyyy* is rounded to the nearest year; rounds down on June 30, rounds up on July 1

Pattern	Description
YYY, YY, Y	
IYYY, IYY, IY, I	Rounds to the beginning of the ISO year which is determined by rounding down if the month and day is on or before June 30th, or by rounding up if the month and day is July 1st or later
Q	Returns the first day of the quarter determined by rounding down if the month and day is on or before the 15th of the second month of the quarter, or by rounding up if the month and day is on the 16th of the second month or later of the quarter
MONTH, MON, MM, RM	Returns the first day of the specified month if the day of the month is on or prior to the 15th; returns the first day of the following month if the day of the month is on the 16th or later
WW	Round to the nearest date that corresponds to the same day of the week as the first day of the year
IW	Round to the nearest date that corresponds to the same day of the week as the first day of the ISO year
W	Round to the nearest date that corresponds to the same day of the week as the first day of the month
DDD, DD, J	Rounds to the start of the nearest day; 11:59:59 AM or earlier rounds to the start of the same day; 12:00:00 PM or later rounds to the start of the next day
DAY, DY, D	Rounds to the nearest Sunday
HH, HH12, HH24	Round to the nearest hour
MI	Round to the nearest minute

Following are examples of usage of the ROUND function.

The following examples round to the nearest hundred years.

```
SELECT TO_CHAR(ROUND(TO_DATE('1950','YYYY'),'CC'),'DD-MON-YYYY') "Century"
FROM DUAL;

   Century
-------------
 01-JAN-1901
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century"
FROM DUAL;

   Century
-------------
 01-JAN-2001
(1 row)
```

The following examples round to the nearest year.

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY')
"Year" FROM DUAL;

    Year
-------------
 01-JAN-1999
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY')
"Year" FROM DUAL;
```

```
    Year
-------------
 01-JAN-2000
(1 row)
```

The following examples round to the nearest ISO year. The first example rounds to 2004 and the ISO year for 2004 begins on December 29th of 2003. The second example rounds to 2005 and the ISO year for 2005 begins on January 3rd of that same year.

(An ISO year begins on the first Monday from which a 7 day span, Monday thru Sunday, contains at least 4 days of the new year. Thus, it is possible for the beginning of an ISO year to start in December of the prior year.)

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-
YYYY') "ISO Year" FROM DUAL;

  ISO Year
-------------
 29-DEC-2003
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-
YYYY') "ISO Year" FROM DUAL;

  ISO Year
-------------
 03-JAN-2005
(1 row)
```

The following examples round to the nearest quarter.

```
SELECT ROUND(TO_DATE('15-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-APR-07 00:00:00
(1 row)
```

The following examples round to the nearest month.

```
SELECT ROUND(TO_DATE('15-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

       Month
--------------------
 01-DEC-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;
```

```
       Month
--------------------
 01-JAN-08 00:00:00
(1 row)
```

The following examples round to the nearest week. The first day of 2007 lands on a Monday so in the first example, January 18th is closest to the Monday that lands on January 15th. In the second example, January 19th is closer to the Monday that falls on January 22nd.

```
SELECT ROUND(TO_DATE('18-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 15-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 22-JAN-07 00:00:00
(1 row)
```

The following examples round to the nearest ISO week. An ISO week begins on a Monday. In the first example, January 1, 2004 is closest to the Monday that lands on December 29, 2003. In the second example, January 2, 2004 is closer to the Monday that lands on January 5, 2004.

```
SELECT ROUND(TO_DATE('01-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 29-DEC-03 00:00:00
(1 row)

SELECT ROUND(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 05-JAN-04 00:00:00
(1 row)
```

The following examples round to the nearest week where a week is considered to start on the same day as the first day of the month.

```
SELECT ROUND(TO_DATE('05-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
--------------------
 08-MAR-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('04-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
--------------------
```

```
 01-MAR-07 00:00:00
(1 row)
```

The following examples round to the nearest day.

```
SELECT ROUND(TO_DATE('04-AUG-07 11:59:59 AM','DD-MON-YY HH:MI:SS AM'),'J')
"Day" FROM DUAL;

        Day
--------------------
 04-AUG-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J')
"Day" FROM DUAL;

        Day
--------------------
 05-AUG-07 00:00:00
(1 row)
```

The following examples round to the start of the nearest day of the week (Sunday).

```
SELECT ROUND(TO_DATE('08-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

    Day of Week
--------------------
 05-AUG-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

    Day of Week
--------------------
 12-AUG-07 00:00:00
(1 row)
```

The following examples round to the nearest hour.

```
SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:29','DD-MON-YY HH:MI'),'HH'),'DD-
MON-YY HH24:MI:SS') "Hour" FROM DUAL;

        Hour
--------------------
 09-AUG-07 08:00:00
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-
MON-YY HH24:MI:SS') "Hour" FROM DUAL;

        Hour
--------------------
 09-AUG-07 09:00:00
(1 row)
```

The following examples round to the nearest minute.

```
SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:29','DD-MON-YY
HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;

       Minute
--------------------
 09-AUG-07 08:30:00
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY
HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;

       Minute
--------------------
 09-AUG-07 08:31:00
(1 row)
```

2.4.8.7 TRUNC

The TRUNC function returns a date truncated according to a specified template pattern. If the template pattern is omitted, the date is truncated to the nearest day. The following table shows the template patterns for the TRUNC function.

Table 2-28 Template Date Patterns for the TRUNC Function

Pattern	Description
CC, SCC	Returns January 1, *cc*01 where *cc* is first 2 digits of the given year
SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	Returns January 1, *yyyy* where *yyyy* is the given year
IYYY, IYY, IY, I	Returns the start date of the ISO year containing the given date
Q	Returns the first day of the quarter containing the given date
MONTH, MON, MM, RM	Returns the first day of the specified month
WW	Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the year
IW	Returns the start of the ISO week containing the given date
W	Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the month
DDD, DD, J	Returns the start of the day for the given date
DAY, DY, D	Returns the start of the week (Sunday) containing the given date
HH, HH12, HH24	Returns the start of the hour
MI	Returns the start of the minute

Following are examples of usage of the TRUNC function.

The following example truncates down to the hundred years unit.

```
SELECT TO_CHAR(TRUNC(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century"
FROM DUAL;

   Century
-------------
 01-JAN-1901
```

```
(1 row)
```

The following example truncates down to the year.

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY')
"Year" FROM DUAL;

    Year
-------------
 01-JAN-1999
(1 row)
```

The following example truncates down to the beginning of the ISO year.

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-
YYYY') "ISO Year" FROM DUAL;

  ISO Year
-------------
 29-DEC-2003
(1 row)
```

The following example truncates down to the start date of the quarter.

```
SELECT TRUNC(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-JAN-07 00:00:00
(1 row)
```

The following example truncates to the start of the month.

```
SELECT TRUNC(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

       Month
--------------------
 01-DEC-07 00:00:00
(1 row)
```

The following example truncates down to the start of the week determined by the first day of the year. The first day of 2007 lands on a Monday so the Monday just prior to January 19th is January 15th.

```
SELECT TRUNC(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 15-JAN-07 00:00:00
(1 row)
```

The following example truncates to the start of an ISO week. An ISO week begins on a Monday. January 2, 2004 falls in the ISO week that starts on Monday, December 29, 2003.

```
SELECT TRUNC(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 29-DEC-03 00:00:00
(1 row)
```

The following example truncates to the start of the week where a week is considered to start on the same day as the first day of the month.

```
SELECT TRUNC(TO_DATE('21-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
--------------------
 15-MAR-07 00:00:00
(1 row)
```

The following example truncates to the start of the day.

```
SELECT TRUNC(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J')
"Day" FROM DUAL;

        Day
--------------------
 04-AUG-07 00:00:00
(1 row)
```

The following example truncates to the start of the week (Sunday).

```
SELECT TRUNC(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

    Day of Week
--------------------
 05-AUG-07 00:00:00
(1 row)
```

The following example truncates to the start of the hour.

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-
MON-YY HH24:MI:SS') "Hour" FROM DUAL;

        Hour
--------------------
 09-AUG-07 08:00:00
(1 row)
```

The following example truncates to the minute.

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY
HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;
```

```
       Minute
--------------------
 09-AUG-07 08:30:00
(1 row)
```

2.4.8.8 CURRENT DATE/TIME

Advanced Server provides a number of functions that return values related to the current date and time. These functions all return values based on the start time of the current transaction.

- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- LOCALTIMESTAMP(precision)

CURRENT_DATE returns the current date and time based on the start time of the current transaction. The value of CURRENT_DATE will not change if called multiple times within a transaction.

```
SELECT CURRENT_DATE FROM DUAL;

   date
-----------
 06-AUG-07
```

CURRENT_TIMESTAMP returns the current date and time. When called from a single SQL statement, it will return the same value for each occurrence within the statement. If called from multiple statements within a transaction, may return different values for each occurrence. If called from a function, may return a different value than the value returned by current_timestamp in the caller.

```
SELECT CURRENT_TIMESTAMP, CURRENT_TIMESTAMP FROM DUAL;

                current_timestamp | current_timestamp
----------------------------------+----------------------------------
 02-SEP-13 17:52:29.261473 +05:00 | 02-SEP-13 17:52:29.261474 +05:00
```

LOCALTIMESTAMP can optionally be given a precision parameter which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

```
SELECT LOCALTIMESTAMP FROM DUAL;

       timestamp
------------------------
 06-AUG-07 16:11:35.973
(1 row)

SELECT LOCALTIMESTAMP(2) FROM DUAL;
```

```
       timestamp
-----------------------
 06-AUG-07 16:11:44.58
(1 row)
```

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp. Other database systems may advance these values more frequently.

2.4.8.9 NUMTODSINTERVAL

The `NUMTODSINTERVAL` function converts a numeric value to a time interval that includes day through second interval units. When calling the function, specify the smallest fractional interval type to be included in the result set. The valid interval types are `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

The following example converts a numeric value to a time interval that includes days and hours:

```
SELECT numtodsinterval(100, 'hour');
numtodsinterval
---------------
4 days 04:00:00
(1 row)
```

The following example converts a numeric value to a time interval that includes minutes and seconds:

```
SELECT numtodsinterval(100, 'second');
numtodsinterval
---------------
1 min 40 secs
(1 row)
```

2.4.8.10 NUMTOYMINTERVAL

The `NUMTOYMINTERVAL` function converts a numeric value to a time interval that includes year through month interval units. When calling the function, specify the smallest fractional interval type to be included in the result set. The valid interval types are `YEAR` and `MONTH`.

The following example converts a numeric value to a time interval that includes years and months:

```
SELECT numtoyminterval(100, 'month');
numtoyminterval
---------------
8 years 4 mons
```

```
(1 row)
```

The following example converts a numeric value to a time interval that includes years only:

```
SELECT numtoyminterval(100, 'year');
numtoyminterval
---------------
100 years
(1 row)
```

2.4.9 Sequence Manipulation Functions

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

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

sequence is the identifier assigned to the sequence in the CREATE SEQUENCE command. The following describes the usage of these functions.

NEXTVAL

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

CURRVAL

> Return the value most recently obtained by NEXTVAL for this sequence in the current session. (An error is reported if NEXTVAL has never been called for this sequence in this session.) Notice that because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed NEXTVAL since the current session did.

If a sequence object has been created with default parameters, NEXTVAL calls on it will return successive values beginning with 1. Other behaviors can be obtained by using special parameters in the CREATE SEQUENCE command.

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

2.4.10 Conditional Expressions

The following section describes the SQL-compliant conditional expressions available in Advanced Server.

2.4.10.1 CASE

The SQL `CASE` expression is a generic conditional expression, similar to if/else statements in other languages:

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

`CASE` clauses can be used wherever an expression is valid. *`condition`* is an expression that returns a `BOOLEAN` result. If the result is `TRUE` then the value of the `CASE` expression is the *`result`* that follows the condition. If the result is `FALSE` any subsequent `WHEN` clauses are searched in the same manner. If no `WHEN` *`condition`* is `TRUE` then the value of the `CASE` expression is the *`result`* in the `ELSE` clause. If the `ELSE` clause is omitted and no condition matches, the result is `NULL`.

An example:

```
SELECT * FROM test;

 a
---
 1
 2
 3
(3 rows)

SELECT a,
    CASE WHEN a=1 THEN 'one'
         WHEN a=2 THEN 'two'
         ELSE 'other'
    END
FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

The data types of all the *`result`* expressions must be convertible to a single output type.

The following "simple" CASE expression is a specialized variant of the general form above:

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

The *expression* is computed and compared to all the *value* specifications in the WHEN clauses until one is found that is equal. If no match is found, the *result* in the ELSE clause (or a null value) is returned.

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

```
SELECT a,
    CASE a WHEN 1 THEN 'one'
           WHEN 2 THEN 'two'
           ELSE 'other'
    END
FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

A CASE expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

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

2.4.10.2 COALESCE

The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null.

```
COALESCE(value [, value2 ] ... )
```

It is often used to substitute a default value for null values when data is retrieved for display or further computation. For example:

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

Like a CASE expression, COALESCE will not evaluate arguments that are not needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which are used in some other database systems.

2.4.10.3 NULLIF

The NULLIF function returns a null value if *value1* and *value2* are equal; otherwise it returns *value1*.

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

This can be used to perform the inverse operation of the COALESCE example given above:

```
SELECT NULLIF(value1, '(none)') ...
```

If *value1* is (none), return a null, otherwise return *value1*.

2.4.10.4 NVL

The NVL function returns the first of its arguments that is not null. NVL evaluates the first expression; if that expression evaluates to NULL, NVL returns the second expression.

```
NVL(expr1, expr2)
```

The return type is the same as the argument types; all arguments must have the same data type (or be coercible to a common type). NVL returns NULL if all arguments are NULL.

The following example computes a bonus for non-commissioned employees, If an employee is a commissioned employee, this expression returns the employees commission; if the employee is not a commissioned employee (that is, his commission is NULL), this expression returns a bonus that is 10% of his salary.

```
bonus = NVL(emp.commission, emp.salary * .10)
```

2.4.10.5 NVL2

NVL2 evaluates an expression, and returns either the second or third expression, depending on the value of the first expression. If the first expression is not NULL, NVL2 returns the value in expr2; if the first expression is NULL, NVL2 returns the value in expr3.

```
NVL2(expr1, expr2, expr3)
```

The return type is the same as the argument types; all arguments must have the same data type (or be coercible to a common type).

The following example computes a bonus for commissioned employees - if a given employee is a commissioned employee, this expression returns an amount equal to 110%

of his commission; if the employee is not a commissioned employee (that is, his commission is NULL), this expression returns 0.

```
bonus = NVL2(emp.commission, emp.commission * 1.1, 0)
```

2.4.10.6 GREATEST and LEAST

The GREATEST and LEAST functions select the largest or smallest value from a list of any number of expressions.

```
GREATEST(value [, value2 ] ... )
LEAST(value [, value2 ] ... )
```

The expressions must all be convertible to a common data type, which will be the type of the result. Null values in the list are ignored. The result will be null only if all the expressions evaluate to null.

Note that GREATEST and LEAST are not in the SQL standard, but are a common extension.

2.4.11 Aggregate Functions

Aggregate functions compute a single result value from a set of input values. The built-in aggregate functions are listed in the following tables.

Table 2-29 General-Purpose Aggregate Functions

Function	Argument Type	Return Type	Description
AVG(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	NUMBER for any integer type, DOUBLE PRECISION for a floating-point argument, otherwise the same as the argument data type	The average (arithmetic mean) of all input values
COUNT(*)		BIGINT	Number of input rows
COUNT(*expression*)	Any	BIGINT	Number of input rows for which the value of expression is not null
MAX(*expression*)	Any numeric, string, date/time, or bytea type	Same as argument type	Maximum value of expression across all input values
MIN(*expression*)	Any numeric, string, date/time, or bytea type	Same as argument type	Minimum value of expression across all input values
SUM(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	BIGINT for SMALLINT or INTEGER arguments, NUMBER for BIGINT arguments, DOUBLE PRECISION for floating-point arguments, otherwise the same as the argument data type	Sum of expression across all input values

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

The following table shows the aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Where the description mentions *N*, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when *N* is zero.

Table 2-30 Aggregate Functions for Statistics

Function	Argument Type	Return Type	Description
CORR(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Correlation coefficient
COVAR_POP(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Population covariance
COVAR_SAMP(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sample covariance
REGR_AVGX(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Average of the independent variable (sum(*X*) / *N*)

Function	Argument Type	Return Type	Description
REGR_AVGY(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Average of the dependent variable (sum(*Y*) / *N*)
REGR_COUNT(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Number of input rows in which both expressions are nonnull
REGR_INTERCEPT(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	y-intercept of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs
REGR_R2(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Square of the correlation coefficient
REGR_SLOPE(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Slope of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs
REGR_SXX(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (X^2) – sum (X)^2 / *N* ("sum of squares" of the independent variable)
REGR_SXY(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (X*Y) – sum (X) * sum (Y) / *N* ("sum of products" of independent times dependent variable)
REGR_SYY(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (Y^2) – sum (Y)^2 / *N* ("sum of squares" of the dependent variable)
STDDEV(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Historic alias for STDDEV_SAMP
STDDEV_POP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Population standard deviation of the input values
STDDEV_SAMP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Sample standard deviation of the input values
VARIANCE(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Historical alias for VAR_SAMP
VAR_POP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Population variance of the input values (square of the population standard deviation)
VAR_SAMP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Sample variance of the input values (square of the sample standard deviation)

2.4.12 Subquery Expressions

This section describes the SQL-compliant subquery expressions available in Advanced Server. All of the expression forms documented in this section return Boolean (TRUE/FALSE) results.

2.4.12.1 EXISTS

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

```
EXISTS(subquery)
```

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

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

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

This simple example is like an inner join on deptno, but it produces at most one output row for each dept row, even though there are multiple matching emp rows:

```
SELECT dname FROM dept WHERE EXISTS (SELECT 1 FROM emp WHERE emp.deptno =
dept.deptno);

   dname
------------
 ACCOUNTING
 RESEARCH
 SALES
(3 rows)
```

2.4.12.2 IN

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

The result of IN is TRUE if any equal subquery row is found. The result is FALSE if no equal row is found (including the special case where the subquery returns no rows).

```
expression IN (subquery)
```

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

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

2.4.12.3 NOT IN

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

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

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

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

2.4.12.4 ANY/SOME

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

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

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

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

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

2.4.12.5 ALL

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of ALL is TRUE if all rows yield true (including the special case where the subquery returns no rows). The result is FALSE if any false result is found. The result is NULL if the comparison does not return FALSE for any row, and it returns NULL for at least one row.

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

NOT IN is equivalent to <> ALL. As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

3 Oracle Catalog Views

The Oracle Catalog Views provide information about database objects in a manner compatible with the Oracle data dictionary views.

3.1 ALL_ALL_TABLES

The ALL_ALL_TABLES view provides information about the tables accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	The name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING (5)	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

3.2 ALL_CONS_COLUMNS

The ALL_CONS_COLUMNS view provides information about the columns specified in constraints placed on tables accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
table_name	TEXT	The name of the table to which the constraint belongs.
column_name	TEXT	The name of the column referenced in the constraint.
position	SMALLINT	The position of the column within the object definition.
constraint_def	TEXT	The definition of the constraint.

3.3 ALL_CONSTRAINTS

The ALL_CONSTRAINTS view provides information about the constraints placed on tables accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
constraint_type	TEXT	The constraint type. Possible values are: C – check constraint F – foreign key constraint P – primary key constraint U – unique key constraint R – referential integrity constraint V – constraint on a view O – with read-only, on a view
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.
r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade R – restrict N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

3.4 ALL_DB_LINKS

The ALL_DB_LINKS view provides information about the database links accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the database link's owner.
db_link	TEXT	The name of the database link.
type	CHARACTER VARYING	Type of remote server. Value will be either REDWOOD or EDB
username	TEXT	User name of the user logging in.
host	TEXT	Name or IP address of the remote server.

3.5 ALL_DIRECTORIES

The ALL_DIRECTORIES view provides information about all directories created with the CREATE DIRECTORY command.

Name	Type	Description
owner	CHARACTER VARYING(30)	User name of the directory's owner.
directory_name	CHARACTER VARYING(30)	The alias name assigned to the directory.
directory_path	CHARACTER VARYING(4000)	The path to the directory.

3.6 ALL_IND_COLUMNS

The ALL_IND_COLUMNS view provides information about columns included in indexes on the tables accessible by the current user.

Name	Type	Description
index_owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
table_owner	TEXT	User name of the table owner.
table_name	TEXT	The name of the table to which the index belongs.
column_name	TEXT	The name of the column.
column_position	SMALLINT	The position of the column within the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

3.7 ALL_INDEXES

The ALL_INDEXES view provides information about the indexes on tables that may be accessed by the current user.

Name	Type	Description
owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
index_type	TEXT	The index type is always BTREE. Included for compatibility only.
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.

Name	Type	Description
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Always set to N (not compressed). Included for compatibility only.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Always set to LOGGING. Included for compatibility only.
status	TEXT	Included for compatibility only; always set to VALID.
partitioned	CHARACTER(3)	Indicates that the index is partitioned. Currently, always set to NO.
temporary	CHARACTER(1)	Indicates that an index is on a temporary table. Always set to N; included for compatibility only.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

3.8 ALL_JOBS

The ALL_JOBS view provides information about all jobs that reside in the database.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

3.9 ALL_OBJECTS

The ALL_OBJECTS view provides information about all objects that reside in the database.

Name	Type	Description
owner	TEXT	User name of the object's owner.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	CHARACTER VARYING	Whether or not the state of the object is valid. Currently, Included for compatibility only; always set to VALID.
temporary	TEXT	Y if a temporary object; N if this is a permanent object.

3.10 ALL_PART_KEY_COLUMNS

The ALL_PART_KEY_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

3.11 ALL_PART_TABLES

The ALL_PART_TABLES view provides information about all of the partitioned tables that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the partitioned table.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The partitioning type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_next_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL

3.12 ALL_POLICIES

The ALL_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

Name	Type	Description
object_owner	TEXT	Name of the owner of the object.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object on which the policy applies.
policy_group	TEXT	Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function (if the function belongs to a package).
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.
del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.
enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Included for compatibility only; always set to NO.
policy_type	TEXT	Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

3.13 ALL_QUEUES

The ALL_QUEUES view provides information about any currently defined queues.

Name	Type	Description
owner	TEXT	User name of the queue owner.
name	TEXT	The name of the queue.
queue_table	TEXT	The name of the queue table in which the queue resides.
qid	OID	The system-assigned object ID of the queue.

Name	Type	Description
queue_type	CHARACTER VARYING	The queue type; may be EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, or NORMAL_QUEUE.
max_retries	NUMERIC	The maximum number of dequeue attempts.
retrydelay	NUMERIC	The maximum time allowed between retries.
enqueue_enabled	CHARACTER VARYING	YES if the queue allows enqueuing; NO if the queue does not.
dequeue_enabled	CHARACTER VARYING	YES if the queue allows dequeuing; NO if the queue does not.
retention	CHARACTER VARYING	The number of seconds that a processed message is retained in the queue.
user_comment	CHARACTER VARYING	A user-specified comment.
network_name	CHARACTER VARYING	The name of the network on which the queue resides.
sharded	CHARACTER VARYING	YES if the queue resides on a sharded network; NO if the queue does not.

3.14 ALL_QUEUE_TABLES

The ALL_QUEUE_TABLES view provides information about all of the queue tables in the database.

Name	Type	Description
owner	TEXT	Role name of the owner of the queue table.
queue_table	TEXT	The user-specified name of the queue table.
type	CHARACTER VARYING	The type of data stored in the queue table.
object_type	TEXT	The user-defined payload type.
sort_order	CHARACTER VARYING	The order in which the queue table is sorted.
recipients	CHARACTER VARYING	Always SINGLE.
message_grouping	CHARACTER VARYING	Always NONE.
compatible	CHARACTER VARYING	The release number of the Advanced Server release with which this queue table is compatible.
primary_instance	NUMERIC	Always 0.
secondary_instance	NUMERIC	Always 0.
owner_instance	NUMERIC	The instance number of the instance that owns the queue table.
user_comment	CHARACTER VARYING	The user comment provided when the table was created.
secure	CHARACTER VARYING	YES indicates that the queue table is secure; NO indicates that it is not.

3.15 ALL_SEQUENCES

The ALL_SEQUENCES view provides information about all user-defined sequences on which the user has SELECT, or UPDATE privileges.

Name	Type	Description
sequence_owner	TEXT	User name of the sequence's owner.
schema_name	TEXT	Name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	Will always return Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers stored in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

3.16 ALL_SOURCE

The ALL_SOURCE view provides a source code listing of the following program types: functions, procedures, triggers, package specifications, and package bodies.

Name	Type	Description
owner	TEXT	User name of the program's owner.
schema_name	TEXT	Name of the schema in which the program belongs.
name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

3.17 ALL_SUBPART_KEY_COLUMNS

The ALL_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

3.18 ALL_SYNONYMS

The ALL_SYNONYMS view provides information on all synonyms that may be referenced by the current user.

Name	Type	Description
owner	TEXT	User name of the synonym's owner.
schema_name	TEXT	The name of the schema in which the synonym resides.
synonym_name	TEXT	Name of the synonym.
table_owner	TEXT	User name of the object's owner.
table_schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the object that the synonym refers to.
db_link	TEXT	The name of any associated database link.

3.19 ALL_TAB_COLUMNS

The ALL_TAB_COLUMNS view provides information on all columns in all user-defined tables and views.

Name	Type	Description
owner	CHARACTER VARYING	User name of the owner of the table or view in which the column resides.
schema_name	CHARACTER VARYING	Name of the schema in which the table or view resides.
table_name	CHARACTER VARYING	Name of the table or view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable. Possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the table or view.
data_default	CHARACTER VARYING	Default value assigned to the column.

3.20 ALL_TAB_PARTITIONS

The ALL_TAB_PARTITIONS view provides information about all of the partitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the partition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.
tablespace_name	TEXT	The name of the tablespace in which the partition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

3.21 ALL_TAB_SUBPARTITIONS

The ALL_TAB_SUBPARTITIONS view provides information about all of the subpartitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the subpartition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the partition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.
subpartition_position	INTEGER	1 for the first subpartition; 2 for the second subpartition, etc.
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the subpartition backing table.

3.22 ALL_TABLES

The ALL_TABLES view provides information on all user-defined tables.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Whether or not the state of the table is valid. Currently, Included for compatibility only; always set to VALID.
temporary	CHARACTER(1)	Y if this is a temporary table; N if this is not a temporary table.

3.23 ALL_TRIGGERS

The ALL_TRIGGERS view provides information about the triggers on tables that may be accessed by the current user.

Name	Type	Description
owner	TEXT	User name of the trigger's owner.
schema_name	TEXT	The name of the schema in which the trigger resides.
trigger_name	TEXT	The name of the trigger.
trigger_type	TEXT	The type of the trigger. Possible values are: BEFORE ROW BEFORE STATEMENT AFTER ROW AFTER STATEMENT
triggering_event	TEXT	The event that fires the trigger.
table_owner	TEXT	The user name of the owner of the table on which the trigger is defined.
base_object_type	TEXT	Included for compatibility only. Value will always be TABLE.
table_name	TEXT	The name of the table on which the trigger is defined.
referencing_name	TEXT	Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD.
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

3.24 ALL_TYPES

The ALL_TYPES view provides information about the object types available to the current user.

Name	Type	Description
owner	TEXT	The owner of the object type.
schema_name	TEXT	The name of the schema in which the type is defined.
type_name	TEXT	The name of the type.
type_oid	OID	The object identifier (OID) of the type.
typecode	TEXT	The typecode of the type. Possible values are: OBJECT COLLECTION OTHER
attributes	INTEGER	The number of attributes in the type.

3.25 ALL_USERS

The ALL_USERS view provides information on all user names.

Name	Type	Description
username	TEXT	Name of the user.
user_id	OID	Numeric user id assigned to the user.
created	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL.

3.26 ALL_VIEW_COLUMNS

The ALL_VIEW_COLUMNS view provides information on all columns in all user-defined views.

Name	Type	Description
owner	CHARACTER VARYING	User name of the view's owner.
schema_name	CHARACTER VARYING	Name of the schema in which the view belongs.
view_name	CHARACTER VARYING	Name of the view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

3.27 ALL_VIEWS

The ALL_VIEWS view provides information about all user-defined views.

Name	Type	Description
owner	TEXT	User name of the view's owner.
schema_name	TEXT	Name of the schema in which the view belongs.
view_name	TEXT	Name of the view.
text	TEXT	The SELECT statement that defines the view.

3.28 DBA_ALL_TABLES

The DBA_ALL_TABLES view provides information about all tables in the database.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

3.29 DBA_CONS_COLUMNS

The DBA_CONS_COLUMNS view provides information about all columns that are included in constraints that are specified in on all tables in the database.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
table_name	TEXT	The name of the table to which the constraint belongs.
column_name	TEXT	The name of the column referenced in the constraint.
position	SMALLINT	The position of the column within the object definition.
constraint_def	TEXT	The definition of the constraint.

3.30 DBA_CONSTRAINTS

The DBA_CONSTRAINTS view provides information about all constraints on tables in the database.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
constraint_type	TEXT	The constraint type. Possible values are: C – check constraint F – foreign key constraint P – primary key constraint U – unique key constraint R – referential integrity constraint V – constraint on a view O – with read-only, on a view
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.
r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade R - restrict N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

3.31 DBA_DB_LINKS

The DBA_DB_LINKS view provides information about all database links in the database.

Name	Type	Description
owner	TEXT	User name of the database link's owner.
db_link	TEXT	The name of the database link.
type	CHARACTER VARYING	Type of remote server. Value will be either REDWOOD or EDB
username	TEXT	User name of the user logging in.
host	TEXT	Name or IP address of the remote server.

3.32 DBA_DIRECTORIES

The DBA_DIRECTORIES view provides information about all directories created with the CREATE DIRECTORY command.

Name	Type	Description
owner	CHARACTER VARYING(30)	User name of the directory's owner.
directory_name	CHARACTER VARYING(30)	The alias name assigned to the directory.
directory_path	CHARACTER VARYING(4000)	The path to the directory.

3.33 DBA_IND_COLUMNS

The DBA_IND_COLUMNS view provides information about all columns included in indexes, on all tables in the database.

Name	Type	Description
index_owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	Name of the index.
table_owner	TEXT	User name of the table's owner.
table_name	TEXT	Name of the table in which the index belongs.
column_name	TEXT	Name of column or attribute of object column.
column_position	SMALLINT	The position of the column in the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

3.34 DBA_INDEXES

The DBA_INDEXES view provides information about all indexes in the database.

Name	Type	Description
owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index resides.
index_name	TEXT	The name of the index.
index_type	TEXT	The index type is always BTREE. Included for compatibility only.
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.

Name	Type	Description
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Always set to N (not compressed). Included for compatibility only.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Included for compatibility only. Always set to LOGGING.
status	TEXT	Whether or not the state of the object is valid. (VALID or INVALID).
partitioned	CHARACTER(3)	Indicates that the index is partitioned. Always set to NO.
temporary	CHARACTER(1)	Indicates that an index is on a temporary table. Always set to N.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

3.35 DBA_JOBS

The DBA_JOBS view provides information about all jobs in the database.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

3.36 DBA_OBJECTS

The DBA_OBJECTS view provides information about all objects in the database.

Name	Type	Description
owner	TEXT	User name of the object's owner.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	CHARACTER VARYING	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

3.37 DBA_PART_KEY_COLUMNS

The DBA_PART_KEY_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

3.38 DBA_PART_TABLES

The DBA_PART_TABLES view provides information about all of the partitioned tables in the database.

Name	Type	Description
owner	TEXT	The owner of the partitioned table.
schema_name	TEXT	The schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_next_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL

3.39 DBA_POLICIES

The DBA_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

Name	Type	Description
object_owner	TEXT	Name of the owner of the object.
schema_name	TEXT	The name of the schema in which the object resides.
object_name	TEXT	Name of the object to which the policy applies.
policy_group	TEXT	Name of the policy group. Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function (if the function belongs to a package).
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.
del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.
Enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Included for compatibility only; always set to NO.
policy_type	TEXT	Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

3.40 DBA_PROFILES

The DBA_PROFILES view provides information about existing profiles. The table includes a row for each profile/resource combination.

Name	Type	Description
profile	CHARACTER VARYING(128)	The name of the profile.
resource_name	CHARACTER VARYING(32)	The name of the resource associated with the profile.
resource_type	CHARACTER VARYING(8)	The type of resource governed by the profile; currently PASSWORD for all supported resources.
limit	CHARACTER VARYING(128)	The limit values of the resource.
common	CHARACTER VARYING(3)	YES for a user-created profile; NO for a system-defined profile.

3.41 DBA_QUEUES

The DBA_QUEUES view provides information about any currently defined queues.

Name	Type	Description
owner	TEXT	User name of the queue owner.
name	TEXT	The name of the queue.
queue_table	TEXT	The name of the queue table in which the queue resides.
qid	OID	The system-assigned object ID of the queue.
queue_type	CHARACTER VARYING	The queue type; may be EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, or NORMAL_QUEUE.
max_retries	NUMERIC	The maximum number of dequeue attempts.
retrydelay	NUMERIC	The maximum time allowed between retries.
enqueue_enabled	CHARACTER VARYING	YES if the queue allows enqueuing; NO if the queue does not.
dequeue_enabled	CHARACTER VARYING	YES if the queue allows dequeuing; NO if the queue does not.
retention	CHARACTER VARYING	The number of seconds that a processed message is retained in the queue.
user_comment	CHARACTER VARYING	A user-specified comment.
network_name	CHARACTER VARYING	The name of the network on which the queue resides.
sharded	CHARACTER VARYING	YES if the queue resides on a sharded network; NO if the queue does not.

3.42 DBA_QUEUE_TABLES

The DBA_QUEUE_TABLES view provides information about all of the queue tables in the database.

Name	Type	Description
owner	TEXT	Role name of the owner of the queue table.
queue_table	TEXT	The user-specified name of the queue table.
type	CHARACTER VARYING	The type of data stored in the queue table.
object_type	TEXT	The user-defined payload type.
sort_order	CHARACTER VARYING	The order in which the queue table is sorted.
recipients	CHARACTER VARYING	Always SINGLE.
message_grouping	CHARACTER VARYING	Always NONE.
compatible	CHARACTER VARYING	The release number of the Advanced Server release with which this queue table is compatible.
primary_instance	NUMERIC	Always 0.
secondary_instance	NUMERIC	Always 0.
owner_instance	NUMERIC	The instance number of the instance that owns the queue table.
user_comment	CHARACTER VARYING	The user comment provided when the table was created.
secure	CHARACTER VARYING	YES indicates that the queue table is secure; NO indicates that it is not.

3.43 DBA_ROLE_PRIVS

The DBA_ROLE_PRIVS view provides information on all roles that have been granted to users. A row is created for each role to which a user has been granted.

Name	Type	Description
grantee	TEXT	User name to whom the role was granted.
granted_role	TEXT	Name of the role granted to the grantee.
admin_option	TEXT	YES if the role was granted with the admin option, NO otherwise.
default_role	TEXT	YES if the role is enabled when the grantee creates a session.

3.44 DBA_ROLES

The DBA_ROLES view provides information on all roles with the NOLOGIN attribute (groups).

Name	Type	Description
role	TEXT	Name of a role having the NOLOGIN attribute – i.e., a group.
password_required	TEXT	Included for compatibility only; always N.

3.45 DBA_SEQUENCES

The DBA_SEQUENCES view provides information about all user-defined sequences.

Name	Type	Description
sequence_owner	TEXT	User name of the sequence's owner.
schema_name	TEXT	The name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	Will always return Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers stored in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

3.46 DBA_SOURCE

The DBA_SOURCE view provides the source code listing of all objects in the database.

Name	Type	Description
owner	TEXT	User name of the program's owner.
schema_name	TEXT	Name of the schema in which the program belongs.
name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

3.47 DBA_SUBPART_KEY_COLUMNS

The DBA_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

3.48 DBA_SYNONYMS

The DBA_SYNONYM view provides information about all synonyms in the database.

Name	Type	Description
owner	TEXT	User name of the synonym's owner.
schema_name	TEXT	Name of the schema in which the synonym belongs.
synonym_name	TEXT	Name of the synonym.
table_owner	TEXT	User name of the table's owner on which the synonym is defined.
table_schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	Name of the table on which the synonym is defined.
db_link	TEXT	Name of any associated database link.

3.49 DBA_TAB_COLUMNS

The DBA_TAB_COLUMNS view provides information about all columns in the database.

Name	Type	Description
owner	CHARACTER VARYING	User name of the owner of the table or view in which the column resides.
schema_name	CHARACTER VARYING	Name of the schema in which the table or view resides.
table_name	CHARACTER VARYING	Name of the table or view in which the column resides.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the table or view.
data_default	CHARACTER VARYING	Default value assigned to the column.

3.50 DBA_TAB_PARTITIONS

The DBA_TAB_PARTITIONS view provides information about all of the partitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the partition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.
tablespace_name	TEXT	The name of the tablespace in which the partition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

3.51 DBA_TAB_SUBPARTITIONS

The DBA_TAB_SUBPARTITIONS view provides information about all of the subpartitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the subpartition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the partition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.
subpartition_position	INTEGER	1 for the first subpartition; 2 for the second subpartition, etc.
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the subpartition backing table.

3.52 DBA_TABLES

The DBA_TABLES view provides information about all tables in the database.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID.
temporary	CHARACTER(1)	Y if the table is temporary; N if the table is permanent.

3.53 DBA_TRIGGERS

The DBA_TRIGGERS view provides information about all triggers in the database.

Name	Type	Description
owner	TEXT	User name of the trigger's owner.
schema_name	TEXT	The name of the schema in which the trigger resides.
trigger_name	TEXT	The name of the trigger.
trigger_type	TEXT	The type of the trigger. Possible values are: BEFORE ROW BEFORE STATEMENT AFTER ROW AFTER STATEMENT
triggering_event	TEXT	The event that fires the trigger.
table_owner	TEXT	The user name of the owner of the table on which the trigger is defined.
base_object_type	TEXT	Included for compatibility only. Value will always be TABLE.
table_name	TEXT	The name of the table on which the trigger is defined.
referencing_names	TEXT	Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD.
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

3.54 DBA_TYPES

The DBA_TYPES view provides information about all object types in the database.

Name	Type	Description
owner	TEXT	The owner of the object type.
schema_name	TEXT	The name of the schema in which the type is defined.
type_name	TEXT	The name of the type.
type_oid	OID	The object identifier (OID) of the type.
typecode	TEXT	The typecode of the type. Possible values are: OBJECT COLLECTION OTHER
attributes	INTEGER	The number of attributes in the type.

3.55 DBA_USERS

The DBA_USERS view provides information about all users of the database.

Name	Type	Description
username	TEXT	User name of the user.
user_id	OID	ID number of the user.
password	CHARACTER VARYING(30)	The password (encrypted) of the user.
account_status	CHARACTER VARYING(32)	The current status of the account. Possible values are: OPEN EXPIRED EXPIRED(GRACE) EXPIRED & LOCKED EXPIRED & LOCKED(TIMED) EXPIRED(GRACE) & LOCKED EXPIRED(GRACE) & LOCKED(TIMED) LOCKED LOCKED(TIMED) Use the edb_get_role_status(*role_id*) function to get the current status of the account.
lock_date	TIMESTAMP WITHOUT TIME ZONE	If the account status is LOCKED, lock_date displays the date and time the account was locked.
expiry_date	TIMESTAMP WITHOUT TIME ZONE	The expiration date of the password. Use the edb_get_password_expiry_date(*role_id*) function to get the current password expiration date.
default_tablespace	TEXT	The default tablespace associated with the account.
temporary_tablespace	CHARACTER VARYING(30)	Included for compatibility only. The value will always be '' (an empty string).
created	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only. The value is always NULL.
profile	CHARACTER VARYING(30)	The profile associated with the user.
initial_rsrc_consumer_group	CHARACTER VARYING(30)	Included for compatibility only. The value is always NULL.
external_name	CHARACTER VARYING(4000)	Included for compatibility only. The value is always NULL.

3.56 DBA_VIEW_COLUMNS

The DBA_VIEW_COLUMNS view provides information on all columns in the database.

Name	Type	Description
owner	CHARACTER VARYING	User name of the view's owner.
schema_name	CHARACTER VARYING	Name of the schema in which the view belongs.
view_name	CHARACTER VARYING	Name of the view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

3.57 DBA_VIEWS

The DBA_VIEWS view provides information about all views in the database.

Name	Type	Description
owner	TEXT	User name of the view's owner.
schema_name	TEXT	Name of the schema in which the view belongs.
view_name	TEXT	Name of the view.
text	TEXT	The text of the SELECT statement that defines the view.

3.58 USER_ALL_TABLES

The USER_ALL_TABLES view provides information about all tables owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID..
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

3.59 USER_CONS_COLUMNS

The USER_CONS_COLUMNS view provides information about all columns that are included in constraints in tables that are owned by the current user.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
table_name	TEXT	The name of the table to which the constraint belongs.
column_name	TEXT	The name of the column referenced in the constraint.
position	SMALLINT	The position of the column within the object definition.
constraint_def	TEXT	The definition of the constraint.

3.60 USER_CONSTRAINTS

The USER_CONSTRAINTS view provides information about all constraints placed on tables that are owned by the current user.

Name	Type	Description
owner	TEXT	The name of the owner of the constraint.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
constraint_type	TEXT	The constraint type. Possible values are: C – check constraint F – foreign key constraint P – primary key constraint U – unique key constraint R – referential integrity constraint V – constraint on a view O – with read-only, on a view
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.
r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade R – restrict N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

3.61 USER_DB_LINKS

The USER_DB_LINKS view provides information about all database links that are owned by the current user.

Name	Type	Description
db_link	TEXT	The name of the database link.
type	CHARACTER VARYING	Type of remote server. Value will be either REDWOOD or EDB
username	TEXT	User name of the user logging in.
password	TEXT	Password used to authenticate on the remote server.
host	TEXT	Name or IP address of the remote server.

3.62 USER_IND_COLUMNS

The USER_IND_COLUMNS view provides information about all columns referred to in indexes on tables that are owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
table_name	TEXT	The name of the table to which the index belongs.
column_name	TEXT	The name of the column.
column_position	SMALLINT	The position of the column within the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

3.63 USER_INDEXES

The USER_INDEXES view provides information about all indexes on tables that are owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
index_type	TEXT	Included for compatibility only. The index type is always BTREE.
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Included for compatibility only. Always set to N (not compressed).
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Included for compatibility only. Always set to LOGGING.
status	TEXT	Whether or not the state of the object is valid. (VALID or INVALID).
partitioned	CHARACTER(3)	Included for compatibility only. Always set to NO.
temporary	CHARACTER(1)	Included for compatibility only. Always set to N.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

3.64 USER_JOBS

The USER_JOBS view provides information about all jobs owned by the current user.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date.
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

3.65 USER_OBJECTS

The USER_OBJECTS view provides information about all objects that are owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	CHARACTER VARYING	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the object is temporary; N if the object is not temporary.

3.66 USER_PART_KEY_COLUMNS

The USER_PART_KEY_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

3.67 USER_PART_TABLES

The USER_PART_TABLES view provides information about all of the partitioned tables in the database that are owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The partitioning type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL

3.68 USER_POLICIES

The USER_POLICIES view provides information on policies where the schema containing the object on which the policy applies has the same name as the current session user. This view is accessible only to superusers.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the object resides.
object_name	TEXT	Name of the object on which the policy applies.
policy_group	TEXT	Name of the policy group. Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function (if the function belongs to a package).
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.
del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.
enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Whether or not the policy is static. Included for compatibility only; always set to NO.
policy_type	TEXT	Policy type. Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

3.69 USER_QUEUES

The USER_QUEUES view provides information about any queue on which the current user has usage privileges.

Name	Type	Description
name	TEXT	The name of the queue.
queue_table	TEXT	The name of the queue table in which the queue resides.
qid	OID	The system-assigned object ID of the queue.
queue_type	CHARACTER VARYING	The queue type; may be EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, or NORMAL_QUEUE.
max_retries	NUMERIC	The maximum number of dequeue attempts.
retrydelay	NUMERIC	The maximum time allowed between retries.
enqueue_enabled	CHARACTER VARYING	YES if the queue allows enqueuing; NO if the queue does not.
dequeue_enabled	CHARACTER VARYING	YES if the queue allows dequeuing; NO if the queue does not.
retention	CHARACTER VARYING	The number of seconds that a processed message is retained in the queue.
user_comment	CHARACTER VARYING	A user-specified comment.
network_name	CHARACTER VARYING	The name of the network on which the queue resides.
sharded	CHARACTER VARYING	YES if the queue resides on a sharded network; NO if the queue does not.

3.70 USER_QUEUE_TABLES

The USER_QUEUE_TABLES view provides information about all of the queue tables accessible by the current user.

Name	Type	Description
queue_table	TEXT	The user-specified name of the queue table.
type	CHARACTER VARYING	The type of data stored in the queue table.
object_type	TEXT	The user-defined payload type.
sort_order	CHARACTER VARYING	The order in which the queue table is sorted.
recipients	CHARACTER VARYING	Always SINGLE.
message_grouping	CHARACTER VARYING	Always NONE.
compatible	CHARACTER VARYING	The release number of the Advanced Server release with which this queue table is compatible.
primary_instance	NUMERIC	Always 0.
secondary_instance	NUMERIC	Always 0.
owner_instance	NUMERIC	The instance number of the instance that owns the queue table.

Name	Type	Description
user_comment	CHARACTER VARYING	The user comment provided when the table was created.
secure	CHARACTER VARYING	YES indicates that the queue table is secure; NO indicates that it is not.

3.71 USER_ROLE_PRIVS

The USER_ROLE_PRIVS view provides information about the privileges that have been granted to the current user. A row is created for each role to which a user has been granted.

Name	Type	Description
username	TEXT	The name of the user to which the role was granted.
granted_role	TEXT	Name of the role granted to the grantee.
admin_option	TEXT	YES if the role was granted with the admin option, NO otherwise.
default_role	TEXT	YES if the role is enabled when the grantee creates a session.
os_granted	CHARACTER VARYING(3)	Included for compatibility only; always NO.

3.72 USER_SEQUENCES

The USER_SEQUENCES view provides information about all user-defined sequences that belong to the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	Included for compatibility only; always Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

3.73 USER_SOURCE

The USER_SOURCE view provides information about all programs owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the program belongs.
name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

3.74 USER_SUBPART_KEY_COLUMNS

The USER_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that belong to the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

3.75 USER_SYNONYMS

The USER_SYNONYMS view provides information about all synonyms owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the synonym resides.
synonym_name	TEXT	Name of the synonym.
table_owner	TEXT	User name of the table's owner on which the synonym is defined.
table_schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	Name of the table on which the synonym is defined.
db_link	TEXT	Name of any associated database link.

3.76 USER_TAB_COLUMNS

The USER_TAB_COLUMNS view displays information about all columns in tables and views owned by the current user.

Name	Type	Description
schema_name	CHARACTER VARYING	Name of the schema in which the table or view resides.
table_name	CHARACTER VARYING	Name of the table or view in which the column resides.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the table.
data_default	CHARACTER VARYING	Default value assigned to the column.

3.77 USER_TAB_PARTITIONS

The USER_TAB_PARTITIONS view provides information about all of the partitions that are owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.
tablespace_name	TEXT	The name of the tablespace in which the partition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

3.78 USER_TAB_SUBPARTITIONS

The USER_TAB_SUBPARTITIONS view provides information about all of the subpartitions owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the partition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.
subpartition_position	INTEGER	1 for the first subpartition; 2 for the second subpartition, etc.
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

3.79 USER_TABLES

The USER_TABLES view displays information about all tables owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID..
temporary	CHARACTER(1)	Y if the table is temporary; N if the table is not temporary.

3.80 USER_TRIGGERS

The USER_TRIGGERS view displays information about all triggers on tables owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the trigger resides.
trigger_name	TEXT	The name of the trigger.
trigger_type	TEXT	The type of the trigger. Possible values are: BEFORE ROW BEFORE STATEMENT AFTER ROW AFTER STATEMENT
triggering_event	TEXT	The event that fires the trigger.
table_owner	TEXT	The user name of the owner of the table on which the trigger is defined.
base_object_type	TEXT	Included for compatibility only. Value will always be TABLE.
table_name	TEXT	The name of the table on which the trigger is defined.
referencing_names	TEXT	Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD.
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

3.81 USER_TYPES

The USER_TYPES view provides information about all object types owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the type is defined.
type_name	TEXT	The name of the type.
type_oid	OID	The object identifier (OID) of the type.
typecode	TEXT	The typecode of the type. Possible values are: OBJECT COLLECTION OTHER
attributes	INTEGER	The number of attributes in the type.

3.82 USER_USERS

The USER_USERS view provides information about the current user.

Name	Type	Description
username	TEXT	User name of the user.
user_id	OID	ID number of the user.
account_status	CHARACTER VARYING(32)	The current status of the account. Possible values are: OPEN EXPIRED EXPIRED(GRACE) EXPIRED & LOCKED EXPIRED & LOCKED(TIMED) EXPIRED(GRACE) & LOCKED EXPIRED(GRACE) & LOCKED(TIMED) LOCKED LOCKED(TIMED) Use the edb_get_role_status(*role_id*) function to get the current status of the account.
lock_date	TIMESTAMP WITHOUT TIME ZONE	If the account status is LOCKED, lock_date displays the date and time the account was locked.
expiry_date	TIMESTAMP WITHOUT TIME ZONE	The expiration date of the account.
default_tablespace	TEXT	The default tablespace associated with the account.
temporary_tablespace	CHARACTER VARYING(30)	Included for compatibility only. The value will always be '' (an empty string).
created	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only. The value will always be NULL.
initial_rsrc_consumer_group	CHARACTER VARYING(30)	Included for compatibility only. The value will always be NULL.
external_name	CHARACTER VARYING(4000)	Included for compatibility only; always set to NULL.

3.83 USER_VIEW_COLUMNS

The USER_VIEW_COLUMNS view provides information about all columns in views owned by the current user.

Name	Type	Description
schema_name	CHARACTER VARYING	Name of the schema in which the view belongs.
view_name	CHARACTER VARYING	Name of the view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

3.84 USER_VIEWS

The USER_VIEWS view provides information about all views owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the view resides.
view_name	TEXT	Name of the view.
text	TEXT	The SELECT statement that defines the view.

3.85 V$VERSION

The V$VERSION view provides information about product compatibility.

Name	Type	Description
banner	TEXT	Displays product compatibility information.

3.86 PRODUCT_COMPONENT_VERSION

The `PRODUCT_COMPONENT_VERSION` view provides version information about product version compatibility.

Name	Type	Description
`product`	`CHARACTER VARYING(74)`	The name of the product.
`version`	`CHARACTER VARYING(74`	The version number of the product.
`status`	`CHARACTER VARYING(74)`	Included for compatibility; always `Available`.

4 System Catalog Tables

The following system catalog tables contain definitions of database objects. The layout of the system tables is subject to change; if you are writing an application that depends on information stored in the system tables, it would be prudent to use an existing catalog view, or create a catalog view to isolate the application from changes to the system table.

4.1 dual

`dual` is a single-row, single-column table that is provided for compatibility with Oracle databases only.

Column	Type	Modifiers	Description
dummy	VARCHAR2(1)		Provided for compatibility only.

4.2 edb_dir

The `edb_dir` table contains one row for each alias that points to a directory created with the `CREATE DIRECTORY` command. A directory is an alias for a pathname that allows a user limited access to the host file system.

You can use a directory to fence a user into a specific directory tree within the file system. For example, the `UTL_FILE` package offers functions that permit a user to read and write files and directories in the host file system, but only allows access to paths that the database administrator has granted access to via a `CREATE DIRECTORY` command.

Column	Type	Modifiers	Description
dirname	"name"	not null	The name of the alias.
dirowner	oid	not null	The OID of the user that owns the alias.
dirpath	text		The directory name to which access is granted.
diracl	aclitem[]		The access control list that determines which users may access the alias.

4.3 edb_password_history

The `edb_password_history` table contains one row for each password change. The table is shared across all databases within a cluster.

Column	Type	References	Description
passhistroleid	oid	pg_authid.oid	The ID of a role.
passhistpassword	text		Role password in md5 encrypted form.
passhistpasswordsetat	timestamptz		The time the password was set.

4.4 edb_policy

The edb_policy table contains one row for each policy.

Column	Type	Modifiers	Description
policyname	name	not null	The policy name.
policygroup	oid	not null	Currently unused.
policyobject	oid	not null	The OID of the table secured by this policy (the object_schema plus the object_name).
policykind	char	not null	The kind of object secured by this policy: 'r' for a table 'v' for a view = for a synonym Currently always 'r'.
policyproc	oid	not null	The OID of the policy function (function_schema plus policy_function).
policyinsert	boolean	not null	True if the policy is enforced by INSERT statements.
policyselect	boolean	not null	True if the policy is enforced by SELECT statements.
policydelete	boolean	not null	True if the policy is enforced by DELETE statements.
policyupdate	boolean	not null	True if the policy is enforced by UPDATE statements.
policyindex	boolean	not null	Currently unused.
policyenabled	boolean	not null	True if the policy is enabled.
policyupdatecheck	boolean	not null	True if rows updated by an UPDATE statement must satisfy the policy.
policystatic	boolean	not null	Currently unused.
policytype	integer	not null	Currently unused.
policyopts	integer	not null	Currently unused.
policyseccols	int2vector	not null	The column numbers for columns listed in sec_relevant_cols.

4.5 edb_profile

The edb_profile table stores information about the available profiles. edb_profiles is shared across all databases within a cluster.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected).
prfname	name		The name of the profile.
prffailedloginattempts	integer		The number of failed login attempts allowed by the profile. -1 indicates that the value from the default profile should be used. -2 indicates

Column	Type	References	Description
			no limit on failed login attempts.
prfpasswordlocktime	integer		The password lock time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the account should be locked permanently.
prfpasswordlifetime	integer		The password life time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
prfpasswordgracetime	integer		The password grace time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
prfpasswordreusetime	integer		The number of seconds that a user must wait before reusing a password. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
prfpasswordreusemax	integer		The number of password changes that have to occur before a password can be reused. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
prfpasswordverifyfuncdb	oid	pg_database.oid	The OID of the database in which the password verify function exists.
prfpasswordverifyfunc	oid	pg_proc.oid	The OID of the password verify function associated with the profile.

4.6 edb_variable

The edb_variable table contains one row for each package level variable (each variable declared within a package).

Column	Type	Modifiers	Description
varname	"name"	not null	The name of the variable.
varpackage	oid	not null	The OID of the pg_namespace row that stores the package.
vartype	oid	not null	The OID of the pg_type row that defines the type of the variable.
varaccess	"char"	not null	+ if the variable is visible outside of the package. - if the variable is only visible within the package.

Column	Type	Modifiers	Description
			Note: Public variables are declared within the package header; private variables are declared within the package body.
varsrc	text		Contains the source of the variable declaration, including any default value expressions for the variable.
varseq	smallint	not null	The order in which the variable was declared in the package.

4.7 pg_synonym

The pg_synonym table contains one row for each synonym created with the CREATE SYNONYM command or CREATE PUBLIC SYNONYM command.

Column	Type	Modifiers	Description
synname	"name"	not null	The name of the synonym.
synnamespace	oid	not null	Replaces synowner. Contains the OID of the pg_namespace row where the synonym is stored
synowner	oid	not null	The OID of the user that owns the synonym.
synobjschema	"name"	not null	The schema in which the referenced object is defined.
synobjname	"name"	not null	The name of the referenced object.
synlink	text		The (optional) name of the database link in which the referenced object is defined.

4.8 product_component_version

The product_component_version table contains information about feature compatibility; an application can query this table at installation or run time to verify that features used by the application are available with this deployment.

Column	Type	Description
product	character varying (74)	The name of the product.
version	character varying (74)	The version number of the product.
status	character varying (74)	The status of the release.

5 Acknowledgements

The PostgreSQL 8.3, 8.4, 9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, and 10 Documentation provided the baseline for the portions of this guide that are common to PostgreSQL, and is hereby acknowledged:

Portions of this EnterpriseDB Software and Documentation may utilize the following copyrighted material, the use of which is hereby acknowledged.

PostgreSQL Documentation, Database Management System

PostgreSQL is Copyright © 1996-2017 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below.

Postgres95 is Copyright © 1994-5 by the Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS-IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.