Standard support for this version of Advanced Server ended as of February 11, 2021. Please contact us for extended support. To view documentation for the current version, click here.

Database Compatibility for Oracle® Developer's Guide

EDB Postgres™ Advanced Server 9.5

formerly Postgres Plus Advanced Server 9.5

February 15, 2021

Database Compatibility for Oracle® Developer's Guide, Version 9.5
by EnterpriseDB® Corporation
Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

EnterpriseDB Corporation, 34 Crosby Drive, Suite 100, Bedford, MA 01730, USA
T +1 781 357 3390 **F** +1 978 589 5701 **E** info@enterprisedb.com www.enterprisedb.com

Table of Contents

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

1 Introduction

Notice: The names for EDB™'s products have changed. The product formerly referred to as Postgres Plus Advanced Server is now referred to as EDB Postgres™ Advanced Server. Until a new version of this documentation is published, wherever you see Postgres Plus Advanced Server you may substitute it with EDB Postgres Advanced Server. Name changes in software and software outputs will be phased in over time.

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.

Advanced Server contains a rich set of features that enables development of database applications for either PostgreSQL or Oracle. This guide focuses solely on the features that provide database compatibility for Oracle. To learn about all of the features of Advanced Server, consult the Advanced Server documentation set.

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 choosing:

- 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

Advanced Server provides these features.

In addition, for applications written using the *Oracle Call Interface* (OCI), EnterpriseDB's Open Client Library (OCL) provides interoperability with these applications.

The remainder of this guide explains each of these areas in more detail.

1.1 What's New

The following database compatibility for Oracle features have been added to Advanced Server 9.4 to create Advanced Server 9.5:

- Advanced Server now supports the `SYS_CONNECT_BY_PATH` for hierarchical queries. For more information, see Section 2.2.5.7.
- Advanced Server now supports `WAIT n` syntax in the `FOR UPDATE` clause of a SQL `SELECT` statement. For more information, see Section 3.3.65.12.
- Advanced Server now supports the `DBMS_SESSION.SET_ROLE` procedure. For more information, see Section 7.13.1.
- Advanced Server now supports the `UTL_RAW` package. For more information, see Section 7.20.
- Advanced Server now supports profile management. For more information about creating and using named profiles, see Section 2.3.
- Advanced Server supports the use of `EDBLDR_ENV_STYLE` to specify the style of environment variables recognized by EDB*Loader. For more information, see Section 11.2.3.
- EDB*Loader now accepts the `ZONED [(precision[,scale])]` field type specification. For more information, see Section 11.2.3.
- Advanced Server now supports the `UTL_HTTP.WRITE_LINE` and `UTL_HTTP.WRITE_TEXT` procedures. For more information, see sections 7.18.25 and 7.18.27, respectively.
- Advanced Server now supports the `DBA_PROFILES` view. For more information, see Section 10.36.
- Advanced Server now supports the `REPLACE` clause in the `ALTER ROLE` and `ALTER USER` commands. For more information, see sections 3.3.4 and 3.3.10, respectively.
- Advanced Server now supports the `FREEZE` keyword in the EDB*Loader control file and on the command line. For more information, see sections 11.2.3 and 11.2.4, respectively.
- Advanced Server now supports the `STRICT`, `LEAKPROOF`, `COST`, `ROWS`, and `SET` keywords in the `CREATE PROCEDURE` and `CREATE FUNCTION` commands. For more information, see sections 3.3.23 and 3.3.18, respectively.

- Advanced Server now supports XA functions `xaoEnv` and `xaoSvcCtx` in the Open Client Library. For more information, see Section 9.4.6.

- Advanced Server now supports the `EDB_ATTR_EMPTY_STRINGS` environment attribute in the Open Client Library. For more information, see Section 9.4.2.1.

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`".

1.3 Configuration Parameters Compatible with Oracle Databases

EDB Postgres Advanced Server supports the development and execution of applications compatible with PostgreSQL and Oracle. Some system behaviors can be altered to act in a more PostgreSQL or in a more Oracle compliant manner; these behaviors are controlled by configuration parameters. Modifying the parameters in the `postgresql.conf` file changes the behavior for all databases in the cluster, while a user or group can `SET` the parameter value on the command line, effecting only their session. These parameters are:

- `edb_redwood_date` – Controls whether or not a time component is stored in `DATE` columns. For behavior compatible with Oracle databases, set `edb_redwood_date` to `TRUE`. See Section 1.3.1.
- `edb_redwood_raw_names` – Controls whether database object names appear in uppercase or lowercase letters when viewed from Oracle system catalogs. For behavior compatible with Oracle databases, `edb_redwood_raw_names` is set to its default value of `FALSE`. To view database object names as they are actually stored in the PostgreSQL system catalogs, set `edb_redwood_raw_names` to `TRUE`. See Section 1.3.2.
- `edb_redwood_strings` – Equates `NULL` to an empty string for purposes of string concatenation operations. For behavior compatible with Oracle databases, set `edb_redwood_strings` to `TRUE`. See Section 1.3.3.
- `edb_stmt_level_tx` – Isolates automatic rollback of an aborted SQL command to statement level rollback only – the entire, current transaction is not automatically rolled back as is the case for default PostgreSQL behavior. For behavior compatible with Oracle databases, set `edb_stmt_level_tx` to `TRUE`; however, use only when absolutely necessary. See Section 1.3.4.
- `oracle_home` – Point Advanced Server to the correct Oracle installation directory. See Section 1.3.5.

1.3.1 edb_redwood_date

When DATE appears as the data type of a column in the commands, it is translated to TIMESTAMP(0) at the time the table definition is stored in the data base if the configuration parameter edb_redwood_date is set to TRUE. Thus, a time component will also be stored in the column along with the date. This is consistent with Oracle's DATE data type.

If edb_redwood_date is set to FALSE the column's data type in a CREATE TABLE or ALTER TABLE command remains as a native PostgreSQL DATE data type and is stored as such in the database. The PostgreSQL DATE data type stores only the date without a time component in the column.

Regardless of the setting of edb_redwood_date, when DATE appears as a data type in any other context such as the data type of a variable in an SPL declaration section, or the data type of a formal parameter in an SPL procedure or SPL function, or the return type of an SPL function, it is always internally translated to a TIMESTAMP(0) and thus, can handle a time component if present.

See Section 3.2.4 for more information on date/time data types.

1.3.2 edb_redwood_raw_names

When edb_redwood_raw_names is set to its default value of FALSE, database object names such as table names, column names, trigger names, program names, user names, etc. appear in uppercase letters when viewed from Oracle catalogs (that is, system catalogs prefixed by ALL_, DBA_, or USER_. See Chapter 10 for a list of such catalogs). In addition, quotation marks enclose names that were created with enclosing quotation marks.

When edb_redwood_raw_names is set to TRUE, the database object names are displayed exactly as they are stored in the PostgreSQL system catalogs when viewed from the Oracle catalogs. Thus, names created without enclosing quotation marks appear in lowercase as expected in PostgreSQL. Names created with enclosing quotation marks appear exactly as they were created, but without the quotation marks.

For example, the following user name is created, and then a session is started with that user.

```
CREATE USER reduser IDENTIFIED BY password;
edb=# \c - reduser
Password for user reduser:
You are now connected to database "edb" as user "reduser".
```

When connected to the database as `reduser`, the following tables are created.

```
CREATE TABLE all_lower (col INTEGER);
CREATE TABLE ALL_UPPER (COL INTEGER);
CREATE TABLE "Mixed_Case" ("Col" INTEGER);
```

When viewed from the Oracle catalog, `USER_TABLES`, with `edb_redwood_raw_names` set to the default value `FALSE`, the names appear in uppercase except for the `Mixed_Case` name, which appears as created and also with enclosing quotation marks.

```
edb=> SELECT * FROM USER_TABLES;
 schema_name |  table_name   | tablespace_name | status | temporary
-------------+--------------+-----------------+--------+-----------
 REDUSER     | ALL_LOWER    |                 | VALID  | N
 REDUSER     | ALL_UPPER    |                 | VALID  | N
 REDUSER     | "Mixed_Case" |                 | VALID  | N
(3 rows)
```

When viewed with `edb_redwood_raw_names` set to `TRUE`, the names appear in lowercase except for the `Mixed_Case` name, which appears as created, but now without the enclosing quotation marks.

```
edb=> SET edb_redwood_raw_names TO true;
SET
edb=> SELECT * FROM USER_TABLES;
 schema_name | table_name | tablespace_name | status | temporary
-------------+------------+-----------------+--------+-----------
 reduser     | all_lower  |                 | VALID  | N
 reduser     | all_upper  |                 | VALID  | N
 reduser     | Mixed_Case |                 | VALID  | N
(3 rows)
```

These names now match the case when viewed from the PostgreSQL `pg_tables` catalog.

```
edb=> SELECT schemaname, tablename, tableowner FROM pg_tables WHERE
tableowner = 'reduser';
 schemaname | tablename  | tableowner
------------+------------+------------
 reduser    | all_lower  | reduser
 reduser    | all_upper  | reduser
 reduser    | Mixed_Case | reduser
(3 rows)
```

1.3.3 edb_redwood_strings

In Oracle, when a string is concatenated with a null variable or null column, the result is the original string; however, in PostgreSQL concatenation of a string with a null variable or null column gives a null result. If the `edb_redwood_strings` parameter is set to `TRUE`, the aforementioned concatenation operation results in the original string as done by Oracle. If `edb_redwood_strings` is set to `FALSE`, the native PostgreSQL behavior is maintained.

The following example illustrates the difference.

The sample application introduced in the next section contains a table of employees. This table has a column named `comm` that is null for most employees. The following query is run with `edb_redwood_string` set to `FALSE`. The concatenation of a null column with non-empty strings produces a final result of null, so only employees that have a commission appear in the query result. The output line for all other employees is null.

```
SET edb_redwood_strings TO off;

SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' ||
TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;

      EMPLOYEE COMPENSATION
----------------------------------

 ALLEN        1,600.00     300.00
 WARD         1,250.00     500.00

 MARTIN       1,250.00   1,400.00




 TURNER       1,500.00        .00




(14 rows)
```

The following is the same query executed when `edb_redwood_strings` is set to `TRUE`. Here, the value of a null column is treated as an empty string. The concatenation of an empty string with a non-empty string produces the non-empty string. This result is consistent with the results produced by Oracle for the same query.

```
SET edb_redwood_strings TO on;

SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' ||
TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;

      EMPLOYEE COMPENSATION
----------------------------------
 SMITH          800.00
 ALLEN        1,600.00     300.00
 WARD         1,250.00     500.00
 JONES        2,975.00
 MARTIN       1,250.00   1,400.00
 BLAKE        2,850.00
 CLARK        2,450.00
 SCOTT        3,000.00
 KING         5,000.00
 TURNER       1,500.00        .00
 ADAMS        1,100.00
 JAMES          950.00
 FORD         3,000.00
 MILLER       1,300.00
(14 rows)
```

1.3.4 edb_stmt_level_tx

In Oracle, when a runtime error occurs in a SQL command, all the updates on the database caused by that single command are rolled back. This is called *statement level transaction isolation*. For example, if a single `UPDATE` command successfully updates five rows, but an attempt to update a sixth row results in an exception, the updates to all six rows made by this `UPDATE` command are rolled back. The effects of prior SQL commands that have not yet been committed or rolled back are pending until a `COMMIT` or `ROLLBACK` command is executed.

In PostgreSQL, if an exception occurs while executing a SQL command, all the updates on the database since the start of the transaction are rolled back. In addition, the transaction is left in an aborted state and either a `COMMIT` or `ROLLBACK` command must be issued before another transaction can be started.

If `edb_stmt_level_tx` is set to `TRUE`, then an exception will not automatically roll back prior uncommitted database updates, emulating the Oracle behavior. If `edb_stmt_level_tx` is set to `FALSE`, then an exception will roll back uncommitted database updates.

Note: Use `edb_stmt_level_tx` set to `TRUE` only when absolutely necessary, as this may cause a negative performance impact.

The following example run in PSQL shows that when `edb_stmt_level_tx` is `FALSE`, the abort of the second `INSERT` command also rolls back the first `INSERT` command. Note that in PSQL, the command `\set AUTOCOMMIT off` must be issued, otherwise every statement commits automatically defeating the purpose of this demonstration of the effect of `edb_stmt_level_tx`.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO off;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);
ERROR:  insert or update on table "emp" violates foreign key constraint
"emp_ref_dept_fk"
DETAIL:  Key (deptno)=(0) is not present in table "dept".

COMMIT;
SELECT empno, ename, deptno FROM emp WHERE empno > 9000;

empno | ename | deptno
-------+-------+--------
(0 rows)
```

In the following example, with `edb_stmt_level_tx` set to `TRUE`, the first `INSERT` command has not been rolled back after the error on the second `INSERT` command. At this point, the first `INSERT` command can either be committed or rolled back.

```
\set AUTOCOMMIT off
```

```
SET edb_stmt_level_tx TO on;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);
ERROR:  insert or update on table "emp" violates foreign key constraint
"emp_ref_dept_fk"
DETAIL:  Key (deptno)=(0) is not present in table "dept".

SELECT empno, ename, deptno FROM emp WHERE empno > 9000;

empno | ename | deptno
-------+-------+--------
  9001 | JONES |     40
(1 row)

COMMIT;
```

A `ROLLBACK` command could have been issued instead of the `COMMIT` command in which case the insert of employee number `9001` would have been rolled back as well.

1.3.5 oracle_home

Before creating a link to an Oracle server, you must direct Advanced Server to the correct Oracle home directory. Set the `LD_LIBRARY_PATH` environment variable on Linux (or `PATH` on Windows) to the `lib` directory of the Oracle client installation directory.

For Windows only, you can instead 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.

The `LD_LIBRARY_PATH` environment variable on Linux (`PATH` environment variable or `oracle_home` configuration parameter on Windows) must be set properly each time you start Advanced Server.

For Windows only: 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`*.

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

1.4 About the Examples Used in this Guide

The examples shown in this guide are illustrated using the PSQL program. The prompt that normally appears when using PSQL is omitted in these examples to provide extra clarity for the point being demonstrated.

```
Examples and output from examples are shown in fixed-width, blue font on a
light blue background.
```

Also note the following points:

- During installation of the EDB Postgres Advanced Server the selection for configuration and defaults compatible with Oracle databases must be chosen in order to reproduce the same results as the examples shown in this guide. A default compatible configuration can be verified by issuing the following commands in PSQL and obtaining the same results as shown below.

```
SHOW edb_redwood_date;

 edb_redwood_date
------------------
 on

SHOW datestyle;

  DateStyle
--------------
 Redwood, DMY

SHOW edb_redwood_strings;

edb_redwood_strings
---------------------
 on
```

- The examples use the sample tables, `dept`, `emp`, and `jobhist`, created and loaded when Advanced Server is installed. The `emp` table is installed with triggers that must be disabled in order to reproduce the same results as shown in this guide. Log onto Advanced Server as the `enterprisedb` superuser and disable the triggers by issuing the following command.

```
ALTER TABLE emp DISABLE TRIGGER USER;
```

The triggers on the `emp` table can later be re-activated with the following command.

```
ALTER TABLE emp ENABLE TRIGGER USER;
```

2 SQL Tutorial

This section is an introduction to the SQL language for those new to relational database management systems. Basic operations such as creating, populating, querying, and updating tables are discussed along with examples.

More advanced concepts such as view, foreign keys, and transactions are discussed as well.

2.1 Getting Started

Advanced Server is a *relational database management system* (RDBMS). That means it is a system for managing data stored in *relations*. A relation is essentially a mathematical term for a *table*. The notion of storing data in tables is so commonplace today that it might seem inherently obvious, but there are a number of other ways of organizing databases. Files and directories on Unix-like operating systems form an example of a hierarchical database. A more modern development is the object-oriented database.

Each table is a named collection of *rows*. Each row of a given table has the same set of named *columns*, and each column is of a specific *data type*. Whereas columns have a fixed order in each row, it is important to remember that SQL does not guarantee the order of the rows within the table in any way (although they can be explicitly sorted for display).

Tables are grouped into *databases*, and a collection of databases managed by a single Advanced Server instance constitutes a database *cluster*.

2.1.1 Sample Database

Throughout this documentation we will be working with a sample database to help explain some basic to advanced level database concepts.

2.1.1.1 Sample Database Installation

When Advanced Server is installed a sample database named, `edb`, is automatically created. This sample database contains the tables and programs used throughout this document.

The tables and programs in the sample database can be re-created at any time by executing the script, `edb-sample.sql`, located in the `samples` subdirectory of the Advanced Server home directory.

This script does the following:

- Creates the sample tables and programs in the currently connected database
- Grants all permissions on the tables to the `PUBLIC` group

The tables and programs will be created in the first schema of the search path in which the current user has permission to create tables and procedures. You can display the search path by issuing the command:

```
SHOW SEARCH_PATH;
```

Altering the search path can be done using commands in PSQL.

2.1.1.2 Sample Database Description

The sample database represents employees in an organization.

It contains three types of records: employees, departments, and historical records of employees.

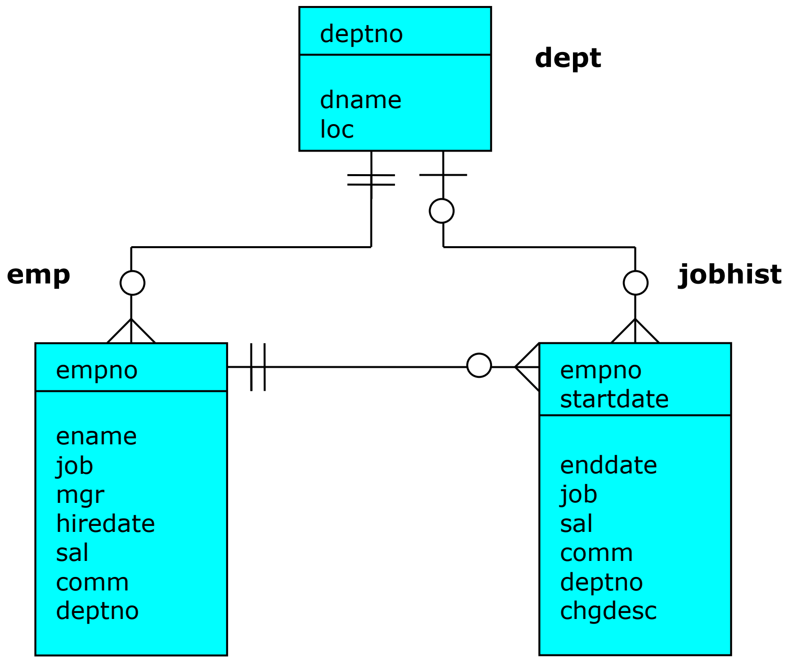
Each employee has an identification number, name, hire date, salary, and manager. Some employees earn a commission in addition to their salary. All employee-related information is stored in the `emp` table.

The sample company is regionally diverse, so the database keeps track of the location of the departments. Each company employee is assigned to a department. Each department is identified by a unique department number and a short name. Each department is associated with one location. All department-related information is stored in the `dept` table.

The company also tracks information about jobs held by the employees. Some employees have been with the company for a long time and have held different positions, received raises, switched departments, etc. When a change in employee status occurs, the company records the end date of the former position. A new job record is added with the start date and the new job title, department, salary, and the reason for the status change. All employee history is maintained in the `jobhist` table.

The following is an entity relationship diagram of the sample database tables.

Figure 1 Sample Database Tables



The following is the `edb-sample.sql` script.

```
--
--  Script that creates the 'sample' tables, views, procedures,
--  functions, triggers, etc.
--
--  Start new transaction - commit all or nothing
--
BEGIN;
/
--
--  Create and load tables used in the documentation examples.
--
--  Create the 'dept' 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)
);
--
--  Create the 'emp' table
--
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)
);
--
--  Create the 'jobhist' 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),
    CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
        REFERENCES emp(empno) ON DELETE CASCADE,
    CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
        REFERENCES dept (deptno) ON DELETE SET NULL,
    CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
);
--
--  Create the 'salesemp' view
--
CREATE OR REPLACE VIEW salesemp AS
    SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
--
--  Sequence to generate values for function 'new_empno'.
--
CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
```

```
--
--  Issue PUBLIC grants
--
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
GRANT ALL ON next_empno TO PUBLIC;
--
--  Load the 'dept' table
--
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');
--
--  Load the 'emp' table
--
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-
81',1600,300,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-
81',2975,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-
81',1250,1400,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-
81',2850,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-
81',2450,NULL,10);
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-
87',3000,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'17-NOV-
81',5000,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3000,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);
--
--  Load the 'jobhist' table
--
INSERT INTO jobhist VALUES (7369,'17-DEC-80',NULL,'CLERK',800,NULL,20,'New
Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,'New
Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,'New
Hire');
INSERT INTO jobhist VALUES (7566,'02-APR-81',NULL,'MANAGER',2975,NULL,20,'New
Hire');
INSERT INTO jobhist VALUES (7654,'28-SEP-
81',NULL,'SALESMAN',1250,1400,30,'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,'New
Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,'New
Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-
88','CLERK',1000,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-
89','CLERK',1040,NULL,20,'Raise');
INSERT INTO jobhist VALUES (7788,'05-MAY-
90',NULL,'ANALYST',3000,NULL,20,'Promoted to Analyst');
```

```
INSERT INTO jobhist VALUES (7839,'17-NOV-
81',NULL,'PRESIDENT',5000,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7844,'08-SEP-81',NULL,'SALESMAN',1500,0,30,'New
Hire');
INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,'New
Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-
83','CLERK',950,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7900,'15-JAN-
83',NULL,'CLERK',950,NULL,30,'Changed to Dept 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,'New
Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,'New
Hire');
--
--  Populate statistics table and view (pg_statistic/pg_stats)
--
ANALYZE dept;
ANALYZE emp;
ANALYZE jobhist;
--
--  Procedure that lists all employees' numbers and names
--  from the 'emp' table using a cursor.
--
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;
/
--
--  Procedure that selects an employee row given the employee
--  number and displays certain columns.
--
CREATE OR REPLACE PROCEDURE select_emp (
    p_empno         IN  NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_dname         dept.dname%TYPE;
    v_disp_date     VARCHAR2(10);
BEGIN
    SELECT ename, hiredate, sal, NVL(comm, 0), dname
        INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
        FROM emp e, dept d
        WHERE empno = p_empno
          AND e.deptno = d.deptno;
    v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
    DBMS_OUTPUT.PUT_LINE('Number    : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name      : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
    DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
    DBMS_OUTPUT.PUT_LINE('Department: ' || v_dname);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
/
--
--  Procedure that queries the 'emp' table based on
--  department number and employee number or name.  Returns
--  employee number and name as IN OUT parameters and job,
--  hire date, and salary as OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query (
    p_deptno        IN     NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
    p_job           OUT    VARCHAR2,
    p_hiredate      OUT    DATE,
    p_sal           OUT    NUMBER
)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
          AND (empno = p_empno
           OR  ename = UPPER(p_ename));
END;
/
--
--  Procedure to call 'emp_query_caller' with IN and IN OUT
--  parameters.  Displays the results received from IN OUT and
--  OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query_caller
IS
    v_deptno        NUMBER(2);
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 0;
    v_ename  := 'Martin';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee was selected');
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No employees were selected');
END;
/
--
--  Function to compute yearly compensation based on semimonthly
--  salary.
--
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;
/
--
--  Function that gets the next number from sequence, 'next_empno',
--  and ensures it is not already in use as an employee number.
--
CREATE OR REPLACE FUNCTION new_empno RETURN NUMBER
IS
    v_cnt           INTEGER := 1;
    v_new_empno     NUMBER;
BEGIN
    WHILE v_cnt > 0 LOOP
        SELECT next_empno.nextval INTO v_new_empno FROM dual;
        SELECT COUNT(*) INTO v_cnt FROM emp WHERE empno = v_new_empno;
    END LOOP;
    RETURN v_new_empno;
END;
/
--
--  EDB-SPL function that adds a new clerk to table 'emp'.  This function
--  uses package 'emp_admin'.
--
CREATE OR REPLACE FUNCTION hire_clerk (
    p_ename         VARCHAR2,
    p_deptno        NUMBER
) RETURN NUMBER
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_mgr           NUMBER(4);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_deptno        NUMBER(2);
BEGIN
    v_empno := new_empno;
    INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
        TRUNC(SYSDATE), 950.00, NULL, p_deptno);
    SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        FROM emp WHERE empno = v_empno;
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Manager    : ' || v_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        RETURN -1;
END;
/
--
--  PostgreSQL PL/pgSQL function that adds a new salesman
--  to table 'emp'.
--
CREATE OR REPLACE FUNCTION hire_salesman (
    p_ename         VARCHAR,
    p_sal           NUMERIC,
    p_comm          NUMERIC
) RETURNS NUMERIC
AS $$
DECLARE
    v_empno         NUMERIC(4);
    v_ename         VARCHAR(10);
    v_job           VARCHAR(9);
    v_mgr           NUMERIC(4);
    v_hiredate      DATE;
    v_sal           NUMERIC(7,2);
    v_comm          NUMERIC(7,2);
    v_deptno        NUMERIC(2);
BEGIN
    v_empno := new_empno();
    INSERT INTO emp VALUES (v_empno, p_ename, 'SALESMAN', 7698,
        CURRENT_DATE, p_sal, p_comm, 30);
    SELECT INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        empno, ename, job, mgr, hiredate, sal, comm, deptno
        FROM emp WHERE empno = v_empno;
    RAISE INFO 'Department : %', v_deptno;
    RAISE INFO 'Employee No: %', v_empno;
    RAISE INFO 'Name       : %', v_ename;
    RAISE INFO 'Job        : %', v_job;
    RAISE INFO 'Manager    : %', v_mgr;
    RAISE INFO 'Hire Date  : %', v_hiredate;
    RAISE INFO 'Salary     : %', v_sal;
    RAISE INFO 'Commission : %', v_comm;
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO 'The following is SQLERRM:';
        RAISE INFO '%', SQLERRM;
        RAISE INFO 'The following is SQLSTATE:';
        RAISE INFO '%', SQLSTATE;
        RETURN -1;
END;
$$ LANGUAGE 'plpgsql';
/
--
```

```
--  Rule to INSERT into view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_i AS ON INSERT TO salesemp
DO INSTEAD
    INSERT INTO emp VALUES (NEW.empno, NEW.ename, 'SALESMAN', 7698,
        NEW.hiredate, NEW.sal, NEW.comm, 30);
--
--  Rule to UPDATE view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_u AS ON UPDATE TO salesemp
DO INSTEAD
    UPDATE emp SET empno    = NEW.empno,
                   ename    = NEW.ename,
                   hiredate = NEW.hiredate,
                   sal      = NEW.sal,
                   comm     = NEW.comm
        WHERE empno = OLD.empno;
--
--  Rule to DELETE from view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_d AS ON DELETE TO salesemp
DO INSTEAD
    DELETE FROM emp WHERE empno = OLD.empno;
--
--  After statement-level trigger that displays a message after
--  an insert, update, or deletion to the 'emp' table.  One message
--  per SQL command is displayed.
--
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;
/
--
--  Before row-level trigger that displays employee number and
--  salary of an employee that is about to be added, updated,
--  or deleted in the 'emp' table.
--
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;
/
--
--  Package specification for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE emp_admin
IS
    FUNCTION get_dept_name (
        p_deptno        NUMBER
    ) RETURN VARCHAR2;
    FUNCTION update_emp_sal (
        p_empno         NUMBER,
        p_raise         NUMBER
    ) RETURN NUMBER;
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    );
    PROCEDURE fire_emp (
        p_empno         NUMBER
    );
END emp_admin;
/
--
--  Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
    --
    --  Function that queries the 'dept' table based on the department
    --  number and returns the corresponding department name.
    --
    FUNCTION get_dept_name (
        p_deptno        IN NUMBER
    ) RETURN VARCHAR2
    IS
        v_dname         VARCHAR2(14);
    BEGIN
        SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
        RETURN v_dname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
            RETURN '';
    END;
    --
    --  Function that updates an employee's salary based on the
    --  employee number and salary increment/decrement passed
    --  as IN parameters.  Upon successful completion the function
    --  returns the new updated salary.
    --
    FUNCTION update_emp_sal (
        p_empno         IN NUMBER,
        p_raise         IN NUMBER
    ) RETURN NUMBER
    IS
        v_sal           NUMBER := 0;
    BEGIN
        SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
        v_sal := v_sal + p_raise;
        UPDATE emp SET sal = v_sal WHERE empno = p_empno;
        RETURN v_sal;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
            RETURN -1;
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
            DBMS_OUTPUT.PUT_LINE(SQLERRM);
            DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
            DBMS_OUTPUT.PUT_LINE(SQLCODE);
            RETURN -1;
    END;
    --
    --  Procedure that inserts a new employee record into the 'emp' table.
    --
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    )
    AS
    BEGIN
        INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
            VALUES(p_empno, p_ename, p_job, p_sal,
                   p_hiredate, p_comm, p_mgr, p_deptno);
    END;
    --
    --  Procedure that deletes an employee record from the 'emp' table based
    --  on the employee number.
    --
    PROCEDURE fire_emp (
        p_empno         NUMBER
    )
    AS
    BEGIN
        DELETE FROM emp WHERE empno = p_empno;
    END;
END;
/
COMMIT;
```

2.1.2 Creating a New Table

A new table is created by specifying the table name, along with all column names and their types. The following is a simplified version of the `emp` sample table with just the minimal information needed to define a table.

```
CREATE TABLE emp (
    empno           NUMBER(4),
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2)
);
```

You can enter this into PSQL with line breaks. PSQL will recognize that the command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) may be used freely in SQL commands. That means you can type the command aligned differently than the above, or even all on one line. Two dashes ("`--`") introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

`VARCHAR2(10)` specifies a data type that can store arbitrary character strings up to 10 characters in length. `NUMBER(7,2)` is a fixed point number with precision 7 and scale 2. `NUMBER(4)` is an integer number with precision 4 and scale 0.

Advanced Server supports the usual SQL data types `INTEGER`, `SMALLINT`, `NUMBER`, `REAL`, `DOUBLE PRECISION`, `CHAR`, `VARCHAR2`, `DATE`, and `TIMESTAMP` as well as various synonyms for these types.

If you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE tablename;
```

2.1.3 Populating a Table With Rows

The `INSERT` statement is used to populate a table with rows:

```
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
```

Note that all data types use rather obvious input formats. Constants that are not simple numeric values usually must be surrounded by single quotes (`'`), as in the example. The `DATE` type is actually quite flexible in what it accepts, but for this tutorial we will stick to the unambiguous format shown here.

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO emp(empno,ename,job,mgr,hiredate,sal,comm,deptno)
    VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the commission is unknown:

```
INSERT INTO emp(empno,ename,job,mgr,hiredate,sal,deptno)
    VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,20);
```

Many developers consider explicitly listing the columns better style than relying on the order implicitly.

2.1.4 Querying a Table

To retrieve data from a table, the table is *queried*. An SQL `SELECT` statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). The following query lists all columns of all employees in the table in no particular order.

```
SELECT * FROM emp;
```

Here, "*" in the select list means all columns. The following is the output from this query.

```
 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)
```

You may specify any arbitrary expression in the select list. For example, you can do:

```
SELECT ename, sal, sal * 24 AS yearly_salary, deptno FROM emp;

 ename  |   sal   | yearly_salary | deptno
--------+---------+---------------+--------
 SMITH  |  800.00 |      19200.00 |     20
 ALLEN  | 1600.00 |      38400.00 |     30
 WARD   | 1250.00 |      30000.00 |     30
 JONES  | 2975.00 |      71400.00 |     20
 MARTIN | 1250.00 |      30000.00 |     30
 BLAKE  | 2850.00 |      68400.00 |     30
 CLARK  | 2450.00 |      58800.00 |     10
 SCOTT  | 3000.00 |      72000.00 |     20
 KING   | 5000.00 |     120000.00 |     10
 TURNER | 1500.00 |      36000.00 |     30
 ADAMS  | 1100.00 |      26400.00 |     20
 JAMES  |  950.00 |      22800.00 |     30
 FORD   | 3000.00 |      72000.00 |     20
 MILLER | 1300.00 |      31200.00 |     10
(14 rows)
```

Notice how the `AS` clause is used to re-label the output column. (The `AS` clause is optional.)

A query can be qualified by adding a WHERE clause that specifies which rows are wanted. The WHERE clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (AND, OR, and NOT) are allowed in the qualification. For example, the following retrieves the employees in department 20 with salaries over $1000.00:

```
SELECT ename, sal, deptno FROM emp WHERE deptno = 20 AND sal > 1000;

 ename |   sal   | deptno
-------+---------+--------
 JONES | 2975.00 |     20
 SCOTT | 3000.00 |     20
 ADAMS | 1100.00 |     20
 FORD  | 3000.00 |     20
(4 rows)
```

You can request that the results of a query be returned in sorted order:

```
SELECT ename, sal, deptno FROM emp ORDER BY ename;

 ename  |   sal   | deptno
--------+---------+--------
 ADAMS  | 1100.00 |     20
 ALLEN  | 1600.00 |     30
 BLAKE  | 2850.00 |     30
 CLARK  | 2450.00 |     10
 FORD   | 3000.00 |     20
 JAMES  |  950.00 |     30
 JONES  | 2975.00 |     20
 KING   | 5000.00 |     10
 MARTIN | 1250.00 |     30
 MILLER | 1300.00 |     10
 SCOTT  | 3000.00 |     20
 SMITH  |  800.00 |     20
 TURNER | 1500.00 |     30
 WARD   | 1250.00 |     30
(14 rows)
```

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT job FROM emp;

    job
-----------
 ANALYST
 CLERK
 MANAGER
 PRESIDENT
 SALESMAN
(5 rows)
```

The following section shows how to obtain rows from more than one table in a single query.

2.1.5 Joins Between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a *join* query. For example, say you wish to list all the employee records together with the name and location of the associated department. To do that, we need to compare the `deptno` column of each row of the `emp` table with the `deptno` column of all rows in the `dept` table, and select the pairs of rows where these values match. This would be accomplished by the following query:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM emp, dept
WHERE emp.deptno = dept.deptno;

 ename  |   sal   | deptno |   dname    |   loc
--------+---------+--------+------------+----------
 MILLER | 1300.00 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 2450.00 |     10 | ACCOUNTING | NEW YORK
 KING   | 5000.00 |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 3000.00 |     20 | RESEARCH   | DALLAS
 JONES  | 2975.00 |     20 | RESEARCH   | DALLAS
 SMITH  |  800.00 |     20 | RESEARCH   | DALLAS
 ADAMS  | 1100.00 |     20 | RESEARCH   | DALLAS
 FORD   | 3000.00 |     20 | RESEARCH   | DALLAS
 WARD   | 1250.00 |     30 | SALES      | CHICAGO
 TURNER | 1500.00 |     30 | SALES      | CHICAGO
 ALLEN  | 1600.00 |     30 | SALES      | CHICAGO
 BLAKE  | 2850.00 |     30 | SALES      | CHICAGO
 MARTIN | 1250.00 |     30 | SALES      | CHICAGO
 JAMES  |  950.00 |     30 | SALES      | CHICAGO
(14 rows)
```

Observe two things about the result set:

- There is no result row for department 40. This is because there is no matching entry in the `emp` table for department 40, so the join ignores the unmatched rows in the `dept` table. Shortly we will see how this can be fixed.
- It is more desirable to list the output columns qualified by table name rather than using * or leaving out the qualification as follows:

```
SELECT ename, sal, dept.deptno, dname, loc FROM emp, dept WHERE emp.deptno =
dept.deptno;
```

Since all the columns had different names (except for `deptno` which therefore must be qualified), the parser automatically found out which table they belong to, but it is good style to fully qualify column names in join queries:

Join queries of the kind seen thus far can also be written in this alternative form:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM emp INNER
JOIN dept ON emp.deptno = dept.deptno;
```

This syntax is not as commonly used as the one above, but we show it here to help you understand the following topics.

You will notice that in all the above results for joins no employees were returned that belonged to department 40 and as a consequence, the record for department 40 never appears. Now we will figure out how we can get the department 40 record in the results despite the fact that there are no matching employees. What we want the query to do is to scan the `dept` table and for each row to find the matching `emp` row. If no matching row is found we want some "empty" values to be substituted for the `emp` table's columns. This kind of query is called an *outer join*. (The joins we have seen so far are *inner joins*.) The command looks like this:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM dept LEFT
OUTER JOIN emp ON emp.deptno = dept.deptno;

 ename  |   sal    | deptno |   dname    |   loc
--------+----------+--------+------------+----------
 MILLER | 1300.00 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 2450.00 |     10 | ACCOUNTING | NEW YORK
 KING   | 5000.00 |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 3000.00 |     20 | RESEARCH   | DALLAS
 JONES  | 2975.00 |     20 | RESEARCH   | DALLAS
 SMITH  |  800.00 |     20 | RESEARCH   | DALLAS
 ADAMS  | 1100.00 |     20 | RESEARCH   | DALLAS
 FORD   | 3000.00 |     20 | RESEARCH   | DALLAS
 WARD   | 1250.00 |     30 | SALES      | CHICAGO
 TURNER | 1500.00 |     30 | SALES      | CHICAGO
 ALLEN  | 1600.00 |     30 | SALES      | CHICAGO
 BLAKE  | 2850.00 |     30 | SALES      | CHICAGO
 MARTIN | 1250.00 |     30 | SALES      | CHICAGO
 JAMES  |  950.00 |     30 | SALES      | CHICAGO
        |          |     40 | OPERATIONS | BOSTON
(15 rows)
```

This query is called a *left outer join* because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When a left-table row is selected for which there is no right-table match, empty (`NULL`) values are substituted for the right-table columns.

An alternative syntax for an outer join is to use the outer join operator, "(+)", in the join condition within the `WHERE` clause. The outer join operator is placed after the column name of the table for which null values should be substituted for unmatched rows. So for all the rows in the `dept` table that have no matching rows in the `emp` table, Advanced Server returns null for any select list expressions containing columns of `emp`. Hence the above example could be rewritten as:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM dept, emp
WHERE emp.deptno(+) = dept.deptno;
```

```
 ename  |   sal    | deptno |   dname    |   loc
--------+---------+--------+------------+----------
 MILLER | 1300.00 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 2450.00 |     10 | ACCOUNTING | NEW YORK
 KING   | 5000.00 |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 3000.00 |     20 | RESEARCH   | DALLAS
 JONES  | 2975.00 |     20 | RESEARCH   | DALLAS
 SMITH  |  800.00 |     20 | RESEARCH   | DALLAS
 ADAMS  | 1100.00 |     20 | RESEARCH   | DALLAS
 FORD   | 3000.00 |     20 | RESEARCH   | DALLAS
 WARD   | 1250.00 |     30 | SALES      | CHICAGO
 TURNER | 1500.00 |     30 | SALES      | CHICAGO
 ALLEN  | 1600.00 |     30 | SALES      | CHICAGO
 BLAKE  | 2850.00 |     30 | SALES      | CHICAGO
 MARTIN | 1250.00 |     30 | SALES      | CHICAGO
 JAMES  |  950.00 |     30 | SALES      | CHICAGO
        |         |     40 | OPERATIONS | BOSTON
(15 rows)
```

We can also join a table against itself. This is called a *self join*. As an example, suppose we wish to find the name of each employee along with the name of that employee's manager. So we need to compare the `mgr` column of each `emp` row to the `empno` column of all other `emp` rows.

```
SELECT e1.ename || ' works for ' || e2.ename AS "Employees and their
Managers" FROM emp e1, emp e2 WHERE e1.mgr = e2.empno;

 Employees and their Managers
------------------------------
 FORD works for JONES
 SCOTT works for JONES
 WARD works for BLAKE
 TURNER works for BLAKE
 MARTIN works for BLAKE
 JAMES works for BLAKE
 ALLEN works for BLAKE
 MILLER works for CLARK
 ADAMS works for SCOTT
 CLARK works for KING
 BLAKE works for KING
 JONES works for KING
 SMITH works for FORD
(13 rows)
```

Here, the `emp` table has been re-labeled as `e1` to represent the employee row in the select list and in the join condition, and also as `e2` to represent the matching employee row acting as manager in the select list and in the join condition. These kinds of aliases can be used in other queries to save some typing, for example:

```
SELECT e.ename, e.mgr, d.deptno, d.dname, d.loc FROM emp e, dept d WHERE
e.deptno = d.deptno;

 ename  | mgr  | deptno |   dname    |   loc
--------+------+--------+------------+----------
 MILLER | 7782 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 7839 |     10 | ACCOUNTING | NEW YORK
 KING   |      |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 7566 |     20 | RESEARCH   | DALLAS
 JONES  | 7839 |     20 | RESEARCH   | DALLAS
 SMITH  | 7902 |     20 | RESEARCH   | DALLAS
 ADAMS  | 7788 |     20 | RESEARCH   | DALLAS
 FORD   | 7566 |     20 | RESEARCH   | DALLAS
 WARD   | 7698 |     30 | SALES      | CHICAGO
 TURNER | 7698 |     30 | SALES      | CHICAGO
 ALLEN  | 7698 |     30 | SALES      | CHICAGO
 BLAKE  | 7839 |     30 | SALES      | CHICAGO
 MARTIN | 7698 |     30 | SALES      | CHICAGO
 JAMES  | 7698 |     30 | SALES      | CHICAGO
(14 rows)
```

This style of abbreviating will be encountered quite frequently.

2.1.6 Aggregate Functions

Like most other relational database products, Advanced Server supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the `COUNT`, `SUM`, `AVG` (average), `MAX` (maximum), and `MIN` (minimum) over a set of rows.

As an example, the highest and lowest salaries can be found with the following query:

```
SELECT MAX(sal) highest_salary, MIN(sal) lowest_salary FROM emp;

 highest_salary | lowest_salary
----------------+---------------
        5000.00 |        800.00
(1 row)
```

If we wanted to find the employee with the largest salary, we may be tempted to try:

```
SELECT ename FROM emp WHERE sal = MAX(sal);

ERROR:  aggregates not allowed in WHERE clause
```

This does not work because the aggregate function, `MAX`, cannot be used in the `WHERE` clause. This restriction exists because the `WHERE` clause determines the rows that will go into the aggregation stage so it has to be evaluated before aggregate functions are computed. However, the query can be restated to accomplish the intended result by using a *subquery*:

```
SELECT ename FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);

 ename
-------
 KING
(1 row)
```

The subquery is an independent computation that obtains its own result separately from the outer query.

Aggregates are also very useful in combination with the `GROUP BY` clause. For example, the following query gets the highest salary in each department.

```
SELECT deptno, MAX(sal) FROM emp GROUP BY deptno;

 deptno |   max
--------+---------
     10 | 5000.00
     20 | 3000.00
     30 | 2850.00
(3 rows)
```

This query produces one output row per department. Each aggregate result is computed over the rows matching that department. These grouped rows can be filtered using the `HAVING` clause.

```
SELECT deptno, MAX(sal) FROM emp GROUP BY deptno HAVING AVG(sal) > 2000;

 deptno |   max
--------+---------
     10 | 5000.00
     20 | 3000.00
(2 rows)
```

This query gives the same results for only those departments that have an average salary greater than 2000.

Finally, the following query takes into account only the highest paid employees who are analysts in each department.

```
SELECT deptno, MAX(sal) FROM emp WHERE job = 'ANALYST' GROUP BY deptno HAVING
AVG(sal) > 2000;

 deptno |   max
--------+---------
     20 | 3000.00
(1 row)
```

There is a subtle distinction between the `WHERE` and `HAVING` clauses. The `WHERE` clause filters out rows before grouping occurs and aggregate functions are applied. The `HAVING` clause applies filters on the results after rows have been grouped and aggregate functions have been computed for each group.

So in the previous example, only employees who are analysts are considered. From this subset, the employees are grouped by department and only those groups where the average salary of analysts in the group is greater than 2000 are in the final result. This is true of only the group for department 20 and the maximum analyst salary in department 20 is 3000.00.

2.1.7 Updates

The column values of existing rows can be changed using the `UPDATE` command. For example, the following sequence of commands shows the before and after results of giving everyone who is a manager a 10% raise:

```
SELECT ename, sal FROM emp WHERE job = 'MANAGER';

 ename |   sal
-------+---------
 JONES | 2975.00
 BLAKE | 2850.00
 CLARK | 2450.00
(3 rows)

UPDATE emp SET sal = sal * 1.1 WHERE job = 'MANAGER';

SELECT ename, sal FROM emp WHERE job = 'MANAGER';

 ename |   sal
-------+---------
 JONES | 3272.50
 BLAKE | 3135.00
 CLARK | 2695.00
(3 rows)
```

2.1.8 Deletions

Rows can be removed from a table using the `DELETE` command. For example, the following sequence of commands shows the before and after results of deleting all employees in department `20`.

```
SELECT ename, deptno FROM emp;

 ename  | deptno
--------+--------
 SMITH  |     20
 ALLEN  |     30
 WARD   |     30
 JONES  |     20
 MARTIN |     30
 BLAKE  |     30
 CLARK  |     10
 SCOTT  |     20
 KING   |     10
 TURNER |     30
 ADAMS  |     20
 JAMES  |     30
 FORD   |     20
 MILLER |     10
(14 rows)

DELETE FROM emp WHERE deptno = 20;

SELECT ename, deptno FROM emp;
 ename  | deptno
--------+--------
 ALLEN  |     30
 WARD   |     30
 MARTIN |     30
 BLAKE  |     30
 CLARK  |     10
 KING   |     10
 TURNER |     30
 JAMES  |     30
 MILLER |     10
(9 rows)
```

Be extremely careful of giving a `DELETE` command without a `WHERE` clause such as the following:

```
DELETE FROM tablename;
```

This statement will remove all rows from the given table, leaving it completely empty. The system will not request confirmation before doing this.

2.2 Advanced Concepts

The previous section discussed the basics of using SQL to store and access your data in Advanced Server. This section discusses more advanced SQL features that may simplify management and prevent loss or corruption of your data.

2.2.1 Views

Consider the following `SELECT` command.

```
SELECT ename, sal, sal * 24 AS yearly_salary, deptno FROM emp;

 ename  |   sal   | yearly_salary | deptno
--------+---------+---------------+--------
 SMITH  |  800.00 |      19200.00 |     20
 ALLEN  | 1600.00 |      38400.00 |     30
 WARD   | 1250.00 |      30000.00 |     30
 JONES  | 2975.00 |      71400.00 |     20
 MARTIN | 1250.00 |      30000.00 |     30
 BLAKE  | 2850.00 |      68400.00 |     30
 CLARK  | 2450.00 |      58800.00 |     10
 SCOTT  | 3000.00 |      72000.00 |     20
 KING   | 5000.00 |     120000.00 |     10
 TURNER | 1500.00 |      36000.00 |     30
 ADAMS  | 1100.00 |      26400.00 |     20
 JAMES  |  950.00 |      22800.00 |     30
 FORD   | 3000.00 |      72000.00 |     20
 MILLER | 1300.00 |      31200.00 |     10
(14 rows)
```

If this is a query that is used repeatedly, a shorthand method of reusing this query without re-typing the entire `SELECT` command each time is to create a *view* as shown below.

```
CREATE VIEW employee_pay AS SELECT ename, sal, sal * 24 AS yearly_salary,
deptno FROM emp;
```

The view name, `employee_pay`, can now be used like an ordinary table name to perform the query.

```
SELECT * FROM employee_pay;

 ename  |   sal   | yearly_salary | deptno
--------+---------+---------------+--------
 SMITH  |  800.00 |      19200.00 |     20
 ALLEN  | 1600.00 |      38400.00 |     30
 WARD   | 1250.00 |      30000.00 |     30
 JONES  | 2975.00 |      71400.00 |     20
 MARTIN | 1250.00 |      30000.00 |     30
 BLAKE  | 2850.00 |      68400.00 |     30
 CLARK  | 2450.00 |      58800.00 |     10
 SCOTT  | 3000.00 |      72000.00 |     20
 KING   | 5000.00 |     120000.00 |     10
 TURNER | 1500.00 |      36000.00 |     30
 ADAMS  | 1100.00 |      26400.00 |     20
 JAMES  |  950.00 |      22800.00 |     30
 FORD   | 3000.00 |      72000.00 |     20
 MILLER | 1300.00 |      31200.00 |     10
(14 rows)
```

Making liberal use of views is a key aspect of good SQL database design. Views provide a consistent interface that encapsulate details of the structure of your tables which may change as your application evolves.

Views can be used in almost any place a real table can be used. Building views upon other views is not uncommon.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

2.2.2 Foreign Keys

Suppose you want to make sure all employees belong to a valid department. This is called maintaining the *referential integrity* of your data. In simplistic database systems this would be implemented (if at all) by first looking at the `dept` table to check if a matching record exists, and then inserting or rejecting the new employee record. This approach has a number of problems and is very inconvenient. Advanced Server can make it easier for you.

A modified version of the `emp` table presented in Section 2.1.2 is shown in this section with the addition of a foreign key constraint. The modified `emp` table looks like the following:

```
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)
);
```

If an attempt is made to issue the following `INSERT` command in the sample `emp` table, the foreign key constraint, `emp_ref_dept_fk`, ensures that department `50` exists in the `dept` table. Since it does not, the command is rejected.

```
INSERT INTO emp VALUES (8000,'JONES','CLERK',7902,'17-AUG-07',1200,NULL,50);

ERROR:  insert or update on table "emp" violates foreign key constraint
"emp_ref_dept_fk"
DETAIL:  Key (deptno)=(50) is not present in table "dept".
```

The behavior of foreign keys can be finely tuned to your application. Making correct use of foreign keys will definitely improve the quality of your database applications, so you are strongly encouraged to learn more about them.

2.2.3 The ROWNUM Pseudo-Column

`ROWNUM` is a pseudo-column that is assigned an incremental, unique integer value for each row based on the order the rows were retrieved from a query. Therefore, the first row retrieved will have `ROWNUM` of `1`; the second row will have `ROWNUM` of `2` and so on.

This feature can be used to limit the number of rows retrieved by a query. This is demonstrated in the following example:

```
SELECT empno, ename, job FROM emp WHERE ROWNUM < 5;

 empno | ename |    job
-------+-------+----------
  7369 | SMITH | CLERK
  7499 | ALLEN | SALESMAN
  7521 | WARD  | SALESMAN
  7566 | JONES | MANAGER
(4 rows)
```

The `ROWNUM` value is assigned to each row before any sorting of the result set takes place. Thus, the result set is returned in the order given by the `ORDER BY` clause, but the `ROWNUM` values may not necessarily be in ascending order as shown in the following example:

```
SELECT ROWNUM, empno, ename, job FROM emp WHERE ROWNUM < 5 ORDER BY ename;

 rownum | empno | ename |    job
--------+-------+-------+----------
      2 |  7499 | ALLEN | SALESMAN
      4 |  7566 | JONES | MANAGER
      1 |  7369 | SMITH | CLERK
      3 |  7521 | WARD  | SALESMAN
(4 rows)
```

The following example shows how a sequence number can be added to every row in the `jobhist` table. First a new column named, `seqno`, is added to the table and then `seqno` is set to `ROWNUM` in the `UPDATE` command.

```
ALTER TABLE jobhist ADD seqno NUMBER(3);
UPDATE jobhist SET seqno = ROWNUM;
```

The following `SELECT` command shows the new `seqno` values.

```
SELECT seqno, empno, TO_CHAR(startdate,'DD-MON-YY') AS start, job FROM
jobhist;

 seqno | empno |   start   |    job
-------+-------+-----------+-----------
     1 |  7369 | 17-DEC-80 | CLERK
     2 |  7499 | 20-FEB-81 | SALESMAN
     3 |  7521 | 22-FEB-81 | SALESMAN
     4 |  7566 | 02-APR-81 | MANAGER
     5 |  7654 | 28-SEP-81 | SALESMAN
     6 |  7698 | 01-MAY-81 | MANAGER
     7 |  7782 | 09-JUN-81 | MANAGER
     8 |  7788 | 19-APR-87 | CLERK
     9 |  7788 | 13-APR-88 | CLERK
    10 |  7788 | 05-MAY-90 | ANALYST
    11 |  7839 | 17-NOV-81 | PRESIDENT
    12 |  7844 | 08-SEP-81 | SALESMAN
    13 |  7876 | 23-MAY-87 | CLERK
    14 |  7900 | 03-DEC-81 | CLERK
    15 |  7900 | 15-JAN-83 | CLERK
    16 |  7902 | 03-DEC-81 | ANALYST
    17 |  7934 | 23-JAN-82 | CLERK
(17 rows)
```

2.2.4 Synonyms

A *synonym* is an identifier that can be used to reference another database object in a SQL statement. A synonym is useful in cases where a database object would normally require full qualification by schema name to be properly referenced in a SQL statement. A synonym defined for that object simplifies the reference to a single, unqualified name.

Advanced Server supports synonyms for:

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

Neither the referenced schema or referenced object must exist at the time that you create the synonym; a synonym may refer to a non-existent object or schema. A synonym will become invalid if you drop the referenced object or schema. You must explicitly drop a synonym to remove it.

As with any other schema object, Advanced Server uses the search path to resolve unqualified synonym names. If you have two synonyms with the same name, an unqualified reference to a synonym will resolve to the first synonym with the given name in the search path. If `public` is in your search path, you can refer to a synonym in that schema without qualifying that name.

When Advanced Server executes an SQL command, the privileges of the current user are checked against the synonym's underlying database object; if the user does not have the proper permissions for that object, the SQL command will fail.

Creating a Synonym

Use the `CREATE SYNONYM` command to create a synonym. The syntax is:

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

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 object resides in.

dblink_name

> *dblink_name* specifies the name of the database link through which a target object may be 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. Compatible with Oracle databases, the CREATE PUBLIC SYNONYM command 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;
```

The following example creates a synonym named personnel that refers to the enterprisedb.emp table.

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

Unless the synonym is schema qualified in the `CREATE SYNONYM` command, it will be created in the first existing schema in your search path. You can view your search path by executing the following command:

```
SHOW SEARCH_PATH;

      search_path
-----------------------
 development,accounting
(1 row)
```

In our example, if a schema named `development` does not exist, the synonym will be created in the schema named `accounting`.

Now, the `emp` table in the `enterprisedb` schema can be referenced in any SQL statement (DDL or DML), by using the synonym, `personnel`:

```
INSERT INTO personnel VALUES (8142,'ANDERSON','CLERK',7902,'17-DEC-06',1300,NULL,20);

SELECT * FROM personnel;

 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
  8142 | ANDERSON | CLERK     | 7902 | 17-DEC-06 00:00:00 | 1300.00 |         |     20
(15 rows)
```

Deleting a Synonym

To delete a synonym, use the command, `DROP SYNONYM`. The syntax is:

```
DROP [PUBLIC] SYNONYM [schema.] syn_name
```

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 in which the synonym resides.

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. Compatible with Oracle databases, the DROP PUBLIC SYNONYM command drops a synonym that resides in the public schema:

```
DROP PUBLIC SYNONYM syn_name;
```

The following example drops the synonym, personnel:

```
DROP SYNONYM personnel;
```

2.2.5 Hierarchical Queries

A *hierarchical query* is a type of query that returns the rows of the result set in a hierarchical order based upon data forming a parent-child relationship. A hierarchy is typically represented by an inverted tree structure. The tree is comprised of interconnected *nodes*. Each node may be connected to none, one, or multiple *child* nodes. Each node is connected to one *parent* node except for the top node which has no parent. This node is the *root* node. Each tree has exactly one root node. Nodes that don't have any children are called *leaf* nodes. A tree always has at least one leaf node - e.g., the trivial case where the tree is comprised of a single node. In this case it is both the root and the leaf.

In a hierarchical query the rows of the result set represent the nodes of one or more trees.

Note: It is possible that a single, given row may appear in more than one tree and thus appear more than once in the result set.

The hierarchical relationship in a query is described by the `CONNECT BY` clause which forms the basis of the order in which rows are returned in the result set. The context of where the `CONNECT BY` clause and its associated optional clauses appear in the `SELECT` command is shown below.

```
SELECT select_list FROM table_expression [ WHERE ...]
  [ START WITH start_expression ]
    CONNECT BY { PRIOR parent_expr = child_expr |
      child_expr = PRIOR parent_expr }
  [ ORDER SIBLINGS BY column1 [ ASC | DESC ]
      [, column2 [ ASC | DESC ] ] ...
  [ GROUP BY ...]
  [ HAVING ...]
  [ other ...]
```

`select_list` is one or more expressions that comprise the fields of the result set. *`table_expression`* is one or more tables or views from which the rows of the result set originate. *`other`* is any additional legal `SELECT` command clauses. The clauses pertinent to hierarchical queries, `START WITH`, `CONNECT BY`, and `ORDER SIBLINGS BY` are described in the following sections.

Note: At this time, Advanced Server does not support the use of `AND` (or other operators) in the `CONNECT BY` clause.

2.2.5.1 Defining the Parent/Child Relationship

For any given row, its parent and its children are determined by the `CONNECT BY` clause. The `CONNECT BY` clause must consist of two expressions compared with the equals (=) operator. In addition, one of these two expressions must be preceded by the keyword, `PRIOR`.

For any given row, to determine its children:

1. Evaluate *`parent_expr`* on the given row
2. Evaluate *`child_expr`* on any other row resulting from the evaluation of *`table_expression`*
3. If *`parent_expr = child_expr`*, then this row is a child node of the given parent row
4. Repeat the process for all remaining rows in *`table_expression`*. All rows that satisfy the equation in step 3 are the children nodes of the given parent row.

Note: The evaluation process to determine if a row is a child node occurs on every row returned by *`table_expression`* before the `WHERE` clause is applied to *`table_expression`*.

By iteratively repeating this process treating each child node found in the prior steps as a parent, an inverted tree of nodes is constructed. The process is complete when the final set of child nodes has no children of their own - these are the leaf nodes.

A `SELECT` command that includes a `CONNECT BY` clause typically includes the `START WITH` clause. The `START WITH` clause determines the rows that are to be the root nodes - i.e., the rows that are the initial parent nodes upon which the algorithm described previously is to be applied. This is further explained in the following section.

2.2.5.2 Selecting the Root Nodes

The `START WITH` clause is used to determine the row(s) selected by *`table_expression`* that are to be used as the root nodes. All rows selected by *`table_expression`* where *`start_expression`* evaluates to true become a root node of a tree. Thus, the number of potential trees in the result set is equal to the number of root nodes. As a consequence, if the `START WITH` clause is omitted, then every row returned by *`table_expression`* is a root of its own tree.

2.2.5.3 Organization Tree in the Sample Application

Consider the `emp` table of the sample application. The rows of the `emp` table form a hierarchy based upon the `mgr` column which contains the employee number of the employee's manager. Each employee has at most, one manager. `KING` is the president of the company so he has no manager, therefore KING's `mgr` column is null. Also, it is possible for an employee to act as a manager for more than one employee. This relationship forms a typical, tree-structured, hierarchical organization chart as illustrated below.

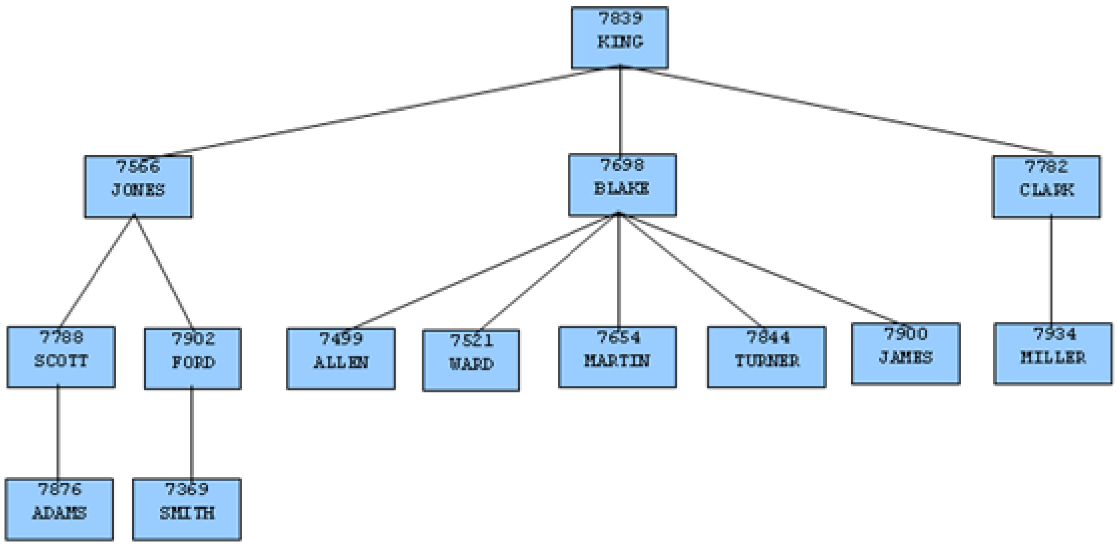


Figure 2 Employee Organization Hierarchy

To form a hierarchical query based upon this relationship, the `SELECT` command includes the clause, `CONNECT BY PRIOR empno = mgr`. For example, given the company president, `KING`, with employee number `7839`, any employee whose `mgr` column is `7839` reports directly to `KING` which is true for `JONES`, `BLAKE`, and `CLARK` (these are the child nodes of `KING`). Similarly, for employee, `JONES`, any other employee with `mgr` column equal to `7566` is a child node of `JONES` - these are `SCOTT` and `FORD` in this example.

The top of the organization chart is `KING` so there is one root node in this tree. The `START WITH mgr IS NULL` clause selects only `KING` as the initial root node.

The complete `SELECT` command is shown below.

```
SELECT ename, empno, mgr
FROM emp
START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr;
```

The rows in the query output traverse each branch from the root to leaf moving in a top-to-bottom, left-to-right order. Below is the output from this query.

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

2.2.5.4 Node Level

`LEVEL` is a pseudo-column that can be used wherever a column can appear in the `SELECT` command. For each row in the result set, `LEVEL` returns a non-zero integer value designating the depth in the hierarchy of the node represented by this row. The `LEVEL` for root nodes is 1. The `LEVEL` for direct children of root nodes is 2, and so on.

The following query is a modification of the previous query with the addition of the `LEVEL` pseudo-column. In addition, using the `LEVEL` value, the employee names are indented to further emphasize the depth in the hierarchy of each row.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr
FROM emp START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr;
```

The output from this query follows.

```
 level |   employee    | empno | mgr
-------+---------------+-------+------
     1 | KING          |  7839 |
     2 |   JONES       |  7566 | 7839
     3 |     SCOTT     |  7788 | 7566
     4 |       ADAMS   |  7876 | 7788
     3 |     FORD      |  7902 | 7566
     4 |       SMITH   |  7369 | 7902
     2 |   BLAKE       |  7698 | 7839
     3 |     ALLEN     |  7499 | 7698
     3 |     WARD      |  7521 | 7698
     3 |     MARTIN    |  7654 | 7698
     3 |     TURNER    |  7844 | 7698
     3 |     JAMES     |  7900 | 7698
     2 |   CLARK       |  7782 | 7839
     3 |     MILLER    |  7934 | 7782
(14 rows)
```

Nodes that share a common parent and are at the same level are called *siblings*. For example in the above output, employees `ALLEN`, `WARD`, `MARTIN`, `TURNER`, and `JAMES` are siblings since they are all at level three with parent, `BLAKE`. `JONES`, `BLAKE`, and `CLARK` are siblings since they are at level two and `KING` is their common parent.

2.2.5.5 Ordering the Siblings

The result set can be ordered so the siblings appear in ascending or descending order by selected column value(s) using the `ORDER SIBLINGS BY` clause. This is a special case of the `ORDER BY` clause that can be used only with hierarchical queries.

The previous query is further modified with the addition of `ORDER SIBLINGS BY` ename `ASC`.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr
FROM emp START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The output from the prior query is now modified so the siblings appear in ascending order by name. Siblings `BLAKE`, `CLARK`, and `JONES` are now alphabetically arranged under `KING`. Siblings `ALLEN`, `JAMES`, `MARTIN`, `TURNER`, and `WARD` are alphabetically arranged under `BLAKE`, and so on.

```
 level |  employee    | empno | mgr
-------+-------------+-------+------
     1 | KING         |  7839 |
     2 |   BLAKE      |  7698 | 7839
     3 |     ALLEN    |  7499 | 7698
     3 |     JAMES    |  7900 | 7698
     3 |     MARTIN   |  7654 | 7698
     3 |     TURNER   |  7844 | 7698
     3 |     WARD     |  7521 | 7698
     2 |   CLARK      |  7782 | 7839
     3 |     MILLER   |  7934 | 7782
     2 |   JONES      |  7566 | 7839
     3 |     FORD     |  7902 | 7566
     4 |       SMITH  |  7369 | 7902
     3 |     SCOTT    |  7788 | 7566
     4 |       ADAMS  |  7876 | 7788
(14 rows)
```

This final example adds the `WHERE` clause and starts with three root nodes. After the node tree is constructed, the `WHERE` clause filters out rows in the tree to form the result set.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr
FROM emp WHERE mgr IN (7839, 7782, 7902, 7788)
START WITH ename IN ('BLAKE','CLARK','JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The output from the query shows three root nodes (level one) - `BLAKE`, `CLARK`, and `JONES`. In addition, rows that do not satisfy the `WHERE` clause have been eliminated from the output.

```
 level | employee   | empno | mgr
-------+-----------+-------+------
     1 | BLAKE      |  7698 | 7839
     1 | CLARK      |   7782 | 7839
     2 |   MILLER   |   7934 | 7782
     1 | JONES      |   7566 | 7839
     3 |     SMITH  |   7369 | 7902
     3 |     ADAMS  |   7876 | 7788
(6 rows)
```

2.2.5.6 Retrieving the Root Node with CONNECT_BY_ROOT

`CONNECT_BY_ROOT` is a unary operator that can be used to qualify a column in order to return the column's value of the row considered to be the root node in relation to the current row.

Note: A *unary operator* operates on a single operand, which in the case of `CONNECT_BY_ROOT`, is the column name following the `CONNECT_BY_ROOT` keyword.

In the context of the `SELECT` list, the `CONNECT_BY_ROOT` operator is shown by the following.

```
SELECT [... ,] CONNECT_BY_ROOT column [, ...]
  FROM table_expression ...
```

The following are some points to note about the `CONNECT_BY_ROOT` operator.

- The `CONNECT_BY_ROOT` operator can be used in the `SELECT` list, the `WHERE` clause, the `GROUP BY` clause, the `HAVING` clause, the `ORDER BY` clause, and the `ORDER SIBLINGS BY` clause as long as the `SELECT` command is for a hierarchical query.
- The `CONNECT_BY_ROOT` operator cannot be used in the `CONNECT BY` clause or the `START WITH` clause of the hierarchical query.
- It is possible to apply `CONNECT_BY_ROOT` to an expression involving a column, but to do so, the expression must be enclosed within parentheses.

The following query shows the use of the `CONNECT_BY_ROOT` operator to return the employee number and employee name of the root node for each employee listed in the result set based on trees starting with employees `BLAKE`, `CLARK`, and `JONES`.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
CONNECT_BY_ROOT empno "mgr empno",
CONNECT_BY_ROOT ename "mgr ename"
FROM emp
START WITH ename IN ('BLAKE','CLARK','JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

Note that the output from the query shows that all of the root nodes in columns `mgr empno` and `mgr ename` are one of the employees, `BLAKE`, `CLARK`, or `JONES`, listed in the `START WITH` clause.

```
 level | employee    | empno | mgr  | mgr empno | mgr ename
-------+-------------+-------+------+-----------+-----------
     1 | BLAKE       |  7698 | 7839 |      7698 | BLAKE
     2 |   ALLEN     |  7499 | 7698 |      7698 | BLAKE
     2 |   JAMES     |  7900 | 7698 |      7698 | BLAKE
     2 |   MARTIN    |  7654 | 7698 |      7698 | BLAKE
     2 |   TURNER    |  7844 | 7698 |      7698 | BLAKE
     2 |   WARD      |  7521 | 7698 |      7698 | BLAKE
     1 | CLARK       |  7782 | 7839 |      7782 | CLARK
     2 |   MILLER    |  7934 | 7782 |      7782 | CLARK
     1 | JONES       |  7566 | 7839 |      7566 | JONES
     2 |   FORD      |  7902 | 7566 |      7566 | JONES
     3 |     SMITH   |  7369 | 7902 |      7566 | JONES
     2 |   SCOTT     |  7788 | 7566 |      7566 | JONES
     3 |     ADAMS   |  7876 | 7788 |      7566 | JONES
(13 rows)
```

The following is a similar query, but producing only one tree starting with the single, top-level, employee where the `mgr` column is null.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
CONNECT_BY_ROOT empno "mgr empno",
CONNECT_BY_ROOT ename "mgr ename"
FROM emp START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

In the following output, all of the root nodes in columns `mgr empno` and `mgr ename` indicate `KING` as the root for this particular query.

```
 level |   employee    | empno | mgr  | mgr empno | mgr ename
-------+---------------+-------+------+-----------+-----------
     1 | KING          |  7839 |      |      7839 | KING
     2 |   BLAKE       |  7698 | 7839 |      7839 | KING
     3 |     ALLEN     |  7499 | 7698 |      7839 | KING
     3 |     JAMES     |  7900 | 7698 |      7839 | KING
     3 |     MARTIN    |  7654 | 7698 |      7839 | KING
     3 |     TURNER    |  7844 | 7698 |      7839 | KING
     3 |     WARD      |  7521 | 7698 |      7839 | KING
     2 |   CLARK       |  7782 | 7839 |      7839 | KING
     3 |     MILLER    |  7934 | 7782 |      7839 | KING
     2 |   JONES       |  7566 | 7839 |      7839 | KING
     3 |     FORD      |  7902 | 7566 |      7839 | KING
     4 |       SMITH   |  7369 | 7902 |      7839 | KING
     3 |     SCOTT     |  7788 | 7566 |      7839 | KING
     4 |       ADAMS   |  7876 | 7788 |      7839 | KING
(14 rows)
```

By contrast, the following example omits the `START WITH` clause thereby resulting in fourteen trees.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
CONNECT_BY_ROOT empno "mgr empno",
CONNECT_BY_ROOT ename "mgr ename"
FROM emp
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The following is the output from the query. Each node appears at least once as a root node under the `mgr empno` and `mgr ename` columns since even the leaf nodes form the top of their own trees.

```
 level |  employee    | empno | mgr  | mgr empno | mgr ename
-------+-------------+-------+------+-----------+-----------
     1 | ADAMS       |  7876 | 7788 |      7876 | ADAMS
     1 | ALLEN       |  7499 | 7698 |      7499 | ALLEN
     1 | BLAKE       |  7698 | 7839 |      7698 | BLAKE
     2 |   ALLEN     |  7499 | 7698 |      7698 | BLAKE
     2 |   JAMES     |  7900 | 7698 |      7698 | BLAKE
     2 |   MARTIN    |  7654 | 7698 |      7698 | BLAKE
     2 |   TURNER    |  7844 | 7698 |      7698 | BLAKE
     2 |   WARD      |  7521 | 7698 |      7698 | BLAKE
     1 | CLARK       |  7782 | 7839 |      7782 | CLARK
     2 |   MILLER    |  7934 | 7782 |      7782 | CLARK
     1 | FORD        |  7902 | 7566 |      7902 | FORD
     2 |   SMITH     |  7369 | 7902 |      7902 | FORD
     1 | JAMES       |  7900 | 7698 |      7900 | JAMES
     1 | JONES       |  7566 | 7839 |      7566 | JONES
     2 |   FORD      |  7902 | 7566 |      7566 | JONES
     3 |     SMITH   |  7369 | 7902 |      7566 | JONES
     2 |   SCOTT     |  7788 | 7566 |      7566 | JONES
     3 |     ADAMS   |  7876 | 7788 |      7566 | JONES
     1 | KING        |  7839 |      |      7839 | KING
     2 |   BLAKE     |  7698 | 7839 |      7839 | KING
     3 |     ALLEN   |  7499 | 7698 |      7839 | KING
     3 |     JAMES   |  7900 | 7698 |      7839 | KING
     3 |     MARTIN  |  7654 | 7698 |      7839 | KING
     3 |     TURNER  |  7844 | 7698 |      7839 | KING
     3 |     WARD    |  7521 | 7698 |      7839 | KING
     2 |   CLARK     |  7782 | 7839 |      7839 | KING
     3 |     MILLER  |  7934 | 7782 |      7839 | KING
     2 |   JONES     |  7566 | 7839 |      7839 | KING
     3 |     FORD    |  7902 | 7566 |      7839 | KING
     4 |       SMITH |  7369 | 7902 |      7839 | KING
     3 |     SCOTT   |  7788 | 7566 |      7839 | KING
     4 |       ADAMS |  7876 | 7788 |      7839 | KING
     1 | MARTIN      |  7654 | 7698 |      7654 | MARTIN
     1 | MILLER      |  7934 | 7782 |      7934 | MILLER
     1 | SCOTT       |  7788 | 7566 |      7788 | SCOTT
     2 |   ADAMS     |  7876 | 7788 |      7788 | SCOTT
     1 | SMITH       |  7369 | 7902 |      7369 | SMITH
     1 | TURNER      |  7844 | 7698 |      7844 | TURNER
     1 | WARD        |  7521 | 7698 |      7521 | WARD
(39 rows)
```

The following illustrates the unary operator effect of `CONNECT_BY_ROOT`. As shown in this example, when applied to an expression that is not enclosed in parentheses, the `CONNECT_BY_ROOT` operator affects only the term, `ename`, immediately following it. The subsequent concatenation of `|| ' manages ' || ename` is not part of the `CONNECT_BY_ROOT` operation, hence the second occurrence of `ename` results in the value of the currently processed row while the first occurrence of `ename` results in the value from the root node.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
CONNECT_BY_ROOT ename || ' manages ' || ename "top mgr/employee"
FROM emp
START WITH ename IN ('BLAKE','CLARK','JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The following is the output from the query. Note the values produced under the `top mgr/employee` column.

```
 level | employee   | empno | mgr  |    top mgr/employee
-------+-----------+-------+------+----------------------
     1 | BLAKE      |  7698 | 7839 | BLAKE manages BLAKE
     2 |   ALLEN    |  7499 | 7698 | BLAKE manages ALLEN
     2 |   JAMES    |  7900 | 7698 | BLAKE manages JAMES
     2 |   MARTIN   |  7654 | 7698 | BLAKE manages MARTIN
     2 |   TURNER   |  7844 | 7698 | BLAKE manages TURNER
     2 |   WARD     |  7521 | 7698 | BLAKE manages WARD
     1 | CLARK      |  7782 | 7839 | CLARK manages CLARK
     2 |   MILLER   |  7934 | 7782 | CLARK manages MILLER
     1 | JONES      |  7566 | 7839 | JONES manages JONES
     2 |   FORD     |  7902 | 7566 | JONES manages FORD
     3 |     SMITH  |  7369 | 7902 | JONES manages SMITH
     2 |   SCOTT    |  7788 | 7566 | JONES manages SCOTT
     3 |     ADAMS  |  7876 | 7788 | JONES manages ADAMS
(13 rows)
```

The following example uses the `CONNECT_BY_ROOT` operator on an expression enclosed in parentheses.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
CONNECT_BY_ROOT ('Manager ' || ename || ' is emp # ' || empno)
"top mgr/empno"
FROM emp
START WITH ename IN ('BLAKE','CLARK','JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The following is the output of the query. Note that the values of both `ename` and `empno` are affected by the `CONNECT_BY_ROOT` operator and as a result, return the values from the root node as shown under the `top mgr/empno` column.

```
 level | employee   | empno | mgr  |         top mgr/empno
-------+-----------+-------+------+-----------------------------
     1 | BLAKE      |  7698 | 7839 | Manager BLAKE is emp # 7698
     2 |   ALLEN    |  7499 | 7698 | Manager BLAKE is emp # 7698
     2 |   JAMES    |  7900 | 7698 | Manager BLAKE is emp # 7698
     2 |   MARTIN   |  7654 | 7698 | Manager BLAKE is emp # 7698
     2 |   TURNER   |  7844 | 7698 | Manager BLAKE is emp # 7698
     2 |   WARD     |  7521 | 7698 | Manager BLAKE is emp # 7698
     1 | CLARK      |  7782 | 7839 | Manager CLARK is emp # 7782
     2 |   MILLER   |  7934 | 7782 | Manager CLARK is emp # 7782
     1 | JONES      |  7566 | 7839 | Manager JONES is emp # 7566
     2 |   FORD     |  7902 | 7566 | Manager JONES is emp # 7566
```

```
      3 |      SMITH |  7369 | 7902 | Manager JONES is emp # 7566
      2 |    SCOTT    |  7788 | 7566 | Manager JONES is emp # 7566
      3 |      ADAMS |  7876 | 7788 | Manager JONES is emp # 7566
(13 rows)
```

2.2.5.7 Retrieving a Path with SYS_CONNECT_BY_PATH

`SYS_CONNECT_BY_PATH` is a function that works within a hierarchical query to retrieve the column values of a specified column that occur between the current node and the root node. The signature of the function is:

    SYS_CONNECT_BY_PATH (column, delimiter)

The function takes two arguments:

> *`column`* is the name of a column that resides within a table specified in the hierarchical query that is calling the function.
>
> *`delimiter`* is the `varchar` value that separates each entry in the specified column.

The following example returns a list of employee names, and their managers; if the manager has a manager, that name is appended to the result:

```
edb=# SELECT level, ename , SYS_CONNECT_BY_PATH(ename, '/') managers
      FROM emp
      CONNECT BY PRIOR empno = mgr
      START WITH mgr IS NULL
      ORDER BY level, ename, managers;

 level | ename  |        managers
-------+--------+-------------------------
     1 | KING   | /KING
     2 | BLAKE  | /KING/BLAKE
     2 | CLARK  | /KING/CLARK
     2 | JONES  | /KING/JONES
     3 | ALLEN  | /KING/BLAKE/ALLEN
     3 | FORD   | /KING/JONES/FORD
     3 | JAMES  | /KING/BLAKE/JAMES
     3 | MARTIN | /KING/BLAKE/MARTIN
     3 | MILLER | /KING/CLARK/MILLER
     3 | SCOTT  | /KING/JONES/SCOTT
     3 | TURNER | /KING/BLAKE/TURNER
     3 | WARD   | /KING/BLAKE/WARD
     4 | ADAMS  | /KING/JONES/SCOTT/ADAMS
     4 | SMITH  | /KING/JONES/FORD/SMITH
(14 rows)
```

Within the result set:

- The `level` column displays the number of levels that the query returned.
- The `ename` column displays the employee name.
- The `managers` column contains the hierarchical list of managers.

The Advanced Server implementation of `SYS_CONNECT_BY_PATH` does not support use of:

- `SYS_CONNECT_BY_PATH` inside `CONNECT_BY_PATH`
- `SYS_CONNECT_BY_PATH` inside `SYS_CONNECT_BY_PATH`

2.2.6 Multidimensional Analysis

Multidimensional analysis refers to the process commonly used in data warehousing applications of examining data using various combinations of dimensions. *Dimensions* are categories used to classify data such as time, geography, a company's departments, product lines, and so forth. The results associated with a particular set of dimensions are called *facts*. Facts are typically figures associated with product sales, profits, volumes, counts, etc.

In order to obtain these facts according to a set of dimensions in a relational database system, SQL aggregation is typically used. *SQL aggregation* basically means data is grouped according to certain criteria (dimensions) and the result set consists of aggregates of facts such as counts, sums, and averages of the data in each group.

The `GROUP BY` clause of the SQL `SELECT` command supports the following extensions that simplify the process of producing aggregate results.

- `ROLLUP` extension
- `CUBE` extension
- `GROUPING SETS` extension

In addition, the `GROUPING` function and the `GROUPING_ID` function can be used in the `SELECT` list or the `HAVING` clause to aid with the interpretation of the results when these extensions are used.

Note: The sample `dept` and `emp` tables are used extensively in this discussion to provide usage examples. The following changes were applied to these tables to provide more informative results.

```
UPDATE dept SET loc = 'BOSTON' WHERE deptno = 20;
INSERT INTO emp (empno,ename,job,deptno) VALUES (9001,'SMITH','CLERK',40);
INSERT INTO emp (empno,ename,job,deptno) VALUES (9002,'JONES','ANALYST',40);
INSERT INTO emp (empno,ename,job,deptno) VALUES (9003,'ROGERS','MANAGER',40);
```

The following rows from a join of the `emp` and `dept` tables are used:

```
SELECT loc, dname, job, empno FROM emp e, dept d
WHERE e.deptno = d.deptno
ORDER BY 1, 2, 3, 4;

   loc     |   dname    |    job    | empno
-----------+------------+-----------+-------
 BOSTON    | OPERATIONS | ANALYST   |  9002
 BOSTON    | OPERATIONS | CLERK     |  9001
 BOSTON    | OPERATIONS | MANAGER   |  9003
 BOSTON    | RESEARCH   | ANALYST   |  7788
 BOSTON    | RESEARCH   | ANALYST   |  7902
 BOSTON    | RESEARCH   | CLERK     |  7369
 BOSTON   | RESEARCH   | CLERK     |  7876
 BOSTON   | RESEARCH   | MANAGER   |  7566
 CHICAGO  | SALES      | CLERK     |  7900
 CHICAGO  | SALES      | MANAGER   |  7698
 CHICAGO  | SALES      | SALESMAN  |  7499
 CHICAGO  | SALES      | SALESMAN  |  7521
 CHICAGO  | SALES      | SALESMAN  |  7654
 CHICAGO  | SALES      | SALESMAN  |  7844
 NEW YORK | ACCOUNTING | CLERK     |  7934
 NEW YORK | ACCOUNTING | MANAGER   |  7782
 NEW YORK | ACCOUNTING | PRESIDENT |  7839
(17 rows)
```

The `loc`, `dname`, and `job` columns are used for the dimensions of the SQL aggregations used in the examples. The resulting facts of the aggregations are the number of employees obtained by using the `COUNT(*)` function.

A basic query grouping the `loc`, `dname`, and `job` columns is given by the following.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc, dname, job
ORDER BY 1, 2, 3;
```

The rows of this result set using the basic `GROUP BY` clause without extensions are referred to as the *base aggregate* rows.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
(12 rows)
```

The `ROLLUP` and `CUBE` extensions add to the base aggregate rows by providing additional levels of subtotals to the result set.

The `GROUPING SETS` extension provides the ability to combine different types of groupings into a single result set.

The `GROUPING` and `GROUPING_ID` functions aid in the interpretation of the result set.

The additions provided by these extensions are discussed in more detail in the subsequent sections.

2.2.6.1 ROLLUP Extension

The `ROLLUP` extension produces a hierarchical set of groups with subtotals for each hierarchical group as well as a grand total. The order of the hierarchy is determined by the order of the expressions given in the `ROLLUP` expression list. The top of the hierarchy is the leftmost item in the list. Each successive item proceeding to the right moves down the hierarchy with the rightmost item being the lowest level.

The syntax for a single `ROLLUP` is as follows:

```
ROLLUP ( { expr_1 | ( expr_1a [, expr_1b ] ...) }
  [, expr_2 | ( expr_2a [, expr_2b ] ...) ] ...)
```

Each *`expr`* is an expression that determines the grouping of the result set. If enclosed within parenthesis as `(` *`expr_1a`*`,` *`expr_1b`*`, ...)` then the combination of values returned by *`expr_1a`* and *`expr_1b`* defines a single grouping level of the hierarchy.

The base level of aggregates returned in the result set is for each unique combination of values returned by the expression list.

In addition, a subtotal is returned for the first item in the list (*`expr_1`* or the combination of `(` *`expr_1a`*`,` *`expr_1b`*`, ...)`, whichever is specified) for each unique value. A subtotal is returned for the second item in the list (*`expr_2`* or the combination of `(` *`expr_2a`*`,` *`expr_2b`*`, ...)`, whichever is specified) for each unique value, within each grouping of the first item and so on. Finally a grand total is returned for the entire result set.

For the subtotal rows, null is returned for the items across which the subtotal is taken.

The `ROLLUP` extension specified within the context of the `GROUP BY` clause is shown by the following:

```
SELECT select_list FROM ...
GROUP BY [... ,] ROLLUP ( expression_list ) [, ...]
```

The items specified in *`select_list`* must also appear in the `ROLLUP` *`expression_list`*; or they must be aggregate functions such as `COUNT`, `SUM`, `AVG`, `MIN`, or `MAX`; or they must be constants or functions whose return values are independent of the individual rows in the group (for example, the `SYSDATE` function).

The `GROUP BY` clause may specify multiple `ROLLUP` extensions as well as multiple occurrences of other `GROUP BY` extensions and individual expressions.

The ORDER BY clause should be used if you want the output to display in a hierarchical or other meaningful structure. There is no guarantee on the order of the result set if no ORDER BY clause is specified.

The number of grouping levels or totals is *n* + 1 where *n* represents the number of items in the ROLLUP expression list. A parenthesized list counts as one item.

The following query produces a rollup based on a hierarchy of columns loc, dname, then job.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, dname, job)
ORDER BY 1, 2, 3;
```

The following is the result of the query. There is a count of the number of employees for each unique combination of loc, dname, and job, as well as subtotals for each unique combination of loc and dname, for each unique value of loc, and a grand total displayed on the last line.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | OPERATIONS |           |         3
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   | RESEARCH   |           |         5
 BOSTON   |            |           |         8
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  | SALES      |           |         6
 CHICAGO  |            |           |         6
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
 NEW YORK | ACCOUNTING |           |         3
 NEW YORK |            |           |         3
          |            |           |        17
(20 rows)
```

The following query shows the effect of combining items in the ROLLUP list within parenthesis.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, (dname, job))
ORDER BY 1, 2, 3;
```

In the output, note that there are no subtotals for loc and dname combinations as in the prior example.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   |            |           |         8
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  |            |           |         6
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
 NEW YORK |            |           |         3
          |            |           |        17
(16 rows)
```

If the first two columns in the `ROLLUP` list are enclosed in parenthesis, the subtotal levels differ as well.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP ((loc, dname), job)
ORDER BY 1, 2, 3;
```

Now there is a subtotal for each unique `loc` and `dname` combination, but none for unique values of `loc`.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | OPERATIONS |           |         3
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   | RESEARCH   |           |         5
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  | SALES      |           |         6
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
 NEW YORK | ACCOUNTING |           |         3
          |            |           |        17
(17 rows)
```

2.2.6.2 CUBE Extension

The CUBE extension is similar to the ROLLUP extension. However, unlike ROLLUP, which produces groupings and results in a hierarchy based on a left to right listing of items in the ROLLUP expression list, a CUBE produces groupings and subtotals based on every permutation of all items in the CUBE expression list. Thus, the result set contains more rows than a ROLLUP performed on the same expression list.

The syntax for a single CUBE is as follows:

```
CUBE ( { expr_1 | ( expr_1a [, expr_1b ] ...) }
  [, expr_2 | ( expr_2a [, expr_2b ] ...) ] ...)
```

Each *expr* is an expression that determines the grouping of the result set. If enclosed within parenthesis as (*expr_1a*, *expr_1b*, ...) then the combination of values returned by *expr_1a* and *expr_1b* defines a single group.

The base level of aggregates returned in the result set is for each unique combination of values returned by the expression list.

In addition, a subtotal is returned for the first item in the list (*expr_1* or the combination of (*expr_1a*, *expr_1b*, ...), whichever is specified) for each unique value. A subtotal is returned for the second item in the list (*expr_2* or the combination of (*expr_2a*, *expr_2b*, ...), whichever is specified) for each unique value. A subtotal is also returned for each unique combination of the first item and the second item. Similarly, if there is a third item, a subtotal is returned for each unique value of the third item, each unique value of the third item and first item combination, each unique value of the third item and second item combination, and each unique value of the third item, second item, and first item combination. Finally a grand total is returned for the entire result set.

For the subtotal rows, null is returned for the items across which the subtotal is taken.

The CUBE extension specified within the context of the GROUP BY clause is shown by the following:

```
SELECT select_list FROM ...
GROUP BY [... ,] CUBE ( expression_list ) [, ...]
```

The items specified in *select_list* must also appear in the CUBE *expression_list*; or they must be aggregate functions such as COUNT, SUM, AVG, MIN, or MAX; or they must be constants or functions whose return values are independent of the individual rows in the group (for example, the SYSDATE function).

The GROUP BY clause may specify multiple CUBE extensions as well as multiple occurrences of other GROUP BY extensions and individual expressions.

The ORDER BY clause should be used if you want the output to display in a meaningful structure. There is no guarantee on the order of the result set if no ORDER BY clause is specified.

The number of grouping levels or totals is 2 raised to the power of *n* where *n* represents the number of items in the CUBE expression list. A parenthesized list counts as one item.

The following query produces a cube based on permutations of columns loc, dname, and job.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, dname, job)
ORDER BY 1, 2, 3;
```

The following is the result of the query. There is a count of the number of employees for each combination of loc, dname, and job, as well as subtotals for each combination of loc and dname, for each combination of loc and job, for each combination of dname and job, for each unique value of loc, for each unique value of dname, for each unique value of job, and a grand total displayed on the last line.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | OPERATIONS |           |         3
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   | RESEARCH   |           |         5
 BOSTON   |            | ANALYST   |         3
 BOSTON   |            | CLERK     |         3
 BOSTON   |            | MANAGER   |         2
 BOSTON   |            |           |         8
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  | SALES      |           |         6
 CHICAGO  |            | CLERK     |         1
 CHICAGO  |            | MANAGER   |         1
 CHICAGO  |            | SALESMAN  |         4
 CHICAGO  |            |           |         6
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
 NEW YORK | ACCOUNTING |           |         3
 NEW YORK |            | CLERK     |         1
 NEW YORK |            | MANAGER   |         1
 NEW YORK |            | PRESIDENT |         1
 NEW YORK |            |           |         3
          | ACCOUNTING | CLERK     |         1
```

```
           | ACCOUNTING | MANAGER   |         1
           | ACCOUNTING | PRESIDENT |         1
           | ACCOUNTING |           |         3
           | OPERATIONS | ANALYST   |         1
           | OPERATIONS | CLERK     |         1
           | OPERATIONS | MANAGER   |         1
           | OPERATIONS |           |         3
           | RESEARCH   | ANALYST   |         2
           | RESEARCH   | CLERK     |         2
           | RESEARCH   | MANAGER   |         1
           | RESEARCH   |           |         5
           | SALES      | CLERK     |         1
           | SALES      | MANAGER   |         1
           | SALES      | SALESMAN  |         4
           | SALES      |           |         6
           |            | ANALYST   |         3
           |            | CLERK     |         5
           |            | MANAGER   |         4
           |            | PRESIDENT |         1
           |            | SALESMAN  |         4
           |            |           |        17
(50 rows)
```

The following query shows the effect of combining items in the `CUBE` list within parenthesis.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, (dname, job))
ORDER BY 1, 2, 3;
```

In the output note that there are no subtotals for permutations involving `loc` and `dname` combinations, `loc` and `job` combinations, or for `dname` by itself, or for `job` by itself.

```
   loc     |   dname    |    job    | employees
-----------+------------+-----------+-----------
 BOSTON    | OPERATIONS | ANALYST   |         1
 BOSTON    | OPERATIONS | CLERK     |         1
 BOSTON    | OPERATIONS | MANAGER   |         1
 BOSTON    | RESEARCH   | ANALYST   |         2
 BOSTON    | RESEARCH   | CLERK     |         2
 BOSTON    | RESEARCH   | MANAGER   |         1
 BOSTON    |            |           |         8
 CHICAGO   | SALES      | CLERK     |         1
 CHICAGO   | SALES      | MANAGER   |         1
 CHICAGO   | SALES      | SALESMAN  |         4
 CHICAGO   |            |           |         6
 NEW YORK  | ACCOUNTING | CLERK     |         1
 NEW YORK  | ACCOUNTING | MANAGER   |         1
 NEW YORK  | ACCOUNTING | PRESIDENT |         1
 NEW YORK  |            |           |         3
           | ACCOUNTING | CLERK     |         1
           | ACCOUNTING | MANAGER   |         1
           | ACCOUNTING | PRESIDENT |         1
           | OPERATIONS | ANALYST   |         1
           | OPERATIONS | CLERK     |         1
           | OPERATIONS | MANAGER   |         1
           | RESEARCH   | ANALYST   |         2
           | RESEARCH   | CLERK     |         2
           | RESEARCH   | MANAGER   |         1
```

```
          | SALES        | CLERK     |         1
          | SALES        | MANAGER   |         1
          | SALES        | SALESMAN  |         4
          |              |           |        17
(28 rows)
```

The following query shows another variation whereby the first expression is specified outside of the `CUBE` extension.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc, CUBE (dname, job)
ORDER BY 1, 2, 3;
```

In this output, the permutations are performed for `dname` and `job` within each grouping of `loc`.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | OPERATIONS |           |         3
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   | RESEARCH   |           |         5
 BOSTON   |            | ANALYST   |         3
 BOSTON   |            | CLERK     |         3
 BOSTON   |            | MANAGER   |         2
 BOSTON   |            |           |         8
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  | SALES      |           |         6
 CHICAGO  |            | CLERK     |         1
 CHICAGO  |            | MANAGER   |         1
 CHICAGO  |            | SALESMAN  |         4
 CHICAGO  |            |           |         6
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
 NEW YORK | ACCOUNTING |           |         3
 NEW YORK |            | CLERK     |         1
 NEW YORK |            | MANAGER   |         1
 NEW YORK |            | PRESIDENT |         1
 NEW YORK |            |           |         3
(28 rows)
```

2.2.6.3 GROUPING SETS Extension

The use of the `GROUPING SETS` extension within the `GROUP BY` clause provides a means to produce one result set that is actually the concatenation of multiple results sets based upon different groupings. In other words, a `UNION ALL` operation is performed combining the result sets of multiple groupings into one result set.

Note that a `UNION ALL` operation, and therefore the `GROUPING SETS` extension, do not eliminate duplicate rows from the result sets that are being combined together.

The syntax for a single `GROUPING SETS` extension is as follows:

```
GROUPING SETS (
  { expr_1 | ( expr_1a [, expr_1b ] ...) |
    ROLLUP ( expr_list ) | CUBE ( expr_list )
  } [, ...] )
```

A `GROUPING SETS` extension can contain any combination of one or more comma-separated expressions, lists of expressions enclosed within parenthesis, `ROLLUP` extensions, and `CUBE` extensions.

The `GROUPING SETS` extension is specified within the context of the `GROUP BY` clause as shown by the following:

```
SELECT select_list FROM ...
GROUP BY [... ,] GROUPING SETS ( expression_list ) [, ...]
```

The items specified in *`select_list`* must also appear in the `GROUPING SETS` *`expression_list`*; or they must be aggregate functions such as `COUNT`, `SUM`, `AVG`, `MIN`, or `MAX`; or they must be constants or functions whose return values are independent of the individual rows in the group (for example, the `SYSDATE` function).

The `GROUP BY` clause may specify multiple `GROUPING SETS` extensions as well as multiple occurrences of other `GROUP BY` extensions and individual expressions.

The `ORDER BY` clause should be used if you want the output to display in a meaningful structure. There is no guarantee on the order of the result set if no `ORDER BY` clause is specified.

The following query produces a union of groups given by columns `loc`, `dname`, and `job`.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY GROUPING SETS (loc, dname, job)
ORDER BY 1, 2, 3;
```

The result is as follows:

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   |            |           |         8
 CHICAGO  |            |           |         6
 NEW YORK |            |           |         3
          | ACCOUNTING |           |         3
          | OPERATIONS |           |         3
          | RESEARCH   |           |         5
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
          |            | MANAGER   |         4
          |            | PRESIDENT |         1
          |            | SALESMAN  |         4
(12 rows)
```

This is equivalent to the following query, which employs the use of the `UNION ALL` operator.

```
SELECT loc AS "loc", NULL AS "dname", NULL AS "job", COUNT(*) AS "employees"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc
  UNION ALL
SELECT NULL, dname, NULL, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY dname
  UNION ALL
SELECT NULL, NULL, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY job
ORDER BY 1, 2, 3;
```

The output from the `UNION ALL` query is the same as the `GROUPING SETS` output.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   |            |           |         8
 CHICAGO  |            |           |         6
 NEW YORK |            |           |         3
          | ACCOUNTING |           |         3
          | OPERATIONS |           |         3
          | RESEARCH   |           |         5
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
          |            | MANAGER   |         4
          |            | PRESIDENT |         1
          |            | SALESMAN  |         4
(12 rows)
```

The following example shows how various types of `GROUP BY` extensions can be used together within a `GROUPING SETS` expression list.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

```
GROUP BY GROUPING SETS (loc, ROLLUP (dname, job), CUBE (job, loc))
ORDER BY 1, 2, 3;
```

The following is the output from this query.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   |            | ANALYST   |         3
 BOSTON   |            | CLERK     |         3
 BOSTON   |            | MANAGER   |         2
 BOSTON   |            |           |         8
 BOSTON   |            |           |         8
 CHICAGO  |            | CLERK     |         1
 CHICAGO  |            | MANAGER   |         1
 CHICAGO  |            | SALESMAN  |         4
 CHICAGO  |            |           |         6
 CHICAGO  |            |           |         6
 NEW YORK |            | CLERK     |         1
 NEW YORK |            | MANAGER   |         1
 NEW YORK |            | PRESIDENT |         1
 NEW YORK |            |           |         3
 NEW YORK |            |           |         3
          | ACCOUNTING | CLERK     |         1
          | ACCOUNTING | MANAGER   |         1
          | ACCOUNTING | PRESIDENT |         1
          | ACCOUNTING |           |         3
          | OPERATIONS | ANALYST   |         1
          | OPERATIONS | CLERK     |         1
          | OPERATIONS | MANAGER   |         1
          | OPERATIONS |           |         3
          | RESEARCH   | ANALYST   |         2
          | RESEARCH   | CLERK     |         2
          | RESEARCH   | MANAGER   |         1
          | RESEARCH   |           |         5
          | SALES      | CLERK     |         1
          | SALES      | MANAGER   |         1
          | SALES      | SALESMAN  |         4
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
          |            | MANAGER   |         4
          |            | PRESIDENT |         1
          |            | SALESMAN  |         4
          |            |           |        17
          |            |           |        17
(38 rows)
```

The output is basically a concatenation of the result sets that would be produced individually from `GROUP BY loc`, `GROUP BY ROLLUP (dname, job)`, and `GROUP BY CUBE (job, loc)`. These individual queries are shown by the following.

```
SELECT loc, NULL AS "dname", NULL AS "job", COUNT(*) AS "employees"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc
ORDER BY 1;
```

The following is the result set from the `GROUP BY loc` clause.

```
   loc     | dname | job | employees
----------+-------+-----+-----------
 BOSTON   |       |     |         8
 CHICAGO  |       |     |         6
 NEW YORK |       |     |         3
(3 rows)
```

The following query uses the `GROUP BY ROLLUP (dname, job)` clause.

```
SELECT NULL AS "loc", dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (dname, job)
ORDER BY 2, 3;
```

The following is the result set from the `GROUP BY ROLLUP (dname, job)` clause.

```
 loc |   dname    |    job    | employees
-----+------------+-----------+-----------
     | ACCOUNTING | CLERK     |         1
     | ACCOUNTING | MANAGER   |         1
     | ACCOUNTING | PRESIDENT |         1
     | ACCOUNTING |           |         3
     | OPERATIONS | ANALYST   |         1
     | OPERATIONS | CLERK     |         1
     | OPERATIONS | MANAGER   |         1
     | OPERATIONS |           |         3
     | RESEARCH   | ANALYST   |         2
     | RESEARCH   | CLERK     |         2
     | RESEARCH   | MANAGER   |         1
     | RESEARCH   |           |         5
     | SALES      | CLERK     |         1
     | SALES      | MANAGER   |         1
     | SALES      | SALESMAN  |         4
     | SALES      |           |         6
     |            |           |        17
(17 rows)
```

The following query uses the `GROUP BY CUBE (job, loc)` clause.

```
SELECT loc, NULL AS "dname", job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (job, loc)
ORDER BY 1, 3;
```

The following is the result set from the `GROUP BY CUBE (job, loc)` clause.

```
   loc     | dname |    job    | employees
----------+-------+-----------+-----------
 BOSTON   |       | ANALYST   |         3
 BOSTON   |       | CLERK     |         3
 BOSTON   |       | MANAGER   |         2
 BOSTON   |       |           |         8
 CHICAGO  |       | CLERK     |         1
 CHICAGO  |       | MANAGER   |         1
 CHICAGO  |       | SALESMAN  |         4
 CHICAGO  |       |           |         6
 NEW YORK |         | CLERK     |         1
 NEW YORK |         | MANAGER   |         1
 NEW YORK |         | PRESIDENT |         1
 NEW YORK |         |           |         3
          |         | ANALYST   |         3
          |         | CLERK     |         5
          |         | MANAGER   |         4
          |         | PRESIDENT |         1
          |         | SALESMAN  |         4
          |         |           |        17
(18 rows)
```

If the previous three queries are combined with the `UNION ALL` operator, a concatenation of the three results sets is produced.

```
SELECT loc AS "loc", NULL AS "dname", NULL AS "job", COUNT(*) AS "employees"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc
  UNION ALL
SELECT NULL, dname, job, count(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (dname, job)
  UNION ALL
SELECT loc, NULL, job, count(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (job, loc)
ORDER BY 1, 2, 3;
```

The following is the output, which is the same as when the `GROUP BY GROUPING SETS (loc, ROLLUP (dname, job), CUBE (job, loc))` clause is used.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   |            | ANALYST   |         3
 BOSTON   |            | CLERK     |         3
 BOSTON   |            | MANAGER   |         2
 BOSTON   |            |           |         8
 BOSTON   |            |           |         8
 CHICAGO  |            | CLERK     |         1
 CHICAGO  |            | MANAGER   |         1
 CHICAGO  |            | SALESMAN  |         4
 CHICAGO  |            |           |         6
 CHICAGO  |            |           |         6
 NEW YORK |            | CLERK     |         1
 NEW YORK |            | MANAGER   |         1
 NEW YORK |            | PRESIDENT |         1
 NEW YORK |            |           |         3
 NEW YORK |            |           |         3
          | ACCOUNTING | CLERK     |         1
          | ACCOUNTING | MANAGER   |         1
          | ACCOUNTING | PRESIDENT |         1
          | ACCOUNTING |           |         3
          | OPERATIONS | ANALYST   |         1
          | OPERATIONS | CLERK     |         1
          | OPERATIONS | MANAGER   |         1
          | OPERATIONS |           |         3
          | RESEARCH   | ANALYST   |         2
          | RESEARCH   | CLERK     |         2
          | RESEARCH   | MANAGER   |         1
          | RESEARCH   |           |         5
          | SALES      | CLERK     |         1
          | SALES      | MANAGER   |         1
          | SALES      | SALESMAN  |         4
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
          |            | MANAGER   |         4
          |            | PRESIDENT |         1
          |            | SALESMAN  |         4
          |            |           |        17
          |            |           |        17
(38 rows)
```

2.2.6.4 GROUPING Function

When using the ROLLUP, CUBE, or GROUPING SETS extensions to the GROUP BY clause, it may sometimes be difficult to differentiate between the various levels of subtotals generated by the extensions as well as the base aggregate rows in the result set. The GROUPING function provides a means of making this distinction.

The general syntax for use of the GROUPING function is shown by the following.

```
SELECT [ expr ...,] GROUPING( col_expr ) [, expr ] ...
FROM ...
GROUP BY [...,]
  { ROLLUP | CUBE | GROUPING SETS }( [...,] col_expr
  [, ...] ) [, ...]
```

The GROUPING function takes a single parameter that must be an expression of a dimension column specified in the expression list of a ROLLUP, CUBE, or GROUPING SETS extension of the GROUP BY clause.

The return value of the GROUPING function is either a 0 or 1. In the result set of a query, if the column expression specified in the GROUPING function is null because the row represents a subtotal over multiple values of that column then the GROUPING function returns a value of 1. If the row returns results based on a particular value of the column specified in the GROUPING function, then the GROUPING function returns a value of 0. In the latter case, the column can be null as well as non-null, but in any case, it is for a particular value of that column, not a subtotal across multiple values.

The following query shows how the return values of the GROUPING function correspond to the subtotal lines.

```
SELECT loc, dname, job, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc",
  GROUPING(dname) AS "gf_dname",
  GROUPING(job) AS "gf_job"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, dname, job)
ORDER BY 1, 2, 3;
```

In the three right-most columns displaying the output of the GROUPING functions, a value of 1 appears on a subtotal line wherever a subtotal is taken across values of the corresponding columns.

```
   loc    |   dname    |    job    | employees | gf_loc | gf_dname | gf_job
----------+------------+-----------+-----------+--------+----------+--------
 BOSTON   | OPERATIONS | ANALYST   |         1 |      0 |        0 |      0
 BOSTON   | OPERATIONS | CLERK     |         1 |      0 |        0 |      0
 BOSTON   | OPERATIONS | MANAGER   |         1 |      0 |        0 |      0
 BOSTON    | OPERATIONS |            |         3 |        0 |          0 |        1
 BOSTON    | RESEARCH   | ANALYST    |         2 |        0 |          0 |        0
 BOSTON    | RESEARCH   | CLERK      |         2 |        0 |          0 |        0
 BOSTON    | RESEARCH   | MANAGER    |         1 |        0 |          0 |        0
 BOSTON    | RESEARCH   |            |         5 |        0 |          0 |        1
 BOSTON    |            |            |         8 |        0 |          1 |        1
 CHICAGO   | SALES      | CLERK      |         1 |        0 |          0 |        0
 CHICAGO   | SALES      | MANAGER    |         1 |        0 |          0 |        0
 CHICAGO   | SALES      | SALESMAN   |         4 |        0 |          0 |        0
 CHICAGO   | SALES      |            |         6 |        0 |          0 |        1
 CHICAGO   |            |            |         6 |        0 |          1 |        1
 NEW YORK  | ACCOUNTING | CLERK      |         1 |        0 |          0 |        0
 NEW YORK  | ACCOUNTING | MANAGER    |         1 |        0 |          0 |        0
 NEW YORK  | ACCOUNTING | PRESIDENT  |         1 |        0 |          0 |        0
 NEW YORK  | ACCOUNTING |            |         3 |        0 |          0 |        1
 NEW YORK  |            |            |         3 |        0 |          1 |        1
           |            |            |        17 |        1 |          1 |        1
(20 rows)
```

These indicators can be used as screening criteria for particular subtotals. For example, using the previous query, you can display only those subtotals for `loc` and `dname` combinations by using the `GROUPING` function in a `HAVING` clause.

```
SELECT loc, dname, job, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc",
  GROUPING(dname) AS "gf_dname",
  GROUPING(job) AS "gf_job"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, dname, job)
HAVING GROUPING(loc) = 0
  AND  GROUPING(dname) = 0
  AND  GROUPING(job) = 1
ORDER BY 1, 2;
```

This query produces the following result:

```
   loc     |   dname    | job | employees | gf_loc | gf_dname | gf_job
----------+------------+-----+-----------+--------+----------+--------
 BOSTON    | OPERATIONS |     |         3 |      0 |        0 |      1
 BOSTON    | RESEARCH   |     |         5 |      0 |        0 |      1
 CHICAGO   | SALES      |     |         6 |      0 |        0 |      1
 NEW YORK  | ACCOUNTING |     |         3 |      0 |        0 |      1
(4 rows)
```

The `GROUPING` function can be used to distinguish a subtotal row from a base aggregate row or from certain subtotal rows where one of the items in the expression list returns null as a result of the column on which the expression is based being null for one or more rows in the table, as opposed to representing a subtotal over the column.

To illustrate this point, the following row is added to the `emp` table. This provides a row with a null value for the `job` column.

```
INSERT INTO emp (empno,ename,deptno) VALUES (9004,'PETERS',40);
```

The following query is issued using a reduced number of rows for clarity.

```
SELECT loc, job, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc",
  GROUPING(job) AS "gf_job"
FROM emp e, dept d
WHERE e.deptno = d.deptno AND loc = 'BOSTON'
GROUP BY CUBE (loc, job)
ORDER BY 1, 2;
```

Note that the output contains two rows containing `BOSTON` in the `loc` column and spaces in the `job` column (fourth and fifth entries in the table).

```
  loc    |   job    | employees | gf_loc | gf_job
--------+---------+-----------+--------+--------
 BOSTON | ANALYST |         3 |      0 |      0
 BOSTON | CLERK   |         3 |      0 |      0
 BOSTON | MANAGER |         2 |      0 |      0
 BOSTON |         |         1 |      0 |      0
 BOSTON |         |         9 |      0 |      1
        | ANALYST |         3 |      1 |      0
        | CLERK   |         3 |      1 |      0
        | MANAGER |         2 |      1 |      0
        |         |         1 |      1 |      0
        |         |         9 |      1 |      1
(10 rows)
```

The fifth row where the `GROUPING` function on the `job` column (`gf_job`) returns 1 indicates this is a subtotal over all jobs. Note that the row contains a subtotal value of 9 in the `employees` column.

The fourth row where the `GROUPING` function on the `job` column as well as on the `loc` column returns 0 indicates this is a base aggregate of all rows where `loc` is `BOSTON` and `job` is null, which is the row inserted for this example. The `employees` column contains 1, which is the count of the single such row inserted.

Also note that in the ninth row (next to last) the `GROUPING` function on the `job` column returns 0 while the `GROUPING` function on the `loc` column returns 1 indicating this is a subtotal over all locations where the `job` column is null, which again, is a count of the single row inserted for this example.

2.2.6.5 GROUPING_ID Function

The GROUPING_ID function provides a simplification of the GROUPING function in order to determine the subtotal level of a row in the result set from a ROLLBACK, CUBE, or GROUPING SETS extension.

The GROUPING function takes only one column expression and returns an indication of whether or not a row is a subtotal over all values of the given column. Thus, multiple GROUPING functions may be required to interpret the level of subtotals for queries with multiple grouping columns.

The GROUPING_ID function accepts one or more column expressions that have been used in the ROLLBACK, CUBE, or GROUPING SETS extensions and returns a single integer that can be used to determine over which of these columns a subtotal has been aggregated.

The general syntax for use of the GROUPING_ID function is shown by the following.

```
SELECT [ expr ...,]
  GROUPING_ID( col_expr_1 [, col_expr_2 ] ... )
  [, expr ] ...
FROM ...
GROUP BY [...,]
  { ROLLUP | CUBE | GROUPING SETS }( [...,] col_expr_1
  [, col_expr_2 ] [, ...] ) [, ...]
```

The GROUPING_ID function takes one or more parameters that must be expressions of dimension columns specified in the expression list of a ROLLUP, CUBE, or GROUPING SETS extension of the GROUP BY clause.

The GROUPING_ID function returns an integer value. This value corresponds to the base-10 interpretation of a bit vector consisting of the concatenated 1's and 0's that would be returned by a series of GROUPING functions specified in the same left-to-right order as the ordering of the parameters specified in the GROUPING_ID function.

The following query shows how the returned values of the GROUPING_ID function represented in column gid correspond to the values returned by two GROUPING functions on columns loc and dname.

```
SELECT loc, dname, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc", GROUPING(dname) AS "gf_dname",
  GROUPING_ID(loc, dname) AS "gid"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, dname)
ORDER BY 6, 1, 2;
```

In the following output, note the relationship between a bit vector consisting of the `gf_loc` value and `gf_dname` value compared to the integer given in `gid`.

```
   loc     |   dname    | employees | gf_loc | gf_dname | gid
-----------+------------+-----------+--------+----------+-----
 BOSTON    | OPERATIONS |         3 |      0 |        0 |   0
 BOSTON    | RESEARCH   |         5 |      0 |        0 |   0
 CHICAGO   | SALES      |         6 |      0 |        0 |   0
 NEW YORK  | ACCOUNTING |         3 |      0 |        0 |   0
 BOSTON    |            |         8 |      0 |        1 |   1
 CHICAGO   |            |         6 |      0 |        1 |   1
 NEW YORK  |            |         3 |      0 |        1 |   1
           | ACCOUNTING |         3 |      1 |        0 |   2
           | OPERATIONS |         3 |      1 |        0 |   2
           | RESEARCH   |         5 |      1 |        0 |   2
           | SALES      |         6 |      1 |        0 |   2
           |            |        17 |      1 |        1 |   3
(12 rows)
```

The following table provides specific examples of the `GROUPING_ID` function calculations based on the `GROUPING` function return values for four rows of the output.

loc	dname	Bit Vector gf_loc gf_dname	GROUPING_ID gid
BOSTON	OPERATIONS	$0 * 2^1 + 0 * 2^0$	0
BOSTON	null	$0 * 2^1 + 1 * 2^0$	1
null	ACCOUNTING	$1 * 2^1 + 0 * 2^0$	2
null	null	$1 * 2^1 + 1 * 2^0$	3

The following table summarizes how the `GROUPING_ID` function return values correspond to the grouping columns over which aggregation occurs.

Aggregation by Column	Bit Vector gf_loc gf_dname	GROUPING_ID gid
loc, dname	0 0	0
loc	0 1	1
dname	1 0	2
Grand Total	1 1	3

So to display only those subtotals by `dname`, the following simplified query can be used with a `HAVING` clause based on the `GROUPING_ID` function.

```
SELECT loc, dname, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc", GROUPING(dname) AS "gf_dname",
  GROUPING_ID(loc, dname) AS "gid"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, dname)
HAVING GROUPING_ID(loc, dname) = 2
ORDER BY 6, 1, 2;
```

The following is the result of the query.

```
loc |   dname    | employees | gf_loc | gf_dname | gid
-----+------------+-----------+--------+----------+-----
     | ACCOUNTING |         3 |      1 |        0 |   2
     | OPERATIONS |         3 |      1 |        0 |   2
     | RESEARCH   |         5 |      1 |        0 |   2
     | SALES      |         6 |      1 |        0 |   2
(4 rows)
```

2.3 Profile Management

Advanced Server 9.5 allows a database superuser to create named *profiles*. Each profile defines rules for password management that augment `password` and `md5` authentication. The rules in a profile can:

- count failed login attempts
- lock an account due to excessive failed login attempts
- mark a password for expiration
- define a grace period after a password expiration
- define rules for password complexity
- define rules that limit password re-use

A profile is a named set of password attributes that allow you to easily manage a group of roles that share comparable authentication requirements. If the password requirements change, you can modify the profile to have the new requirements applied to each user that is associated with that profile.

After creating the profile, you can associate the profile with one or more users. When a user connects to the server, the server enforces the profile that is associated with their login role. Profiles are shared by all databases within a cluster, but each cluster may have multiple profiles. A single user with access to multiple databases will use the same profile when connecting to each database within the cluster.

Advanced Server 9.5 creates a profile named `default` that is associated with a new role when the role is created unless an alternate profile is specified. If you upgrade to Advanced Server 9.5 from a previous server version, existing roles will automatically be assigned to the `default` profile. You cannot delete the `default` profile.

The `default` profile specifies the following attributes:

```
FAILED_LOGIN_ATTEMPTS      UNLIMITED
PASSWORD_LOCK_TIME         UNLIMITED
PASSWORD_LIFE_TIME         UNLIMITED
PASSWORD_GRACE_TIME        UNLIMITED
PASSWORD_REUSE_TIME        UNLIMITED
PASSWORD_REUSE_MAX         UNLIMITED
PASSWORD_VERIFY_FUNCTION   NULL
```

A database superuser can use the `ALTER PROFILE` command to modify the values specified by the `default` profile. For more information about modifying a profile, see Section 2.3.2.

2.3.1 Creating a New Profile

Use the CREATE PROFILE command to create a new profile. The syntax is:

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

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

Parameters:

profile_name specifies the name of the profile.

parameter specifies the attribute limited by the profile.

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

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;
```

2.3.1.1 Creating a Password Function

When specifying `PASSWORD_VERIFY_FUNCTION`, you can provide a customized function that specifies the security rules that will be applied when your users change their password. For example, you can specify rules that stipulate that the new password must be at least *n* characters long, and may not contain a specific value.

The password function has the following signature:

```
function_name (user_name VARCHAR2,
               new_password VARCHAR2,
               old_password VARCHAR2) RETURN boolean
```

Where:

user_name is the name of the user.

new_password is the new password.

old_password is the user's previous password. If you reference this parameter within your function:

When a database superuser changes their password, the third parameter will always be `NULL`.

When a user with the `CREATEROLE` attribute changes their password, the parameter will pass the previous password if the statement includes the `REPLACE` clause. Note that the `REPLACE` clause is optional syntax for a user with the `CREATEROLE` privilege.

When a user that is not a database superuser and does not have the `CREATEROLE` attribute changes their password, the third parameter will contain the previous password for the role.

The function returns a Boolean value. If the function returns true and does not raise an exception, the password is accepted; if the function returns false or raises an exception, the password is rejected. If the function raises an exception, the specified error message is displayed to the user. If the function does not raise an exception, but returns false, the following error message is displayed:

```
ERROR:  password verification for the specified password failed
```

The function must be owned by a database superuser, and reside in the `sys` schema.

Example:

The following example creates a profile and a custom function; then, the function is associated with the profile. The following `CREATE PROFILE` command creates a profile named `acctg_pwd_profile`:

```
CREATE PROFILE acctg_pwd_profile;
```

The following commands create a (schema-qualified) function named `verify_password`:

```
CREATE OR REPLACE FUNCTION sys.verify_password(user_name varchar2,
new_password varchar2, old_password varchar2)
RETURN boolean IMMUTABLE
IS
BEGIN
  IF (length(new_password) < 5)
  THEN
    raise_application_error(-20001, 'too short');
  END IF;

  IF substring(new_password FROM old_password) IS NOT NULL
  THEN
    raise_application_error(-20002, 'includes old password');
  END IF;

  RETURN true;
END;
```

The function first ensures that the password is at least 5 characters long, and then compares the new password to the old password. If the new password contains fewer than 5 characters, or contains the old password, the function raises an error.

The following statement sets the ownership of the `verify_password` function to the `enterprisedb` database superuser:

```
ALTER FUNCTION verify_password(varchar2, varchar2, varchar2) OWNER TO
enterprisedb;
```

Then, the verify_password function is associated with the profile:

```
ALTER PROFILE acctg_pwd_profile LIMIT PASSWORD_VERIFY_FUNCTION
verify_password;
```

The following statements confirm that the function is working by first creating a test user (alice), and then attempting to associate invalid and valid passwords with her role:

```
CREATE ROLE alice WITH LOGIN PASSWORD 'temp_password' PROFILE
acctg_pwd_profile;
```

Then, when alice connects to the database and attempts to change her password, she must adhere to the rules established by the profile function. A non-superuser without CREATEROLE must include the REPLACE clause when changing a password:

```
edb=> ALTER ROLE alice PASSWORD 'hey';
ERROR:  missing REPLACE clause
```

The new password must be at least 5 characters long:

```
edb=> ALTER USER alice PASSWORD 'hey' REPLACE 'temp_password';
ERROR:  EDB-20001: too short
CONTEXT:  edb-spl function verify_password(character varying,character
varying,character varying) line 5 at procedure/function invocation statement
```

If the new password is acceptable, the command completes without error:

```
edb=> ALTER USER alice PASSWORD 'hello' REPLACE 'temp_password';
ALTER ROLE
```

If alice decides to change her password, the new password must not contain the old password:

```
edb=> ALTER USER alice PASSWORD 'helloworld' REPLACE 'hello';
ERROR:  EDB-20002: includes old password
CONTEXT:  edb-spl function verify_password(character varying,character
varying,character varying) line 10 at procedure/function invocation statement
```

To remove the verify function, set password_verify_function to NULL:

```
ALTER PROFILE acctg_pwd_profile LIMIT password_verify_function NULL;
```

Then, all password constraints will be lifted:

```
edb=# ALTER ROLE alice PASSWORD 'hey';
ALTER ROLE
```

2.3.2 Altering a Profile

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

```
ALTER PROFILE profile_name RENAME TO new_name;

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

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` specifies the name of the profile.

`new_name` specifies the new name of the profile.

`parameter` specifies the attribute limited by the profile.

`value` specifies the parameter limit.

See the table in Section 2.3.1 for a complete list of accepted parameter/value pairs.

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;
```

2.3.3 Dropping a Profile

Use the `DROP PROFILE` command to drop a profile. The syntax is:

```
DROP PROFILE [IF EXISTS] profile_name [CASCADE|RESTRICT];
```

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.

Examples

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.

2.3.4 Associating a Profile with an Existing Role

After creating a profile, you can use the `ALTER USER… PROFILE` or `ALTER ROLE… PROFILE` command to associate the profile with a role. The command syntax related to profile management functionality is:

```
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:

http://www.postgresql.org/docs/9.5/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.

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;
```

2.3.5 Unlocking a Locked Account

A database superuser can use clauses of the `ALTER USER|ROLE…` command to lock or unlock a role. The syntax is:

```
ALTER USER|ROLE name
      ACCOUNT {LOCK|UNLOCK}
      LOCK TIME 'timestamp'
```

Include the `ACCOUNT LOCK` clause to lock a role immediately; when locked, a role's `LOGIN` functionality is disabled. When you specify the `ACCOUNT LOCK` clause without the `LOCK TIME` clause, the state of the role will not change until a superuser uses the `ACCOUNT UNLOCK` clause to unlock the role.

Use the `ACCOUNT UNLOCK` clause to unlock a role.

Use the `LOCK TIME 'timestamp'` clause to instruct the server to lock the account at the time specified by the given timestamp for the length of time specified by the `PASSWORD_LOCK_TIME` parameter of the profile associated with this role.

Combine the `LOCK TIME 'timestamp'` clause and the `ACCOUNT LOCK` clause to lock an account at a specified time until the account is unlocked by a superuser invoking the `ACCOUNT UNLOCK` clause.

Parameters

`name`

The name of the role that is being locked or unlocked.

`timestamp`

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

Note

This command (available only in Advanced Server) is implemented to support Oracle-styled profile management.

Examples

The following example uses the ACCOUNT LOCK clause to lock the role named john. The account will remain locked until the account is unlocked with the ACCOUNT UNLOCK clause:

```
ALTER ROLE john ACCOUNT LOCK;
```

The following example uses the ACCOUNT UNLOCK clause to unlock the role named john:

```
ALTER USER john ACCOUNT UNLOCK;
```

The following example uses the LOCK TIME '*timestamp*' clause to lock the role named john on September 4, 2015:

```
ALTER ROLE john LOCK TIME ‘September 4 12:00:00 2015’;
```

The role will remain locked for the length of time specified by the PASSWORD_LOCK_TIME parameter.

The following example combines the LOCK TIME '*timestamp*' clause and the ACCOUNT LOCK clause to lock the role named john on September 4, 2015:

```
ALTER ROLE john LOCK TIME ‘September 4 12:00:00 2015’ ACCOUNT LOCK;
```

The role will remain locked until a database superuser uses the ACCOUNT UNLOCK command to unlock the role.

2.3.6 Creating a New Role Associated with a Profile

A database superuser can use clauses of the CREATE USER|ROLE command to assign a named profile to a role when creating the role, or to specify profile management details for a role. The command syntax related to profile management functionality is:

```
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:

http://www.postgresql.org/docs/9.5/static/sql-commands.html

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`.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

2.3.7 Backing up Profile Management Functions

A profile may include a `PASSWORD_VERIFY_FUNCTION` clause that refers to a user-defined function that specifies the behavior enforced by Advanced Server. Profiles are global objects; they are shared by all of the databases within a cluster. While profiles are global objects, user-defined functions are database objects.

Invoking `pg_dumpall` with the `–g` or `–r` option will create a script that recreates the definition of any existing profiles, but that does not recreate the user-defined functions that are referred to by the `PASSWORD_VERIFY_FUNCTION` clause. You should use the `pg_dump` utility to explicitly dump (and later restore) the database in which those functions reside.

The script created by `pg_dump` will contain a command that includes the clause and function name:

```
    ALTER PROFILE… LIMIT PASSWORD_VERIFY_FUNCTION function_name
```

to associate the restored function with the profile with which it was previously associated.

If the `PASSWORD_VERIFY_FUNCTION` clause is set to `DEFAULT` or `NULL`, the behavior will be replicated by the script generated by the `pg_dumpall –g` or `pg_dumpall –r` command.

3 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

3.1 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.

3.1.1 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 3.3.

3.1.2 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.

3.1.3 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.

3.1.3.1 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 (").

3.1.3.2 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.

3.1.3.3 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.

3.1.4 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 `*/`.

3.2 Data Types

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

Table 3-3-1 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(0)`	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

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.2.1 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 3-3-2 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.

3.2.1.1 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.

3.2.1.2 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.

3.2.1.3 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`.

3.2.2 Character Types

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

Table 3-3-3 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.

3.2.3 Binary Data

The following data types allows storage of binary strings.

Table 3-3-4 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
`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.

3.2.4 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 3-3-5 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(0)` 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(0)` 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.

3.2.4.1 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:

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.

3.2.4.2 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.

3.2.4.2.1 Dates

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

Table 3-3-6 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.

3.2.4.2.2 Times

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

Table 3-3-7 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

3.2.4.2.3 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 3-3-6 Date Input. The time portion of the time stamp can be formatted according to any of examples shown in Table 3-3-7 Time Input.

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
```

3.2.4.3 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 3-3-8 Date/Time Output Styles

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

3.2.4.4 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.

3.2.5 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 3-3-9 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`.

3.2.6 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)
```

3.3 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.1 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:

http://www.postgresql.org/docs/9.5/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

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.2 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;
```

3.3.3 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:

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

- 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;
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.4 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.

For more information about Profile Management, see Section 2.3.

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;
```

3.3.5 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

3.3.6 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

3.3.7 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. See Section 3.4 for more information about optimizer hints.

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';
```

3.3.8 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

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

3.3.9 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

3.3.10 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.

For more information about Profile Management, see Section 2.3.

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

3.3.11 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:

http://www.postgresql.org/docs/9.5/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

These clauses are implemented to support Oracle-styled profile management. For more information about Profile Management, see Section 2.3.

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

http://www.postgresql.org/docs/9.5/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;
```

3.3.12 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);
```

3.3.13 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 '';
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.14 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

3.3.15 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;
```

3.3.16 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 { libpq 'libpq_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/9.5/static/high-availability.html

Note: For Advanced Server 9.5, the `CREATE DATABASE LINK` command is tested against and certified for use with Oracle version 10g Release 2 (10.2) and Oracle version 11g Release 2 (11.2).

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`*.

`libpq`

Specifies a `libpq` connection to a remote Advanced Server database.

`libpq_connection_string`

Specify the connection information for a libpq connection.

`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.

If you are executing a SQL command that references a database link to a remote Oracle database, Advanced Server needs a way to know where the correct Oracle installation resides on disk. Set the `LD_LIBRARY_PATH` environment variable on Linux (or `PATH` on Windows) to the `lib` directory of the Oracle client installation directory.

For Windows only, you can instead 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.

The `LD_LIBRARY_PATH` environment variable on Linux (`PATH` environment variable or `oracle_home` configuration parameter on Windows) must be set properly each time you start Advanced Server.

For Windows only: 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`*.

After setting 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 libpq Database Link

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

```
CREATE DATABASE LINK boston
  CONNECT TO admin IDENTIFIED BY 'mypassword'
  USING libpq 'host=127.0.0.1 dbname=sales';
```

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, `edblink`, to the Advanced Server database named, `edb`, located on localhost, running on port 5443. Connect to the Advanced Server database with username, `enterprisedb`, and password, `password`.

```
CREATE DATABASE LINK edblink CONNECT TO enterprisedb IDENTIFIED BY 'password'
USING libpq 'host=localhost port=5443 dbname=edb';
```

Display attributes of database links, `oralink` and `edblink`, 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
 edblink | enterprisedb | host=localhost port=5443 dbname=edb
(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@edblink 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)
```

See Also

DROP DATABASE LINK

3.3.17 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. See Section 7.16 for information about the `UTL_FILE` package.

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.

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 |    dirpath
---------+----------------
 empdir  | C:/TEMP/EMPDIR
(1 row)
```

See Also

DROP DIRECTORY

3.3.18 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
        | 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.

See Section 4.2.4 for more information on functions.

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.

`declarations`

> *`declarations`* are variable, cursor, or 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 `LEAKPROOK` 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`

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

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 INVOKER`. 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.

`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`, `COST`, `ROWS` and `SET` keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

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

3.3.19 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).

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.

If you create an index on a partitioned table, the `CREATE INDEX` command does not propagate indexes to the table's subpartitions.

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

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.20 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 using the `DBMS_MVIEW` package to refresh a materialized view, please see Section 7.6.

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

http://www.postgresql.org/docs/9.5/static/sql-refreshmaterializedview.html

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

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`.

3.3.21 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 ]
        [, ...]) ];
    |
      FUNCTION func_name
      [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
        [, ...]) ]
      RETURN rettype;
    }
  ] [, ...]
  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.

See Section 6 for more information about packages.

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.

Examples

The package specification, `empinfo`, contains three public components - a public variable, a public procedure, and a public function. See the `CREATE PACKAGE BODY` command for the package body for this example.

```
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

3.3.22 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 ]
        [, ...]) ]
    { IS | AS }
        program_body
      END [ proc_name ];
    |
      FUNCTION func_name
      [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
        [, ...]) ]
      RETURN rettype
    { 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`.

See Sections 6.1.2 and 6.2.2 for more information on the 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.

`program_body`

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

`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.

statement

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

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

3.3.23 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
        | 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`.

The user that creates the procedure becomes the owner of the procedure. See Section 4.2.3 for more information about procedures.

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, or 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 LEAKPROOK 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 INVOKER. 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.

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, 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

3.3.24 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 9.5, 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;
```

3.3.25 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

3.3.26 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;
```

3.3.27 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 3.5.9.

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

3.3.28 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
- sequences
- stored procedures
- stored functions
- types
- objects that are accessible through a database link
- other synonyms

A synonym is an alternate name that refers to a database object. See Section 2.2.4 for additional information about 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

3.3.29 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 3.2.

`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:

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

`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`.)

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

`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),
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

```
    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

3.3.30 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.

3.3.31 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
  [ 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.

See Section 5 for more information about triggers.

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.

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, or REF CURSOR declaration.

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

3.3.32 CREATE TYPE

Name

`CREATE TYPE` -- define a new user-defined type

Synopsis

```
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
  }
```

```
CREATE [ OR REPLACE ] TYPE name { IS | AS } TABLE OF
  { datatype | objtype | collecttype }
```

```
CREATE [ OR REPLACE ] TYPE name { IS | AS }
  { VARRAY | VARYING ARRAY } (maxsize) OF { datatype | objtype }
```

Description

`CREATE TYPE` defines a new user-defined data type. The types that can be created are an object type, a nested table type, or a varray type. (Nested table and varray types belong to the category of types known as *collections*. See Section 4.10 for information on collections.) `CREATE OR REPLACE TYPE` will either create a new type definition, or replace an existing type definition.

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.

See Section 4.10.2 for more information on nested table types. See Section 4.10.3 for more information on varray types. See Section 8 for more information on object types.

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.

`datatype`

The data type that defines an attribute of the object 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

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
);
```

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);
```

See Also

CREATE TYPE BODY, DROP TYPE

3.3.33 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`.

See Section 8.2.2 for more information on the object 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

3.3.34 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

3.3.35 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:

http://www.postgresql.org/docs/9.5/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.

For more information about Profile Management, see Section 2.3.

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`.

3.3.36 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

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. See the `CREATE RULE` command in the Advanced Server documentation set.

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

3.3.37 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. See Section 3.4 for information on optimizer hints.

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

3.3.38 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

3.3.39 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

3.3.40 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

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.41 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

3.3.42 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

3.3.43 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

3.3.44 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.

3.3.45 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. See Section 2.2.4 for additional information about 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.

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;
```

3.3.46 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

3.3.47 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

3.3.48 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

3.3.49 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

3.3.50 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

3.3.51 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. 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

3.3.52 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

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.53 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:

http://www.postgresql.org/docs/9.5/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

3.3.54 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);
```

3.3.55 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.56 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.

3.3.57 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

3.3.58 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. See Section 7.11 for information about DBMS_RLS security policies.

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. For more information about using the `ALTER ROLE` command to manage database link privileges, see Section 3.3.3, `ALTER ROLE`.

See Also

REVOKE

3.3.59 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);
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

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;
```

3.3.60 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.61 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. For more information about using the ALTER ROLE command to manage system privileges, see Section 3.3.3, ALTER ROLE.

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

3.3.62 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

3.3.63 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

3.3.64 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

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.65 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.

3.3.65.1 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, `JOIN`s 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)
```

3.3.65.2 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)
```

3.3.65.3 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 3.3.65.3 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)
```

3.3.65.4 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)
```

3.3.65.5 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)
```

3.3.65.6 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`.

3.3.65.7 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)`.

3.3.65.8 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`.

3.3.65.9 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.

(See Section 2.2.5 for additional information on hierarchical queries.)

3.3.65.10 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.

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)
```

3.3.65.11 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.

3.3.65.12 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.

3.3.66 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.3.67 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;
```

3.3.68 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.

3.3.69 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

3.3.70 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. See Section 3.4 for information on optimizer hints.

table

The name (optionally schema-qualified) of the table to update.

dblink

Database link name identifying a remote database. See the CREATE DATABASE LINK 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';
```

3.4 Optimizer Hints

When you invoke a `DELETE`, `INSERT`, `SELECT` or `UPDATE` command, the server generates a set of execution plans; after analyzing those execution plans, the server selects a plan that will (generally) return the result set in the least amount of time. The server's choice of plan is dependent upon several factors:

- The estimated execution cost of data handling operations.
- Parameter values assigned to parameters in the `Query Tuning` section of the `postgresql.conf` file.
- Column statistics that have been gathered by the `ANALYZE` command.

As a rule, the query planner will select the least expensive plan. You can use an *optimizer hint* to influence the server as it selects a query plan.

An optimizer hint is a directive (or multiple directives) embedded in a comment-like syntax that immediately follows a `DELETE`, `INSERT`, `SELECT` or `UPDATE` command. Keywords in the comment instruct the server to employ or avoid a specific plan when producing the result set.

Synopsis

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

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

Optimizer hints may be included in either of the forms shown above. Note that in both forms, a plus sign (+) must immediately follow the `/*` or `--` opening comment symbols, with no intervening space, or the server will not interpret the following tokens as hints.

If you are using the first form, the hint and optional comment may span multiple lines. The second form requires all hints and comments to occupy a single line; the remainder of the statement must start on a new line.

Description

Please Note:

- The database server will always try to use the specified hints if at all possible.
- If a planner method parameter is set so as to disable a certain plan type, then this plan will not be used even if it is specified in a hint, unless there are no other possible options for the planner. Examples of planner method parameters are

enable_indexscan, enable_seqscan, enable_hashjoin, enable_mergejoin, and enable_nestloop. These are all Boolean parameters.

- Remember that the hint is embedded within a comment. As a consequence, if the hint is misspelled or if any parameter to a hint such as view, table, or column name is misspelled, or non-existent in the SQL command, there will be no indication that any sort of error has occurred. No syntax error will be given and the entire hint is simply ignored.
- If an alias is used for a table or view name in the SQL command, then the alias name, not the original object name, must be used in the hint. For example, in the command, `SELECT /*+ FULL(acct) */ * FROM accounts acct ...`, `acct`, the alias for `accounts`, must be specified in the `FULL` hint, not the table name, `accounts`.

Use the `EXPLAIN` command to ensure that the hint is correctly formed and the planner is using the hint. See the Advanced Server documentation set for information on the `EXPLAIN` command.

- In general, optimizer hints should not be used in production applications. Typically, the table data changes throughout the life of the application. By ensuring that the more dynamic columns are `ANALYZE`d frequently, the column statistics will be updated to reflect value changes and the planner will use such information to produce the least cost plan for any given command execution. Use of optimizer hints defeats the purpose of this process and will result in the same plan regardless of how the table data changes.

Parameters

hint

An optimizer hint directive.

comment

A string with additional information. Note that there are restrictions as to what characters may be included in the comment. Generally, *comment* may only consist of alphabetic, numeric, the underscore, dollar sign, number sign and space characters. These must also conform to the syntax of an identifier. See Section 3.1.2 for more information on identifiers. Any subsequent hint will be ignored if the comment is not in this form.

statement_body

The remainder of the `DELETE`, `INSERT`, `SELECT`, or `UPDATE` command.

The following sections describe the optimizer hint directives in more detail.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.4.1 Default Optimization Modes

There are a number of optimization modes that can be chosen as the default setting for an Advanced Server database cluster. This setting can also be changed on a per session basis by using the `ALTER SESSION` command as well as in individual `DELETE`, `SELECT`, and `UPDATE` commands within an optimizer hint. The configuration parameter that controls these default modes is named `OPTIMIZER_MODE`. The following table shows the possible values.

Table 3-3-10 Default Optimization Modes

Hint	Description
`ALL_ROWS`	Optimizes for retrieval of all rows of the result set.
`CHOOSE`	Does no default optimization based on assumed number of rows to be retrieved from the result set. This is the default.
`FIRST_ROWS`	Optimizes for retrieval of only the first row of the result set.
`FIRST_ROWS_10`	Optimizes for retrieval of the first 10 rows of the results set.
`FIRST_ROWS_100`	Optimizes for retrieval of the first 100 rows of the result set.
`FIRST_ROWS_1000`	Optimizes for retrieval of the first 1000 rows of the result set.
`FIRST_ROWS(`*n*`)`	Optimizes for retrieval of the first *n* rows of the result set. This form may not be used as the object of the `ALTER SESSION SET OPTIMIZER_MODE` command. It may only be used in the form of a hint in a SQL command.

These optimization modes are based upon the assumption that the client submitting the SQL command is interested in viewing only the first "n" rows of the result set and will then abandon the remainder of the result set. Resources allocated to the query are adjusted as such.

Examples

Alter the current session to optimize for retrieval of the first 10 rows of the result set.

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_10;
```

The current value of the `OPTIMIZER_MODE` parameter can be shown by using the `SHOW` command. Note that this command is a utility dependent command. In PSQL, the `SHOW` command is used as follows:

```
SHOW OPTIMIZER_MODE;

optimizer_mode
----------------
 first_rows_10
(1 row)
```

The SHOW command, compatible with Oracle databases, has the following syntax:

```
SHOW PARAMETER OPTIMIZER_MODE;

NAME
-------------------------------------------------
VALUE
-------------------------------------------------
optimizer_mode
first_rows_10
```

The following example shows an optimization mode used in a SELECT command as a hint:

```
SELECT /*+ FIRST_ROWS(7) */ * FROM emp;

 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)
```

3.4.2 Access Method Hints

The following hints influence how the optimizer accesses relations to create the result set.

Table 3-3-11 Access Method Hints

Hint	Description
`FULL(table)`	Perform a full sequential scan on *table*.
`INDEX(table [ index ] [...])`	Use *index* on *table* to access the relation.
`NO_INDEX(table [ index ] [...])`	Do not use *index* on *table* to access the relation.

In addition, the `ALL_ROWS`, `FIRST_ROWS`, and `FIRST_ROWS(n)` hints of Table 3-3-10 can be used.

Examples

The sample application does not have sufficient data to illustrate the effects of optimizer hints so the remainder of the examples in this section will use a banking database created by the `pgbench` application located in the `PostgresPlus\9.5AS\bin` subdirectory.

The following steps create a database named, `bank`, populated by the tables, `accounts`, `branches`, `tellers`, and `history`. The `-s 5` option specifies a scaling factor of five which results in the creation of five branches, each with 100,000 accounts, resulting in a total of 500,000 rows in the `accounts` table and five rows in the `branches` table. Ten tellers are assigned to each branch resulting in a total of 50 rows in the `tellers` table.

Note, if using Linux use the `export` command instead of the `SET PATH` command as shown below.

```
export PATH=/opt/PostgresPlus/9.5AS/bin:$PATH
```

The following example was run in Windows.

```
SET PATH=C:\PostgresPlus\9.5AS\bin;%PATH%

createdb -U enterprisedb bank
CREATE DATABASE

pgbench -i -s 5 -U enterprisedb -d bank

creating tables...
10000 tuples done.
20000 tuples done.
30000 tuples done.
        .
        .
        .
470000 tuples done.
480000 tuples done.
```

```
490000 tuples done.
500000 tuples done.
set primary key...
vacuum...done.
```

Ten transactions per client are then processed for eight clients for a total of 80 transactions. This will populate the `history` table with 80 rows.

```
pgbench -U enterprisedb -d bank -c 8 -t 10
        .
        .
        .
transaction type: TPC-B (sort of)
scaling factor: 5
number of clients: 8
number of transactions per client: 10
number of transactions actually processed: 80/80
tps = 6.023189 (including connections establishing)
tps = 7.140944 (excluding connections establishing)
```

The table definitions are shown below:

```
\d accounts

       Table "public.accounts"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 aid      | integer       | not null
 bid      | integer       |
 abalance | integer       |
 filler   | character(84) |
Indexes:
    "accounts_pkey" PRIMARY KEY, btree (aid)

\d branches

       Table "public.branches"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 bid      | integer       | not null
 bbalance | integer       |
 filler   | character(88) |
Indexes:
    "branches_pkey" PRIMARY KEY, btree (bid)

\d tellers

        Table "public.tellers"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 tid      | integer       | not null
 bid      | integer       |
 tbalance | integer       |
 filler   | character(84) |
Indexes:
    "tellers_pkey" PRIMARY KEY, btree (tid)

\d history

              Table "public.history"
 Column |            Type             | Modifiers
--------+-----------------------------+-----------
 tid    | integer                     |
 bid    | integer                     |
 aid    | integer                     |
 delta  | integer                     |
 mtime  | timestamp without time zone |
 filler | character(22)               |
```

The EXPLAIN command shows the plan selected by the query planner. In the following example, aid is the primary key column, so an indexed search is used on index, accounts_pkey.

```
EXPLAIN SELECT * FROM accounts WHERE aid = 100;

                                  QUERY PLAN
-----------------------------------------------------------------------------
--
 Index Scan using accounts_pkey on accounts  (cost=0.00..8.32 rows=1
width=97)
   Index Cond: (aid = 100)
(2 rows)
```

The FULL hint is used to force a full sequential scan instead of using the index as shown below:

```
EXPLAIN SELECT /*+ FULL(accounts) */ * FROM accounts WHERE aid = 100;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on accounts  (cost=0.00..14461.10 rows=1 width=97)
   Filter: (aid = 100)
(2 rows)
```

The NO_INDEX hint also forces a sequential scan as shown below:

```
EXPLAIN SELECT /*+ NO_INDEX(accounts accounts_pkey) */ * FROM accounts WHERE
aid = 100;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on accounts  (cost=0.00..14461.10 rows=1 width=97)
   Filter: (aid = 100)
(2 rows)
```

In addition to using the EXPLAIN command as shown in the prior examples, more detailed information regarding whether or not a hint was used by the planner can be obtained by setting the client_min_messages and trace_hints configuration parameters as follows:

```
SET client_min_messages TO info;
SET trace_hints TO true;
```

The SELECT command with the NO_INDEX hint is repeated below to illustrate the additional information produced when the aforementioned configuration parameters are set.

```
EXPLAIN SELECT /*+ NO_INDEX(accounts accounts_pkey) */ * FROM accounts WHERE
aid = 100;

INFO:  [HINTS] Index Scan of [accounts].[accounts_pkey] rejected because of
NO_INDEX hint.

INFO:  [HINTS] Bitmap Heap Scan of [accounts].[accounts_pkey] rejected
because of NO_INDEX hint.
                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on accounts  (cost=0.00..14461.10 rows=1 width=97)
   Filter: (aid = 100)
(2 rows)
```

Note that if a hint is ignored, the INFO: [HINTS] line will not appear. This may be an indication that there was a syntax error or some other misspelling in the hint as shown in the following example where the index name is misspelled.

```
EXPLAIN SELECT /*+ NO_INDEX(accounts accounts_xxx) */ * FROM accounts WHERE
aid = 100;

                                  QUERY PLAN
-----------------------------------------------------------------------------
--
 Index Scan using accounts_pkey on accounts  (cost=0.00..8.32 rows=1
 width=97)
   Index Cond: (aid = 100)
(2 rows
```

3.4.3 Specifying a Join Order

Include the ORDERED directive to instruct the query optimizer to join tables in the order in which they are listed in the FROM clause. If you do not include the ORDERED keyword, the query optimizer will choose the order in which to join the tables.

For example, the following command allows the optimizer to choose the order in which to join the tables listed in the FROM clause:

```
SELECT e.ename, d.dname, h.startdate
  FROM emp e, dept d, jobhist h
  WHERE d.deptno = e.deptno
  AND h.empno = e.empno;
```

The following command instructs the optimizer to join the tables in the ordered specified:

```
SELECT /*+ ORDERED */ e.ename, d.dname, h.startdate
  FROM emp e, dept d, jobhist h
  WHERE d.deptno = e.deptno
  AND h.empno = e.empno;
```

In the ORDERED version of the command, Advanced Server will first join emp e with dept d before joining the results with jobhist h. Without the ORDERED directive, the join order is selected by the query optimizer.

Please note: the ORDERED directive does not work for Oracle-style outer joins (those joins that contain a '+' sign).

3.4.4 Joining Relations Hints

When two tables are to be joined, there are three possible plans that may be used to perform the join.

- *Nested Loop Join* – The right table is scanned once for every row in the left table.
- *Merge Sort Join* – Each table is sorted on the join attributes before the join starts. The two tables are then scanned in parallel and the matching rows are combined to form the join rows.
- *Hash Join* – The right table is scanned and its join attributes are loaded into a hash table using its join attributes as hash keys. The left table is then scanned and its join attributes are used as hash keys to locate the matching rows from the right table.

The following table lists the optimizer hints that can be used to influence the planner to use one type of join plan over another.

Table 3-3-12 Join Hints

Hint	Description
`USE_HASH(`*`table`*` [...])`	Use a hash join with a hash table created from the join attributes of *`table`*.
`NO_USE_HASH(`*`table`*` [...])`	Do not use a hash join created from the join attributes of *`table`*.
`USE_MERGE(`*`table`*` [...])`	Use a merge sort join for *`table`*.
`NO_USE_MERGE(`*`table`*` [...])`	Do not use a merge sort join for *`table`*.
`USE_NL(`*`table`*` [...])`	Use a nested loop join for *`table`*.
`NO_USE_NL(`*`table`*` [...])`	Do not use a nested loop join for *`table`*.

Examples

In the following example, a join is performed on the `branches` and `accounts` tables. The query plan shows that a hash join is used by creating a hash table from the join attribute of the `branches` table.

```
EXPLAIN SELECT b.bid, a.aid, abalance FROM branches b, accounts a WHERE b.bid
= a.bid;

                                QUERY PLAN
--------------------------------------------------------------------------
 Hash Join  (cost=1.11..20092.70 rows=500488 width=12)
   Hash Cond: (a.bid = b.bid)
   ->  Seq Scan on accounts a  (cost=0.00..13209.88 rows=500488 width=12)
   ->  Hash  (cost=1.05..1.05 rows=5 width=4)
         ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
(5 rows)
```

By using the USE_HASH(a) hint, the planner is forced to create the hash table from the accounts join attribute instead of from the branches table. Note the use of the alias, a, for the accounts table in the USE_HASH hint.

```
EXPLAIN SELECT /*+ USE_HASH(a) */ b.bid, a.aid, abalance FROM branches b,
accounts a WHERE b.bid = a.bid;

                                   QUERY PLAN
-----------------------------------------------------------------------------
---
 Hash Join  (cost=21909.98..30011.52 rows=500488 width=12)
   Hash Cond: (b.bid = a.bid)
   ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
   ->  Hash  (cost=13209.88..13209.88 rows=500488 width=12)
         ->  Seq Scan on accounts a  (cost=0.00..13209.88 rows=500488
width=12)
(5 rows)
```

Next, the NO_USE_HASH(a b) hint forces the planner to use an approach other than hash tables. The result is a nested loop.

```
EXPLAIN SELECT /*+ NO_USE_HASH(a b) */ b.bid, a.aid, abalance FROM branches
b, accounts a WHERE b.bid = a.bid;

                                QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop  (cost=1.05..69515.84 rows=500488 width=12)
   Join Filter: (b.bid = a.bid)
   ->  Seq Scan on accounts a  (cost=0.00..13209.88 rows=500488 width=12)
   ->  Materialize  (cost=1.05..1.11 rows=5 width=4)
         ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
(5 rows)
```

Finally, the USE_MERGE hint forces the planner to use a merge join.

```
EXPLAIN SELECT /*+ USE_MERGE(a) */ b.bid, a.aid, abalance FROM branches b,
accounts a WHERE b.bid = a.bid;

                                   QUERY PLAN
-----------------------------------------------------------------------------
---
 Merge Join  (cost=69143.62..76650.97 rows=500488 width=12)
   Merge Cond: (b.bid = a.bid)
   ->  Sort  (cost=1.11..1.12 rows=5 width=4)
         Sort Key: b.bid
         ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
   ->  Sort  (cost=69142.52..70393.74 rows=500488 width=12)
         Sort Key: a.bid
         ->  Seq Scan on accounts a  (cost=0.00..13209.88 rows=500488
width=12)
(8 rows)
```

In this three-table join example, the planner first performs a hash join on the branches and history tables, then finally performs a nested loop join of the result with the accounts_pkey index of the accounts table.

```
EXPLAIN SELECT h.mtime, h.delta, b.bid, a.aid FROM history h, branches b,
accounts a WHERE h.bid = b.bid AND h.aid = a.aid;

                                      QUERY PLAN
-----------------------------------------------------------------------------
---------
 Nested Loop  (cost=1.11..207.95 rows=26 width=20)
   ->  Hash Join  (cost=1.11..25.40 rows=26 width=20)
         Hash Cond: (h.bid = b.bid)
         ->  Seq Scan on history h  (cost=0.00..20.20 rows=1020 width=20)
         ->  Hash  (cost=1.05..1.05 rows=5 width=4)
               ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
   ->  Index Scan using accounts_pkey on accounts a  (cost=0.00..7.01 rows=1
       width=4)
         Index Cond: (h.aid = a.aid)
(8 rows)
```

This plan is altered by using hints to force a combination of a merge sort join and a hash join.

```
EXPLAIN SELECT /*+ USE_MERGE(h b) USE_HASH(a) */ h.mtime, h.delta, b.bid,
a.aid FROM history h, branches b, accounts a WHERE h.bid = b.bid AND h.aid =
a.aid;

                                        QUERY PLAN
-----------------------------------------------------------------------------
--------------
 Merge Join  (cost=23480.11..23485.60 rows=26 width=20)
   Merge Cond: (h.bid = b.bid)
   ->  Sort  (cost=23479.00..23481.55 rows=1020 width=20)
         Sort Key: h.bid
         ->  Hash Join  (cost=21421.98..23428.03 rows=1020 width=20)
               Hash Cond: (h.aid = a.aid)
               ->  Seq Scan on history h  (cost=0.00..20.20 rows=1020
                   width=20)
               ->  Hash  (cost=13209.88..13209.88 rows=500488 width=4)
                     ->  Seq Scan on accounts a  (cost=0.00..13209.88
                         rows=500488 width=4)
   ->  Sort  (cost=1.11..1.12 rows=5 width=4)
         Sort Key: b.bid
         ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
(12 rows)
```

3.4.5 Global Hints

Thus far, hints have been applied directly to tables that are referenced in the SQL command. It is also possible to apply hints to tables that appear in a view when the view is referenced in the SQL command. The hint does not appear in the view, itself, but rather in the SQL command that references the view.

When specifying a hint that is to apply to a table within a view, the view and table names are given in dot notation within the hint argument list.

Synopsis

hint(*view*.*table*)

Parameters

hint

Any of the hints in Table 3-3-11 or Table 3-3-12.

view

The name of the view containing *table*.

table

The table on which the hint is to be applied.

Examples

A view named, `tx`, is created from the three-table join of `history`, `branches`, and `accounts` shown in the final example of Section 3.4.3.

```
CREATE VIEW tx AS SELECT h.mtime, h.delta, b.bid, a.aid FROM history h,
branches b, accounts a WHERE h.bid = b.bid AND h.aid = a.aid;
```

The query plan produced by selecting from this view is show below:

```
EXPLAIN SELECT * FROM tx;

                                      QUERY PLAN
-----------------------------------------------------------------------------
---------
 Nested Loop  (cost=1.11..207.95 rows=26 width=20)
   ->  Hash Join  (cost=1.11..25.40 rows=26 width=20)
         Hash Cond: (h.bid = b.bid)
         ->  Seq Scan on history h  (cost=0.00..20.20 rows=1020 width=20)
         ->  Hash  (cost=1.05..1.05 rows=5 width=4)
```

```
               ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
   ->  Index Scan using accounts_pkey on accounts a  (cost=0.00..7.01 rows=1
       width=4)
         Index Cond: (h.aid = a.aid)
(8 rows)
```

The same hints that were applied to this join at the end of Section 3.4.3 can be applied to the view as follows:

```
EXPLAIN SELECT /*+ USE_MERGE(tx.h tx.b) USE_HASH(tx.a) */ * FROM tx;

                                        QUERY PLAN

-----------------------------------------------------------------------------
-------------
-
 Merge Join  (cost=23480.11..23485.60 rows=26 width=20)
   Merge Cond: (h.bid = b.bid)
   ->  Sort  (cost=23479.00..23481.55 rows=1020 width=20)
         Sort Key: h.bid
         ->  Hash Join  (cost=21421.98..23428.03 rows=1020 width=20)
               Hash Cond: (h.aid = a.aid)
               ->  Seq Scan on history h  (cost=0.00..20.20 rows=1020
                   width=20)
               ->  Hash  (cost=13209.88..13209.88 rows=500488 width=4)
                     ->  Seq Scan on accounts a  (cost=0.00..13209.88
                         rows=500488 width=4)
   ->  Sort  (cost=1.11..1.12 rows=5 width=4)
         Sort Key: b.bid
         ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
(12 rows)
```

In addition to applying hints to tables within stored views, hints can be applied to tables within subqueries as illustrated by the following example. In this query on the sample application `emp` table, employees and their managers are listed by joining the `emp` table with a subquery of the `emp` table identified by the alias, `b`.

```
SELECT a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename" FROM emp a,
(SELECT * FROM emp) b WHERE a.mgr = b.empno;

empno | ename  | mgr empno | mgr ename
-------+--------+-----------+-----------
  7902 | FORD   |      7566 | JONES
  7788 | SCOTT  |      7566 | JONES
  7521 | WARD   |      7698 | BLAKE
  7844 | TURNER |      7698 | BLAKE
  7654 | MARTIN |      7698 | BLAKE
  7900 | JAMES  |      7698 | BLAKE
  7499 | ALLEN  |      7698 | BLAKE
  7934 | MILLER |      7782 | CLARK
  7876 | ADAMS  |      7788 | SCOTT
  7782 | CLARK  |      7839 | KING
  7698 | BLAKE  |      7839 | KING
  7566 | JONES  |      7839 | KING
  7369 | SMITH  |      7902 | FORD
(13 rows)
```

The plan chosen by the query planner is shown below:

```
EXPLAIN SELECT a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename"
FROM emp a, (SELECT * FROM emp) b WHERE a.mgr = b.empno;

                            QUERY PLAN
-------------------------------------------------------------------
 Merge Join  (cost=2.81..3.08 rows=13 width=26)
   Merge Cond: (a.mgr = emp.empno)
   ->  Sort  (cost=1.41..1.44 rows=14 width=20)
         Sort Key: a.mgr
         ->  Seq Scan on emp a  (cost=0.00..1.14 rows=14 width=20)
   ->  Sort  (cost=1.41..1.44 rows=14 width=13)
         Sort Key: emp.empno
         ->  Seq Scan on emp  (cost=0.00..1.14 rows=14 width=13)
(8 rows)
```

A hint can be applied to the `emp` table within the subquery to perform an index scan on index, `emp_pk`, instead of a table scan. Note the difference in the query plans.

```
EXPLAIN SELECT /*+ INDEX(b.emp emp_pk) */ a.empno, a.ename, b.empno "mgr
empno", b.ename "mgr ename" FROM emp a, (SELECT * FROM emp) b WHERE a.mgr =
b.empno;

                                QUERY PLAN
---------------------------------------------------------------------------
 Merge Join  (cost=1.41..13.21 rows=13 width=26)
   Merge Cond: (a.mgr = emp.empno)
   ->  Sort  (cost=1.41..1.44 rows=14 width=20)
         Sort Key: a.mgr
         ->  Seq Scan on emp a  (cost=0.00..1.14 rows=14 width=20)
   ->  Index Scan using emp_pk on emp  (cost=0.00..12.46 rows=14 width=13)
(6 rows)
```

3.4.6 Using the APPEND Optimizer Hint

By default, Advanced Server will add new data into the first available free-space in a table (vacated by vacuumed records). Include the `APPEND` directive after an `INSERT` or `SELECT` command to instruct the server to bypass mid-table free space, and affix new rows to the end of the table. This optimizer hint can be particularly useful when bulk loading data.

The syntax is:

```
/*+APPEND*/
```

For example, the following command, compatible with Oracle databases, instructs the server to append the data in the `INSERT` statement to the end of the `sales` table:

```
INSERT /*+APPEND*/ INTO sales VALUES
(10, 10, '01-Mar-2011', 10, 'OR');
```

Note that Advanced Server supports the `APPEND` hint when adding multiple rows in a single `INSERT` statement:

```
INSERT /*+APPEND*/ INTO sales VALUES
(20, 20, '01-Aug-2011', 20, 'NY'),
(30, 30, '01-Feb-2011', 30, 'FL'),
(40, 40, '01-Nov-2011', 40, 'TX');
```

The `APPEND` hint can also be included in the `SELECT` clause of an `INSERT INTO` statement:

```
INSERT INTO sales_history SELECT /*+APPEND*/ FROM sales;
```

3.4.7 Conflicting Hints

If a command includes two or more conflicting hints, the server will ignore the contradictory hints. The following table lists hints that are contradictory to each other.

Table 3-3-13 Conflicting Hints

Hint	Conflicting Hint
`ALL_ROWS`	`FIRST_ROWS` - all formats
`FULL(`*`table`*`)`	`INDEX(`*`table`*` [ `*`index`*` ])`

`INDEX(`*`table`*`)`	`FULL(`*`table`*`)`
	`NO_INDEX(`*`table`*`)`

`INDEX(`*`table index`*`)`	`FULL(`*`table`*`)`
	`NO_INDEX(`*`table index`*`)`

`USE_HASH(`*`table`*`)`	`NO_USE_HASH(`*`table`*`)`
`USE_MERGE(`*`table`*`)`	`NO_USE_MERGE(`*`table`*`)`
`USE_NL(`*`table`*`)`	`NO_USE_NL(`*`table`*`)`

3.5 Functions and Operators

Advanced Server provides a large number of functions and operators for the built-in data types.

3.5.1 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 3-3-14 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 3-3-15 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.

3.5.2 Comparison Operators

The usual comparison operators are shown in the following table.

Table 3-3-16 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.5.3 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 3-3-17 Mathematical Operators

Operator	Description	Example	Result
`+`	Addition	`2 + 3`	`5`
`-`	Subtraction	`2 - 3`	`-1`
`*`	Multiplication	`2 * 3`	`6`
`/`	Division (integer division truncates results)	`4 / 2`	`2`
`**`	Exponentiation Operator	`2 ** 3`	`8`

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 3-3-18 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)`	`2.7182818284590452`
`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)`	`0.6931471805599453`
`LOG(`*b* `NUMBER,` *x* `NUMBER)`	`NUMBER`	Logarithm to base *b*	`LOG(2.0, 64.0)`	`6.0000000000000000`
`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 the same data type	If *x* is null, then NVL returns *y*	`NVL(9, 0)`	`9`
`POWER(`*a* `DOUBLE PRECISION,` *b* `DOUBLE PRECISION)`	`DOUBLE PRECISION`	*a* raised to the power of *b*	`POWER(9.0, 3.0)`	`729.0000000000000000`

Function	Return Type	Description	Example	Result
`POWER(`*a* `NUMBER,` *b* `NUMBER)`	`NUMBER`	*a* raised to the power of *b*	`POWER(9.0, 3.0)`	`729.0000000000000000`
`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 3-3-19 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

3.5.4 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 3-3-20 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 3-3-20.

Table 3-3-21 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`

3.5.5 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:

http://www.postgresql.org/docs/9.5/static/functions-matching.html

3.5.5.1 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:
>
> http://www.postgresql.org/docs/9.5/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 occurances of the letter i that occur before the 6th position.

3.5.5.2 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:

http://www.postgresql.org/docs/9.5/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.

3.5.5.3 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:

http://www.postgresql.org/docs/9.5/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.

3.5.6 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.

3.5.7 Data Type Formatting Functions

The Advanced Server formatting functions (described in Table 3-3-22) 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 3-3-22 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 3-3-23 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 3-3-24 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 3-3-25 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 3-3-26 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'`
`TO_CHAR(0.1, '0.9')`	`' 0.1'`
`TO_CHAR(12, '9990999.9')`	`'    0012.0'`

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

Expression	Result
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'

3.5.7.1 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:

http://www.postgresql.org/docs/9.5/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
```

3.5.8 Date/Time Functions and Operators

Table 3-3-28 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 3-3-27 illustrates the behaviors of the basic arithmetic operators (+, -). For formatting functions, refer to Section 3.5.7. You should be familiar with the background information on date/time data types from Section 3.2.4.

Table 3-3-27 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 Table 3-3-28 the use of the `DATE` and `TIMESTAMP` data types are interchangeable.

Table 3-3-28 Date/Time Functions

Function	Return Type	Description	Example	Result
`ADD_MONTHS(DATE, NUMBER)`	`DATE`	Add months to a date; see Section 3.5.8.1	`ADD_MONTHS('28-FEB-97', 3.8)`	`31-MAY-97 00:00:00`
`CURRENT_DATE`	`DATE`	Current date; see Section 3.5.8.8	`CURRENT_DATE`	`04-JUL-07`
`CURRENT_TIMESTAMP`	`TIMESTAMP`	Returns the current date and time; see Section 3.5.8.8	`CURRENT_TIMESTAMP`	`04-JUL-07 15:33:23.484`
`EXTRACT(`*`field`*` FROM TIMESTAMP)`	`DOUBLE PRECISION`	Get subfield; see Section 3.5.8.2	`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 3.5.8.8	`LOCALTIMESTAMP`	`04-JUL-07 15:33:23.484`
`MONTHS_BETWEEN(DATE, DATE)`	`NUMBER`	Number of months between two dates; see Section 3.5.8.3	`MONTHS_BETWEEN('28-FEB-07', '30-NOV-06')`	`3`

Function	Return Type	Description	Example	Result
`NEXT_DAY(DATE, dayofweek)`	`DATE`	Date falling on *dayofweek* following specified date; see Section 3.5.8.4	`NEXT_DAY('16-APR-07','FRI')`	`20-APR-07 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 3.5.8.9.	`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 3.5.8.10.	`SELECT numtoyminterval(100, 'month');`	`8 years 4 mons`
`ROUND(DATE [, format ])`	`DATE`	Date rounded according to *format*; see Section 3.5.8.6	`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 3.5.8.7	`TRUNC(TO_DATE('29-MAY-05'), 'MON')`	`01-MAY-05 00:00:00`

3.5.8.1 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)
```

3.5.8.2 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)
```

3.5.8.3 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)
```

3.5.8.4 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)
```

3.5.8.5 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:

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)
```

3.5.8.6 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 3-3-29 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, YYY, YY, Y	Returns January 1, *yyyy* where *yyyy* is rounded to the nearest year; rounds down on June 30, rounds up on July 1
IYYY, IYY, IY, I	Rounds to the beginning of the ISO year which is determined by rounding down if

Pattern	Description
	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)
```

3.5.8.7 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 3-3-30 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)
```

3.5.8.8 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.

3.5.8.9 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)
```

3.5.8.10 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)
```

3.5.9 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.

3.5.10 Conditional Expressions

The following section describes the SQL-compliant conditional expressions available in Advanced Server.

3.5.10.1 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;
```

3.5.10.2 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.

3.5.10.3 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`*.

3.5.10.4 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)
```

3.5.10.5 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)
```

3.5.10.6 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.

3.5.11 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 3-3-31 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 3-3-32 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)

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

3.5.12 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.

3.5.12.1 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)
```

3.5.12.2 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.

3.5.12.3 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.

3.5.12.4 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.

3.5.12.5 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.

4 Stored Procedure Language

This chapter describes the Stored Procedure Language (SPL). SPL is a highly productive, procedural programming language for writing custom procedures, functions, triggers, and packages for Advanced Server that provides:

- full procedural programming functionality to complement the SQL language
- a single, common language to create stored procedures, functions, triggers, and packages for the Advanced Server database
- a seamless development and testing environment
- the use of reusable code
- ease of use

This chapter describes the basic elements of an SPL program, before providing an overview of the organization of an SPL program and how it is used to create a procedure or a function. Triggers, while still utilizing SPL, are sufficiently different to warrant a separate discussion (see Section 5 for information about triggers). Packages are discussed in Section 6.

The remaining sections of this chapter delve into the details of the SPL language and provide examples of its application.

4.1 Basic SPL Elements

This section discusses the basic programming elements of an SPL program.

4.1.1 Character Set

SPL programs are written using the following set of characters:

- Uppercase letters `A` thru `Z` and lowercase letters `a` thru `z`
- Digits `0` thru `9`
- Symbols `( ) + - * / < > = ! ~ ^ ; : . ' @ % , " # $ & _ | { } ? [ ]`
- White space characters tabs, spaces, and carriage returns

Identifiers, expressions, statements, control structures, etc. that comprise the SPL language are written using these characters.

Note: The data that can be manipulated by an SPL program is determined by the character set supported by the database encoding.

4.1.2 Case Sensitivity

Keywords and user-defined identifiers that are used in an SPL program are case insensitive. So for example, the statement `DBMS_OUTPUT.PUT_LINE('Hello World');` is interpreted to mean the same thing as `dbms_output.put_line('Hello World');` or `Dbms_Output.Put_Line('Hello World');` or `DBMS_output.Put_line('Hello World');`.

Character and string constants, however, are case sensitive as well as any data retrieved from the Advanced Server database or data obtained from other external sources. The statement `DBMS_OUTPUT.PUT_LINE('Hello World!');` produces the following output:

```
Hello World!
```

However the statement `DBMS_OUTPUT.PUT_LINE('HELLO WORLD!');` produces the output:

```
HELLO WORLD!
```

4.1.3 Identifiers

Identifiers are user-defined names that are used to identify various elements of an SPL program including variables, cursors, labels, programs, and parameters.

The syntax rules for valid identifiers are the same as for identifiers in the SQL language. See Section 3.1.2 for a discussion of SQL identifiers.

An identifier must not be the same as an SPL keyword or a keyword of the SQL language. The following are some examples of valid identifiers:

```
x
last___name
a_$_Sign
Many$$$$$$$$signs_____
THIS_IS_AN_EXTREMELY_LONG_NAME
A1
```

4.1.4 Qualifiers

A *qualifier* is a name that specifies the owner or context of an entity that is the object of the qualification. A qualified object is specified as the qualifier name followed by a dot with no intervening white space, followed by the name of the object being qualified with no intervening white space. This syntax is called *dot notation*.

The following is the syntax of a qualified object.

```
    qualifier. [ qualifier. ]... object
```

`qualifier` is the name of the owner of the object. *`object`* is the name of the entity belonging to *`qualifier`*. It is possible to have a chain of qualifications where the preceding qualifier owns the entity identified by the subsequent qualifier(s) and object.

Almost any identifier can be qualified. What an identifier is qualified by depends upon what the identifier represents and the context of its usage.

Some examples of qualification follow:

- Procedure and function names qualified by the schema to which they belong - e.g., *`schema_name.procedure_name`*`(...)`
- Trigger names qualified by the schema to which they belong - e.g., *`schema_name.trigger_name`*
- Column names qualified by the table to which they belong - e.g., `emp.empno`
- Table names qualified by the schema to which they belong - e.g., `public.emp`
- Column names qualified by table and schema - e.g., `public.emp.empno`

As a general rule, wherever a name appears in the syntax of an SPL statement, its qualified name can be used as well. Typically a qualified name would only be used if there is some ambiguity associated with the name. For example, if two procedures with the same name belonging to two different schemas are invoked from within a program or if the same name is used for a table column and SPL variable within the same program.

You should avoid using qualified names if at all possible. In this chapter, the following conventions are adopted to avoid naming conflicts:

- All variables declared in the declaration section of an SPL program are prefixed by `v_`. E.g., `v_empno`
- All formal parameters declared in a procedure or function definition are prefixed by `p_`. E.g., `p_empno`
- Column names and table names do not have any special prefix conventions. E.g., column `empno` in table `emp`

4.1.5 Constants

Constants or *literals* are fixed values that can be used in SPL programs to represent values of various types - e.g., numbers, strings, dates, etc. Constants come in the following types:

- Numeric (Integer and Real) – see Section 3.1.3.2 for information on numeric constants.
- Character and String – see Section 3.1.3.1 for information on character and string constants.
- Date/time – see Section 3.2.4 for information on date/time data types and constants.

4.1.6 User-Defined PL/SQL Subtypes

Advanced Server supports user-defined PL/SQL subtypes and (subtype) aliases. A subtype is a data type with an optional set of constraints that restrict the values that can be stored in a column of that type. The rules that apply to the type on which the subtype is based are still enforced, but you can use additional constraints to place limits on the precision or scale of values stored in the type.

You can define a subtype in the declaration of a PL function, procedure, anonymous block or package. The syntax is:

```
SUBTYPE subtype_name IS type_name[(constraint)] [NOT NULL]
```

Where `constraint` is:

```
{precision [, scale]} | length
```

Where:

subtype_name

> *subtype_name* specifies the name of the subtype.

type_name

> *type_name* specifies the name of the original type on which the subtype is based. *type_name* may be:
>
> - The name of any of the type supported by Advanced Server.
> - The name of any composite type.
> - A column anchored by a `%TYPE` operator.
> - The name of another subtype.

Include the *constraint* clause to define restrictions for types that support precision or scale.

precision

> *precision* specifies the total number of digits permitted in a value of the subtype.

scale

> *scale* specifies the number of fractional digits permitted in a value of the subtype.

length

> *length* specifies the total length permitted in a value of CHARACTER, VARCHAR, or TEXT base types

Include the NOT NULL clause to specify that NULL values may not be stored in a column of the specified subtype.

Note that a subtype that is based on a column will inherit the column size constraints, but the subtype will not inherit NOT NULL or CHECK constraints.

Unconstrained Subtypes

To create an unconstrained subtype, use the SUBTYPE command to specify the new subtype name and the name of the type on which the subtype is based. For example, the following command creates a subtype named address that has all of the attributes of the type, CHAR:

```
SUBTYPE address IS CHAR;
```

You can also create a subtype (constrained or unconstrained) that is a subtype of another subtype:

```
SUBTYPE cust_address IS address NOT NULL;
```

This command creates a subtype named cust_address that shares all of the attributes of the address subtype. Include the NOT NULL clause to specify that a value of the cust_address may not be NULL.

Constrained Subtypes

Include a *length* value when creating a subtype that is based on a character type to define the maximum length of the subtype. For example:

```
SUBTYPE acct_name IS VARCHAR (15);
```

This example creates a subtype named acct_name that is based on a VARCHAR data type, but is limited to 15 characters in length.

Include values for *precision* (to specify the maximum number of digits in a value of the subtype) and optionally, *scale* (to specify the number of digits to the right of the decimal point) when constraining a numeric base type. For example:

```
SUBTYPE acct_balance IS NUMBER (5, 2);
```

This example creates a subtype named `acct_balance` that shares all of the attributes of a `NUMBER` type, but that may not exceed 3 digits to the left of the decimal point and 2 digits to the right of the decimal.

An argument declaration (in a function or procedure header) is a *formal argument*. The value passed to a function or procedure is an *actual argument*. When invoking a function or procedure, the caller provides (0 or more) actual arguments. Each actual argument is assigned to a formal argument that holds the value within the body of the function or procedure.

If a formal argument is declared as a constrained subtype:

- Advanced Server does not enforce subtype constraints when assigning an actual argument to a formal argument when invoking a function.
- Advanced Server enforces subtype constraints when assigning an actual argument to a formal argument when invoking a procedure.

Using the %TYPE Operator

You can use `%TYPE` notation to declare a subtype anchored to a column. For example:

```
SUBTYPE emp_type IS emp.empno%TYPE
```

This command creates a subtype named `emp_type` whose base type matches the type of the `empno` column in the `emp` table. A subtype that is based on a column will share the column size constraints; `NOT NULL` and `CHECK` constraints are not inherited.

Subtype Conversion

Unconstrained subtypes are aliases for the type on which they are based. Any variable of type subtype (unconstrained) is interchangeable with a variable of the base type without conversion, and vice versa.

A variable of a constrained subtype may be interchanged with a variable of the base type without conversion, but a variable of the base type may only be interchanged with a constrained subtype if it complies with the constraints of the subtype. A variable of a constrained subtype may be implicitly converted to another subtype if it is based on the same subtype, and the constraint values are within the values of the subtype to which it is being converted.

4.2 SPL Programs

SPL is a procedural, block-structured language. There are four different types of programs that can be created using SPL, namely *procedures*, *functions*, *triggers*, and *packages*.

Procedures and functions are discussed in more detail later in this section. Triggers are discussed in Section 5 and packages are addressed in Section 6.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

4.2.1 SPL Block Structure

Regardless of whether the program is a procedure, function, or trigger, an SPL program has the same *block* structure. A block consists of up to three sections - an optional declaration section, a mandatory executable section, and an optional exception section. Minimally, a block has an executable section that consists of one or more SPL statements within the keywords, `BEGIN` and `END`.

The optional declaration section is used to declare variables, cursors, and types that are used by the statements within the executable and exception sections. Declarations appear just prior to the `BEGIN` keyword of the executable section. Depending upon the context of where the block is used, the declaration section may begin with the keyword `DECLARE`.

You can include an exception section within the `BEGIN` - `END` block. The exception section begins with the keyword, `EXCEPTION`, and continues until the end of the block in which it appears. If an exception is thrown by a statement within the block, program control goes to the exception section where the thrown exception may or may not be handled depending upon the exception and the contents of the exception section.

The following is the general structure of a block:

```
[ [ DECLARE ]
      declarations ]
    BEGIN
      statements
  [ EXCEPTION
      WHEN exception_condition THEN
        statements [, ...] ]
    END;
```

`declarations` are one or more variable, cursor, or type declarations that are local to the block. Each declaration must be terminated by a semicolon. The use of the keyword `DECLARE` depends upon the context in which the block appears.

`statements` are one or more SPL statements. Each statement must be terminated by a semicolon. The end of the block denoted by the keyword `END` must also be terminated by a semicolon.

If present, the keyword `EXCEPTION` marks the beginning of the exception section. *`exception_condition`* is a conditional expression testing for one or more types of exceptions. If a thrown exception matches one of the exceptions in *`exception_condition`*, the *`statements`* following the `WHEN` *`exception_condition`* clause are executed. There may be one or more `WHEN` *`exception_condition`* clauses, each followed by *`statements`*.

Note: A BEGIN/END block in itself, is considered a statement; thus, blocks may be nested. The exception section may also contain nested blocks.

The following is the simplest possible block consisting of the NULL statement within the executable section. The NULL statement is an executable statement that does nothing.

```
BEGIN
    NULL;
END;
```

The following block contains a declaration section as well as the executable section.

```
DECLARE
    v_numerator     NUMBER(2);
    v_denominator   NUMBER(2);
    v_result        NUMBER(5,2);
BEGIN
    v_numerator := 75;
    v_denominator := 14;
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator ||
        ' is ' || v_result);
END;
```

In this example, three numeric variables are declared of data type NUMBER. In the executable section, values are assigned to two of the variables and then one number is divided by the other, storing the results in a third variable which is then displayed. If executed, the output would be:

```
75 divided by 14 is 5.36
```

The following block consists of a declaration, an executable, and an exception:

```
DECLARE
    v_numerator     NUMBER(2);
    v_denominator   NUMBER(2);
    v_result        NUMBER(5,2);
BEGIN
    v_numerator := 75;
    v_denominator := 0;
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator ||
        ' is ' || v_result);
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('An exception occurred');
END;
```

The following output shows that the statement within the exception section is executed as a result of the division by zero.

```
An exception occurred
```

4.2.2 Anonymous Blocks

Blocks are typically written as part of a procedure, function, or trigger. These programs are named and stored in the database for re-use. For quick (one-time) execution (such as testing), you can simply enter the block without providing a name or storing it in the database.

A block of this type is called an *anonymous block*. An anonymous block is unnamed and is not stored in the database. Once the block has been executed and erased from the application buffer, it cannot be re-executed unless the block code is re-entered into the application.

Typically, the same block of code will be re-executed many times. In order to run a block of code repeatedly without the necessity of re-entering the code each time, with some simple modifications, an anonymous block can be turned into a procedure or function. The following sections discuss how to create a procedure or function that can be stored in the database and invoked repeatedly by another procedure, function, or application program.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

4.2.3 Procedures Overview

Procedures are SPL programs that are invoked or called as an individual SPL program statement. When called, procedures may optionally receive values from the caller in the form of input parameters and optionally return values to the caller in the form of output parameters.

4.2.3.1 Creating a Procedure

The `CREATE PROCEDURE` command defines and names a procedure that will be stored in the database.

```
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
        | COST execution_cost
        | ROWS result_rows
        | SET configuration_parameter
          { TO value | = value | FROM CURRENT }
   ...]
{ IS | AS }
    [ declarations ]
    BEGIN
    statements
    END [ name ];
```

Where:

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, or 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 `LEAKPROOK` 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 INVOKER`. 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.

`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, COST, ROWS and SET keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Example

The following is an example of a simple procedure that takes no parameters.

```
CREATE OR REPLACE PROCEDURE simple_procedure
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('That''s all folks!');
END simple_procedure;
```

The procedure is stored in the database by entering the procedure code in Advanced Server.

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.

4.2.3.2 Calling a Procedure

A procedure can be invoked from another SPL program by simply specifying the procedure name followed by its parameters, if any, followed by a semicolon.

```
name [ ([ parameters ]) ];
```

Where:

name is the identifier of the procedure.

parameters is a list of actual parameters.

Note: If there are no actual parameters to be passed, the procedure may be called with an empty parameter list, or the opening and closing parenthesis may be omitted entirely.

Note: The syntax for calling a procedure is the same as in the preceding syntax diagram when executing it with the `EXEC` command in PSQL or EDB*Plus. See Section 11.1.2.12 for information about the `EXEC` command.

The following is an example of calling the procedure from an anonymous block:

```
BEGIN
    simple_procedure;
END;

That's all folks!
```

Note: Each application has its own unique way to call a procedure. For example, in a Java application, the application programming interface, JDBC, is used.

4.2.3.3 Deleting a Procedure

A procedure can be deleted from the database using the `DROP PROCEDURE` command.

```
DROP PROCEDURE name;
```

Where *`name`* is the name of the procedure to be dropped.

The previously created procedure is dropped in this example:

```
DROP PROCEDURE simple_procedure;
```

4.2.4 Functions Overview

Functions are SPL programs that are invoked as expressions. When evaluated, a function returns a value that is substituted in the expression in which the function is embedded. Functions may optionally take values from the calling program in the form of input parameters. In addition to the fact that the function, itself, returns a value, a function may optionally return additional values to the caller in the form of output parameters. The use of output parameters in functions, however, is not an encouraged programming practice.

4.2.4.1 Creating a Function

The `CREATE FUNCTION` command defines and names a function that will be stored in the database.

```
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
        | COST execution_cost
        | ROWS result_rows
        | SET configuration_parameter
          { TO value | = value | FROM CURRENT }
   ...]
{ IS | AS }
    [ declarations ]
    BEGIN
    statements
    END [ name ];
```

Where:

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.

`declarations`

`declarations` are variable, cursor, or 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 `LEAKPROOK` 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 INVOKER`. 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.

`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`, `COST`, `ROWS` and `SET` keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Examples

The following is an example of a simple function that takes no parameters.

```
CREATE OR REPLACE FUNCTION simple_function
    RETURN VARCHAR2
IS
BEGIN
    RETURN 'That''s All Folks!';
END simple_function;
```

The following function takes two input parameters. Parameters are discussed in more detail in subsequent sections.

```
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 emp_comp;
```

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.

4.2.4.2 Calling a Function

A function can be used anywhere an expression can appear within an SPL statement. A function is invoked by simply specifying its name followed by its parameters enclosed in parenthesis, if any.

```
    name [ ([ parameters ]) ]
```

name is the name of the function. *parameters* is a list of actual parameters.

Note: If there are no actual parameters to be passed, the function may be called with an empty parameter list, or the opening and closing parenthesis may be omitted entirely.

The following shows how the function can be called from another SPL program.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(simple_function);
END;

That's All Folks!
```

A function is typically used within a SQL statement as shown in the following.

```
SELECT empno "EMPNO", ename "ENAME", sal "SAL", comm "COMM",
    emp_comp(sal, comm) "YEARLY COMPENSATION" FROM emp;

 EMPNO | ENAME  |   SAL   |  COMM   | YEARLY COMPENSATION
-------+--------+---------+---------+---------------------
  7369 | SMITH  |  800.00 |         |            19200.00
  7499 | ALLEN  | 1600.00 |  300.00 |            45600.00
  7521 | WARD   | 1250.00 |  500.00 |            42000.00
  7566 | JONES  | 2975.00 |         |            71400.00
  7654 | MARTIN | 1250.00 | 1400.00 |            63600.00
  7698 | BLAKE  | 2850.00 |         |            68400.00
  7782 | CLARK  | 2450.00 |         |            58800.00
  7788 | SCOTT  | 3000.00 |         |            72000.00
  7839 | KING   | 5000.00 |         |           120000.00
  7844 | TURNER | 1500.00 |    0.00 |            36000.00
  7876 | ADAMS  | 1100.00 |         |            26400.00
  7900 | JAMES  |  950.00 |         |            22800.00
  7902 | FORD   | 3000.00 |         |            72000.00
  7934 | MILLER | 1300.00 |         |            31200.00
(14 rows)
```

4.2.4.3 Deleting a Function

A function can be deleted from the database using the `DROP FUNCTION` command.

```
DROP FUNCTION name [ (parameters) ];
```

Where *`name`* is the name of the function to be dropped.

Note: The specification of the parameter list is required in Advanced Server under certain circumstances. Oracle requires that the parameter list always be omitted.

The previously created function is dropped in this example:

```
DROP FUNCTION simple_function;
```

4.2.5 Procedure and Function Parameters

An important aspect of using procedures and functions is the capability to pass data from the calling program to the procedure or function and to receive data back from the procedure or function. This is accomplished by using *parameters*.

Parameters are declared in the procedure or function definition, enclosed within parenthesis following the procedure or function name. Parameters declared in the procedure or function definition are known as *formal parameters*. When the procedure or function is invoked, the calling program supplies the actual data that is to be used in the called program's processing as well as the variables that are to receive the results of the called program's processing. The data and variables supplied by the calling program when the procedure or function is called are referred to as the *actual parameters*.

The following is the general format of a formal parameter declaration.

```
(name [ IN | OUT | IN OUT ] data_type [ DEFAULT value ])
```

`name` is an identifier assigned to the formal parameter. If specified, `IN` defines the parameter for receiving input data into the procedure or function. An `IN` parameter can also be initialized to a default value. If specified, `OUT` defines the parameter for returning data from the procedure or function. If specified, `IN OUT` allows the parameter to be used for both input and output. If all of `IN`, `OUT`, and `IN OUT` are omitted, then the parameter acts as if it were defined as `IN` by default. Whether a parameter is `IN`, `OUT`, or `IN OUT` is referred to as the parameter's *mode*. `data_type` defines the data type of the parameter. `value` is a default value assigned to an `IN` parameter in the called program if an actual parameter is not specified in the call.

The following is an example of a procedure that takes parameters:

```
CREATE OR REPLACE PROCEDURE emp_query (
    p_deptno        IN      NUMBER,
    p_empno         IN OUT  NUMBER,
    p_ename         IN OUT  VARCHAR2,
    p_job           OUT     VARCHAR2,
    p_hiredate      OUT     DATE,
    p_sal           OUT     NUMBER
)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
          AND (empno = p_empno
           OR  ename = UPPER(p_ename));
END;
```

In this example, `p_deptno` is an `IN` formal parameter, `p_empno` and `p_ename` are `IN OUT` formal parameters, and `p_job`, `p_hiredate`, and `p_sal` are `OUT` formal parameters.

Note: In the previous example, no maximum length was specified on the `VARCHAR2` parameters and no precision and scale were specified on the `NUMBER` parameters. It is illegal to specify a length, precision, scale or other constraints on parameter declarations. These constraints are automatically inherited from the actual parameters that are used when the procedure or function is called.

The `emp_query` procedure can be called by another program, passing it the actual parameters. The following is an example of another SPL program that calls `emp_query`.

```
DECLARE
    v_deptno        NUMBER(2);
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 7900;
    v_ename  := '';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
END;
```

In this example, `v_deptno`, `v_empno`, `v_ename`, `v_job`, `v_hiredate`, and `v_sal` are the actual parameters.

The output from the preceding example is shown as follows:

```
Department : 30
Employee No: 7900
Name       : JAMES
Job        : CLERK
Hire Date  : 03-DEC-81
Salary     : 950
```

4.2.5.1 Positional vs. Named Parameter Notation

You can use either *positional* or *named* parameter notation when passing parameters to a function or procedure. If you specify parameters using positional notation, you must list the parameters in the order that they are declared; if you specify parameters with named notation, the order of the parameters is not significant.

To specify parameters using named notation, list the name of each parameter followed by an arrow (=>) and the parameter value. Named notation is more verbose, but makes your code easier to read and maintain.

A simple example that demonstrates using positional and named parameter notation follows:

```
CREATE OR REPLACE PROCEDURE emp_info (
    p_deptno        IN     NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
)
IS
BEGIN
    dbms_output.put_line('Department Number =' || p_deptno);
    dbms_output.put_line('Employee Number =' || p_empno);
    dbms_output.put_line('Employee Name =' || p_ename;
END;
```

To call the procedure using positional notation, pass the following:

```
emp_info(30, 7455, 'Clark');
```

To call the procedure using named notation, pass the following:

```
emp_info(p_ename =>'Clark', p_empno=>7455, p_deptno=>30);
```

Using named notation can alleviate the need to re-arrange a procedure's parameter list if the parameter list changes, if the parameters are reordered or if a new optional parameter is added.

In a case where you have a default value for an argument and the argument is not a trailing argument, you must use named notation to call the procedure or function. The following case demonstrates a procedure with two, leading, default arguments.

```
CREATE OR REPLACE PROCEDURE check_balance (
    p_customerID  IN NUMBER DEFAULT NULL,
    p_balance     IN NUMBER DEFAULT NULL,
    p_amount      IN NUMBER
)
IS
DECLARE
    balance NUMBER;
BEGIN
   IF (p_balance IS NULL AND p_customerID IS NULL) THEN
      RAISE_APPLICATION_ERROR
          (-20010, 'Must provide balance or customer');
   ELSEIF (p_balance IS NOT NULL AND p_customerID IS NOT NULL) THEN
      RAISE_APPLICATION_ERROR
          (-20020,'Must provide balance or customer, not both');
   ELSEIF (p_balance IS NULL) THEN
```

```
      balance := getCustomerBalance(p_customerID);
   ELSE
      balance := p_balance;
   END IF;

   IF (amount > balance) THEN
      RAISE_APPLICATION_ERROR
        (-20030, 'Balance insufficient');
   END IF;
END;
```

You can only omit non-trailing argument values (when you call this procedure) by using named notation; when using positional notation, only trailing arguments are allowed to default. You can call this procedure with the following arguments:

```
check_balance(p_customerID => 10, p_amount = 500.00)

check_balance(p_balance => 1000.00, p_amount = 500.00)
```

You can use a combination of positional and named notation (mixed notation) to specify parameters. A simple example that demonstrates using mixed parameter notation follows:

```
CREATE OR REPLACE PROCEDURE emp_info (
    p_deptno        IN     NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
)
IS
BEGIN
    dbms_output.put_line('Department Number =' || p_deptno);
    dbms_output.put_line('Employee Number =' || p_empno);
    dbms_output.put_line('Employee Name =' || p_ename;
END;
```

You can call the procedure using mixed notation:

```
emp_info(30, p_ename =>'Clark', p_empno=>7455);
```

If you do use mixed notation, remember that named arguments cannot precede positional arguments.

4.2.5.2 Parameter Modes

As previously discussed, a parameter has one of three possible modes - `IN`, `OUT`, or `IN OUT`. The following characteristics of a formal parameter are dependent upon its mode:

- Its initial value when the procedure or function is called.
- Whether or not the called procedure or function can modify the formal parameter.

- How the actual parameter value is passed from the calling program to the called program.
- What happens to the formal parameter value when an unhandled exception occurs in the called program.

The following table summarizes the behavior of parameters according to their mode.

Table 4-4-1 Parameter Modes

Mode Property	IN	IN OUT	OUT
Formal parameter initialized to:	Actual parameter value	Actual parameter value	Actual parameter value
Formal parameter modifiable by the called program?	No	Yes	Yes
Actual parameter contains: (after normal called program termination)	Original actual parameter value prior to the call	Last value of the formal parameter	Last value of the formal parameter
Actual parameter contains: (after a handled exception in the called program)	Original actual parameter value prior to the call	Last value of the formal parameter	Last value of the formal parameter
Actual parameter contains: (after an unhandled exception in the called program)	Original actual parameter value prior to the call	Original actual parameter value prior to the call	Original actual parameter value prior to the call

As shown by the table, an `IN` formal parameter is initialized to the actual parameter with which it is called unless it was explicitly initialized with a default value. The `IN` parameter may be referenced within the called program, however, the called program may not assign a new value to the `IN` parameter. After control returns to the calling program, the actual parameter always contains the same value as it was set to prior to the call.

The `OUT` formal parameter is initialized to the actual parameter with which it is called. The called program may reference and assign new values to the formal parameter. If the called program terminates without an exception, the actual parameter takes on the value last set in the formal parameter. If a handled exception occurs, the value of the actual parameter takes on the last value assigned to the formal parameter. If an unhandled exception occurs, the value of the actual parameter remains as it was prior to the call.

Like an `IN` parameter, an `IN OUT` formal parameter is initialized to the actual parameter with which it is called. Like an `OUT` parameter, an `IN OUT` formal parameter is modifiable by the called program and the last value in the formal parameter is passed to the calling program's actual parameter if the called program terminates without an exception. If a handled exception occurs, the value of the actual parameter takes on the last value assigned to the formal parameter. If an unhandled exception occurs, the value of the actual parameter remains as it was prior to the call.

4.2.5.3 Using Default Values in Parameters

You can set a default value of a formal parameter by including the `DEFAULT` clause or using the assignment operator (`:=`) in the `CREATE PROCEDURE` or `CREATE FUNCTION` statement.

The general form of a formal parameter declaration is:

```
    (name [ IN|OUT|IN OUT ] data_type [{DEFAULT | := } expr ])
```

`name` is an identifier assigned to the parameter.

`IN|OUT|IN OUT` specifies the parameter mode.

`data_type` is the data type assigned to the variable.

`expr` is the default value assigned to the parameter. If you do not include a `DEFAULT` clause, the caller must provide a value for the parameter.

The default value is evaluated every time the function or procedure is invoked. For example, assigning `SYSDATE` to a parameter of type `DATE` causes the parameter to have the time of the current invocation, not the time when the procedure or function was created.

The following simple procedure demonstrates using the assignment operator to set a default value of `SYSDATE` into the parameter, `hiredate`:

```
CREATE OR REPLACE PROCEDURE hire_emp (
    p_empno         NUMBER,
    p_ename         VARCHAR2,
    p_hiredate      DATE := SYSDATE
) RETURN
IS
BEGIN
    INSERT INTO emp(empno, ename, hiredate)
                   VALUES(p_empno, p_ename, p_hiredate);

    DBMS_OUTPUT.PUT_LINE('Hired!');
END emp_comp;
```

If the parameter declaration includes a default value, you can omit the parameter from the actual parameter list when you call the function. Calls to the sample procedure (`hire_emp`) must include two arguments: the employee number (`p_empno`) and employee name (`p_empno`). The third parameter (`p_hiredate`) defaults to the value of `SYSDATE`:

```
 hire_emp 7575, Clark
```

If you do include a value for the actual parameter when you call the function, that value takes precedence over the default value:

```
hire_emp 7575, Clark, 15-FEB-2010
```

Adds a new employee with a hiredate of `February 15, 2010`, regardless of the current value of `SYSDATE`.

You can write the same function by substituting the `DEFAULT` keyword for the assignment operator:

```
CREATE OR REPLACE PROCEDURE hire_emp (
    p_empno         NUMBER,
    p_ename         VARCHAR2,
    p_hiredate      DATE DEFAULT SYSDATE
) RETURN
IS
BEGIN
    INSERT INTO emp(empno, ename, hiredate)
                   VALUES(p_empno, p_ename, p_hiredate);

    DBMS_OUTPUT.PUT_LINE('Hired!');
END emp_comp;
```

4.2.6 Compilation Errors in Procedures and Functions

When the Advanced Server parsers compile a procedure or function, they confirm that both the `CREATE` statement and the program body (that portion of the program that follows the `AS` keyword) conforms to the grammar rules for SPL and SQL constructs. By default, the server will terminate the compilation process if a parser detects an error. Note that the parsers detect syntax errors in expressions, but not semantic errors (i.e. an expression referencing a non-existent column, table, or function, or a value of incorrect type).

`spl.max_error_count` instructs the server to stop parsing if it encounters the specified number of errors in SPL code, or when it encounters an error in SQL code. The default value of `spl.max_error_count` is `10`; the maximum value is `1000`. Setting `spl.max_error_count` to a value of `1` instructs the server to stop parsing when it encounters the first error in either SPL or SQL code.

You can use the `SET` command to specify a value for `spl.max_error_count` for your current session. The syntax is:

```
SET spl.max_error_count = number_of_errors
```

Where *`number_of_errors`* specifies the number of SPL errors that may occur before the server halts the compilation process. For example:

```
SET spl.max_error_count = 6
```

The example instructs the server to continue past the first five SPL errors it encounters. When the server encounters the sixth error it will stop validating, and print six detailed error messages, and one error summary.

To save time when developing new code, or when importing existing code from another source, you may want to set the `spl.max_error_count` configuration parameter to a relatively high number of errors.

Please note that if you instruct the server to continue parsing in spite of errors in the SPL code in a program body, and the parser encounters an error in a segment of SQL code, there may still be errors in any SPL or SQL code that follows the erroneous SQL code. For example, the following function results in two errors:

```
CREATE FUNCTION computeBonus(baseSalary number) RETURN
number AS
BEGIN

    bonus := baseSalary * 1.10;
    total := bonus + 100;
    RETURN bonus;
END;

ERROR:  "bonus" is not a known variable
LINE 4:     bonus := baseSalary * 1.10;
            ^
ERROR:  "total" is not a known variable
LINE 5:     total := bonus + 100;
            ^
ERROR:  compilation of SPL function/procedure
"computebonus" failed due to 2 errors
```

The following example adds a `SELECT` statement to the previous example. The error in the `SELECT` statement masks the other errors that follow:

```
CREATE FUNCTION computeBonus(employeeName number) RETURN
number AS
BEGIN
    SELECT salary INTO baseSalary FROM emp
      WHERE ename = employeeName;

    bonus := baseSalary * 1.10;
    total := bonus + 100;

    RETURN bonus;

END;

ERROR:  "basesalary" is not a known variable
LINE 3:     SELECT salary INTO baseSalary FROM emp WHERE
ename = emp...
```

4.2.7 Program Security

Security over what user may execute an SPL program and what database objects an SPL program may access for any given user executing the program is controlled by the following:

- Privilege to execute a program.
- Privileges granted on the database objects (including other SPL programs) which a program attempts to access.
- Whether the program is defined with definer's rights or invoker's rights.

These aspects are discussed in the following sections.

4.2.7.1 EXECUTE Privilege

An SPL program (function, procedure, or package) can begin execution only if any of the following are true:

- The current user is a superuser, or
- The current user has been granted `EXECUTE` privilege on the SPL program, or
- The current user inherits `EXECUTE` privilege on the SPL program by virtue of being a member of a group which does have such privilege, or
- `EXECUTE` privilege has been granted to the `PUBLIC` group.

Whenever an SPL program is created in Advanced Server, `EXECUTE` privilege is automatically granted to the `PUBLIC` group by default, therefore, any user can immediately execute the program.

This default privilege can be removed by using the `REVOKE EXECUTE` command. See th `REVOKE` command for details. The following is an example:

```
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
```

Explicit `EXECUTE` privilege on the program can then be granted to individual users or groups.

```
GRANT EXECUTE ON PROCEDURE list_emp TO john;
```

Now, user, `john`, can execute the `list_emp` program; other users who do not meet any of the conditions listed at the beginning of this section cannot.

Once a program begins execution, the next aspect of security is what privilege checks occur if the program attempts to perform an action on any database object including:

- Reading or modifying table or view data.
- Creating, modifying, or deleting a database object such as a table, view, index, or sequence.
- Obtaining the current or next value from a sequence.
- Calling another program (function, procedure, or package).

Each such action can be protected by privileges on the database object either allowed or disallowed for the user.

Note that it is possible for a database to have more than one object of the same type with the same name, but each such object belonging to a different schema in the database. If this is the case, which object is being referenced by an SPL program? This is the topic of the next section.

4.2.7.2 Database Object Name Resolution

A database object inside an SPL program may either be referenced by its qualified name or by an unqualified name. A qualified name is in the form of `*schema*.*name*` where *schema* is the name of the schema under which the database object with identifier, *name*, exists. An unqualified name does not have the "*schema*." portion. When a reference is made to a qualified name, there is absolutely no ambiguity as to exactly which database object is intended – it either does or does not exist in the specified schema.

Locating an object with an unqualified name, however, requires the use of the current user's search path. When a user becomes the current user of a session, a default search path is always associated with that user. The search path consists of a list of schemas which are searched in left-to-right order for locating an unqualified database object reference. The object is considered non-existent if it can't be found in any of the schemas in the search path. The default search path can be displayed in PSQL using the `SHOW search_path` command.

```
SHOW search_path;

     search_path
----------------------
 $user,public,sys,dbo
(1 row)
```

`$user` in the above search path is a generic placeholder that refers to the current user so if the current user of the above session is `enterprisedb`, an unqualified database object would be searched for in the following schemas in this order – first, `enterprisedb`, then `public`, then `sys`, and finally, `dbo`.

Once an unqualified name has been resolved in the search path, it can be determined if the current user has the appropriate privilege to perform the desired action on that specific object.

Note: The concept of the search path is not compatible with Oracle databases. For an unqualified reference, Oracle simply looks in the schema of the current user for the named database object. It also important to note that in Oracle, a user and his or her schema is the same entity while in Advanced Server, a user and a schema are two distinct objects.

4.2.7.3 Database Object Privileges

Once an SPL program begins execution, any attempt to access a database object from within the program results in a check to ensure the current user has the authorization to perform the intended action against the referenced object. Privileges on database objects are bestowed and removed using the `GRANT` and `REVOKE` commands, respectively. If the current user attempts unauthorized access on a database object, then the program will throw an exception. See Section 4.5.7 for information about exception handling.

The final topic discusses exactly who is the current user.

4.2.7.4 Definer's vs. Invokers Rights

When an SPL program is about to begin execution, a determination is made as to what user is to be associated with this process. This user is referred to as the *current user*. The current user's database object privileges are used to determine whether or not access to database objects referenced in the program will be permitted. The current, prevailing search path in effect when the program is invoked will be used to resolve any unqualified object references.

The selection of the current user is influenced by whether the SPL program was created with definer's right or invoker's rights. The `AUTHID` clause determines that selection. Appearance of the clause `AUTHID DEFINER` gives the program definer's rights. This is also the default if the `AUTHID` clause is omitted. Use of the clause `AUTHID CURRENT_USER` gives the program invoker's rights. The difference between the two is summarized as follows:

- If a program has *definer's rights*, then the owner of the program becomes the current user when program execution begins. The program owner's database object privileges are used to determine if access to a referenced object is permitted. In a definer's rights program, it is irrelevant as to which user actually invoked the program.
- If a program has *invoker's rights*, then the current user at the time the program is called remains the current user while the program is executing (but not necessarily within called subprograms – see the following bullet points). When an invoker's rights program is invoked, the current user is typically the user that started the

session (i.e., made the database connection) although it is possible to change the current user after the session has started using the

- SET ROLE command. In an invoker's rights program, it is irrelevant as to which user actually owns the program.

From the previous definitions, the following observations can be made:

- If a definer's rights program calls a definer's rights program, the current user changes from the owner of the calling program to the owner of the called program during execution of the called program.
- If a definer's rights program calls an invoker's rights program, the owner of the calling program remains the current user during execution of both the calling and called programs.
- If an invoker's rights program calls an invoker's rights program, the current user of the calling program remains the current user during execution of the called program.
- If an invokers' rights program calls a definer's rights program, the current user switches to the owner of the definer's rights program during execution of the called program.

The same principles apply if the called program in turn calls another program in the cases cited above.

This section on security concludes with an example using the sample application.

4.2.7.5 Security Example

In the following example, a new database will be created along with two users – `hr_mgr` who will own a copy of the entire sample application in schema, `hr_mgr`; and `sales_mgr` who will own a schema named, `sales_mgr`, that will have a copy of only the `emp` table containing only the employees who work in sales.

The procedure `list_emp`, function `hire_clerk`, and package `emp_admin` will be used in this example. All of the default privileges that are granted upon installation of the sample application will be removed and then be explicitly re-granted so as to present a more secure environment in this example.

Programs `list_emp` and `hire_clerk` will be changed from the default of definer's rights to invoker's rights. It will be then illustrated that when `sales_mgr` runs these programs, they act upon the `emp` table in `sales_mgr`'s schema since `sales_mgr`'s search path and privileges will be used for name resolution and authorization checking.

Programs `get_dept_name` and `hire_emp` in the `emp_admin` package will then be executed by `sales_mgr`. In this case, the `dept` table and `emp` table in `hr_mgr`'s schema will be accessed as `hr_mgr` is the owner of the `emp_admin` package which is using definer's rights. Since the default search path is in effect with the `$user` placeholder, the schema matching the user (in this case, `hr_mgr`) is used to find the tables.

Step 1 – Create Database and Users

As user `enterprisedb`, create the `hr` database:

```
CREATE DATABASE hr;
```

Switch to the hr database and create the users:

```
\c hr enterprisedb
CREATE USER hr_mgr IDENTIFIED BY password;
CREATE USER sales_mgr IDENTIFIED BY password;
```

Step 2 – Create the Sample Application

Create the entire sample application, owned by `hr_mgr`, in `hr_mgr`'s schema.

```
\c - hr_mgr
\i C:/Program Files/PostgresPlus/9.5AS/installer/server/edb-sample.sql

BEGIN
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE VIEW
CREATE SEQUENCE
        .
        .
        .
CREATE PACKAGE
CREATE PACKAGE BODY
COMMIT
```

Step 3 – Create the emp Table in Schema sales_mgr

Create a subset of the `emp` table owned by `sales_mgr` in `sales_mgr`'s schema.

```
\c – hr_mgr
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
\c – sales_mgr
CREATE TABLE emp AS SELECT * FROM hr_mgr.emp WHERE job = 'SALESMAN';
```

In the above example, the `GRANT USAGE ON SCHEMA` command is given to allow `sales_mgr` access into `hr_mgr`'s schema to make a copy of `hr_mgr`'s `emp` table. This step is required in Advanced Server and is not compatible with Oracle databases since Oracle does not have the concept of a schema that is distinct from its user.

Step 4 – Remove Default Privileges

Remove all privileges to later illustrate the minimum required privileges needed.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

```
\c - hr_mgr
REVOKE USAGE ON SCHEMA hr_mgr FROM sales_mgr;
REVOKE ALL ON dept FROM PUBLIC;
REVOKE ALL ON emp FROM PUBLIC;
REVOKE ALL ON next_empno FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION new_empno() FROM PUBLIC;
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) FROM PUBLIC;
REVOKE EXECUTE ON PACKAGE emp_admin FROM PUBLIC;
```

Step 5 – Change list_emp to Invoker's Rights

While connected as user, hr_mgr, add the AUTHID CURRENT_USER clause to the list_emp program and resave it in Advanced Server. When performing this step, be sure you are logged on as hr_mgr, otherwise the modified program may wind up in the public schema instead of in hr_mgr's schema.

```
CREATE OR REPLACE PROCEDURE list_emp
AUTHID CURRENT_USER
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;
```

Step 6 – Change hire_clerk to Invoker's Rights and Qualify Call to new_empno

While connected as user, hr_mgr, add the AUTHID CURRENT_USER clause to the hire_clerk program.

Also, after the BEGIN statement, fully qualify the reference, new_empno, to hr_mgr.new_empno in order to ensure the hire_clerk function call to the new_empno function resolves to the hr_mgr schema.

When resaving the program, be sure you are logged on as hr_mgr, otherwise the modified program may wind up in the public schema instead of in hr_mgr's schema.

```
CREATE OR REPLACE FUNCTION hire_clerk (
    p_ename         VARCHAR2,
    p_deptno        NUMBER
) RETURN NUMBER
```

```
AUTHID CURRENT_USER
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_mgr           NUMBER(4);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_deptno        NUMBER(2);
BEGIN
    v_empno := hr_mgr.new_empno;
    INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
        TRUNC(SYSDATE), 950.00, NULL, p_deptno);
    SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        FROM emp WHERE empno = v_empno;
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Manager    : ' || v_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        RETURN -1;
END;
```

Step 7 – Grant Required Privileges

While connected as user, `hr_mgr`, grant the privileges needed so `sales_mgr` can execute the `list_emp` procedure, `hire_clerk` function, and `emp_admin` package. Note that the only data object `sales_mgr` has access to is the `emp` table in the `sales_mgr` schema. `sales_mgr` has no privileges on any table in the `hr_mgr` schema.

```
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
GRANT EXECUTE ON PROCEDURE list_emp TO sales_mgr;
GRANT EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) TO sales_mgr;
GRANT EXECUTE ON FUNCTION new_empno() TO sales_mgr;
GRANT EXECUTE ON PACKAGE emp_admin TO sales_mgr;
```

Step 8 – Run Programs list_emp and hire_clerk

Connect as user, `sales_mgr`, and run the following anonymous block:

```
\c – sales_mgr
DECLARE
    v_empno         NUMBER(4);
BEGIN
    hr_mgr.list_emp;
    DBMS_OUTPUT.PUT_LINE('*** Adding new employee ***');
    v_empno := hr_mgr.hire_clerk('JONES',40);
    DBMS_OUTPUT.PUT_LINE('*** After new employee added ***');
    hr_mgr.list_emp;
END;

EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
*** Adding new employee ***
Department : 40
Employee No: 8000
Name       : JONES
Job        : CLERK
Manager    : 7782
Hire Date  : 08-NOV-07 00:00:00
Salary     : 950.00
*** After new employee added ***
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
8000     JONES
```

The table and sequence accessed by the programs of the anonymous block are illustrated in the following diagram. The gray ovals represent the schemas of `sales_mgr` and `hr_mgr`. The current user during each program execution is shown within parenthesis in bold red font.

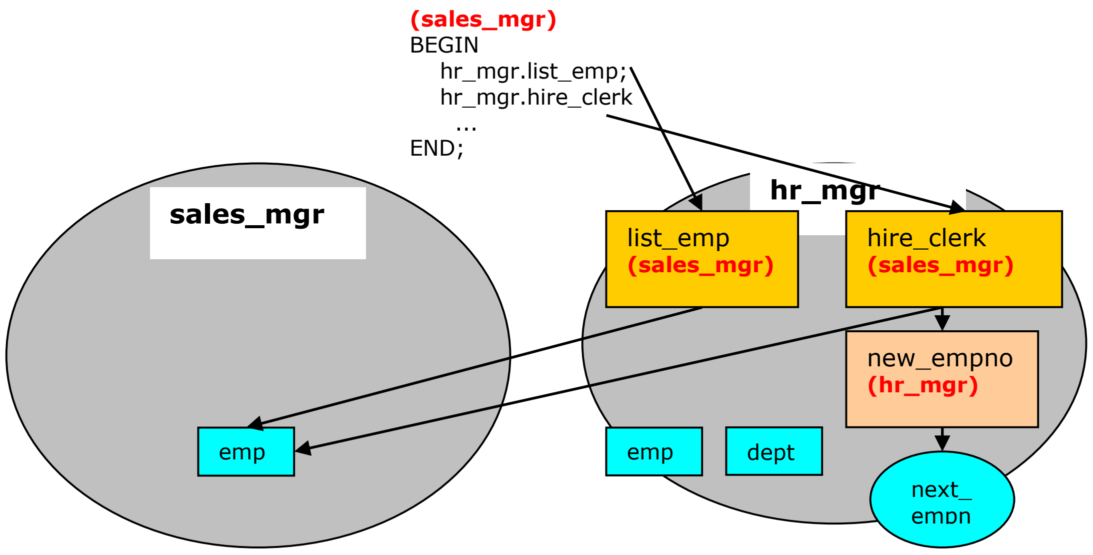


Figure 3 - Invoker's Rights Programs

Selecting from `sales_mgr`'s `emp` table shows that the update was made in this table.

```
SELECT empno, ename, hiredate, sal, deptno,
hr_mgr.emp_admin.get_dept_name(deptno) FROM sales_mgr.emp;

empno | ename  |      hiredate      |   sal   | deptno | get_dept_name
-------+--------+--------------------+---------+--------+---------------
  7499 | ALLEN  | 20-FEB-81 00:00:00 | 1600.00 |     30 | SALES
  7521 | WARD   | 22-FEB-81 00:00:00 | 1250.00 |     30 | SALES
  7654 | MARTIN | 28-SEP-81 00:00:00 | 1250.00 |     30 | SALES
  7844 | TURNER | 08-SEP-81 00:00:00 | 1500.00 |     30 | SALES
  8000 | JONES  | 08-NOV-07 00:00:00 |  950.00 |     40 | OPERATIONS
(5 rows)
```

The following diagram shows that the `SELECT` command references the `emp` table in the `sales_mgr` schema, but the `dept` table referenced by the `get_dept_name` function in the `emp_admin` package is from the `hr_mgr` schema since the `emp_admin` package has definer's rights and is owned by `hr_mgr`. The default search path setting with the `$user` placeholder resolves the access by user `hr_mgr` to the `dept` table in the `hr_mgr` schema.

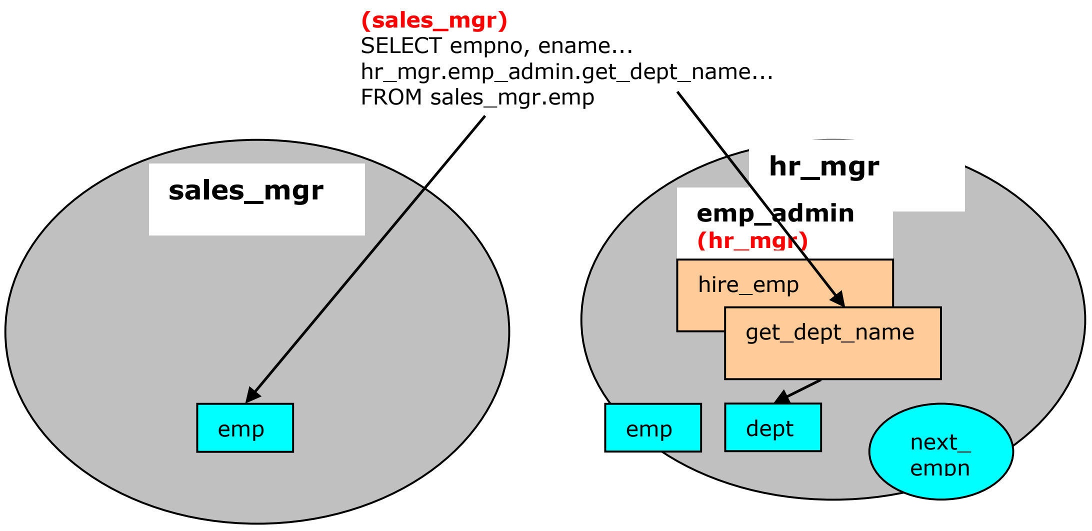


Figure 4 Definer's Rights Package

Step 9 – Run Program hire_emp in the emp_admin Package

While connected as user, `sales_mgr`, run the `hire_emp` procedure in the `emp_admin` package.

```
EXEC hr_mgr.emp_admin.hire_emp(9001,
'ALICE','SALESMAN',8000,TRUNC(SYSDATE),1000,7369,40);
```

This diagram illustrates that the `hire_emp` procedure in the `emp_admin` definer's rights package updates the `emp` table belonging to `hr_mgr` since the object privileges of hr_mgr are used, and the default search path setting with the $user placeholder resolves to the schema of hr_mgr.

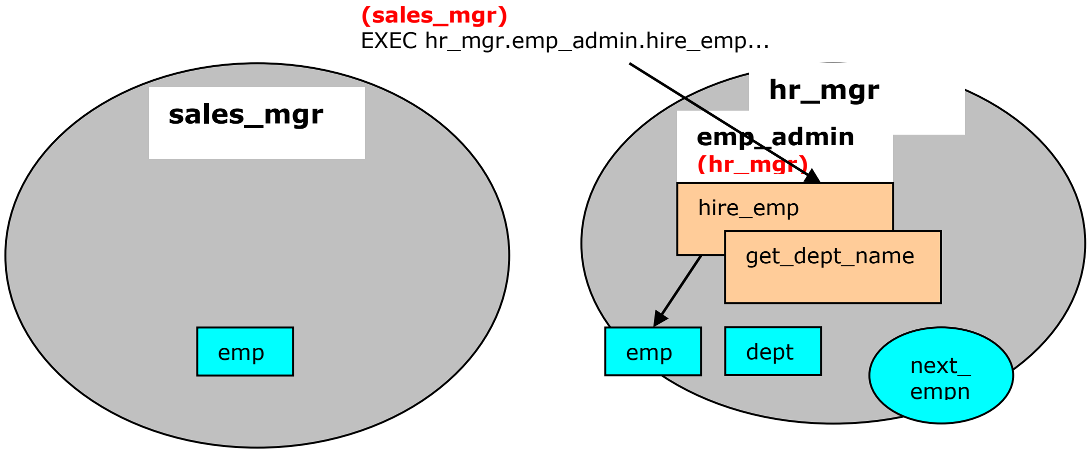


Figure 5 Definer's Rights Package

Now connect as user, hr_mgr. The following SELECT command verifies that the new employee was added to hr_mgr's emp table since the emp_admin package has definer's rights and hr_mgr is emp_admin's owner.

```
\c - hr_mgr
SELECT empno, ename, hiredate, sal, deptno,
hr_mgr.emp_admin.get_dept_name(deptno) FROM hr_mgr.emp;

empno | ename  |      hiredate      |   sal   | deptno | get_dept_name
-------+--------+--------------------+---------+--------+---------------
  7369 | SMITH  | 17-DEC-80 00:00:00 |  800.00 |     20 | RESEARCH
  7499 | ALLEN  | 20-FEB-81 00:00:00 | 1600.00 |     30 | SALES
  7521 | WARD   | 22-FEB-81 00:00:00 | 1250.00 |     30 | SALES
  7566 | JONES  | 02-APR-81 00:00:00 | 2975.00 |     20 | RESEARCH
  7654 | MARTIN | 28-SEP-81 00:00:00 | 1250.00 |     30 | SALES
  7698 | BLAKE  | 01-MAY-81 00:00:00 | 2850.00 |     30 | SALES
  7782 | CLARK  | 09-JUN-81 00:00:00 | 2450.00 |     10 | ACCOUNTING
  7788 | SCOTT  | 19-APR-87 00:00:00 | 3000.00 |     20 | RESEARCH
  7839 | KING   | 17-NOV-81 00:00:00 | 5000.00 |     10 | ACCOUNTING
  7844 | TURNER | 08-SEP-81 00:00:00 | 1500.00 |     30 | SALES
  7876 | ADAMS  | 23-MAY-87 00:00:00 | 1100.00 |     20 | RESEARCH
  7900 | JAMES  | 03-DEC-81 00:00:00 |  950.00 |     30 | SALES
  7902 | FORD   | 03-DEC-81 00:00:00 | 3000.00 |     20 | RESEARCH
  7934 | MILLER | 23-JAN-82 00:00:00 | 1300.00 |     10 | ACCOUNTING
  9001 | ALICE  | 08-NOV-07 00:00:00 | 8000.00 |     40 | OPERATIONS
(15 rows)
```

4.3 Variable Declarations

SPL is a block-structured language. The first section that can appear in a block is the declaration. The declaration contains the definition of variables, cursors, and other types that can be used in SPL statements contained in the block.

4.3.1 Declaring a Variable

Generally, all variables used in a block must be declared in the declaration section of the block. A variable declaration consists of a name that is assigned to the variable and its data type. (See Section 3.2 for a discussion of data types.) Optionally, the variable can be initialized to a default value in the variable declaration.

The general syntax of a variable declaration is:

```
name type [ { := | DEFAULT } { expression | NULL } ];
```

`name` is an identifier assigned to the variable.

`type` is the data type assigned to the variable.

[:= `expression`], if given, specifies the initial value assigned to the variable when the block is entered. If the clause is not given then the variable is initialized to the SQL `NULL` value.

The default value is evaluated every time the block is entered. So, for example, assigning `SYSDATE` to a variable of type `DATE` causes the variable to have the time of the current invocation, not the time when the procedure or function was precompiled.

The following procedure illustrates some variable declarations that utilize defaults consisting of string and numeric expressions.

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno        NUMBER
)
IS
    todays_date     DATE := SYSDATE;
    rpt_title       VARCHAR2(60) := 'Report For Department # ' || p_deptno
                              || ' on ' || todays_date;
    base_sal        INTEGER := 35525;
    base_comm_rate  NUMBER := 1.33333;
    base_annual     NUMBER := ROUND(base_sal * base_comm_rate, 2);
BEGIN
    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

The following output of the above procedure shows that default values in the variable declarations are indeed assigned to the variables.

```
EXEC dept_salary_rpt(20);

Report For Department # 20 on 10-JUL-07 16:44:45
Base Annual Salary: 47366.55
```

4.3.2 Using %TYPE in Variable Declarations

Often, variables will be declared in SPL programs that will be used to hold values from tables in the database. In order to ensure compatibility between the table columns and the SPL variables, the data types of the two should be the same.

However, as quite often happens, a change might be made to the table definition. If the data type of the column is changed, the corresponding change may be required to the variable in the SPL program.

Instead of coding the specific column data type into the variable declaration the column attribute, `%TYPE`, can be used instead. A qualified column name in dot notation or the name of a previously declared variable must be specified as a prefix to `%TYPE`. The data type of the column or variable prefixed to `%TYPE` is assigned to the variable being declared. If the data type of the given column or variable changes, the new data type will be associated with the variable without the need to modify the declaration code.

Note: The `%TYPE` attribute can be used with formal parameter declarations as well.

```
name { { table | view }.column | variable }%TYPE;
```

name is the identifier assigned to the variable or formal parameter that is being declared. *column* is the name of a column in *table* or *view*. *variable* is the name of a variable that was declared prior to the variable identified by *name*.

Note: The variable does not inherit any of the column's other attributes such as might be specified on the column with the `NOT NULL` clause or the `DEFAULT` clause.

In the following example a procedure queries the `emp` table using an employee number, displays the employee's data, finds the average salary of all employees in the department to which the employee belongs, and then compares the chosen employee's salary with the department average.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno          IN NUMBER
)
IS
    v_ename          VARCHAR2(10);
    v_job            VARCHAR2(9);
    v_hiredate       DATE;
    v_sal            NUMBER(7,2);
    v_deptno         NUMBER(2);
    v_avgsal         NUMBER(7,2);
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
            || 'department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
            || 'department average of ' || v_avgsal);
    END IF;
END;
```

Instead of the above, the procedure could be written as follows without explicitly coding the `emp` table data types into the declaration section of the procedure.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_avgsal        v_sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
            || 'department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
            || 'department average of ' || v_avgsal);
    END IF;
END;
```

Note: `p_empno` shows an example of a formal parameter defined using `%TYPE`.

`v_avgsal` illustrates the usage of `%TYPE` referring to another variable instead of a table column.

The following is sample output from executing this procedure.

```
EXEC emp_sal_query(7698);

Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
```

4.3.3 Using %ROWTYPE in Record Declarations

The `%TYPE` attribute provides an easy way to create a variable dependent upon a column's data type. Using the `%ROWTYPE` attribute, you can define a record that contains fields that correspond to all columns of a given table. Each field takes on the data type of its corresponding column. The fields in the record do not inherit any of the columns' other attributes such as might be specified with the `NOT NULL` clause or the `DEFAULT` clause.

A *record* is a named, ordered collection of fields. A *field* is similar to a variable; it has an identifier and data type, but has the additional property of belonging to a record, and must be referenced using dot notation with the record name as its qualifier.

You can use the `%ROWTYPE` attribute to declare a record. The `%ROWTYPE` attribute is prefixed by a table name. Each column in the named table defines an identically named field in the record with the same data type as the column.

```
record table%ROWTYPE;
```

`record` is an identifier assigned to the record. *`table`* is the name of a table (or view) whose columns are to define the fields in the record. The following example shows how the `emp_sal_query` procedure from the prior section can be modified to use `emp%ROWTYPE` to create a record named `r_emp` instead of declaring individual variables for the columns in `emp`.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno          IN emp.empno%TYPE
)
IS
    r_emp            emp%ROWTYPE;
    v_avgsal         emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);
    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
            || 'department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
            || 'department average of ' || v_avgsal);
    END IF;
END;
```

4.3.4 User-Defined Record Types and Record Variables

Records can be declared based upon a table definition using the `%ROWTYPE` attribute as shown in Section 4.3.3. This section describes how a new record structure can be defined that is not tied to any particular table definition.

The `TYPE IS RECORD` statement is used to create the definition of a record type. A *record type* is a definition of a record comprised of one or more identifiers and their corresponding data types. A record type cannot, by itself, be used to manipulate data.

The syntax for a `TYPE IS RECORD` statement is:

```
TYPE rec_type IS RECORD ( fields )
```

Where *`fields`* is a comma-separated list of one or more field definitions of the following form:

```
field_name data_type [NOT NULL][{:= | DEFAULT} default_value]
```

Where:

`rec_type`

> *`rec_type`* is an identifier assigned to the record type.

`field_name`

> *`field_name`* is the identifier assigned to the field of the record type.

`data_type`

> *`data_type`* specifies the data type of *`field_name`*.

`DEFAULT` *`default_value`*

> The `DEFAULT` clause assigns a default data value for the corresponding field. The data type of the default expression must match the data type of the column. If no default is specified, then the default is `NULL`.

A *record variable* or simply put, a *record*, is an instance of a record type. A record is declared from a record type. The properties of the record such as its field names and types are inherited from the record type.

The following is the syntax for a record declaration.

```
record rectype
```

`record` is an identifier assigned to the record variable. `rectype` is the identifier of a previously defined record type. Once declared, a record can then be used to hold data.

Dot notation is used to make reference to the fields in the record.

```
record.field
```

`record` is a previously declared record variable and `field` is the identifier of a field belonging to the record type from which `record` is defined.

The `emp_sal_query` is again modified – this time using a user-defined record type and record variable.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    TYPE emp_typ IS RECORD (
        ename       emp.ename%TYPE,
        job         emp.job%TYPE,
        hiredate    emp.hiredate%TYPE,
        sal         emp.sal%TYPE,
        deptno      emp.deptno%TYPE
    );
    r_emp           emp_typ;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
            || 'department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
            || 'department average of ' || v_avgsal);
    END IF;
END;
```

Note that instead of specifying data type names, the `%TYPE` attribute can be used for the field data types in the record type definition.

The following is the output from executing this stored procedure.

```
EXEC emp_sal_query(7698);

Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
```

4.4 Basic Statements

This section begins the discussion of the programming statements that can be used in an SPL program.

4.4.1 NULL

The simplest statement is the `NULL` statement. This statement is an executable statement that does nothing.

```
NULL;
```

The following is the simplest, possible valid SPL program.

```
BEGIN
    NULL;
END;
```

The `NULL` statement can act as a placeholder where an executable statement is required such as in a branch of an `IF-THEN-ELSE` statement.

For example:

```
CREATE OR REPLACE PROCEDURE divide_it (
    p_numerator      IN   NUMBER,
    p_denominator    IN   NUMBER,
    p_result         OUT  NUMBER
)
IS
BEGIN
    IF p_denominator = 0 THEN
        NULL;
    ELSE
        p_result := p_numerator / p_denominator;
    END IF;
END;
```

4.4.2 Assignment

The assignment statement sets a variable or a formal parameter of mode `OUT` or `IN OUT` specified on the left side of the assignment, `:=`, to the evaluated expression specified on the right side of the assignment.

```
variable := expression;
```

`variable` is an identifier for a previously declared variable, `OUT` formal parameter, or `IN OUT` formal parameter.

`expression` is an expression that produces a single value. The value produced by the expression must have a compatible data type with that of `variable`.

The following example shows the typical use of assignment statements in the executable section of the procedure.

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno        NUMBER
)
IS
    todays_date     DATE;
    rpt_title       VARCHAR2(60);
    base_sal        INTEGER;
    base_comm_rate  NUMBER;
    base_annual     NUMBER;
BEGIN
    todays_date := SYSDATE;
    rpt_title := 'Report For Department # ' || p_deptno || ' on '
        || todays_date;
    base_sal := 35525;
    base_comm_rate := 1.33333;
    base_annual := ROUND(base_sal * base_comm_rate, 2);

    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

4.4.3 SELECT INTO

The `SELECT INTO` statement is an SPL variation of the SQL `SELECT` command, the differences being:

- That `SELECT INTO` is designed to assign the results to variables or records where they can then be used in SPL program statements.
- The accessible result set of `SELECT INTO` is at most one row.

Other than the above, all of the clauses of the `SELECT` command such as `WHERE`, `ORDER BY`, `GROUP BY`, `HAVING`, etc. are valid for `SELECT INTO`. The following are the two variations of `SELECT INTO`.

```
SELECT select_expressions INTO target FROM ...;
```

`target` is a comma-separated list of simple variables. `select_expressions` and the remainder of the statement are the same as for the `SELECT` command. The selected values must exactly match in data type, number, and order the structure of the target or a runtime error occurs.

```
SELECT * INTO record FROM table ...;
```

record is a record variable that has previously been declared.

If the query returns zero rows, null values are assigned to the target(s). If the query returns multiple rows, the first row is assigned to the target(s) and the rest are discarded. (Note that "the first row" is not well-defined unless you've used `ORDER BY`.)

Note: In either cases, where no row is returned or more than one row is returned, SPL throws an exception.

Note: There is a variation of `SELECT INTO` using the `BULK COLLECT` clause that allows a result set of more than one row that is returned into a collection. See Section 4.12.4.1 for more information on using the `BULK COLLECT` clause with the `SELECT INTO` statement.

You can use the `WHEN NO_DATA_FOUND` clause in an `EXCEPTION` block to determine whether the assignment was successful (that is, at least one row was returned by the query).

This version of the `emp_sal_query` procedure uses the variation of `SELECT INTO` that returns the result set into a record. Also note the addition of the `EXCEPTION` block containing the `WHEN NO_DATA_FOUND` conditional expression.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT * INTO r_emp
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
            || 'department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
            || 'department average of ' || v_avgsal);
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
END;
```

If the query is executed with a non-existent employee number the results appear as follows.

```
EXEC emp_sal_query(0);

Employee # 0 not found
```

Another conditional clause of use in the `EXCEPTION` section with `SELECT INTO` is the `TOO_MANY_ROWS` exception. If more than one row is selected by the `SELECT INTO` statement an exception is thrown by SPL.

When the following block is executed, the `TOO_MANY_ROWS` exception is thrown since there are many employees in the specified department.

```
DECLARE
    v_ename         emp.ename%TYPE;
BEGIN
    SELECT ename INTO v_ename FROM emp WHERE deptno = 20 ORDER BY ename;
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee found');
        DBMS_OUTPUT.PUT_LINE('First employee returned is ' || v_ename);
END;

More than one employee found
First employee returned is ADAMS
```

Note: See Section 4.5.7 or more information on exception handling.

4.4.4 INSERT

The `INSERT` command available in the SQL language can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL `INSERT` command. Thus, SPL variables and parameters can be used to supply values to the insert operation.

The following is an example of a procedure that performs an insert of a new employee using data passed from a calling program.

```
CREATE OR REPLACE PROCEDURE emp_insert (
    p_empno         IN emp.empno%TYPE,
    p_ename         IN emp.ename%TYPE,
    p_job           IN emp.job%TYPE,
    p_mgr           IN emp.mgr%TYPE,
    p_hiredate      IN emp.hiredate%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE,
    p_deptno        IN emp.deptno%TYPE
)
IS
BEGIN
    INSERT INTO emp VALUES (
        p_empno,
        p_ename,
        p_job,
        p_mgr,
        p_hiredate,
        p_sal,
        p_comm,
        p_deptno);

    DBMS_OUTPUT.PUT_LINE('Added employee...');
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || p_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || p_job);
    DBMS_OUTPUT.PUT_LINE('Manager    : ' || p_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || p_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || p_sal);
    DBMS_OUTPUT.PUT_LINE('Commission : ' || p_comm);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || p_deptno);
    DBMS_OUTPUT.PUT_LINE('----------------------');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('OTHERS exception on INSERT of employee # '
            || p_empno);
        DBMS_OUTPUT.PUT_LINE('SQLCODE : ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM : ' || SQLERRM);
END;
```

If an exception occurs all database changes made in the procedure are automatically rolled back. In this example the `EXCEPTION` section with the `WHEN OTHERS` clause catches all exceptions. Two variables are displayed. `SQLCODE` is a number that identifies the specific exception that occurred. `SQLERRM` is a text message explaining the error. See Section 4.5.7 for more information on exception handling.

The following shows the output when this procedure is executed.

```
EXEC emp_insert(9503,'PETERSON','ANALYST',7902,'31-MAR-05',5000,NULL,40);

Added employee...
Employee # : 9503
Name       : PETERSON
Job        : ANALYST
Manager    : 7902
Hire Date  : 31-MAR-05 00:00:00
Salary     : 5000
Dept #     : 40
----------------------

SELECT * FROM emp WHERE empno = 9503;

 empno |  ename   |   job   | mgr  |      hiredate      |   sal   | comm | deptno
-------+----------+---------+------+--------------------+---------+------+--------
  9503 | PETERSON | ANALYST | 7902 | 31-MAR-05 00:00:00 | 5000.00 |      |     40
(1 row)
```

Note: The `INSERT` command can be included in a `FORALL` statement. A `FORALL` statement allows a single `INSERT` command to insert multiple rows from values supplied in one or more collections. See Section 4.12.3 for more information on the `FORALL` statement.

4.4.5 UPDATE

The `UPDATE` command available in the SQL language can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL `UPDATE` command. Thus, SPL variables and parameters can be used to supply values to the update operation.

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno         IN emp.empno%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE
)
IS
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || p_empno);
        DBMS_OUTPUT.PUT_LINE('New Salary         : ' || p_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission     : ' || p_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

The `SQL%FOUND` conditional expression returns `TRUE` if a row is updated, `FALSE` otherwise. See Section 4.4.8 for a discussion of `SQL%FOUND` and other similar expressions.

The following shows the update on the employee using this procedure.

```
EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
New Salary         : 6540
New Commission     : 1200

SELECT * FROM emp WHERE empno = 9503;

 empno |  ename   |   job   | mgr  |      hiredate      |   sal   |  comm   | deptno
-------+----------+---------+------+--------------------+---------+---------+--------
  9503 | PETERSON | ANALYST | 7902 | 31-MAR-05 00:00:00 | 6540.00 | 1200.00 |     40
(1 row)
```

Note: The `UPDATE` command can be included in a `FORALL` statement. A `FORALL` statement allows a single `UPDATE` command to update multiple rows from values supplied in one or more collections. See Section 4.12.3 for more information on the `FORALL` statement.

4.4.6 DELETE

The `DELETE` command (available in the SQL language) can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL `DELETE` command. Thus, SPL variables and parameters can be used to supply values to the delete operation.

```
CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno         IN emp.empno%TYPE
)
IS
BEGIN
    DELETE FROM emp WHERE empno = p_empno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || p_empno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

The `SQL%FOUND` conditional expression returns `TRUE` if a row is deleted, `FALSE` otherwise. See Section 4.4.8 for a discussion of `SQL%FOUND` and other similar expressions.

The following shows the deletion of an employee using this procedure.

```
EXEC emp_delete(9503);

Deleted Employee # : 9503

SELECT * FROM emp WHERE empno = 9503;

 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 rows)
```

Note: The `DELETE` command can be included in a `FORALL` statement. A `FORALL` statement allows a single `DELETE` command to delete multiple rows from values supplied in one or more collections. See Section 4.12.3 for more information on the `FORALL` statement.

4.4.7 Using the RETURNING INTO Clause

The `INSERT`, `UPDATE`, and `DELETE` commands may be appended by the optional `RETURNING INTO` clause. This clause allows the SPL program to capture the newly added, modified, or deleted values from the results of an `INSERT`, `UPDATE`, or `DELETE` command, respectively.

The following is the syntax.

```
    { insert | update | delete }
      RETURNING { * | expr_1 [, expr_2 ] ...}
        INTO { record | field_1 [, field_2 ] ...};
```

`insert` is a valid `INSERT` command. *`update`* is a valid `UPDATE` command. *`delete`* is a valid `DELETE` command. If `*` is specified, then the values from the row affected by the `INSERT, UPDATE, or DELETE` command are made available for assignment to the record or fields to the right of the `INTO` keyword. (Note that the use of `*` is an Advanced

Server extension and is not compatible with Oracle databases.) *expr_1*, *expr_2*... are expressions evaluated upon the row affected by the INSERT, UPDATE, or DELETE command. The evaluated results are assigned to the record or fields to the right of the INTO keyword. *record* is the identifier of a record that must contain fields that match in number and order, and are data type compatible with the values in the RETURNING clause. *field_1*, *field_2*,... are variables that must match in number and order, and are data type compatible with the set of values in the RETURNING clause.

If the INSERT, UPDATE, or DELETE command returns a result set with more than one row, then an exception is thrown with SQLCODE 01422, query returned more than one row. If no rows are in the result set, then the variables following the INTO keyword are set to null.

Note: There is a variation of RETURNING INTO using the BULK COLLECT clause that allows a result set of more than one row that is returned into a collection. See Section 4.12.4 for more information on the BULK COLLECT clause.

The following example is a modification of the emp_comp_update procedure introduced in Section 4.4.5, with the addition of the RETURNING INTO clause.

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno         IN emp.empno%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_deptno        emp.deptno%TYPE;
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
    RETURNING
        empno,
        ename,
        job,
        sal,
        comm,
        deptno
    INTO
        v_empno,
        v_ename,
        v_job,
        v_sal,
        v_comm,
        v_deptno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || v_empno);
        DBMS_OUTPUT.PUT_LINE('Name              : ' || v_ename);
        DBMS_OUTPUT.PUT_LINE('Job               : ' || v_job);
        DBMS_OUTPUT.PUT_LINE('Department        : ' || v_deptno);
        DBMS_OUTPUT.PUT_LINE('New Salary        : ' || v_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission     : ' || v_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

The following is the output from this procedure (assuming employee `9503` created by the `emp_insert` procedure still exists within the table).

```
EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Department         : 40
New Salary         : 6540.00
New Commission     : 1200.00
```

The following example is a modification of the `emp_delete` procedure, with the addition of the `RETURNING INTO` clause using record types.

```
CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
BEGIN
    DELETE FROM emp WHERE empno = p_empno
    RETURNING
        *
    INTO
        r_emp;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || r_emp.empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || r_emp.ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || r_emp.job);
        DBMS_OUTPUT.PUT_LINE('Manager            : ' || r_emp.mgr);
        DBMS_OUTPUT.PUT_LINE('Hire Date          : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary             : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Commission         : ' || r_emp.comm);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || r_emp.deptno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

The following is the output from this procedure.

```
EXEC emp_delete(9503);

Deleted Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Manager            : 7902
Hire Date          : 31-MAR-05 00:00:00
Salary             : 6540.00
Commission         : 1200.00
Department         : 40
```

4.4.8 Obtaining the Result Status

There are several attributes that can be used to determine the effect of a command. `SQL%FOUND` is a Boolean that returns `TRUE` if at least one row was affected by an `INSERT`, `UPDATE` or `DELETE` command or a `SELECT INTO` command retrieved one or more rows.

The following anonymous block inserts a row and then displays the fact that the row has been inserted.

```
BEGIN
    INSERT INTO emp (empno,ename,job,sal,deptno) VALUES (
        9001, 'JONES', 'CLERK', 850.00, 40);
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Row has been inserted');
    END IF;
END;

Row has been inserted
```

`SQL%ROWCOUNT` provides the number of rows affected by an `INSERT`, `UPDATE` or `DELETE` command. The following example updates the row that was just inserted and displays `SQL%ROWCOUNT`.

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9001;
    DBMS_OUTPUT.PUT_LINE('# rows updated: ' || SQL%ROWCOUNT);
END;

# rows updated: 1
```

`SQL%NOTFOUND` is the opposite of `SQL%FOUND`. `SQL%NOTFOUND` returns `TRUE` if no rows were affected by an `INSERT`, `UPDATE` or `DELETE` command or a `SELECT INTO` command retrieved no rows.

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9000;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No rows were updated');
    END IF;
END;

No rows were updated
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

4.5 Control Structures

The programming statements in SPL that make it a full procedural complement to SQL are described in this section.

4.5.1 IF Statement

`IF` statements let you execute commands based on certain conditions. SPL has four forms of `IF`:

- `IF ... THEN`
- `IF ... THEN ... ELSE`
- `IF ... THEN ... ELSE IF`
- `IF ... THEN ... ELSIF ... THEN ... ELSE`

4.5.1.1 IF-THEN

```
IF boolean-expression THEN
  statements
END IF;
```

`IF-THEN` statements are the simplest form of `IF`. The statements between `THEN` and `END IF` will be executed if the condition is `TRUE`. Otherwise, they are skipped.

In the following example an `IF-THEN` statement is used to test and display employees who have a commission.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following is the output from this program.

```
EMPNO    COMM
-----    -------
7499     $300.00
7521     $500.00
7654    $1400.00
```

4.5.1.2 IF-THEN-ELSE

```
IF boolean-expression THEN
  statements
ELSE
  statements
END IF;
```

`IF-THEN-ELSE` statements add to `IF-THEN` by letting you specify an alternative set of statements that should be executed if the condition evaluates to false.

The previous example is modified so an `IF-THEN-ELSE` statement is used to display the text `Non-commission` if the employee does not get a commission.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || 'Non-commission');
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following is the output from this program.

```
EMPNO    COMM
-----    -------
7369     Non-commission
7499  $   300.00
7521  $   500.00
7566     Non-commission
7654  $  1400.00
7698     Non-commission
7782     Non-commission
```

```
7788     Non-commission
7839     Non-commission
7844     Non-commission
7876     Non-commission
7900     Non-commission
7902     Non-commission
7934     Non-commission
```

4.5.1.3 IF-THEN-ELSE IF

`IF` statements can be nested so that alternative `IF` statements can be invoked once it is determined whether or not the conditional of an outer `IF` statement is `TRUE` or `FALSE`.

In the following example the outer `IF-THEN-ELSE` statement tests whether or not an employee has a commission. The inner `IF-THEN-ELSE` statements then test whether the employee's total compensation exceeds or is less than the company average.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_avg           NUMBER(7,2);
    CURSOR emp_cursor IS SELECT empno, sal, comm FROM emp;
BEGIN
--
--  Calculate the average yearly compensation in the company
--
    SELECT AVG((sal + NVL(comm,0)) * 24) INTO v_avg FROM emp;
    DBMS_OUTPUT.PUT_LINE('Average Yearly Compensation: ' ||
        TO_CHAR(v_avg,'$999,999.99'));
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    YEARLY COMP');
    DBMS_OUTPUT.PUT_LINE('-----    -----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_sal, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
--
--  Test if the employee's compensation with commission exceeds the average
--
            IF (v_sal + v_comm) * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Below Average');
            END IF;
        ELSE
--
--  Test if the employee's compensation without commission exceeds the
average
--
            IF v_sal * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') || ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') || ' Below Average');
            END IF;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

Note: The logic in this program can be simplified considerably by calculating the employee's yearly compensation using the `NVL` function within the `SELECT` command of the cursor declaration, however, the purpose of this example is to demonstrate how `IF` statements can be used.

The following is the output from this program.

```
Average Yearly Compensation: $  53,528.57
EMPNO    YEARLY COMP
-----    -----------
7369  $  19,200.00 Below Average
7499  $  45,600.00 Below Average
7521  $  42,000.00 Below Average
7566  $  71,400.00 Exceeds Average
7654  $  63,600.00 Exceeds Average
7698  $  68,400.00 Exceeds Average
7782  $  58,800.00 Exceeds Average
7788  $  72,000.00 Exceeds Average
7839  $ 120,000.00 Exceeds Average
7844  $  36,000.00 Below Average
7876  $  26,400.00 Below Average
7900  $  22,800.00 Below Average
7902  $  72,000.00 Exceeds Average
7934  $  31,200.00 Below Average
```

When you use this form, you are actually nesting an `IF` statement inside the `ELSE` part of an outer `IF` statement. Thus you need one `END IF` statement for each nested `IF` and one for the parent `IF-ELSE`.

4.5.1.4 IF-THEN-ELSIF-ELSE

```
  IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements ] ...]
[ ELSE
    statements ]
  END IF;
```

`IF-THEN-ELSIF-ELSE` provides a method of checking many alternatives in one statement. Formally it is equivalent to nested `IF-THEN-ELSE-IF-THEN` commands, but only one `END IF` is needed.

The following example uses an `IF-THEN-ELSIF-ELSE` statement to count the number of employees by compensation ranges of $25,000.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comp          NUMBER(8,2);
    v_lt_25K        SMALLINT := 0;
    v_25K_50K       SMALLINT := 0;
    v_50K_75K       SMALLINT := 0;
    v_75K_100K      SMALLINT := 0;
    v_ge_100K       SMALLINT := 0;
    CURSOR emp_cursor IS SELECT empno, (sal + NVL(comm,0)) * 24 FROM emp;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_empno, v_comp;
        EXIT WHEN emp_cursor%NOTFOUND;
        IF v_comp < 25000 THEN
            v_lt_25K := v_lt_25K + 1;
        ELSIF v_comp < 50000 THEN
            v_25K_50K := v_25K_50K + 1;
        ELSIF v_comp < 75000 THEN
            v_50K_75K := v_50K_75K + 1;
        ELSIF v_comp < 100000 THEN
            v_75K_100K := v_75K_100K + 1;
        ELSE
            v_ge_100K := v_ge_100K + 1;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
    DBMS_OUTPUT.PUT_LINE('Number of employees by yearly compensation');
    DBMS_OUTPUT.PUT_LINE('Less than 25,000 : ' || v_lt_25K);
    DBMS_OUTPUT.PUT_LINE('25,000 - 49,9999 : ' || v_25K_50K);
    DBMS_OUTPUT.PUT_LINE('50,000 - 74,9999 : ' || v_50K_75K);
    DBMS_OUTPUT.PUT_LINE('75,000 - 99,9999 : ' || v_75K_100K);
    DBMS_OUTPUT.PUT_LINE('100,000 and over : ' || v_ge_100K);
END;
```

The following is the output from this program.

```
Number of employees by yearly compensation
Less than 25,000 : 2
25,000 - 49,9999 : 5
50,000 - 74,9999 : 6
75,000 - 99,9999 : 0
100,000 and over : 1
```

4.5.2 RETURN Statement

The `RETURN` statement terminates the current function, procedure or anonymous block and returns control to the caller.

There are two forms of the `RETURN` Statement. The first form of the `RETURN` statement is used to terminate a procedure or function that returns `void`. The syntax of the first form is:

```
RETURN;
```

The second form of `RETURN` returns a value to the caller. The syntax of the second form of the `RETURN` statement is:

```
RETURN expression;
```

`expression` must evaluate to the same data type as the return type of the function.

The following example uses the `RETURN` statement returns a value to the caller:

```
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 emp_comp;
```

4.5.3 GOTO Statement

The `GOTO` statement causes the point of execution to jump to the statement with the specified label. The syntax of a `GOTO` statement is:

```
GOTO label
```

`label` is a name assigned to an executable statement. *`label`* must be unique within the scope of the function, procedure or anonymous block.

To label a statement, use the syntax:

```
<<label>> statement
```

`statement` is the point of execution that the program jumps to.

You can label assignment statements, any SQL statement (like `INSERT`, `UPDATE`, `CREATE`, etc.) and selected procedural language statements. The procedural language statements that can be labeled are:

- `IF`
- `EXIT`
- `RETURN`
- `RAISE`
- `EXECUTE`
- `PERFORM`
- `GET DIAGNOSTICS`
- `OPEN`
- `FETCH`
- `MOVE`
- `CLOSE`
- `NULL`
- `COMMIT`
- `ROLLBACK`
- `GOTO`
- `CASE`
- `LOOP`
- `WHILE`
- `FOR`

Please note that `exit` is considered a keyword, and cannot be used as the name of a label.

`GOTO` statements cannot transfer control *into* a conditional block or sub-block, but can transfer control *from* a conditional block or sub-block.

The following example verifies that an employee record contains a name, job description, and employee hire date; if any piece of information is missing, a `GOTO` statement transfers the point of execution to a statement that prints a message that the employee is not valid.

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno         NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_hiredate      emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, hiredate
        INTO v_ename, v_job, v_hiredate FROM emp
        WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        GOTO invalid_emp;
    END IF;
    IF v_job IS NULL THEN
        GOTO invalid_emp;
    END IF;
    IF v_hiredate IS NULL THEN
        GOTO invalid_emp;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' validated without errors.');
    RETURN;
    <<invalid_emp>> DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' is not a valid employee.');
END;
```

`GOTO` statements have the following restrictions:

- A `GOTO` statement cannot jump to a declaration.
- A `GOTO` statement cannot transfer control to another function or procedure.
- A *`label`* should not be placed at the end of a block, function or procedure.

4.5.4 CASE Expression

The CASE expression returns a value that is substituted where the CASE expression is located within an expression.

There are two formats of the CASE expression - one that is called a *searched* CASE and the other that uses a *selector*.

4.5.4.1 Selector CASE Expression

The selector CASE expression attempts to match an expression called the selector to the expression specified in one or more WHEN clauses. *result* is an expression that is type-compatible in the context where the CASE expression is used. If a match is found, the value given in the corresponding THEN clause is returned by the CASE expression. If there are no matches, the value following ELSE is returned. If ELSE is omitted, the CASE expression returns null.

```
CASE selector-expression
  WHEN match-expression THEN
    result
[ WHEN match-expression THEN
    result
[ WHEN match-expression THEN
    result ] ...]
[ ELSE
    result ]
END;
```

match-expression is evaluated in the order in which it appears within the CASE expression. *result* is an expression that is type-compatible in the context where the CASE expression is used. When the first *match-expression* is encountered that equals *selector-expression*, *result* in the corresponding THEN clause is returned as the value of the CASE expression. If none of *match-expression* equals *selector-expression* then *result* following ELSE is returned. If no ELSE is specified, the CASE expression returns null.

The following example uses a selector CASE expression to assign the department name to a variable based upon the department number.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        v_dname :=
            CASE v_deptno
                WHEN 10 THEN 'Accounting'
                WHEN 20 THEN 'Research'
                WHEN 30 THEN 'Sales'
                WHEN 40 THEN 'Operations'
                ELSE 'unknown'
            END;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '       ' || v_dname);
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following is the output from this program.

```
EMPNO    ENAME     DEPTNO    DNAME
-----    -------   ------    ----------
7369     SMITH       20      Research
7499     ALLEN       30      Sales
7521     WARD        30      Sales
7566     JONES       20      Research
7654     MARTIN      30      Sales
7698     BLAKE       30      Sales
7782     CLARK       10      Accounting
7788     SCOTT       20      Research
7839     KING        10      Accounting
7844     TURNER      30      Sales
7876     ADAMS       20      Research
7900     JAMES       30      Sales
7902     FORD        20      Research
7934     MILLER      10      Accounting
```

4.5.4.2 Searched CASE Expression

A searched `CASE` expression uses one or more Boolean expressions to determine the resulting value to return.

```
CASE WHEN boolean-expression THEN
     result
[ WHEN boolean-expression THEN
     result
 [ WHEN boolean-expression THEN
     result ] ...]
[ ELSE
     result ]
END;
```

boolean-expression is evaluated in the order in which it appears within the `CASE` expression. *result* is an expression that is type-compatible in the context where the `CASE` expression is used. When the first *boolean-expression* is encountered that evaluates to `TRUE`, *`result`* in the corresponding `THEN` clause is returned as the value of the `CASE` expression. If none of *`boolean-expression`* evaluates to true then *`result`* following `ELSE` is returned. If no `ELSE` is specified, the `CASE` expression returns null.

The following example uses a searched `CASE` expression to assign the department name to a variable based upon the department number.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        v_dname :=
            CASE
                WHEN v_deptno = 10 THEN 'Accounting'
                WHEN v_deptno = 20 THEN 'Research'
                WHEN v_deptno = 30 THEN 'Sales'
                WHEN v_deptno = 40 THEN 'Operations'
                ELSE 'unknown'
            END;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || v_dname);
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following is the output from this program.

```
EMPNO    ENAME     DEPTNO    DNAME
-----    -------   ------    ----------
7369     SMITH       20      Research
7499     ALLEN       30      Sales
7521     WARD        30      Sales
7566     JONES       20      Research
7654     MARTIN      30      Sales
7698     BLAKE       30      Sales
7782     CLARK       10      Accounting
7788     SCOTT       20      Research
7839     KING        10      Accounting
7844     TURNER      30      Sales
7876     ADAMS       20      Research
7900     JAMES       30      Sales
7902     FORD        20      Research
7934     MILLER      10      Accounting
```

4.5.5 CASE Statement

The CASE statement executes a set of one or more statements when a specified search condition is TRUE. The CASE statement is a stand-alone statement in itself while the previously discussed CASE expression must appear as part of an expression.

There are two formats of the CASE statement - one that is called a *searched* CASE and the other that uses a *selector*.

4.5.5.1 Selector CASE Statement

The selector CASE statement attempts to match an expression called the selector to the expression specified in one or more WHEN clauses. When a match is found one or more corresponding statements are executed.

```
  CASE selector-expression
  WHEN match-expression THEN
    statements
[ WHEN match-expression THEN
    statements
[ WHEN match-expression THEN
    statements ] ...]
[ ELSE
    statements ]
  END CASE;
```

selector-expression returns a value type-compatible with each *match-expression*. *match-expression* is evaluated in the order in which it appears within the CASE statement. *statements* are one or more SPL statements, each terminated by a semi-colon. When the value of *selector-expression* equals the first *match-expression*, the statement(s) in the corresponding THEN clause are executed and control continues following the END CASE keywords. If there are no matches, the statement(s) following ELSE are executed. If there are no matches and there is no ELSE clause, an exception is thrown.

The following example uses a selector CASE statement to assign a department name and location to a variable based upon the department number.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
```

```
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE v_deptno
            WHEN 10 THEN v_dname := 'Accounting';
                         v_loc   := 'New York';
            WHEN 20 THEN v_dname := 'Research';
                         v_loc   := 'Dallas';
            WHEN 30 THEN v_dname := 'Sales';
                         v_loc   := 'Chicago';
            WHEN 40 THEN v_dname := 'Operations';
                         v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                         v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following is the output from this program.

```
EMPNO    ENAME     DEPTNO    DNAME          LOC
-----    -------   ------    ----------     ---------
7369     SMITH       20      Research       Dallas
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
7698     BLAKE       30      Sales          Chicago
7782     CLARK       10      Accounting     New York
7788     SCOTT       20      Research       Dallas
7839     KING        10      Accounting     New York
7844     TURNER      30      Sales          Chicago
7876     ADAMS       20      Research       Dallas
7900     JAMES       30      Sales          Chicago
7902     FORD        20      Research       Dallas
7934     MILLER      10      Accounting     New York
```

4.5.5.2 Searched CASE statement

A searched `CASE` statement uses one or more Boolean expressions to determine the resulting set of statements to execute.

```
  CASE WHEN boolean-expression THEN
     statements
[ WHEN boolean-expression THEN
     statements
[ WHEN boolean-expression THEN
     statements ] ...]
[ ELSE
```

```
      statements ]
   END CASE;
```

boolean-expression is evaluated in the order in which it appears within the CASE statement. When the first *boolean-expression* is encountered that evaluates to TRUE, the statement(s) in the corresponding THEN clause are executed and control continues following the END CASE keywords. If none of *boolean-expression* evaluates to TRUE, the statement(s) following ELSE are executed. If none of *boolean-expression* evaluates to TRUE and there is no ELSE clause, an exception is thrown.

The following example uses a searched CASE statement to assign a department name and location to a variable based upon the department number.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE
            WHEN v_deptno = 10 THEN v_dname := 'Accounting';
                                    v_loc   := 'New York';
            WHEN v_deptno = 20 THEN v_dname := 'Research';
                                    v_loc   := 'Dallas';
            WHEN v_deptno = 30 THEN v_dname := 'Sales';
                                    v_loc   := 'Chicago';
            WHEN v_deptno = 40 THEN v_dname := 'Operations';
                                    v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                                    v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '       ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following is the output from this program.

```
EMPNO    ENAME     DEPTNO    DNAME          LOC
-----    -------   ------    ----------     ---------
7369     SMITH       20      Research       Dallas
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
```

```
7698     BLAKE      30     Sales         Chicago
7782     CLARK      10     Accounting    New York
7788     SCOTT      20     Research      Dallas
7839     KING       10     Accounting    New York
7844     TURNER     30     Sales         Chicago
7876     ADAMS      20     Research      Dallas
7900     JAMES      30     Sales         Chicago
7902     FORD       20     Research      Dallas
7934     MILLER     10     Accounting    New York
```

4.5.6 Loops

With the `LOOP`, `EXIT`, `CONTINUE`, `WHILE`, and `FOR` statements, you can arrange for your SPL program to repeat a series of commands.

4.5.6.1 LOOP

```
LOOP
    statements
END LOOP;
```

`LOOP` defines an unconditional loop that is repeated indefinitely until terminated by an `EXIT` or `RETURN` statement.

4.5.6.2 EXIT

```
EXIT [ WHEN expression ];
```

The innermost loop is terminated and the statement following `END LOOP` is executed next.

If `WHEN` is present, loop exit occurs only if the specified condition is `TRUE`, otherwise control passes to the statement after `EXIT`.

`EXIT` can be used to cause early exit from all types of loops; it is not limited to use with unconditional loops.

The following is a simple example of a loop that iterates ten times and then uses the `EXIT` statement to terminate.

```
DECLARE
    v_counter       NUMBER(2);
BEGIN
    v_counter := 1;
    LOOP
        EXIT WHEN v_counter > 10;
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
        v_counter := v_counter + 1;
    END LOOP;
END;
```

The following is the output from this program.

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

4.5.6.3 CONTINUE

The `CONTINUE` statement provides a way to proceed with the next iteration of a loop while skipping intervening statements.

When the `CONTINUE` statement is encountered, the next iteration of the innermost loop is begun, skipping all statements following the `CONTINUE` statement until the end of the loop. That is, control is passed back to the loop control expression, if any, and the body of the loop is re-evaluated.

If the `WHEN` clause is used, then the next iteration of the loop is begun only if the specified expression in the `WHEN` clause evaluates to `TRUE`. Otherwise, control is passed to the next statement following the `CONTINUE` statement.

The `CONTINUE` statement may not be used outside of a loop.

The following is a variation of the previous example that uses the `CONTINUE` statement to skip the display of the odd numbers.

```
DECLARE
    v_counter       NUMBER(2);
BEGIN
    v_counter := 0;
    LOOP
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 10;
        CONTINUE WHEN MOD(v_counter,2) = 1;
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
    END LOOP;
END;
```

The following is the output from above program.

```
Iteration # 2
Iteration # 4
Iteration # 6
Iteration # 8
Iteration # 10
```

4.5.6.4 WHILE

```
WHILE expression LOOP
    statements
END LOOP;
```

The `WHILE` statement repeats a sequence of statements so long as the condition expression evaluates to `TRUE`. The condition is checked just before each entry to the loop body.

The following example contains the same logic as in the previous example except the `WHILE` statement is used to take the place of the `EXIT` statement to determine when to exit the loop.

Note: The conditional expression used to determine when to exit the loop must be altered. The `EXIT` statement terminates the loop when its conditional expression is true. The `WHILE` statement terminates (or never begins the loop) when its conditional expression is false.

```
DECLARE
    v_counter       NUMBER(2);
BEGIN
    v_counter := 1;
    WHILE v_counter <= 10 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
        v_counter := v_counter + 1;
    END LOOP;
END;
```

The same result is generated by this example as in the prior example.

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

4.5.6.5 FOR (integer variant)

```
FOR name IN [REVERSE] expression .. expression LOOP
    statements
END LOOP;
```

This form of `FOR` creates a loop that iterates over a range of integer values. The variable *name* is automatically defined as type `INTEGER` and exists only inside the loop. The two expressions giving the loop range are evaluated once when entering the loop. The iteration step is +1 and *name* begins with the value of *expression* to the left of `..` and terminates once *name* exceeds the value of *expression* to the right of `..`. Thus the two expressions take on the following roles: *start-value* .. *end-value*.

The optional `REVERSE` clause specifies that the loop should iterate in reverse order. The first time through the loop, *name* is set to the value of the right-most *expression*; the loop terminates when the *name* is less than the left-most *expression*.

The following example simplifies the `WHILE` loop example even further by using a `FOR` loop that iterates from 1 to 10.

```
BEGIN
    FOR i IN 1 .. 10 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
    END LOOP;
END;
```

Here is the output using the `FOR` statement.

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

If the start value is greater than the end value the loop body is not executed at all. No error is raised as shown by the following example.

```
BEGIN
    FOR i IN 10 .. 1 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
    END LOOP;
END;
```

There is no output from this example as the loop body is never executed.

Note: SPL also supports `CURSOR FOR` loops (see Section 4.8.7).

4.5.7 Exception Handling

By default, any error occurring in an SPL program aborts execution of the program. You can trap errors and recover from them by using a `BEGIN` block with an `EXCEPTION` section. The syntax is an extension of the normal syntax for a `BEGIN` block:

```
[ DECLARE
    declarations ]
  BEGIN
    statements
  EXCEPTION
    WHEN condition [ OR condition ]... THEN
      handler_statements
  [ WHEN condition [ OR condition ]... THEN
      handler_statements ]...
  END;
```

If no error occurs, this form of block simply executes all the *statements*, and then control passes to the next statement after `END`. If an error occurs within the *statements*, further processing of the *statements* is abandoned, and control passes to the `EXCEPTION` list. The list is searched for the first *condition* matching the error that occurred. If a match is found, the corresponding *handler_statements* are executed, and then control passes to the next statement after `END`. If no match is found, the error propagates out as though the `EXCEPTION` clause were not there at all. The error can be caught by an enclosing block with `EXCEPTION`; if there is no enclosing block, it aborts processing of the subprogram.

The special condition name `OTHERS` matches every error type. Condition names are not case-sensitive.

If a new error occurs within the selected *handler_statements*, it cannot be caught by this `EXCEPTION` clause, but is propagated out. A surrounding `EXCEPTION` clause could catch it.

The following table lists the condition names that may be used:

Table 4-4-2 Exception Condition Names

Condition Name	Description
`CASE_NOT_FOUND`	The application has encountered a situation where none of the cases in a `CASE` statement evaluates to `TRUE` and there is no `ELSE` condition.
`COLLECTION_IS_NULL`	The application has attempted to invoke a collection method on a null collection such as an uninitialized nested table.
`CURSOR_ALREADY_OPEN`	The application has attempted to open a cursor that is already open.

Condition Name	Description
`DUP_VAL_ON_INDEX`	The application has attempted to store a duplicate value that currently exists within a constrained column.
`INVALID_CURSOR`	The application has attempted to access an unopened cursor.
`INVALID_NUMBER`	The application has attempted to convert a string to a numeric literal that cannot be converted (applicable only to SQL statements).
`NO_DATA_FOUND`	No rows satisfy the selection criteria.
`OTHERS`	The application has encountered an exception that hasn't been caught by a prior condition in the exception section.
`SUBSCRIPT_BEYOND_COUNT`	The application has attempted to reference a subscript of a nested table or varray beyond its initialized or extended size.
`SUBSCRIPT_OUTSIDE_LIMIT`	The application has attempted to reference a subscript or extend a varray beyond its maximum size limit.
`TOO_MANY_ROWS`	The application has encountered more than one row that satisfies the selection criteria (where only one row is allowed to be returned).
`VALUE_ERROR`	The application has tried to convert a string to a numeric literal that cannot be converted (applicable only to procedural statements).
`ZERO_DIVIDE`	The application has tried to divide by zero.
User-defined Exception	See Section 4.5.8

4.5.8 User-defined Exceptions

Any number of errors (referred to in PL/SQL as *exceptions*) can occur during program execution. When an exception is *thrown*, normal execution of the program stops, and control of the program transfers to the error-handling portion of the program. An *exception* may be a pre-defined error that is generated by the server, or may be a logical error that raises a user-defined exception.

User-defined exceptions are never raised by the server; they are raised explicitly by a `RAISE` statement. A user-defined exception is raised when a developer-defined logical rule is broken; a common example of a logical rule being broken occurs when a check is presented against an account with insufficient funds. An attempt to cash a check against an account with insufficient funds will provoke a user-defined exception.

You can define exceptions in functions, procedures, packages or anonymous blocks. While you cannot declare the same exception twice in the same block, you can declare the same exception in two different blocks.

Before implementing a user-defined exception, you must declare the exception in the declaration section of a function, procedure, package or anonymous block. You can then raise the exception using the `RAISE` statement:

```
DECLARE
    exception_name EXCEPTION;

BEGIN
    ...
    RAISE exception_name;
    ...
END;
```

`exception_name` is the name of the exception.

Unhandled exceptions propagate back through the call stack. If the exception remains unhandled, the exception is eventually reported to the client application.

User-defined exceptions declared in a block are considered to be local to that block, and global to any nested blocks within the block. To reference an exception that resides in an outer block, you must assign a label to the outer block; then, preface the name of the exception with the block name:

```
block_name.exception_name
```

Conversely, outer blocks cannot reference exceptions declared in nested blocks.

The scope of a declaration is limited to the block in which it is declared *unless* it is created in a package, and when referenced, qualified by the package name. For example, to raise an exception named `out_of_stock` that resides in a package named `inventory_control` a program must raise an error named:

```
inventory_control.out_of_stock
```

The following example demonstrates declaring a user-defined exception in a package. The user-defined exception does not require a package-qualifier when it is raised in `check_balance`, since it resides in the same package as the exception:

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
   PROCEDURE check_balance(p_balance NUMBER, p_amount  NUMBER)
   IS
   BEGIN
       IF (p_amount > p_balance) THEN
         RAISE overdrawn;
       END IF;
    END;
```

The following procedure (`purchase`) calls the `check_balance` procedure. If `p_amount` is greater than `p_balance`, `check_balance` raises an exception; `purchase` catches the `ar.overdrawn` exception. `purchase` must refer to the exception with a package-qualified name (`ar.overdrawn`) because `purchase` is not defined within the `ar` package.

```
CREATE PROCEDURE purchase(customerID INT, amount NUMERIC)
AS
  BEGIN
     ar.check_ balance(getcustomerbalance(customerid), amount);
       record_purchase(customerid, amount);
  EXCEPTION
     WHEN ar.overdrawn THEN
       raise_credit_limit(customerid, amount*1.5);
  END;
```

When `ar.check_balance` raises an exception, execution jumps to the exception handler defined in `purchase`:

```
EXCEPTION
     WHEN ar.overdrawn THEN
       raise_credit_limit(customerid, amount*1.5);
```

The exception handler raises the customer's credit limit and ends. When the exception handler ends, execution resumes with the statement that follows `ar.check_balance`.

4.5.9 PRAGMA EXCEPTION_INIT

`PRAGMA EXCEPTION_INIT` associates a user-defined error code with an exception. A `PRAGMA EXCEPTION_INIT` declaration may be included in any block, sub-block or package. You can only assign an error code to an exception (using `PRAGMA EXCEPTION_INIT`) after declaring the exception. The format of a `PRAGMA EXCEPTION_INIT` declaration is:

```
PRAGMA EXCEPTION_INIT(exception_name,
                      {exception_number | exception_code})
```

Where:

exception_name is the name of the associated exception.

exception_number is a user-defined error code associated with the pragma. If you specify an unmapped *exception_number*, the server will return a warning.

exception_code is the name of a pre-defined exception. For a complete list of valid exceptions, see the Postgres core documentation available at:

http://www.postgresql.org/docs/9.5/static/errcodes-appendix.html

The previous section (*User-defined Exceptions*) included an example that demonstrates declaring a user-defined exception in a package. The following example uses the same basic structure, but adds a `PRAGMA EXCEPTION_INIT` declaration:

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PRAGMA EXCEPTION_INIT (overdrawn, -20100);
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
   PROCEDURE check_balance(p_balance NUMBER, p_amount  NUMBER)
   IS
   BEGIN
       IF (p_amount > p_balance) THEN
         RAISE overdrawn;
       END IF;
    END;
```

The following procedure (`purchase`) calls the `check_balance` procedure. If `p_amount` is greater than `p_balance`, `check_balance` raises an exception; `purchase` catches the `ar.overdrawn` exception.

```
CREATE PROCEDURE purchase(customerID int, amount NUMERIC)
AS
  BEGIN
     ar.check_ balance(getcustomerbalance(customerid), amount);
       record_purchase(customerid, amount);
  EXCEPTION
     WHEN ar.overdrawn THEN
      DBMS_OUTPUT.PUT_LINE ('This account is overdrawn.');
      DBMS_OUTPUT.PUT_LINE ('SQLCode :'||SQLCODE||' '||SQLERRM );
END;
```

When `ar.check_balance` raises an exception, execution jumps to the exception handler defined in `purchase`.

```
EXCEPTION
     WHEN ar.overdrawn THEN
      DBMS_OUTPUT.PUT_LINE ('This account is overdrawn.');
      DBMS_OUTPUT.PUT_LINE ('SQLCode :'||SQLCODE||' '||SQLERRM );
```

The exception handler returns an error message, followed by `SQLCODE` information:

```
This account is overdrawn.
SQLCODE: -20100 User-Defined Exception
```

The following example demonstrates using a pre-defined exception. The code creates a more meaningful name for the `no_data_found exception`; if the given customer does not exist, the code catches the exception, calls `DBMS_OUTPUT.PUT_LINE` to report the error, and then re-raises the original exception:

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PRAGMA EXCEPTION_INIT (unknown_customer, no_data_found);
  PROCEDURE check_balance(p_customer_id NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
   PROCEDURE check_balance(p_customer_id NUMBER)
   IS
   DECLARE
     v_balance NUMBER;
   BEGIN
     SELECT balance INTO v_balance FROM customer
       WHERE cust_id = p_customer_id;
   EXCEPTION WHEN unknown_customer THEN
     DBMS_OUTPUT.PUT_LINE('invalid customer id');
     RAISE;
   END;
```

4.5.10 RAISE_APPLICATION_ERROR

The procedure, RAISE_APPLICATION_ERROR, allows a developer to intentionally abort processing within an SPL program from which it is called by causing an exception. The exception is handled in the same manner as described in Section 4.5.7. In addition, the RAISE_APPLICATION_ERROR procedure makes a user-defined code and error message available to the program which can then be used to identify the exception.

```
RAISE_APPLICATION_ERROR(error_number, message);
```

Where:

error_number is an integer value or expression that is returned in a variable named SQLCODE when the procedure is executed. *error_number* must be a value between -20000 and -20999.

message is a string literal or expression that is returned in a variable named SQLERRM.

For additional information on the SQLCODE and SQLERRM variables, see Section 4.13, *Errors and Messages*.

The following example uses the RAISE_APPLICATION_ERROR procedure to display a different code and message depending upon the information missing from an employee.

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno         NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_mgr           emp.mgr%TYPE;
    v_hiredate      emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, mgr, hiredate
        INTO v_ename, v_job, v_mgr, v_hiredate FROM emp
        WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        RAISE_APPLICATION_ERROR(-20010, 'No name for ' || p_empno);
    END IF;
    IF v_job IS NULL THEN
        RAISE_APPLICATION_ERROR(-20020, 'No job for' || p_empno);
    END IF;
    IF v_mgr IS NULL THEN
        RAISE_APPLICATION_ERROR(-20030, 'No manager for ' || p_empno);
    END IF;
    IF v_hiredate IS NULL THEN
        RAISE_APPLICATION_ERROR(-20040, 'No hire date for ' || p_empno);
    END IF;
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' validated without errors');
EXCEPTION
```

```
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
END;
```

The following shows the output in a case where the manager number is missing from an employee record.

```
EXEC verify_emp(7839);

SQLCODE: -20030
SQLERRM: EDB-20030: No manager for 7839
```

4.6 Transaction Control

There may be circumstances where it is desired that all updates to a database are to occur successfully, or none are to occur at all if any error occurs. A set of database updates that are to all occur successfully as a single unit, or are not to occur at all, is said to be a *transaction*.

A common example in banking is a funds transfer between two accounts. The two parts of the transaction are the withdrawal of funds from one account, and the deposit of the funds in another account. Both parts of this transaction must occur otherwise the bank's books will be out of balance. The deposit and withdrawal are one transaction.

An SPL application can be created that uses a style of transaction control compatible with Oracle databases if the following conditions are met:

- The `edb_stmt_level_tx` parameter must be set to `TRUE`. This prevents the action of unconditionally rolling back all database updates within the `BEGIN`/`END` block if any exception occurs. See Section 1.3.4 for more information on the `edb_stmt_level_tx` parameter.
- The application must not be running in autocommit mode. If autocommit mode is on, each successful database update is immediately committed and cannot be undone. The manner in which autocommit mode is turned on or off is application dependent.

A transaction begins when the first SQL command is encountered in the SPL program. All subsequent SQL commands are included as part of that transaction. The transaction ends when one of the following occurs:

- An unhandled exception occurs in which case the effects of all database updates made during the transaction are rolled back and the transaction is aborted.
- A `COMMIT` command is encountered in which case the effect of all database updates made during the transaction become permanent.
- A `ROLLBACK` command is encountered in which case the effects of all database updates made during the transaction are rolled back and the transaction is aborted. If a new SQL command is encountered, a new transaction begins.
- Control returns to the calling application (such as Java, PSQL, etc.) in which case the action of the application determines whether the transaction is committed or rolled back.

Note: Unlike Oracle, DDL commands such as `CREATE TABLE` do not implicitly occur within their own transaction. Therefore, DDL commands do not automatically cause an immediate database commit as in Oracle, and DDL commands may be rolled back just like DML commands.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

A transaction may span one or more BEGIN/END blocks, or a single BEGIN/END block may contain one or more transactions.

The following sections discuss the COMMIT and ROLLBACK commands in more detail.

4.6.1 COMMIT

The COMMIT command makes all database updates made during the current transaction permanent, and ends the current transaction.

```
COMMIT [ WORK ];
```

The COMMIT command may be used within anonymous blocks, stored procedures, or functions. Within an SPL program, it may appear in the executable section and/or the exception section.

In the following example, the third INSERT command in the anonymous block results in an error. The effect of the first two INSERT commands are retained as shown by the first SELECT command. Even after issuing a ROLLBACK command, the two rows remain in the table as shown by the second SELECT command verifying that they were indeed committed.

Note: The edb_stmt_level_tx configuration parameter shown in the example below can be set for the entire database using the ALTER DATABASE command, or it can be set for the entire database server by changing it in the postgresql.conf file.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

BEGIN
    INSERT INTO dept VALUES (50, 'FINANCE', 'DALLAS');
    INSERT INTO dept VALUES (60, 'MARKETING', 'CHICAGO');
    COMMIT;
    INSERT INTO dept VALUES (70, 'HUMAN RESOURCES', 'CHICAGO');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

SQLERRM: value too long for type character varying(14)
SQLCODE: 22001

SELECT * FROM dept;

deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | FINANCE    | DALLAS
     60 | MARKETING  | CHICAGO
(6 rows)

ROLLBACK;

SELECT * FROM dept;

deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | FINANCE    | DALLAS
     60 | MARKETING  | CHICAGO
(6 rows)
```

4.6.2 ROLLBACK

The ROLLBACK command undoes all database updates made during the current transaction, and ends the current transaction.

```
ROLLBACK [ WORK ];
```

The ROLLBACK command may be used within anonymous blocks, stored procedures, or functions. Within an SPL program, it may appear in the executable section and/or the exception section.

In the following example, the exception section contains a ROLLBACK command. Even though the first two INSERT commands are executed successfully, the third results in an exception that results in the rollback of all the INSERT commands in the anonymous block.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

BEGIN
    INSERT INTO dept VALUES (50, 'FINANCE', 'DALLAS');
    INSERT INTO dept VALUES (60, 'MARKETING', 'CHICAGO');
    INSERT INTO dept VALUES (70, 'HUMAN RESOURCES', 'CHICAGO');
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

SQLERRM: value too long for type character varying(14)
SQLCODE: 22001

SELECT * FROM dept;

deptno |   dname    |   loc
```

```
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)
```

The following is a more complex example using both `COMMIT` and `ROLLBACK`. First, the following stored procedure is created which inserts a new employee.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

CREATE OR REPLACE PROCEDURE emp_insert (
    p_empno         IN emp.empno%TYPE,
    p_ename         IN emp.ename%TYPE,
    p_job           IN emp.job%TYPE,
    p_mgr           IN emp.mgr%TYPE,
    p_hiredate      IN emp.hiredate%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE,
    p_deptno        IN emp.deptno%TYPE
)
IS
BEGIN
    INSERT INTO emp VALUES (
        p_empno,
        p_ename,
        p_job,
        p_mgr,
        p_hiredate,
        p_sal,
        p_comm,
        p_deptno);

    DBMS_OUTPUT.PUT_LINE('Added employee...');
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || p_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || p_job);
    DBMS_OUTPUT.PUT_LINE('Manager    : ' || p_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || p_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || p_sal);
    DBMS_OUTPUT.PUT_LINE('Commission : ' || p_comm);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || p_deptno);
    DBMS_OUTPUT.PUT_LINE('----------------------');
END;
```

Note that this procedure has no exception section so any error that may occur is propagated up to the calling program.

The following anonymous block is run. Note the use of the `COMMIT` command after all calls to the `emp_insert` procedure and the `ROLLBACK` command in the exception section.

```
BEGIN
    emp_insert(9601,'FARRELL','ANALYST',7902,'03-MAR-08',5000,NULL,40);
    emp_insert(9602,'TYLER','ANALYST',7900,'25-JAN-08',4800,NULL,40);
    COMMIT;
EXCEPTION
```

```
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('An error occurred - roll back inserts');
        ROLLBACK;
END;

Added employee...
Employee # : 9601
Name       : FARRELL
Job        : ANALYST
Manager    : 7902
Hire Date  : 03-MAR-08 00:00:00
Salary     : 5000
Commission :
Dept #     : 40
----------------------
Added employee...
Employee # : 9602
Name       : TYLER
Job        : ANALYST
Manager    : 7900
Hire Date  : 25-JAN-08 00:00:00
Salary     : 4800
Commission :
Dept #     : 40
----------------------
```

The following `SELECT` command shows that employees Farrell and Tyler were successfully added.

```
SELECT * FROM emp WHERE empno > 9600;

empno |  ename  |   job   | mgr  |      hiredate      |   sal   | comm | deptno
-------+---------+---------+------+--------------------+---------+------+--------
  9601 | FARRELL | ANALYST | 7902 | 03-MAR-08 00:00:00 | 5000.00 |      |     40
  9602 | TYLER   | ANALYST | 7900 | 25-JAN-08 00:00:00 | 4800.00 |      |     40
(2 rows)
```

Now, execute the following anonymous block:

```
BEGIN
    emp_insert(9603,'HARRISON','SALESMAN',7902,'13-DEC-07',5000,3000,20);
    emp_insert(9604,'JARVIS','SALESMAN',7902,'05-MAY-08',4800,4100,11);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('An error occurred - roll back inserts');
        ROLLBACK;
END;

Added employee...
Employee # : 9603
Name       : HARRISON
Job        : SALESMAN
Manager    : 7902
Hire Date  : 13-DEC-07 00:00:00
Salary     : 5000
Commission : 3000
Dept #     : 20
----------------------
```

```
SQLERRM: insert or update on table "emp" violates foreign key constraint
"emp_ref_dept_fk"
An error occurred - roll back inserts
```

A `SELECT` command run against the table yields the following:

```
SELECT * FROM emp WHERE empno > 9600;

empno |  ename   |   job    | mgr  |      hiredate      |   sal    | comm | deptno
-------+---------+---------+------+--------------------+---------+------+--------
  9601 | FARRELL | ANALYST | 7902 | 03-MAR-08 00:00:00 | 5000.00 |      |     40
  9602 | TYLER   | ANALYST | 7900 | 25-JAN-08 00:00:00 | 4800.00 |      |     40
(2 rows)
```

The `ROLLBACK` command in the exception section successfully undoes the insert of employee Harrison. Also note that employees Farrell and Tyler are still in the table as their inserts were made permanent by the `COMMIT` command in the first anonymous block.

4.7 Dynamic SQL

Dynamic SQL is a technique that provides the ability to execute SQL commands that are not known until the commands are about to be executed. Up to this point, the SQL commands that have been illustrated in SPL programs have been static SQL - the full command (with the exception of variables) must be known and coded into the program before the program, itself, can begin to execute. Thus using dynamic SQL, the executed SQL can change during program runtime.

In addition, dynamic SQL is the only method by which data definition commands, such as `CREATE TABLE`, can be executed from within an SPL program.

Note, however, that the runtime performance of dynamic SQL will be slower than static SQL.

The `EXECUTE IMMEDIATE` command is used to run SQL commands dynamically.

```
EXECUTE IMMEDIATE 'sql_expression;'
  [ INTO { variable [, ...] | record } ]
  [ USING expression [, ...] ]
```

`sql_expression` is a string expression containing the SQL command to be dynamically executed. *`variable`* receives the output of the result set, typically from a `SELECT` command, created as a result of executing the SQL command in *`sql_expression`*. The number, order, and type of variables must match the number, order, and be type-compatible with the fields of the result set. Alternatively, a record can be specified as long as the record's fields match the number, order, and are type-compatible with the result set. When using the `INTO` clause, exactly one row must be returned in the result set, otherwise an exception occurs. When using the `USING` clause the value of *`expression`* is passed to a *placeholder*. Placeholders appear embedded within the SQL command in *`sql_expression`* where variables may be used. Placeholders are denoted by an identifier with a colon (`:`) prefix - `:`*`name`*. The number, order, and resultant data types of the evaluated expressions must match the number, order and be type-compatible with the placeholders in *`sql_expression`*. Note that placeholders are not declared anywhere in the SPL program – they only appear in *`sql_expression`*.

The following example shows basic dynamic SQL commands as string literals.

```
DECLARE
    v_sql           VARCHAR2(50);
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE job (jobno NUMBER(3),' ||
        ' jname VARCHAR2(9))';
    v_sql := 'INSERT INTO job VALUES (100, ''ANALYST'')';
    EXECUTE IMMEDIATE v_sql;
    v_sql := 'INSERT INTO job VALUES (200, ''CLERK'')';
    EXECUTE IMMEDIATE v_sql;
END;
```

The following example illustrates the `USING` clause to pass values to placeholders in the SQL string.

```
DECLARE
    v_sql           VARCHAR2(50) := 'INSERT INTO job VALUES ' ||
                        '(:p_jobno, :p_jname)';
    v_jobno         job.jobno%TYPE;
    v_jname         job.jname%TYPE;
BEGIN
    v_jobno := 300;
    v_jname := 'MANAGER';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
    v_jobno := 400;
    v_jname := 'SALESMAN';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
    v_jobno := 500;
    v_jname := 'PRESIDENT';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
END;
```

The following example shows both the `INTO` and `USING` clauses. Note the last execution of the `SELECT` command returns the results into a record instead of individual variables.

```
DECLARE
    v_sql           VARCHAR2(60);
    v_jobno         job.jobno%TYPE;
    v_jname         job.jname%TYPE;
    r_job           job%ROWTYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('JOBNO    JNAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    v_sql := 'SELECT jobno, jname FROM job WHERE jobno = :p_jobno';
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 100;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 200;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 300;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 400;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO r_job USING 500;
    DBMS_OUTPUT.PUT_LINE(r_job.jobno || '      ' || r_job.jname);
END;
```

The following is the output from the previous anonymous block:

```
JOBNO    JNAME
-----    -------
100      ANALYST
200      CLERK
300      MANAGER
400      SALESMAN
```

```
500       PRESIDENT
```

You can use the `BULK COLLECT` clause to assemble the result set from an `EXECUTE IMMEDIATE` statement into a named collection. See Section 4.12.4, `EXECUTE IMMEDIATE BULK COLLECT` for information about using the `BULK COLLECT` clause.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

4.8 Static Cursors

Rather than executing a whole query at once, it is possible to set up a *cursor* that encapsulates the query, and then read the query result set one row at a time. This allows the creation of SPL program logic that retrieves a row from the result set, does some processing on the data in that row, and then retrieves the next row and repeats the process.

Cursors are most often used in the context of a `FOR` or `WHILE` loop. A conditional test should be included in the SPL logic that detects when the end of the result set has been reached so the program can exit the loop.

4.8.1 Declaring a Cursor

In order to use a cursor, it must first be declared in the declaration section of the SPL program. A cursor declaration appears as follows:

```
CURSOR name IS query;
```

`name` is an identifier that will be used to reference the cursor and its result set later in the program. *`query`* is a SQL `SELECT` command that determines the result set retrievable by the cursor.

Note: An extension of this syntax allows the use of parameters. This is discussed in more detail in Section 4.8.8.

The following are some examples of cursor declarations:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_1 IS SELECT * FROM emp;
    CURSOR emp_cur_2 IS SELECT empno, ename FROM emp;
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    ...
END;
```

4.8.2 Opening a Cursor

Before a cursor can be used to retrieve rows, it must first be opened. This is accomplished with the `OPEN` statement.

```
OPEN name;
```

`name` is the identifier of a cursor that has been previously declared in the declaration section of the SPL program. The `OPEN` statement must not be executed on a cursor that has already been, and still is open.

The following shows an `OPEN` statement with its corresponding cursor declaration.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
        ...
END;
```

4.8.3 Fetching Rows From a Cursor

Once a cursor has been opened, rows can be retrieved from the cursor's result set by using the `FETCH` statement.

```
FETCH name INTO { record | variable [, variable_2 ]... };
```

name is the identifier of a previously opened cursor. *record* is the identifier of a previously defined record (for example, using *table*%ROWTYPE). *variable*, *variable_2*... are SPL variables that will receive the field data from the fetched row. The fields in *record* or *variable*, *variable_2*... must match in number and order, the fields returned in the `SELECT` list of the query given in the cursor declaration. The data types of the fields in the `SELECT` list must match, or be implicitly convertible to the data types of the fields in *record* or the data types of *variable*, *variable_2*...

Note: There is a variation of `FETCH INTO` using the `BULK COLLECT` clause that can return multiple rows at a time into a collection. See Section 4.12.4.2 for more information on using the `BULK COLLECT` clause with the `FETCH INTO` statement.

The following shows the `FETCH` statement.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

Instead of explicitly declaring the data type of a target variable, `%TYPE` can be used instead. In this way, if the data type of the database column is changed, the target variable declaration in the SPL program does not have to be changed. `%TYPE` will automatically pick up the new data type of the specified column.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

If all the columns in a table are retrieved in the order defined in the table, `%ROWTYPE` can be used to define a record into which the `FETCH` statement will place the retrieved data. Each field within the record can then be accessed using dot notation.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    FETCH emp_cur_1 INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
    DBMS_OUTPUT.PUT_LINE('Employee Name   : ' || v_emp_rec.ename);
        ...
END;
```

4.8.4 Closing a Cursor

Once all the desired rows have been retrieved from the cursor result set, the cursor must be closed. Once closed, the result set is no longer accessible. The `CLOSE` statement appears as follows:

```
CLOSE name;
```

name is the identifier of a cursor that is currently open. Once a cursor is closed, it must not be closed again. However, once the cursor is closed, the `OPEN` statement can be issued again on the closed cursor and the query result set will be rebuilt after which the `FETCH` statement can then be used to retrieve the rows of the new result set.

The following example illustrates the use of the `CLOSE` statement:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    FETCH emp_cur_1 INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
    DBMS_OUTPUT.PUT_LINE('Employee Name   : ' || v_emp_rec.ename);
    CLOSE emp_cur_1;
END;
```

This procedure produces the following output when invoked. Employee number `7369`, `SMITH` is the first row of the result set.

```
EXEC cursor_example;

Employee Number: 7369
Employee Name  : SMITH
```

4.8.5 Using %ROWTYPE With Cursors

Using the `%ROWTYPE` attribute, a record can be defined that contains fields corresponding to all columns fetched from a cursor or cursor variable. Each field takes on the data type of its corresponding column. The `%ROWTYPE` attribute is prefixed by a cursor name or cursor variable name.

```
record cursor%ROWTYPE;
```

`record` is an identifier assigned to the record. *`cursor`* is an explicitly declared cursor within the current scope.

The following example shows how you can use a cursor with `%ROWTYPE` to get information about which employee works in which department.

```
CREATE OR REPLACE PROCEDURE emp_info
IS
    CURSOR empcur IS SELECT ename, deptno FROM emp;
    myvar           empcur%ROWTYPE;
BEGIN
    OPEN empcur;
    LOOP
        FETCH empcur INTO myvar;
        EXIT WHEN empcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( myvar.ename || ' works in department '
            || myvar.deptno );
    END LOOP;
    CLOSE empcur;
END;
```

The following is the output from this procedure.

```
EXEC emp_info;

SMITH works in department 20
ALLEN works in department 30
WARD works in department 30
JONES works in department 20
MARTIN works in department 30
BLAKE works in department 30
CLARK works in department 10
SCOTT works in department 20
KING works in department 10
TURNER works in department 30
ADAMS works in department 20
JAMES works in department 30
FORD works in department 20
MILLER works in department 10
```

4.8.6 Cursor Attributes

Each cursor has a set of attributes associated with it that allows the program to test the state of the cursor. These attributes are `%ISOPEN`, `%FOUND`, `%NOTFOUND`, and `%ROWCOUNT`. These attributes are described in the following sections.

4.8.6.1 %ISOPEN

The `%ISOPEN` attribute is used to test whether or not a cursor is open.

```
cursor_name%ISOPEN
```

cursor_name is the name of the cursor for which a `BOOLEAN` data type of `TRUE` will be returned if the cursor is open, `FALSE` otherwise.

The following is an example of using `%ISOPEN`.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
        ...
    CURSOR emp_cur_1 IS SELECT * FROM emp;
        ...
BEGIN
        ...
    IF emp_cur_1%ISOPEN THEN
        NULL;
    ELSE
        OPEN emp_cur_1;
    END IF;
    FETCH emp_cur_1 INTO ...
        ...
END;
```

4.8.6.2 %FOUND

The `%FOUND` attribute is used to test whether or not a row is retrieved from the result set of the specified cursor after a `FETCH` on the cursor.

```
cursor_name%FOUND
```

cursor_name is the name of the cursor for which a `BOOLEAN` data type of `TRUE` will be returned if a row is retrieved from the result set of the cursor after a `FETCH`.

After the last row of the result set has been `FETCH`ed the next `FETCH` results in `%FOUND` returning FALSE. FALSE is also returned after the first `FETCH` if there are no rows in the result set to begin with.

Referencing `%FOUND` on a cursor before it is opened or after it is closed results in an `INVALID_CURSOR` exception being thrown.

`%FOUND` returns `null` if it is referenced when the cursor is open, but before the first `FETCH`.

The following example uses `%FOUND`.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FETCH emp_cur_1 INTO v_emp_rec;
    WHILE emp_cur_1%FOUND LOOP
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
        FETCH emp_cur_1 INTO v_emp_rec;
    END LOOP;
    CLOSE emp_cur_1;
END;
```

When the previous procedure is invoked, the output appears as follows:

```
EXEC cursor_example;

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
```

4.8.6.3 %NOTFOUND

The `%NOTFOUND` attribute is the logical opposite of `%FOUND`.

```
    cursor_name%NOTFOUND
```

`cursor_name` is the name of the cursor for which a `BOOLEAN` data type of FALSE will be returned if a row is retrieved from the result set of the cursor after a `FETCH`.

After the last row of the result set has been FETCHed the next FETCH results in %NOTFOUND returning TRUE. TRUE is also returned after the first FETCH if there are no rows in the result set to begin with.

Referencing %NOTFOUND on a cursor before it is opened or after it is closed, results in an INVALID_CURSOR exception being thrown.

%NOTFOUND returns null if it is referenced when the cursor is open, but before the first FETCH.

The following example uses %NOTFOUND.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur_1 INTO v_emp_rec;
        EXIT WHEN emp_cur_1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
    END LOOP;
    CLOSE emp_cur_1;
END;
```

Similar to the prior example, this procedure produces the same output when invoked.

```
EXEC cursor_example;

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
```

4.8.6.4 %ROWCOUNT

The `%ROWCOUNT` attribute returns an integer showing the number of rows `FETCH`ed so far from the specified cursor.

```
cursor_name%ROWCOUNT
```

`cursor_name` is the name of the cursor for which `%ROWCOUNT` returns the number of rows retrieved thus far. After the last row has been retrieved, `%ROWCOUNT` remains set to the total number of rows returned until the cursor is closed at which point `%ROWCOUNT` will throw an `INVALID_CURSOR` exception if referenced.

Referencing `%ROWCOUNT` on a cursor before it is opened or after it is closed, results in an `INVALID_CURSOR` exception being thrown.

`%ROWCOUNT` returns `0` if it is referenced when the cursor is open, but before the first `FETCH`. `%ROWCOUNT` also returns `0` after the first `FETCH` when there are no rows in the result set to begin with.

The following example uses `%ROWCOUNT`.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur_1 INTO v_emp_rec;
        EXIT WHEN emp_cur_1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('*********************');
    DBMS_OUTPUT.PUT_LINE(emp_cur_1%ROWCOUNT || ' rows were retrieved');
    CLOSE emp_cur_1;
END;
```

This procedure prints the total number of rows retrieved at the end of the employee list as follows:

```
EXEC cursor_example;

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
**********************
14 rows were retrieved
```

4.8.6.5 Summary of Cursor States and Attributes

The following table summarizes the possible cursor states and the values returned by the cursor attributes.

Table 4-4-3 Cursor Attributes

Cursor State	%ISOPEN	%FOUND	%NOTFOUND	%ROWCOUNT
Before `OPEN`	False	`INVALID_CURSOR` Exception	`INVALID_CURSOR` Exception	`INVALID_CURSOR` Exception
After `OPEN` & Before 1st `FETCH`	True	Null	Null	0
After 1st Successful `FETCH`	True	True	False	1
After *n*th Successful `FETCH` (last row)	True	True	False	n
After *n*+1st `FETCH` (after last row)	True	False	True	n
After `CLOSE`	False	`INVALID_CURSOR` Exception	`INVALID_CURSOR` Exception	`INVALID_CURSOR` Exception

4.8.7 Cursor FOR Loop

In the cursor examples presented so far, the programming logic required to process the result set of a cursor included a statement to open the cursor, a loop construct to retrieve each row of the result set, a test for the end of the result set, and finally a statement to close the cursor. The *cursor FOR loop* is a loop construct that eliminates the need to individually code the statements just listed.

The cursor `FOR` loop opens a previously declared cursor, fetches all rows in the cursor result set, and then closes the cursor.

The syntax for creating a cursor `FOR` loop is as follows.

```
FOR record IN cursor
LOOP
   statements
END LOOP;
```

record is an identifier assigned to an implicitly declared record with definition, *cursor*%ROWTYPE. *cursor* is the name of a previously declared cursor. *statements* are one or more SPL statements. There must be at least one statement.

The following example shows the example from Section 4.8.6.3, modified to use a cursor FOR loop.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR v_emp_rec IN emp_cur_1 LOOP
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
    END LOOP;
END;
```

The same results are achieved as shown in the output below.

```
EXEC cursor_example;

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
```

4.8.8 Parameterized Cursors

A user can also declare a static cursor that accepts parameters, and can pass values for those parameters when opening that cursor. In the following example we have created a parameterized cursor which will display the name and salary of all employees from the emp table that have a salary less than a specified value which is passed as a parameter.

```
DECLARE
    my_record       emp%ROWTYPE;
    CURSOR c1 (max_wage NUMBER) IS
        SELECT * FROM emp WHERE sal < max_wage;
BEGIN
    OPEN c1(2000);
    LOOP
        FETCH c1 INTO my_record;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = '
            || my_record.sal);
    END LOOP;
    CLOSE c1;
END;
```

So for example if we pass the value 2000 as `max_wage`, then we will only be shown the name and salary of all employees that have a salary less than 2000. The result of the above query is the following:

```
Name = SMITH, salary = 800.00
Name = ALLEN, salary = 1600.00
Name = WARD, salary = 1250.00
Name = MARTIN, salary = 1250.00
Name = TURNER, salary = 1500.00
Name = ADAMS, salary = 1100.00
Name = JAMES, salary = 950.00
Name = MILLER, salary = 1300.00
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

4.9 REF CURSORs and Cursor Variables

This section discusses another type of cursor that provides far greater flexibility than the previously discussed static cursors.

4.9.1 REF CURSOR Overview

A *cursor variable* is a cursor that actually contains a pointer to a query result set. The result set is determined by the execution of the `OPEN FOR` statement using the cursor variable.

A cursor variable is not tied to a single particular query like a static cursor. The same cursor variable may be opened a number of times with `OPEN FOR` statements containing different queries. Each time, a new result set is created from that query and made available via the cursor variable.

`REF CURSOR` types may be passed as parameters to or from stored procedures and functions. The return type of a function may also be a `REF CURSOR` type. This provides the capability to modularize the operations on a cursor into separate programs by passing a cursor variable between programs.

4.9.2 Declaring a Cursor Variable

SPL supports the declaration of a cursor variable using both the `SYS_REFCURSOR` built-in data type as well as creating a type of `REF CURSOR` and then declaring a variable of that type. `SYS_REFCURSOR` is a `REF CURSOR` type that allows any result set to be associated with it. This is known as a *weakly-typed* `REF CURSOR`.

Only the declaration of `SYS_REFCURSOR` and user-defined `REF CURSOR` variables are different. The remaining usage like opening the cursor, selecting into the cursor and closing the cursor is the same across both the cursor types. For the rest of this chapter our examples will primarily be making use of the `SYS_REFCURSOR` cursors. All you need to change in the examples to make them work for user defined `REF CURSOR`s is the declaration section.

Note: *Strongly-typed* `REF CURSOR`s require the result set to conform to a declared number and order of fields with compatible data types and can also optionally return a result set.

4.9.2.1 Declaring a SYS_REFCURSOR Cursor Variable

The following is the syntax for declaring a `SYS_REFCURSOR` cursor variable:

```
    name SYS_REFCURSOR;
```

name is an identifier assigned to the cursor variable.

The following is an example of a `SYS_REFCURSOR` variable declaration.

```
DECLARE
    emp_refcur      SYS_REFCURSOR;
        ...
```

4.9.2.2 Declaring a User Defined REF CURSOR Type Variable

You must perform two distinct declaration steps in order to use a user defined `REF CURSOR` variable:

- Create a referenced cursor `TYPE`
- Declare the actual cursor variable based on that `TYPE`

The syntax for creating a user defined `REF CURSOR` type is as follows:

```
    TYPE cursor_type_name IS REF CURSOR [RETURN return_type];
```

The following is an example of a cursor variable declaration.

```
DECLARE
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    my_rec emp_cur_type;
        ...
```

4.9.3 Opening a Cursor Variable

Once a cursor variable is declared, it must be opened with an associated `SELECT` command. The `OPEN FOR` statement specifies the `SELECT` command to be used to create the result set.

```
    OPEN name FOR query;
```

name is the identifier of a previously declared cursor variable. *query* is a `SELECT` command that determines the result set when the statement is executed. The value of the cursor variable after the `OPEN FOR` statement is executed identifies the result set.

In the following example, the result set is a list of employee numbers and names from a selected department. Note that a variable or parameter can be used in the SELECT command anywhere an expression can normally appear. In this case a parameter is used in the equality test for department number.

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
    p_deptno        emp.deptno%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
        ...
```

4.9.4 Fetching Rows From a Cursor Variable

After a cursor variable is opened, rows may be retrieved from the result set using the FETCH statement. See Section 4.8.3 for details on using the FETCH statement to retrieve rows from a result set.

In the example below, a FETCH statement has been added to the previous example so now the result set is returned into two variables and then displayed. Note that the cursor attributes used to determine cursor state of static cursors can also be used with cursor variables. See Section 4.8.6 for details on cursor attributes.

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
    p_deptno        emp.deptno%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
        ...
```

4.9.5 Closing a Cursor Variable

Use the CLOSE statement described in Section 4.8.4 to release the result set.

Note: Unlike static cursors, a cursor variable does not have to be closed before it can be re-opened again. The result set from the previous open will be lost.

The example is completed with the addition of the CLOSE statement.

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
    p_deptno        emp.deptno%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following is the output when this procedure is executed.

```
EXEC emp_by_dept(20)

EMPNO    ENAME
-----    -------
7369     SMITH
7566     JONES
7788     SCOTT
7876     ADAMS
7902     FORD
```

4.9.6 Usage Restrictions

The following are restrictions on cursor variable usage.

- Comparison operators cannot be used to test cursor variables for equality, inequality, null, or not null
- Null cannot be assigned to a cursor variable
- The value of a cursor variable cannot be stored in a database column
- Static cursors and cursor variables are not interchangeable. For example, a static cursor cannot be used in an `OPEN FOR` statement.

In addition the following table shows the permitted parameter modes for a cursor variable used as a procedure or function parameter depending upon the operations on the cursor variable within the procedure or function.

Table 4-4-4 Permitted Cursor Variable Parameter Modes

Operation	IN	IN OUT	OUT
`OPEN`	No	Yes	No
`FETCH`	Yes	Yes	No
`CLOSE`	Yes	Yes	No

So for example, if a procedure performs all three operations, `OPEN FOR`, `FETCH`, and `CLOSE` on a cursor variable declared as the procedure's formal parameter, then that parameter must be declared with `IN OUT` mode.

4.9.7 Examples

The following examples demonstrate cursor variable usage.

4.9.7.1 Returning a REF CURSOR From a Function

In the following example the cursor variable is opened with a query that selects employees with a given job. Note that the cursor variable is specified in this function's `RETURN` statement so the result set is made available to the caller of the function.

```
CREATE OR REPLACE FUNCTION emp_by_job (p_job VARCHAR2)
RETURN SYS_REFCURSOR
IS
    emp_refcur      SYS_REFCURSOR;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
    RETURN emp_refcur;
END;
```

This function is invoked in the following anonymous block by assigning the function's return value to a cursor variable declared in the anonymous block's declaration section. The result set is fetched using this cursor variable and then it is closed.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE := 'SALESMAN';
    v_emp_refcur    SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES WITH JOB ' || v_job);
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    v_emp_refcur := emp_by_job(v_job);
    LOOP
        FETCH v_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN v_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE v_emp_refcur;
END;
```

The following is the output when the anonymous block is executed.

```
EMPLOYEES WITH JOB SALESMAN
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
```

4.9.7.2 Modularizing Cursor Operations

The following example illustrates how the various operations on cursor variables can be modularized into separate programs.

The following procedure opens the given cursor variable with a `SELECT` command that retrieves all rows.

```
CREATE OR REPLACE PROCEDURE open_all_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp;
END;
```

This variation opens the given cursor variable with a `SELECT` command that retrieves all rows, but of a given department.

```
CREATE OR REPLACE PROCEDURE open_emp_by_dept (
    p_emp_refcur    IN OUT SYS_REFCURSOR,
    p_deptno        emp.deptno%TYPE
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp
        WHERE deptno = p_deptno;
END;
```

This third variation opens the given cursor variable with a `SELECT` command that retrieves all rows, but from a different table. Also note that the function's return value is the opened cursor variable.

```
CREATE OR REPLACE FUNCTION open_dept (
    p_dept_refcur    IN OUT SYS_REFCURSOR
) RETURN SYS_REFCURSOR
IS
    v_dept_refcur    SYS_REFCURSOR;
BEGIN
    v_dept_refcur := p_dept_refcur;
    OPEN v_dept_refcur FOR SELECT deptno, dname FROM dept;
    RETURN v_dept_refcur;
END;
```

This procedure fetches and displays a cursor variable result set consisting of employee number and name.

```
CREATE OR REPLACE PROCEDURE fetch_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH p_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN p_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
END;
```

This procedure fetches and displays a cursor variable result set consisting of department number and name.

```
CREATE OR REPLACE PROCEDURE fetch_dept (
    p_dept_refcur   IN SYS_REFCURSOR
)
IS
    v_deptno        dept.deptno%TYPE;
    v_dname         dept.dname%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('DEPT   DNAME');
    DBMS_OUTPUT.PUT_LINE('----   ---------');
    LOOP
        FETCH p_dept_refcur INTO v_deptno, v_dname;
        EXIT WHEN p_dept_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_deptno || '     ' || v_dname);
    END LOOP;
END;
```

This procedure closes the given cursor variable.

```
CREATE OR REPLACE PROCEDURE close_refcur (
    p_refcur        IN OUT SYS_REFCURSOR
)
IS
BEGIN
    CLOSE p_refcur;
END;
```

The following anonymous block executes all the previously described programs.

```
DECLARE
    gen_refcur      SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('ALL EMPLOYEES');
    open_all_emp(gen_refcur);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('****************');

    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #10');
    open_emp_by_dept(gen_refcur, 10);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('****************');

    DBMS_OUTPUT.PUT_LINE('DEPARTMENTS');
    fetch_dept(open_dept(gen_refcur));
    DBMS_OUTPUT.PUT_LINE('*****************');

    close_refcur(gen_refcur);
END;
```

The following is the output from the anonymous block.

```
ALL EMPLOYEES
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
****************
EMPLOYEES IN DEPT #10
EMPNO    ENAME
-----    -------
7782     CLARK
7839     KING
7934     MILLER
****************
DEPARTMENTS
DEPT   DNAME
----   ---------
10     ACCOUNTING
20     RESEARCH
30     SALES
40     OPERATIONS
*****************
```

4.9.8 Dynamic Queries With REF CURSORs

Advanced Server also supports dynamic queries via the `OPEN FOR USING` statement. A string literal or string variable is supplied in the `OPEN FOR USING` statement to the `SELECT` command.

```
    OPEN name FOR dynamic_string
      [ USING bind_arg [, bind_arg_2 ] ...];
```

name is the identifier of a previously declared cursor variable. *dynamic_string* is a string literal or string variable containing a `SELECT` command (without the terminating semi-colon). *bind_arg*, *bind_arg_2*... are bind arguments that are used to pass variables to corresponding placeholders in the `SELECT` command when the cursor variable is opened. The placeholders are identifiers prefixed by a colon character.

The following is an example of a dynamic query using a string literal.

```
CREATE OR REPLACE PROCEDURE dept_query
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = 30' ||
        ' AND sal >= 1500';
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following is the output when the procedure is executed.

```
EXEC dept_query;

EMPNO    ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

In the next example, the previous query is modified to use bind arguments to pass the query parameters.

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = :dept'
        || ' AND sal >= :sal' USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following is the resulting output.

```
EXEC dept_query(30, 1500);

EMPNO    ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

Finally, a string variable is used to pass the `SELECT` providing the most flexibility.

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    p_query_string  VARCHAR2(100);
BEGIN
    p_query_string := 'SELECT empno, ename FROM emp WHERE ' ||
        'deptno = :dept AND sal >= :sal';
    OPEN emp_refcur FOR p_query_string USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
EXEC dept_query(20, 1500);

EMPNO    ENAME
-----    -------
7566     JONES
7788     SCOTT
7902     FORD
```

4.10 Collections

A *collection* is a set of ordered data items with the same data type. Generally, the data item is a scalar field, but may also be a user-defined type such as a record type or an object type (see Section 8 for information about object types) as long as the structure and the data types that comprise each field of the user-defined type are the same for each element in the set. Each particular data item in the set is referenced by using subscript notation within a pair of parentheses.

Note: Multilevel collections (that is, where the data item of a collection is another collection) are not supported.

The most commonly known type of collection is an array. In Advanced Server, the supported collection types are *associative arrays* (formerly called *index-by-tables* in Oracle), *nested tables*, and *varrays*.

The general steps for using a collection are the following:

- A collection of the desired type must be defined. This can be done in the declaration section of an SPL program, which results in a *local type* that is accessible only within that program. For nested table and varray types this can also be done using the `CREATE TYPE` command, which creates a persistent, *standalone type* that can be referenced by any SPL program in the database.
- Variables of the collection type are declared. The collection associated with the declared variable is said to be *uninitialized* at this point if there is no value assignment made as part of the variable declaration.
- Uninitialized collections of nested tables and varrays are null. A *null collection* does not yet exist. Generally, a `COLLECTION_IS_NULL` exception is thrown if a collection method is invoked on a null collection.
- Uninitialized collections of associative arrays exist, but have no elements. An existing collection with no elements is called an *empty collection*.
- To initialize a null collection, you must either make it an empty collection or assign a non-null value to it. Generally, a null collection is initialized by using its *constructor*.
- To add elements to an empty associative array, you can simply assign values to its keys. For nested tables and varrays, generally its constructor is used to assign initial values to the nested table or varray. For nested tables and varrays, the `EXTEND` method is then used to grow the collection beyond its initial size established by the constructor.

The specific process for each collection type is described in the following sections.

4.10.1 Associative Arrays

An *associative array* is a type of collection that associates a unique key with a value. The key does not have to be numeric, but can be character data as well.

An associative array has the following characteristics:

- An *associative array type* must be defined after which *array variables* can be declared of that array type. Data manipulation occurs using the array variable.
- When an array variable is declared, the associative array is created, but it is empty - just start assigning values to key values.
- The key can be any negative integer, positive integer, or zero if `INDEX BY BINARY_INTEGER` or `PLS_INTEGER` is specified.
- The key can be character data if `INDEX BY VARCHAR2` is specified.
- There is no pre-defined limit on the number of elements in the array - it grows dynamically as elements are added.
- The array can be sparse - there may be gaps in the assignment of values to keys.
- An attempt to reference an array element that has not been assigned a value will result in an exception.

The `TYPE IS TABLE OF ... INDEX BY` statement is used to define an associative array type.

```
TYPE assoctype IS TABLE OF { datatype | rectype | objtype }
  INDEX BY { BINARY_INTEGER | PLS_INTEGER | VARCHAR2(n) };
```

`assoctype` is an identifier assigned to the array type. *`datatype`* is a scalar data type such as `VARCHAR2` or `NUMBER`. *`rectype`* is a previously defined record type. *`objtype`* is a previously defined object type. *`n`* is the maximum length of a character key.

In order to make use of the array, a *variable* must be declared with that array type. The following is the syntax for declaring an array variable.

```
array assoctype
```

`array` is an identifier assigned to the associative array. *`assoctype`* is the identifier of a previously defined array type.

An element of the array is referenced using the following syntax.

```
array(n)[.field ]
```

`array` is the identifier of a previously declared array. *`n`* is the key value, type-compatible with the data type given in the `INDEX BY` clause. If the array type of *`array`* is defined from a record type or object type, then [.*`field`*] must reference an individual field within the record type or attribute within the object type from which the array type is defined. Alternatively, the entire record can be referenced by omitting [.*`field`*].

The following example reads the first ten employee names from the `emp` table, stores them in an array, then displays the results from the array.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

```
DECLARE
    TYPE emp_arr_typ IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END;
```

The above example produces the following output:

```
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
```

The previous example is now modified to use a record type in the array definition.

```
DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR2(10)
    );
    TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END;
```

The following is the output from this anonymous block.

```
EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
```

```
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
```

The `emp%ROWTYPE` attribute could be used to define `emp_arr_typ` instead of using the `emp_rec_typ` record type as shown in the following.

```
DECLARE
    TYPE emp_arr_typ IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END;
```

The results are the same as in the prior example.

Instead of assigning each field of the record individually, a record level assignment can be made from `r_emp` to `emp_arr`.

```
DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR2(10)
    );
    TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END;
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

The key of an associative array can be character data as shown in the following example.

```
DECLARE
    TYPE job_arr_typ IS TABLE OF NUMBER INDEX BY VARCHAR2(9);
    job_arr         job_arr_typ;
BEGIN
    job_arr('ANALYST')   := 100;
    job_arr('CLERK')     := 200;
    job_arr('MANAGER')   := 300;
    job_arr('SALESMAN')  := 400;
    job_arr('PRESIDENT') := 500;
    DBMS_OUTPUT.PUT_LINE('ANALYST  : ' || job_arr('ANALYST'));
    DBMS_OUTPUT.PUT_LINE('CLERK    : ' || job_arr('CLERK'));
    DBMS_OUTPUT.PUT_LINE('MANAGER  : ' || job_arr('MANAGER'));
    DBMS_OUTPUT.PUT_LINE('SALESMAN : ' || job_arr('SALESMAN'));
    DBMS_OUTPUT.PUT_LINE('PRESIDENT: ' || job_arr('PRESIDENT'));
END;

ANALYST  : 100
CLERK    : 200
MANAGER  : 300
SALESMAN : 400
PRESIDENT: 500
```

4.10.2 Nested Tables

A *nested table* is a type of collection that associates a positive integer with a value. A nested table has the following characteristics:

- A *nested table type* must be defined after which *nested table variables* can be declared of that nested table type. Data manipulation occurs using the nested table variable, or simply, "table" for short.
- When a nested table variable is declared, the nested table initially does not exist (it is a null collection). The null table must be initialized with a *constructor*. You can also initialize the table by using an assignment statement where the right-hand side of the assignment is an initialized table of the same type. **Note:** Initialization of a nested table is mandatory in Oracle, but optional in SPL.
- The key is a positive integer.
- The constructor establishes the number of elements in the table. The `EXTEND` method adds additional elements to the table. See Section 4.11 for information on collection methods. **Note:** Usage of the constructor to establish the number of elements in the table and usage of the `EXTEND` method to add additional elements to the table are mandatory in Oracle, but optional in SPL.
- The table can be sparse - there may be gaps in the assignment of values to keys.
- An attempt to reference a table element beyond its initialized or extended size will result in a `SUBSCRIPT_BEYOND_COUNT` exception.

The `TYPE IS TABLE` statement is used to define a nested table type within the declaration section of an SPL program.

```
    TYPE tbltype IS TABLE OF { datatype | rectype | objtype };
```

tbltype is an identifier assigned to the nested table type. *datatype* is a scalar data type such as VARCHAR2 or NUMBER. *rectype* is a previously defined record type. *objtype* is a previously defined object type.

Note: You can use the CREATE TYPE command to define a nested table type that is available to all SPL programs in the database. See the CREATE TYPE command for more information.

In order to make use of the table, a *variable* must be declared of that nested table type. The following is the syntax for declaring a table variable.

```
    table tbltype
```

table is an identifier assigned to the nested table. *tbltype* is the identifier of a previously defined nested table type.

A nested table is initialized using the nested table type's constructor.

```
    tbltype ([ { expr1 | NULL } [, { expr2 | NULL } ] [, ...] ])
```

tbltype is the identifier of the nested table type's constructor, which has the same name as the nested table type. *expr1*, *expr2*, … are expressions that are type-compatible with the element type of the table. If NULL is specified, the corresponding element is set to null. If the parameter list is empty, then an empty nested table is returned, which means there are no elements in the table. If the table is defined from an object type, then *exprn* must return an object of that object type. The object can be the return value of a function or the object type's constructor, or the object can be an element of another nested table of the same type.

If a collection method other than EXISTS is applied to an uninitialized nested table, a COLLECTION_IS_NULL exception is thrown. See Section 4.11 for information on collection methods.

The following is an example of a constructor for a nested table:

```
DECLARE
    TYPE nested_typ IS TABLE OF CHAR(1);
    v_nested        nested_typ := nested_typ('A','B');
```

An element of the table is referenced using the following syntax.

```
    table(n)[.element ]
```

table is the identifier of a previously declared table. *n* is a positive integer. If the table type of *table* is defined from a record type or object type, then [.*element*] must reference an individual field within the record type or attribute within the object type from which the nested table type is defined. Alternatively, the entire record or object can be referenced by omitting [.*element*].

The following is an example of a nested table where it is known that there will be four elements.

```
DECLARE
    TYPE dname_tbl_typ IS TABLE OF VARCHAR2(14);
    dname_tbl       dname_tbl_typ;
    CURSOR dept_cur IS SELECT dname FROM dept ORDER BY dname;
    i               INTEGER := 0;
BEGIN
    dname_tbl := dname_tbl_typ(NULL, NULL, NULL, NULL);
    FOR r_dept IN dept_cur LOOP
        i := i + 1;
        dname_tbl(i) := r_dept.dname;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('DNAME');
    DBMS_OUTPUT.PUT_LINE('----------');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(dname_tbl(j));
    END LOOP;
END;
```

The above example produces the following output:

```
DNAME
----------
ACCOUNTING
OPERATIONS
RESEARCH
SALES
```

The following example reads the first ten employee names from the `emp` table, stores them in a nested table, then displays the results from the table. The SPL code is written to assume that the number of employees to be returned is not known beforehand.

```
DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR2(10)
    );
    TYPE emp_tbl_typ IS TABLE OF emp_rec_typ;
    emp_tbl         emp_tbl_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    emp_tbl := emp_tbl_typ();
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_tbl.EXTEND;
        emp_tbl(i) := r_emp;
    END LOOP;
    FOR j IN 1..10 LOOP
```

```
        DBMS_OUTPUT.PUT_LINE(emp_tbl(j).empno || '     ' ||
            emp_tbl(j).ename);
    END LOOP;
END;
```

Note the creation of an empty table with the constructor `emp_tbl_typ()` as the first statement in the executable section of the anonymous block. The `EXTEND` collection method is then used to add an element to the table for each employee returned from the result set. See Section 4.11.4 for information on `EXTEND`.

The following is the output.

```
EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
```

The following example shows how a nested table of an object type can be used. See Section 8 for information about object types and objects. First, an object type is created with attributes for the department name and location.

```
CREATE TYPE dept_obj_typ AS OBJECT (
    dname           VARCHAR2(14),
    loc             VARCHAR2(13)
);
```

The following anonymous block defines a nested table type whose element consists of the `dept_obj_typ` object type. A nested table variable is declared, initialized, and then populated from the `dept` table. Finally, the elements from the nested table are displayed.

```
DECLARE
    TYPE dept_tbl_typ IS TABLE OF dept_obj_typ;
    dept_tbl        dept_tbl_typ;
    CURSOR dept_cur IS SELECT dname, loc FROM dept ORDER BY dname;
    i               INTEGER := 0;
BEGIN
    dept_tbl := dept_tbl_typ(
        dept_obj_typ(NULL,NULL),
        dept_obj_typ(NULL,NULL),
        dept_obj_typ(NULL,NULL),
        dept_obj_typ(NULL,NULL)
    );
    FOR r_dept IN dept_cur LOOP
        i := i + 1;
        dept_tbl(i).dname := r_dept.dname;
        dept_tbl(i).loc   := r_dept.loc;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('DNAME          LOC');
    DBMS_OUTPUT.PUT_LINE('----------     ----------');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(RPAD(dept_tbl(j).dname,14) || ' ' ||
            dept_tbl(j).loc);
    END LOOP;
END;
```

Note: The parameters comprising the nested table's constructor, `dept_tbl_typ`, are calls to the object type's constructor `dept_obj_typ`.

The following is the output from the anonymous block.

```
DNAME          LOC
----------     ----------
ACCOUNTING     NEW YORK
OPERATIONS     BOSTON
RESEARCH       DALLAS
SALES          CHICAGO
```

4.10.3 Varrays

A *varray* or *variable-size array* is a type of collection that associates a positive integer with a value. In many respects, it is similar to a nested table.

A varray has the following characteristics:

- A *varray type* must be defined along with a maximum size limit. After the varray type is defined, *varray variables* can be declared of that varray type. Data manipulation occurs using the varray variable, or simply, "varray" for short. The number of elements in the varray cannot exceed the maximum size limit established in the varray type definition.
- When a varray variable is declared, the varray initially does not exist (it is a null collection). The null varray must be initialized with a *constructor*. You can also initialize the varray by using an assignment statement where the right-hand side of the assignment is an initialized varray of the same type.
- The key is a positive integer.
- The constructor establishes the number of elements in the varray, which must not exceed the maximum size limit. The `EXTEND` method can add additional elements to the varray up to the maximum size limit. See Section 4.11 for information on collection methods.
- Unlike a nested table, a varray cannot be sparse - there are no gaps in the assignment of values to keys.
- An attempt to reference a varray element beyond its initialized or extended size, but within the maximum size limit will result in a `SUBSCRIPT_BEYOND_COUNT` exception.

- An attempt to reference a varray element beyond the maximum size limit or extend a varray beyond the maximum size limit will result in a `SUBSCRIPT_OUTSIDE_LIMIT` exception.

The `TYPE IS VARRAY` statement is used to define a varray type within the declaration section of an SPL program.

```
TYPE varraytype IS { VARRAY | VARYING ARRAY }(maxsize)
  OF { datatype | objtype };
```

varraytype is an identifier assigned to the varray type. *datatype* is a scalar data type such as `VARCHAR2` or `NUMBER`. *maxsize* is the maximum number of elements permitted in varrays of that type. *objtype* is a previously defined object type.

Note: The `CREATE TYPE` command can be used to define a varray type that is available to all SPL programs in the database. See the

CREATE TYPE command.

In order to make use of the varray, a *variable* must be declared of that varray type. The following is the syntax for declaring a varray variable.

```
varray varraytype
```

varray is an identifier assigned to the varray. *varraytype* is the identifier of a previously defined varray type.

A varray is initialized using the varray type's constructor.

```
varraytype ([ { expr1 | NULL } [, { expr2 | NULL } ]
  [, ...] ])
```

varraytype is the identifier of the varray type's constructor, which has the same name as the varray type. *expr1*, *expr2*, … are expressions that are type-compatible with the element type of the varray. If `NULL` is specified, the corresponding element is set to null. If the parameter list is empty, then an empty varray is returned, which means there are no elements in the varray. If the varray is defined from an object type, then *exprn* must return an object of that object type. The object can be the return value of a function or the return value of the object type's constructor. The object can also be an element of another varray of the same varray type.

If a collection method other than `EXISTS` is applied to an uninitialized varray, a `COLLECTION_IS_NULL` exception is thrown. See Section 4.11 for information on collection methods.

The following is an example of a constructor for a varray:

```
DECLARE
    TYPE varray_typ IS VARRAY(2) OF CHAR(1);
    v_varray        varray_typ := varray_typ('A','B');
```

An element of the varray is referenced using the following syntax.

```
varray(n)[.element ]
```

`varray` is the identifier of a previously declared varray. `n` is a positive integer. If the varray type of `varray` is defined from an object type, then `[.element ]` must reference an attribute within the object type from which the varray type is defined. Alternatively, the entire object can be referenced by omitting `[.element ]`.

The following is an example of a varray where it is known that there will be four elements.

```
DECLARE
    TYPE dname_varray_typ IS VARRAY(4) OF VARCHAR2(14);
    dname_varray    dname_varray_typ;
    CURSOR dept_cur IS SELECT dname FROM dept ORDER BY dname;
    i               INTEGER := 0;
BEGIN
    dname_varray := dname_varray_typ(NULL, NULL, NULL, NULL);
    FOR r_dept IN dept_cur LOOP
        i := i + 1;
        dname_varray(i) := r_dept.dname;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('DNAME');
    DBMS_OUTPUT.PUT_LINE('----------');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(dname_varray(j));
    END LOOP;
END;
```

The above example produces the following output:

```
DNAME
----------
ACCOUNTING
OPERATIONS
RESEARCH
SALES
```

4.11 Collection Methods

Collection methods are functions and procedures that provide useful information about a collection that can aid in the processing of data in the collection. The following sections discuss the collection methods supported by Advanced Server.

4.11.1 COUNT

COUNT is a method that returns the number of elements in a collection. The syntax for using COUNT is as follows:

```
collection.COUNT
```

collection is the name of a collection.

For a varray, COUNT always equals LAST.

The following example shows that an associative array can be sparsely populated (i.e., there are "gaps" in the sequence of assigned elements). COUNT includes only the elements that have been assigned a value.

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
END;
```

The following output shows that there are five populated elements included in COUNT.

```
COUNT: 5
```

4.11.2 DELETE

The DELETE method deletes entries from a collection. You can call the DELETE method in three different ways.

Use the first form of the DELETE method to remove all entries from a collection:

```
collection.DELETE
```

Use the second form of the `DELETE` method to remove the specified entry from a collection:

```
collection.DELETE(subscript)
```

Use the third form of the `DELETE` method to remove the entries that are within the range specified by *`first_subscript`* and *`last_subscript`* (including the entries for the *`first_subscript`* and the *`last_subscript`*) from a collection.

```
collection.DELETE(first_subscript, last_subscript)
```

If `first_subscript` and `last_subscript` refer to non-existent elements, elements that are in the range between the specified subscripts are deleted. If `first_subscript` is greater than `last_subscript`, or if you specify a value of `NULL` for one of the arguments, `DELETE` has no effect.

Note that when you delete an entry, the subscript remains in the collection; you can re-use the subscript with an alternate entry. If you specify a subscript that does not exist in the call to the `DELETE` method, `DELETE` does not raise an exception.

The following example demonstrates using the `DELETE` method to remove the element with subscript `0` from the collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
    v_results       VARCHAR2(50);
    v_sub           NUMBER;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.DELETE(0);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    v_sub := sparse_arr.FIRST;
    WHILE v_sub IS NOT NULL LOOP
        IF sparse_arr(v_sub) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(v_sub) || ' ';
        END IF;
        v_sub := sparse_arr.NEXT(v_sub);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 4
Results: -100 -10 10 100
```

`COUNT` indicates that before the `DELETE` method, there were `5` elements in the collection; after the `DELETE` method was invoked, the collection contains `4` elements.

4.11.3 EXISTS

The `EXISTS` method verifies that a subscript exists within a collection. `EXISTS` returns `TRUE` if the subscript exists; if the subscript does not exist, `EXISTS` returns `FALSE`. The method takes a single argument; the `subscript` that you are testing for. The syntax is:

```
collection.EXISTS(subscript)
```

`collection` is the name of the collection.

`subscript` is the value that you are testing for. If you specify a value of `NULL`, `EXISTS` returns `false`.

The following example verifies that subscript number `10` exists within the associative array:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('The index exists: ' ||
        CASE WHEN sparse_arr.exists(10) = TRUE THEN 'true' ELSE 'false' END);
END;

The index exists: true
```

Some collection methods raise an exception if you call them with a subscript that does not exist within the specified collection. Rather than raising an error, the `EXISTS` method returns a value of `FALSE`.

4.11.4 EXTEND

The `EXTEND` method increases the size of a collection. There are three variations of the `EXTEND` method. The first variation appends a single `NULL` element to a collection; the syntax for the first variation is:

```
collection.EXTEND
```

`collection` is the name of a collection.

The following example demonstrates using the `EXTEND` method to append a single, null element to a collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr      sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results       VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 6
Results: -100 -10 0 10 100 NULL
```

`COUNT` indicates that before the `EXTEND` method, there were `5` elements in the collection; after the `EXTEND` method was invoked, the collection contains `6` elements.

The second variation of the `EXTEND` method appends a specified number of elements to the end of a collection.

```
collection.EXTEND(count)
```

`collection` is the name of a collection.

`count` is the number of null elements added to the end of the collection.

The following example demonstrates using the `EXTEND` method to append multiple null elements to a collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr      sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results       VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND(3);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
```

```
END;

COUNT: 5
COUNT: 8
Results: -100 -10 0 10 100 NULL NULL NULL
```

`COUNT` indicates that before the `EXTEND` method, there were `5` elements in the collection; after the `EXTEND` method was invoked, the collection contains `8` elements.

The third variation of the `EXTEND` method appends a specified number of copies of a particular element to the end of a collection.

```
    collection.EXTEND(count, index_number)
```

`collection` is the name of a collection.

`count` is the number of elements added to the end of the collection.

`index_number` is the subscript of the element that is being copied to the collection.

The following example demonstrates using the `EXTEND` method to append multiple copies of the second element to the collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr      sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results       VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND(3, 2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 8
Results: -100 -10 0 10 100 -10 -10 -10
```

`COUNT` indicates that before the `EXTEND` method, there were `5` elements in the collection; after the `EXTEND` method was invoked, the collection contains `8` elements.

Note: The `EXTEND` method cannot be used on a null or empty collection.

4.11.5 FIRST

`FIRST` is a method that returns the subscript of the first element in a collection. The syntax for using `FIRST` is as follows:

```
    collection.FIRST
```

collection is the name of a collection.

The following example displays the first element of the associative array.

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('FIRST element: ' || sparse_arr(sparse_arr.FIRST));
END;

FIRST element: -100
```

4.11.6 LAST

`LAST` is a method that returns the subscript of the last element in a collection. The syntax for using `LAST` is as follows:

```
    collection.LAST
```

collection is the name of a collection.

The following example displays the last element of the associative array.

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('LAST element: ' || sparse_arr(sparse_arr.LAST));
END;

LAST element: 100
```

4.11.7 LIMIT

`LIMIT` is a method that returns the maximum number of elements permitted in a collection. `LIMIT` is applicable only to varrays. The syntax for using `LIMIT` is as follows:

```
collection.LIMIT
```

`collection` is the name of a collection.

For an initialized varray, `LIMIT` returns the maximum size limit determined by the varray type definition. If the varray is uninitialized (that is, it is a null varray), an exception is thrown.

For an associative array or an initialized nested table, `LIMIT` returns `NULL`. If the nested table is uninitialized (that is, it is a null nested table), an exception is thrown.

4.11.8 NEXT

`NEXT` is a method that returns the subscript that follows a specified subscript. The method takes a single argument; the `subscript` that you are testing for.

```
collection.NEXT(subscript)
```

`collection` is the name of the collection.

If the specified subscript is less than the first subscript in the collection, the function returns the first subscript. If the subscript does not have a successor, `NEXT` returns `NULL`. If you specify a `NULL` subscript, `PRIOR` does not return a value.

The following example demonstrates using `NEXT` to return the subscript that follows subscript `10` in the associative array, `sparse_arr`:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('NEXT element: ' || sparse_arr.next(10));
END;

NEXT element: 100
```

4.11.9 PRIOR

The `PRIOR` method returns the subscript that precedes a specified subscript in a collection. The method takes a single argument; the `subscript` that you are testing for. The syntax is:

```
collection.PRIOR(subscript)
```

`collection` is the name of the collection.

If the subscript specified does not have a predecessor, `PRIOR` returns `NULL`. If the specified subscript is greater than the last subscript in the collection, the method returns the last subscript. If you specify a `NULL` subscript, `PRIOR` does not return a value.

The following example returns the subscript that precedes subscript `100` in the associative array, `sparse_arr`:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('PRIOR element: ' || sparse_arr.prior(100));
END;

PRIOR element: 10
```

4.11.10 TRIM

The `TRIM` method removes an element or elements from the end of a collection. The syntax for the `TRIM` method is:

```
collection.TRIM[(count)]
```

`collection` is the name of a collection.

`count` is the number of elements removed from the end of the collection. Advanced Server will return an error if *`count`* is less than `0` or greater than the number of elements in the collection.

The following example demonstrates using the `TRIM` method to remove an element from the end of a collection:

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr      sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.TRIM;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
END;

COUNT: 5
COUNT: 4
```

`COUNT` indicates that before the `TRIM` method, there were `5` elements in the collection; after the `TRIM` method was invoked, the collection contains `4` elements.

You can also specify the number of elements to remove from the end of the collection with the `TRIM` method:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr      sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results       VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.TRIM(2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 3
Results: -100 -10 0
```

`COUNT` indicates that before the `TRIM` method, there were `5` elements in the collection; after the `TRIM` method was invoked, the collection contains `3` elements.

4.12 Working with Collections

Collection operators allow you to transform, query and manipulate the contents of a collection.

4.12.1 TABLE()

Use the `TABLE()` function to transform the members of an array into a set of rows. The signature is:

```
TABLE(collection_value)
```

Where:

`collection_value`

`collection_value` is an expression that evaluates to a value of collection type.

The `TABLE()` function expands the nested contents of a collection into a table format. You can use the `TABLE()` function anywhere you use a regular table expression.

The `TABLE()` function returns a `SETOF ANYELEMENT` (a set of values of any type). For example, if the argument passed to this function is an array of `dates`, `TABLE()` will return a `SETOF dates`. If the argument passed to this function is an array of `paths`, `TABLE()` will return a `SETOF paths`.

You can use the `TABLE()` function to expand the contents of a collection into table form:

```
postgres=# SELECT * FROM TABLE(monthly_balance(445.00, 980.20, 552.00));

 monthly_balance
----------------
  445.00
  980.20
  552.00
(3 rows)
```

4.12.2 Using the MULTISET UNION Operator

The `MULTISET UNION` operator combines two collections to form a third collection. The signature is:

```
coll_1 MULTISET UNION [ALL | DISTINCT] coll_2
```

`coll_1` and *`coll_2`* specify the names of the collections to combine.

Include the ALL keyword to specify that duplicate elements (elements that are present in both *coll_1* and *coll_2*) should be represented in the result, once for each time they are present in the original collections. This is the default behavior of MULTISET UNION.

Include the DISTINCT keyword to specify that duplicate elements should be included in the result only once.

The following example demonstrates using the MULTISET UNION operator to combine two collections (collection_1 and collection_2) into a third collection (collection_3):

```
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1    int_arr_typ;
    collection_2    int_arr_typ;
    collection_3    int_arr_typ;
    v_results       VARCHAR2(50);
BEGIN
    collection_1 := int_arr_typ(10,20,30);
    collection_2 := int_arr_typ(30,40);
    collection_3 := collection_1 MULTISET UNION ALL collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || collection_3(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
Results: 10 20 30 30 40
```

The resulting collection includes one entry for each element in collection_1 and collection_2. If the DISTINCT keyword is used, the results are the following:

```
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1    int_arr_typ;
    collection_2    int_arr_typ;
    collection_3    int_arr_typ;
    v_results       VARCHAR2(50);
BEGIN
    collection_1 := int_arr_typ(10,20,30);
    collection_2 := int_arr_typ(30,40);
    collection_3 := collection_1 MULTISET UNION DISTINCT collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || collection_3(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
```

```
END;

COUNT: 4
Results: 10 20 30 40
```

The resulting collection includes only those members with distinct values. Note in the following example that the `MULTISET UNION DISTINCT` operator also removes duplicate entries that are stored within the same collection:

```
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1    int_arr_typ;
    collection_2    int_arr_typ;
    collection_3    int_arr_typ;
    v_results       VARCHAR2(50);
BEGIN
    collection_1 := int_arr_typ(10,20,30,30);
    collection_2 := int_arr_typ(40,50);
    collection_3 := collection_1 MULTISET UNION DISTINCT collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || collection_3(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
Results: 10 20 30 40 50
```

4.12.3 Using the FORALL Statement

Collections can be used to more efficiently process DML commands by passing all the values to be used for repetitive execution of a `DELETE`, `INSERT`, or `UPDATE` command in one pass to the database server rather than re-iteratively invoking the DML command with new values. The DML command to be processed in such a manner is specified with the `FORALL` statement. In addition, one or more collections are given in the DML command where different values are to be substituted each time the command is executed.

```
FORALL index IN lower_bound .. upper_bound
  { insert_stmt | update_stmt | delete_stmt };
```

`index` is the position in the collection given in the *`insert_stmt`*, *`update_stmt`*, or *`delete_stmt`* DML command that iterates from the integer value given as *`lower_bound`* up to and including *`upper_bound`*.

Note: If an exception occurs during any iteration of the `FORALL` statement, all updates that occurred since the start of the execution of the `FORALL` statement are automatically rolled back. This behavior is not compatible with Oracle databases. Oracle allows explicit use of the COMMIT or ROLLBACK commands to control whether or not to commit or roll back updates that occurred prior to the exception.

The FORALL statement creates a loop – each iteration of the loop increments the *index* variable (you typically use the *index* within the loop to select a member of a collection). The number of iterations is controlled by the *lower_bound .. upper_bound* clause. The loop is executes once for each integer between the *lower_bound* and *upper_bound* (inclusive) and the index is incremented by one for each iteration. For example:

```
FORALL i IN 2 .. 5
```

Creates a loop that executes four times – in the first iteration, the index (i) is set to the value 2; in the second iteration, the index is set to the value 3, and so on. The loop executes for the value 5 and then terminates.

The following example creates a table (emp_copy) that is an empty copy of the emp table. The example declares a type (emp_tbl) that is an array where each element in the array is of composite type, composed of the column definitions used to create the table, emp. The example also creates an index on the emp_tbl type.

t_emp is an associative array, of type emp_tbl. The SELECT statement uses the BULK COLLECT INTO command to populate the t_emp array. After the t_emp array is populated, the FORALL statement iterates through the values (i) in the t_emp array index and inserts a row for each record into emp_copy.

```
CREATE TABLE emp_copy(LIKE emp);

DECLARE

    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;

    t_emp emp_tbl;

BEGIN
    SELECT * FROM emp BULK COLLECT INTO t_emp;

    FORALL i IN t_emp.FIRST .. t_emp.LAST
     INSERT INTO emp_copy VALUES t_emp(i);

END;
```

The following example uses a FORALL statement to update the salary of three employees:

```
DECLARE
    TYPE empno_tbl  IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl    IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
    t_empno         EMPNO_TBL;
    t_sal           SAL_TBL;
BEGIN
```

```
    t_empno(1)  := 9001;
    t_sal(1)    := 3350.00;
    t_empno(2)  := 9002;
    t_sal(2)    := 2000.00;
    t_empno(3)  := 9003;
    t_sal(3)    := 4100.00;
    FORALL i IN t_empno.FIRST..t_empno.LAST
        UPDATE emp SET sal = t_sal(i) WHERE empno = t_empno(i);
END;

SELECT * FROM emp WHERE empno > 9000;

 empno | ename  |   job   | mgr | hiredate |   sal   | comm | deptno
-------+--------+---------+-----+----------+---------+------+--------
  9001 | JONES  | ANALYST |     |          | 3350.00 |      |     40
  9002 | LARSEN | CLERK   |     |          | 2000.00 |      |     40
  9003 | WILSON | MANAGER |     |          | 4100.00 |      |     40
(3 rows)
```

The following example deletes three employees in a FORALL statement:

```
DECLARE
    TYPE empno_tbl  IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    t_empno         EMPNO_TBL;
BEGIN
    t_empno(1)  := 9001;
    t_empno(2)  := 9002;
    t_empno(3)  := 9003;
    FORALL i IN t_empno.FIRST..t_empno.LAST
        DELETE FROM emp WHERE empno = t_empno(i);
END;

SELECT * FROM emp WHERE empno > 9000;

 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 rows)
```

4.12.4 Using the BULK COLLECT Clause

SQL commands that return a result set consisting of a large number of rows may not be operating as efficiently as possible due to the constant context switching that must occur between the database server and the client in order to transfer the entire result set. This inefficiency can be mitigated by using a collection to gather the entire result set in memory which the client can then access. The BULK COLLECT clause is used to specify the aggregation of the result set into a collection.

The BULK COLLECT clause can be used with the SELECT INTO, FETCH INTO and EXECUTE IMMEDIATE commands, and with the RETURNING INTO clause of the DELETE, INSERT, and UPDATE commands. Each of these is illustrated in the following sections.

4.12.4.1 SELECT BULK COLLECT

The BULK COLLECT clause can be used with the SELECT INTO statement as follows. (Refer to Section 4.4.3 for additional information on the SELECT INTO statement.)

```
SELECT select_expressions BULK COLLECT INTO collection
  [, ...] FROM ...;
```

If a single collection is specified, then *collection* may be a collection of a single field, or it may be a collection of a record type. If more than one collection is specified, then each *collection* must consist of a single field. *select_expressions* must match in number, order, and type-compatibility all fields in the target collections.

The following example shows the use of the BULK COLLECT clause where the target collections are associative arrays consisting of a single field.

```
DECLARE
    TYPE empno_tbl    IS TABLE OF emp.empno%TYPE    INDEX BY BINARY_INTEGER;
    TYPE ename_tbl    IS TABLE OF emp.ename%TYPE    INDEX BY BINARY_INTEGER;
    TYPE job_tbl      IS TABLE OF emp.job%TYPE      INDEX BY BINARY_INTEGER;
    TYPE hiredate_tbl IS TABLE OF emp.hiredate%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl      IS TABLE OF emp.sal%TYPE      INDEX BY BINARY_INTEGER;
    TYPE comm_tbl     IS TABLE OF emp.comm%TYPE     INDEX BY BINARY_INTEGER;
    TYPE deptno_tbl   IS TABLE OF emp.deptno%TYPE   INDEX BY BINARY_INTEGER;
    t_empno           EMPNO_TBL;
    t_ename           ENAME_TBL;
    t_job             JOB_TBL;
    t_hiredate        HIREDATE_TBL;
    t_sal             SAL_TBL;
    t_comm            COMM_TBL;
    t_deptno          DEPTNO_TBL;
BEGIN
    SELECT empno, ename, job, hiredate, sal, comm, deptno BULK COLLECT
       INTO t_empno, t_ename, t_job, t_hiredate, t_sal, t_comm, t_deptno
       FROM emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
        'SAL        ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------   ' || '--------  ------');
    FOR i IN 1..t_empno.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_empno(i) || '   ' ||
            RPAD(t_ename(i),8) || ' ' ||
            RPAD(t_job(i),10) || ' ' ||
            TO_CHAR(t_hiredate(i),'DD-MON-YY') || ' ' ||
            TO_CHAR(t_sal(i),'99,999.99') || ' ' ||
            TO_CHAR(NVL(t_comm(i),0),'99,999.99') || '  ' ||
            t_deptno(i));
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL        COMM      DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80     800.00        .00  20
7499   ALLEN    SALESMAN   20-FEB-81   1,600.00     300.00  30
7521   WARD     SALESMAN   22-FEB-81   1,250.00     500.00  30
7566   JONES    MANAGER    02-APR-81   2,975.00        .00  20
7654   MARTIN   SALESMAN   28-SEP-81   1,250.00   1,400.00  30
```

```
7698   BLAKE    MANAGER    01-MAY-81  2,850.00        .00  30
7782   CLARK    MANAGER    09-JUN-81  2,450.00        .00  10
7788   SCOTT    ANALYST    19-APR-87  3,000.00        .00  20
7839   KING     PRESIDENT  17-NOV-81  5,000.00        .00  10
7844   TURNER   SALESMAN   08-SEP-81  1,500.00        .00  30
7876   ADAMS    CLERK      23-MAY-87  1,100.00        .00  20
7900   JAMES    CLERK      03-DEC-81    950.00        .00  30
7902   FORD     ANALYST    03-DEC-81  3,000.00        .00  20
7934   MILLER   CLERK      23-JAN-82  1,300.00        .00  10
```

The following example produces the same result, but uses an associative array on a record type defined with the `%ROWTYPE` attribute.

```
DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    t_emp           EMP_TBL;
BEGIN
    SELECT * BULK COLLECT INTO t_emp FROM emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
        'SAL         ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------   ' || '--------  ------');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '   ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            RPAD(t_emp(i).job,10) || ' ' ||
            TO_CHAR(t_emp(i).hiredate,'DD-MON-YY') || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99') || ' ' ||
            TO_CHAR(NVL(t_emp(i).comm,0),'99,999.99') || '  ' ||
            t_emp(i).deptno);
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL         COMM      DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80    800.00        .00  20
7499   ALLEN    SALESMAN   20-FEB-81  1,600.00     300.00  30
7521   WARD     SALESMAN   22-FEB-81  1,250.00     500.00  30
7566   JONES    MANAGER    02-APR-81  2,975.00        .00  20
7654   MARTIN   SALESMAN   28-SEP-81  1,250.00   1,400.00  30
7698   BLAKE    MANAGER    01-MAY-81  2,850.00        .00  30
7782   CLARK    MANAGER    09-JUN-81  2,450.00        .00  10
7788   SCOTT    ANALYST    19-APR-87  3,000.00        .00  20
7839   KING     PRESIDENT  17-NOV-81  5,000.00        .00  10
7844   TURNER   SALESMAN   08-SEP-81  1,500.00        .00  30
7876   ADAMS    CLERK      23-MAY-87  1,100.00        .00  20
7900   JAMES    CLERK      03-DEC-81    950.00        .00  30
7902   FORD     ANALYST    03-DEC-81  3,000.00        .00  20
7934   MILLER   CLERK      23-JAN-82  1,300.00        .00  10
```

4.12.4.2 FETCH BULK COLLECT

The `BULK COLLECT` clause can be used with a `FETCH` statement. (See Section 4.8.3 for information on the `FETCH` statement.) Instead of returning a single row at a time from the result set, the `FETCH BULK COLLECT` will return all rows at once from the result set into the specified collection unless restricted by the `LIMIT` clause.

```
FETCH name BULK COLLECT INTO collection [, ...] [ LIMIT n ];
```

If a single collection is specified, then `collection` may be a collection of a single field, or it may be a collection of a record type. If more than one collection is specified, then each `collection` must consist of a single field. The expressions in the `SELECT` list of the cursor identified by `name` must match in number, order, and type-compatibility all fields in the target collections. If `LIMIT n` is specified, the number of rows returned into the collection on each `FETCH` will not exceed `n`.

The following example uses the `FETCH BULK COLLECT` statement to retrieve rows into an associative array.

```
DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    t_emp           EMP_TBL;
    CURSOR emp_cur IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur;
    FETCH emp_cur BULK COLLECT INTO t_emp;
    CLOSE emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
        'SAL         ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------   ' || '--------  ------');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '    ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            RPAD(t_emp(i).job,10) || ' ' ||
            TO_CHAR(t_emp(i).hiredate,'DD-MON-YY') || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99') || ' ' ||
            TO_CHAR(NVL(t_emp(i).comm,0),'99,999.99') || '  ' ||
            t_emp(i).deptno);
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL        COMM      DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80     800.00        .00  20
7499   ALLEN    SALESMAN   20-FEB-81   1,600.00     300.00  30
7521   WARD     SALESMAN   22-FEB-81   1,250.00     500.00  30
7566   JONES    MANAGER    02-APR-81   2,975.00        .00  20
7654   MARTIN   SALESMAN   28-SEP-81   1,250.00   1,400.00  30
7698   BLAKE    MANAGER    01-MAY-81   2,850.00        .00  30
7782   CLARK    MANAGER    09-JUN-81   2,450.00        .00  10
7788   SCOTT    ANALYST    19-APR-87   3,000.00        .00  20
7839   KING     PRESIDENT  17-NOV-81   5,000.00        .00  10
7844   TURNER   SALESMAN   08-SEP-81   1,500.00        .00  30
7876   ADAMS    CLERK      23-MAY-87   1,100.00        .00  20
7900   JAMES    CLERK      03-DEC-81     950.00        .00  30
7902   FORD     ANALYST    03-DEC-81   3,000.00        .00  20
7934   MILLER   CLERK      23-JAN-82   1,300.00        .00  10
```

4.12.4.3 EXECUTE IMMEDIATE BULK COLLECT

The BULK COLLECT clause can be used with a EXECUTE IMMEDIATE statement to specify a collection to receive the returned rows.

```
EXECUTE IMMEDIATE 'sql_expression;'
   BULK COLLECT INTO collection [,...]
   [USING {[bind_type] bind_argument} [, ...]}];
```

collection specifies the name of a collection.

bind_type specifies the parameter mode of the *bind_argument*.

- A *bind_type* of IN specifies that the *bind_argument* contains a value that is passed to the *sql_expression*.

- A *bind_type* of OUT specifies that the *bind_argument* receives a value from the *sql_expression*.

- A *bind_type* of IN OUT specifies that the *bind_argument* is passed to *sql_expression*, and then stores the value returned by *sql_expression*.

bind_argument specifies a parameter that contains a value that is either passed to the *sql_expression* (specified with a *bind_type* of IN), or that receives a value from the *sql_expression* (specified with a *bind_type* of OUT), or both (specified with a *bind_type* of IN OUT).

If a single collection is specified, then *collection* may be a collection of a single field, or a collection of a record type; if more than one collection is specified, each *collection* must consist of a single field.

4.12.4.4 RETURNING BULK COLLECT

The BULK COLLECT clause can be added to the RETURNING INTO clause of a DELETE, INSERT, or UPDATE command. (See Section 4.4.7 for information on the RETURNING INTO clause.)

```
{ insert | update | delete }
  RETURNING { * | expr_1 [, expr_2 ] ...}
    BULK COLLECT INTO collection [, ...];
```

insert, *update*, and *delete* are the INSERT, UPDATE, and DELETE commands as described in Sections 4.4.4, 4.4.5, and 4.4.6, respectively. If a single collection is specified, then *collection* may be a collection of a single field, or it may be a collection of a record type. If more than one collection is specified, then each `collection` must consist of a single field. The expressions following the `RETURNING` keyword must match in number, order, and type-compatibility all fields in the target collections. If `*` is specified, then all columns in the affected table are returned. (Note that the use of `*` is an Advanced Server extension and is not compatible with Oracle databases.)

The `clerkemp` table created by copying the `emp` table is used in the remaining examples in this section as shown below.

```
CREATE TABLE clerkemp AS SELECT * FROM emp WHERE job = 'CLERK';

SELECT * FROM clerkemp;

 empno | ename  |  job  | mgr  |      hiredate      |   sal   | comm | deptno
-------+--------+-------+------+--------------------+---------+------+-------
-
  7369 | SMITH  | CLERK | 7902 | 17-DEC-80 00:00:00 |  800.00 |      |     20
  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
  7934 | MILLER | CLERK | 7782 | 23-JAN-82 00:00:00 | 1300.00 |      |     10
(4 rows)
```

The following example increases everyone's salary by 1.5, stores the employees' numbers, names, and new salaries in three associative arrays, and finally, displays the contents of these arrays.

```
DECLARE
    TYPE empno_tbl IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    TYPE ename_tbl IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl   IS TABLE OF emp.sal%TYPE   INDEX BY BINARY_INTEGER;
    t_empno         EMPNO_TBL;
    t_ename         ENAME_TBL;
    t_sal           SAL_TBL;
BEGIN
    UPDATE clerkemp SET sal = sal * 1.5 RETURNING empno, ename, sal
        BULK COLLECT INTO t_empno, t_ename, t_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL        ');
    DBMS_OUTPUT.PUT_LINE('-----  -------    --------   ');
    FOR i IN 1..t_empno.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_empno(i) || '   ' || RPAD(t_ename(i),8) ||
            ' ' || TO_CHAR(t_sal(i),'99,999.99'));
    END LOOP;
END;

EMPNO  ENAME      SAL
-----  -------    --------
7369   SMITH      1,200.00
7876   ADAMS      1,650.00
7900   JAMES      1,425.00
7934   MILLER     1,950.00
```

The following example performs the same functionality as the previous example, but uses a single collection defined with a record type to store the employees' numbers, names, and new salaries.

```
DECLARE
```

```
    TYPE emp_rec IS RECORD (
        empno       emp.empno%TYPE,
        ename       emp.ename%TYPE,
        sal         emp.sal%TYPE
    );
    TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
    t_emp           EMP_TBL;
BEGIN
    UPDATE clerkemp SET sal = sal * 1.5 RETURNING empno, ename, sal
        BULK COLLECT INTO t_emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL        ');
    DBMS_OUTPUT.PUT_LINE('-----  -------    --------   ');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '   ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99'));
    END LOOP;
END;

EMPNO  ENAME      SAL
-----  -------    --------
7369   SMITH      1,200.00
7876   ADAMS      1,650.00
7900   JAMES      1,425.00
7934   MILLER     1,950.00
```

The following example deletes all rows from the `clerkemp` table, and returns information on the deleted rows into an associative array, which is then displayed.

```
DECLARE
    TYPE emp_rec IS RECORD (
        empno       emp.empno%TYPE,
        ename       emp.ename%TYPE,
        job         emp.job%TYPE,
        hiredate    emp.hiredate%TYPE,
        sal         emp.sal%TYPE,
        comm        emp.comm%TYPE,
        deptno      emp.deptno%TYPE
    );
    TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
    r_emp           EMP_TBL;
BEGIN
    DELETE FROM clerkemp RETURNING empno, ename, job, hiredate, sal,
        comm, deptno BULK COLLECT INTO r_emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
        'SAL         ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------    ' || '--------  ------');
    FOR i IN 1..r_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(r_emp(i).empno || '   ' ||
            RPAD(r_emp(i).ename,8) || ' ' ||
            RPAD(r_emp(i).job,10) || ' ' ||
            TO_CHAR(r_emp(i).hiredate,'DD-MON-YY') || ' ' ||
            TO_CHAR(r_emp(i).sal,'99,999.99') || ' ' ||
            TO_CHAR(NVL(r_emp(i).comm,0),'99,999.99') || '  ' ||
            r_emp(i).deptno);
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL         COMM      DEPTNO
-----  -------  ---------  ---------   --------    --------  ------
7369   SMITH    CLERK      17-DEC-80   1,200.00         .00  20
```

```
7876   ADAMS     CLERK      23-MAY-87   1,650.00        .00  20
7900   JAMES     CLERK      03-DEC-81   1,425.00        .00  30
7934   MILLER    CLERK      23-JAN-82   1,950.00        .00  10
```

4.13 Errors and Messages

Use the `DBMS_OUTPUT.PUT_LINE` statement to report messages.

```
DBMS_OUTPUT.PUT_LINE ( message );
```

`message` is any expression evaluating to a string.

This example displays the message on the user's output display:

```
DBMS_OUTPUT.PUT_LINE('My name is John');
```

The special variables `SQLCODE` and `SQLERRM` contain a numeric code and a text message, respectively, that describe the outcome of the last SQL command issued. If any other error occurs in the program such as division by zero, these variables contain information pertaining to the error.

5 Triggers

This chapter describes *triggers* in Advanced Server. As with procedures and functions, triggers are written in the SPL language.

5.1 Overview

A trigger is a named SPL code block that is associated with a table and stored in the database. When a specified event occurs on the associated table, the SPL code block is executed. The trigger is said to be *fired* when the code block is executed.

The event that causes a trigger to fire can be any combination of an insert, update, or deletion carried out on the table, either directly or indirectly. If the table is the object of a SQL `INSERT`, `UPDATE`, or `DELETE` command the trigger is directly fired assuming that the corresponding insert, update, or deletion event is defined as a *triggering event*. The events that fire the trigger are defined in the `CREATE TRIGGER` command.

A trigger can be fired indirectly if a triggering event occurs on the table as a result of an event initiated on another table. For example, if a trigger is defined on a table containing a foreign key defined with the `ON DELETE CASCADE` clause and a row in the parent table is deleted, all children of the parent would be deleted as well. If deletion is a triggering event on the child table, deletion of the children will cause the trigger to fire.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

5.2 Types of Triggers

Advanced Server supports both *row-level* and *statement-level* triggers. A row-level trigger fires once for each row that is affected by a triggering event. For example, if deletion is defined as a triggering event on a table and a single `DELETE` command is issued that deletes five rows from the table, then the trigger will fire five times, once for each row.

In contrast, a statement-level trigger fires once per triggering statement regardless of the number of rows affected by the triggering event. In the prior example of a single `DELETE` command deleting five rows, a statement-level trigger would fire only once.

The sequence of actions can be defined regarding whether the trigger code block is executed before or after the triggering statement, itself, in the case of statement-level triggers; or before or after each row is affected by the triggering statement in the case of row-level triggers.

In a *before* row-level trigger, the trigger code block is executed before the triggering action is carried out on each affected row. In a *before* statement-level trigger, the trigger code block is executed before the action of the triggering statement is carried out.

In an *after* row-level trigger, the trigger code block is executed after the triggering action is carried out on each affected row. In an *after* statement-level trigger, the trigger code block is executed after the action of the triggering statement is carried out.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

5.3 Creating Triggers

The `CREATE TRIGGER` command defines and names a trigger that will be stored in the database.

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
  [ 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 compatible with Oracle databases 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.

`WHEN` clauses cannot contain subqueries.

`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, or `REF CURSOR` declaration.

`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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

5.4 Trigger Variables

In the trigger code block, several special variables are available for use.

`NEW`

> `NEW` is a pseudo-record name that refers to the new table row for insert and update operations in row-level triggers. This variable is not applicable in statement-level triggers and in delete operations of row-level triggers.
>
> Its usage is: `:NEW.`*`column`* where *`column`* is the name of a column in the table on which the trigger is defined.
>
> The initial content of `:NEW.`*`column`* is the value in the named column of the new row to be inserted or of the new row that is to replace the old one when used in a before row-level trigger. When used in an after row-level trigger, this value has already been stored in the table since the action has already occurred on the affected row.
>
> In the trigger code block, `:NEW.`*`column`* can be used like any other variable. If a value is assigned to `:NEW.`*`column`*, in the code block of a before row-level trigger, the assigned value will be used in the new inserted or updated row.

`OLD`

> `OLD` is a pseudo-record name that refers to the old table row for update and delete operations in row-level triggers. This variable is not applicable in statement-level triggers and in insert operations of row-level triggers.
>
> Its usage is: `:OLD.`*`column`* where *`column`* is the name of a column in the table on which the trigger is defined.
>
> The initial content of `:OLD.`*`column`* is the value in the named column of the row to be deleted or of the old row that is to be replaced by the new one when used in a before row-level trigger. When used in an after row-level trigger, this value is no longer stored in the table since the action has already occurred on the affected row.
>
> In the trigger code block, `:OLD.`*`column`* can be used like any other variable. Assigning a value to `:OLD.`*`column`*, has no effect on the action of the trigger.

`INSERTING`

> `INSERTING` is a conditional expression that returns `TRUE` if an insert operation fired the trigger, otherwise it returns `FALSE`.

`UPDATING`

> `UPDATING` is a conditional expression that returns `TRUE` if an update operation fired the trigger, otherwise it returns `FALSE`.

`DELETING`

> `DELETING` is a conditional expression that returns `TRUE` if a delete operation fired the trigger, otherwise it returns `FALSE`.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

5.5 Transactions and Exceptions

A trigger is always executed as part of the same transaction within which the triggering statement is executing. When no exceptions occur within the trigger code block, the effects of any DML commands within the trigger are committed if and only if the transaction containing the triggering statement is committed. Therefore, if the transaction is rolled back, the effects of any DML commands within the trigger are also rolled back.

If an exception does occur within the trigger code block, but it is caught and handled in an exception section, the effects of any DML commands within the trigger are still rolled back nonetheless. The triggering statement itself, however, is not rolled back unless the application forces a roll back of the encapsulating transaction.

If an unhandled exception occurs within the trigger code block, the transaction that encapsulates the trigger is aborted and rolled back. Therefore the effects of any DML commands within the trigger and the triggering statement, itself are all rolled back.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

5.6 Trigger Examples

The following sections illustrate an example of each type of trigger.

5.6.1 Before Statement-Level Trigger

The following is an example of a simple before statement-level trigger that displays a message prior to an insert operation on the `emp` table.

```
CREATE OR REPLACE TRIGGER emp_alert_trig
    BEFORE INSERT ON emp
BEGIN
    DBMS_OUTPUT.PUT_LINE('New employees are about to be added');
END;
```

The following `INSERT` is constructed so that several new rows are inserted upon a single execution of the command. For each row that has an employee id between 7900 and 7999, a new row is inserted with an employee id incremented by 1000. The following are the results of executing the command when three new rows are inserted.

```
INSERT INTO emp (empno, ename, deptno) SELECT empno + 1000, ename, 40
    FROM emp WHERE empno BETWEEN 7900 AND 7999;
New employees are about to be added

SELECT empno, ename, deptno FROM emp WHERE empno BETWEEN 8900 AND 8999;

     EMPNO ENAME          DEPTNO
---------- ---------- ----------
      8900 JAMES              40
      8902 FORD               40
      8934 MILLER             40
```

The message, `New employees are about to be added`, is displayed once by the firing of the trigger even though the result is the addition of three new rows.

5.6.2 After Statement-Level Trigger

The following is an example of an after statement-level trigger. Whenever an insert, update, or delete operation occurs on the `emp` table, a row is added to the `empauditlog` table recording the date, user, and action.

```
CREATE TABLE empauditlog (
    audit_date      DATE,
    audit_user      VARCHAR2(20),
    audit_desc      VARCHAR2(20)
);
CREATE OR REPLACE TRIGGER emp_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action        VARCHAR2(20);
BEGIN
    IF INSERTING THEN
        v_action := 'Added employee(s)';
    ELSIF UPDATING THEN
        v_action := 'Updated employee(s)';
    ELSIF DELETING THEN
        v_action := 'Deleted employee(s)';
    END IF;
    INSERT INTO empauditlog VALUES (SYSDATE, USER,
        v_action);
END;
```

In the following sequence of commands, two rows are inserted into the `emp` table using two `INSERT` commands. The `sal` and `comm` columns of both rows are updated with one `UPDATE` command. Finally, both rows are deleted with one `DELETE` command.

```
INSERT INTO emp VALUES (9001,'SMITH','ANALYST',7782,SYSDATE,NULL,NULL,10);

INSERT INTO emp VALUES (9002,'JONES','CLERK',7782,SYSDATE,NULL,NULL,10);

UPDATE emp SET sal = 4000.00, comm = 1200.00 WHERE empno IN (9001, 9002);

DELETE FROM emp WHERE empno IN (9001, 9002);

SELECT TO_CHAR(AUDIT_DATE,'DD-MON-YY HH24:MI:SS') AS "AUDIT DATE",
    audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;

AUDIT DATE         AUDIT_USER           AUDIT_DESC
------------------ -------------------- --------------------
31-MAR-05 14:59:48 SYSTEM               Added employee(s)
31-MAR-05 15:00:07 SYSTEM               Added employee(s)
31-MAR-05 15:00:19 SYSTEM               Updated employee(s)
31-MAR-05 15:00:34 SYSTEM               Deleted employee(s)
```

The contents of the `empauditlog` table show how many times the trigger was fired - once each for the two inserts, once for the update (even though two rows were changed) and once for the deletion (even though two rows were deleted).

5.6.3 Before Row-Level Trigger

The following example is a before row-level trigger that calculates the commission of every new employee belonging to department 30 that is inserted into the `emp` table.

```
CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
BEGIN
    IF :NEW.deptno = 30 THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;
END;
```

The listing following the addition of the two employees shows that the trigger computed their commissions and inserted it as part of the new employee rows.

```
INSERT INTO emp VALUES (9005,'ROBERS','SALESMAN',7782,SYSDATE,3000.00,NULL,30);

INSERT INTO emp VALUES (9006,'ALLEN','SALESMAN',7782,SYSDATE,4500.00,NULL,30);
```

```
SELECT * FROM emp WHERE empno IN (9005, 9006);

     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
      9005 ROBERS     SALESMAN        7782 01-APR-05       3000       1200         30
      9006 ALLEN      SALESMAN        7782 01-APR-05       4500       1800         30
```

5.6.4 After Row-Level Trigger

The following example is an after row-level trigger. When a new employee row is inserted, the trigger adds a new row to the `jobhist` table for that employee. When an existing employee is updated, the trigger sets the `enddate` column of the latest `jobhist` row (assumed to be the one with a null `enddate`) to the current date and inserts a new `jobhist` row with the employee's new information.

Finally, trigger adds a row to the `empchglog` table with a description of the action.

```
CREATE TABLE empchglog (
    chg_date        DATE,
    chg_desc        VARCHAR2(30)
);
CREATE OR REPLACE TRIGGER emp_chg_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    IF INSERTING THEN
        v_action := 'Added';
        v_empno := :NEW.empno;
        v_deptno := :NEW.deptno;
        INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
            :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, 'New Hire');
    ELSIF UPDATING THEN
        v_action := 'Updated';
        v_empno := :NEW.empno;
        v_deptno := :NEW.deptno;
        v_chgdesc := '';
        IF NVL(:OLD.ename, '-null-') != NVL(:NEW.ename, '-null-') THEN
            v_chgdesc := v_chgdesc || 'name, ';
        END IF;
        IF NVL(:OLD.job, '-null-') != NVL(:NEW.job, '-null-') THEN
            v_chgdesc := v_chgdesc || 'job, ';
        END IF;
        IF NVL(:OLD.sal, -1) != NVL(:NEW.sal, -1) THEN
            v_chgdesc := v_chgdesc || 'salary, ';
        END IF;
        IF NVL(:OLD.comm, -1) != NVL(:NEW.comm, -1) THEN
            v_chgdesc := v_chgdesc || 'commission, ';
        END IF;
        IF NVL(:OLD.deptno, -1) != NVL(:NEW.deptno, -1) THEN
            v_chgdesc := v_chgdesc || 'department, ';
        END IF;
        v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ', ');
        UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
```

```
            AND enddate IS NULL;
        INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
            :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, v_chgdesc);
    ELSIF DELETING THEN
        v_action := 'Deleted';
        v_empno := :OLD.empno;
        v_deptno := :OLD.deptno;
    END IF;

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;
```

In the first sequence of commands shown below, two employees are added using two separate `INSERT` commands and then both are updated using a single `UPDATE` command. The contents of the `jobhist` table shows the action of the trigger for each affected row - two new hire entries for the two new employees and two changed commission records for the updated commissions on the two employees. The `empchglog` table also shows the trigger was fired a total of four times, once for each action on the two rows.

```
INSERT INTO emp VALUES (9003,'PETERS','ANALYST',7782,SYSDATE,5000.00,NULL,40);

INSERT INTO emp VALUES (9004,'AIKENS','ANALYST',7782,SYSDATE,4500.00,NULL,40);

UPDATE emp SET comm = sal * 1.1 WHERE empno IN (9003, 9004);

SELECT * FROM jobhist WHERE empno IN (9003, 9004);

     EMPNO STARTDATE ENDDATE   JOB              SAL       COMM     DEPTNO CHGDESC
---------- --------- --------- --------- ---------- ---------- ---------- -------------
      9003 31-MAR-05 31-MAR-05 ANALYST         5000                    40 New Hire
      9004 31-MAR-05 31-MAR-05 ANALYST         4500                    40 New Hire
      9003 31-MAR-05           ANALYST         5000       5500         40 Changed
commission
      9004 31-MAR-05           ANALYST         4500       4950         40 Changed
commission

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
```

Finally, both employees are deleted with a single `DELETE` command. The `empchglog` table now shows the trigger was fired twice, once for each deleted employee.

```
DELETE FROM emp WHERE empno IN (9003, 9004);

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
31-MAR-05 Deleted employee # 9003
31-MAR-05 Deleted employee # 9004
```

6 Packages

This chapter discusses the concept of packages in Advanced Server. A *package* is a named collection of functions, procedures, variables, cursors, user-defined record types, and records that are referenced using a common qualifier – the package identifier. Packages have the following characteristics:

- Packages provide a convenient means of organizing the functions and procedures that perform a related purpose. Permission to use the package functions and procedures is dependent upon one privilege granted to the entire package. All of the package programs must be referenced with a common name.
- Certain functions, procedures, variables, types, etc. in the package can be declared as *public*. Public entities are visible and can be referenced by other programs that are given `EXECUTE` privilege on the package. For public functions and procedures, only their signatures are visible - the program names, parameters if any, and return types of functions). The SPL code of these functions and procedures is not accessible to others, therefore applications that utilize a package are dependent only upon the information available in the signature – not in the procedural logic itself.
- Other functions, procedures, variables, types, etc. in the package can be declared as *private*. Private entities can be referenced and using by function and procedures within the package, but not by other external applications. Private entities are for use only by programs within the package.
- Function and procedure names can be overloaded within a package. One or more functions/procedures can be defined with the same name, but with different signatures. This provides the capability to create identically named programs that perform the same job, but on different types of input.

6.1 Package Components

Packages consist of two main components:

- The *package specification*: This is the public interface, (these are the elements which can be referenced outside the package). We declare all database objects that are to be a part of our package within the specification.
- The *package body*: This contains the actual implementation of all the database objects declared within the package specification.

The package body implements the specifications in the package specification. It contains implementation details and private declarations which are invisible to the application. You can debug, enhance or replace a package body without changing the specifications. Similarly, you can change the body without recompiling the calling programs because the implementation details are invisible to the application.

6.1.1 Package Specification Syntax

The package specification defines the user interface for a package (the API). The specification lists the functions, procedures, types, exceptions and cursors that are visible to a user of the package.

The syntax used to define the interface for a package is:

```
CREATE [ OR REPLACE ] PACKAGE package_name
  [ authorization_clause ]
  { IS | AS }
  [ declaration; ] ...
  [ procedure_or_function_declaration; ] ...
END [ package_name ] ;
```

Where *authorization_clause* :=

```
{ AUTHID DEFINER } | { AUTHID CURRENT_USER }
```

Where *procedure_or_function_declaration* :=

```
procedure_declaration | function_declaration
```

Where *procedure_declaration* :=

```
PROCEDURE proc_name[ argument_list ] [restriction_pragma];
```

Where *function_declaration* :=

```
FUNCTION func_name [ argument_list ]
  RETURN rettype [ restriction_pragma ];
```

Where *argument_list* :=

```
( argument_declaration [, ...] )
```

Where *argument_declaration* :=

```
argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
```

Where *restriction_pragma* :=

```
PRAGMA RESTRICT_REFERENCES(name, restrictions)
```

Where *restrictions* :=

```
restriction [, ... ]
```

Parameters

package_name

> *package_name* is an identifier assigned to the package - each package must have a name unique within the schema.

AUTHID DEFINER

> If you omit the AUTHID clause or specify AUTHID DEFINER, the privileges of the package owner are used to determine access privileges to database objects.

AUTHID CURRENT_USER

> If you specify AUTHID CURRENT_USER, the privileges of the current user executing a program in the package are used to determine access privileges.

declaration

> *declaration* is an identifier of a public variable. A public variable can be accessed from outside of the package using the syntax *package_name.variable*. There can be zero, one, or more public variables. Public variable definitions must come before procedure or function declarations.
>
> *declaration* can be any of the following:
>
> - Variable Declaration
> - Record Declaration (see Section 4.3.4)
> - Collection Declaration (see Section 4.10)
> - REF CURSOR and Cursor Variable Declaration
> - TYPE Definitions for Records, Collections, and REF CURSORs
> - Exception
> - Object Variable Declaration (see Section 8.4)

argname

> The name of an argument. The argument is referenced by this name within the function or procedure body.

IN | IN OUT | OUT

> The argument mode. IN declares the argument for input only. This is the default. IN OUT allows the argument to receive a value as well as return a value. OUT specifies the argument is for output only.

argtype

The data type(s) of an argument. An argument type may be a base data type, a copy of the type of an existing column using `%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)`.

The type of a column is referenced by writing *`tablename.columnname`*`%TYPE`; using this can sometimes help make a procedure independent from changes to the definition of a table.

`DEFAULT` *`value`*

The `DEFAULT` clause supplies a default value for an input argument if one is not supplied in the invocation. `DEFAULT` may not be specified for arguments with modes `IN OUT` or `OUT`.

`name`

`name` is the name of the function or procedure.

`restriction`

The following keywords are accepted for compatibility and ignored:

`RNDS`

`RNPS`

`TRUST`

`WNDS`

`WNPS`

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

6.1.2 Package Body Syntax

Package implementation details reside in the package body; the package body may contain objects that are not visible to the package user. Advanced Server supports the following syntax for the package body:

```
CREATE [ OR REPLACE ] PACKAGE BODY package_name
  { IS | AS }
  [ private_declaration; ] ...
  [ procedure_or_function_definition; ] ...
  [ package_initializer ]
END [ package_name ] ;
```

Where *procedure_or_function_definition* :=

```
procedure_definition | function_definition
```

Where *procedure_definition* :=

```
PROCEDURE proc_name[ argument_list ]
  [ options_list ]
  { IS | AS }
    procedure_body
  END [ proc_name ] ;
```

Where *procedure_body* :=

```
[ declaration; ] [, ...]
BEGIN
  statement; [...]
[ EXCEPTION
   { WHEN exception [OR exception] [...]] THEN statement; }
   [...]
]
```

Where *function_definition* :=

```
FUNCTION func_name [ argument_list ]
  RETURN rettype [DETERMINISTIC]
  [ options_list ]
  { IS | AS }
    function_body
  END [ func_name ] ;
```

Where *function_body* :=

```
[ declaration; ] [, ...]
BEGIN
  statement; [...]
```

```
[ EXCEPTION
  { WHEN exception [ OR exception ] [...] THEN statement; }
  [...]
]
```

Where *argument_list* :=

```
( argument_declaration [, ...] )
```

Where *argument_declaration* :=

```
argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
```

Where *options_list* :=

```
option [ ... ]
```

Where *option* :=

```
STRICT
LEAKPROOF
COST execution_cost
ROWS result_rows
SET config_param { TO value | = value | FROM CURRENT }
```

Where *package_initializer* :=

```
BEGIN
 statement; [...]
END;
```

Parameters

package_name

package_name is the name of the package for which this is the package body. There must be an existing package specification with this name.

private_declaration

private_declaration is an identifier of a private variable that can be accessed by any procedure or function within the package. There can be zero, one, or more private variables. *private_declaration* can be any of the following:

- Variable Declaration
- Record Declaration (see Section 4.3.4)
- Collection Declaration (see Section 4.10)
- `REF CURSOR` and Cursor Variable Declaration
- `TYPE` Definitions for Records, Collections, and `REF CURSOR`s
- Exception
- Object Variable Declaration (see Section 8.4)

proc_name

The name of the procedure being created.

declaration

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

statement

An SPL program statement. Note that a `DECLARE - BEGIN - END` block is considered an SPL statement unto itself. Thus, the function body may contain nested blocks.

exception

An exception condition name such as `NO_DATA_FOUND`, `OTHERS`, etc.

func_name

The name of the function being created.

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*.

`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.

declaration

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

argname

The name of a formal argument. The argument is referenced by this name within the procedure body.

`IN | IN OUT | OUT`

The argument mode. `IN` declares the argument for input only. This is the default. `IN OUT` allows the argument to receive a value as well as return a value. `OUT` specifies the argument is for output only.

argtype

The data type(s) of an argument. An argument type may be a base data type, a copy of the type of an existing column using `%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)`.

The type of a column is referenced by writing *tablename.columnname*`%TYPE`; using this can sometimes help make a procedure independent from changes to the definition of a table.

`DEFAULT` *value*

The `DEFAULT` clause supplies a default value for an input argument if one is not supplied in the procedure call. `DEFAULT` may not be specified for arguments with modes `IN OUT` or `OUT`.

Please note: the following options are not compatible with Oracle databases; they are extensions to Oracle package syntax provided by Advanced Server only.

`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.

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.

package_initializer

> The statements in the *package_initializer* are executed once per user's session when the package is first referenced.

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

6.2 Creating Packages

A package is not an executable piece of code; rather it is a repository of code. When you use a package, you actually execute or make reference to an element within a package.

6.2.1 Creating the Package Specification

The package specification contains the definition of all the elements in the package that can be referenced from outside of the package. These are called the public elements of the package, and they act as the package interface. The following code sample is a package specification:

```
--
--  Package specification for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE emp_admin
IS

   FUNCTION get_dept_name (
      p_deptno         NUMBER DEFAULT 10
   )
   RETURN VARCHAR2;
   FUNCTION update_emp_sal (
      p_empno          NUMBER,
      p_raise          NUMBER
   )
   RETURN NUMBER;
   PROCEDURE hire_emp (
      p_empno          NUMBER,
      p_ename          VARCHAR2,
      p_job            VARCHAR2,
      p_sal            NUMBER,
      p_hiredate       DATE DEFAULT sysdate,
      p_comm           NUMBER DEFAULT 0,
      p_mgr            NUMBER,
      p_deptno         NUMBER DEFAULT 10
   );
   PROCEDURE fire_emp (
      p_empno          NUMBER
   );

END emp_admin;
```

This code sample creates the `emp_admin` package specification. This package specification consists of two functions and two stored procedures. We can also add the `OR REPLACE` clause to the `CREATE PACKAGE` statement for convenience.

6.2.2 Creating the Package Body

The body of the package contains the actual implementation behind the package specification. For the above `emp_admin` package specification, we shall now create a package body which will implement the specifications. The body will contain the implementation of the functions and stored procedures in the specification.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

```
--
--  Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
   --
   --  Function that queries the 'dept' table based on the department
   --  number and returns the corresponding department name.
   --
   FUNCTION get_dept_name (
      p_deptno        IN NUMBER DEFAULT 10
   )
   RETURN VARCHAR2
   IS
      v_dname         VARCHAR2(14);
   BEGIN
      SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
      RETURN v_dname;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
         RETURN '';
   END;
   --
   --  Function that updates an employee's salary based on the
   --  employee number and salary increment/decrement passed
   --  as IN parameters.  Upon successful completion the function
   --  returns the new updated salary.
   --
   FUNCTION update_emp_sal (
      p_empno         IN NUMBER,
      p_raise         IN NUMBER
   )
   RETURN NUMBER
   IS
      v_sal           NUMBER := 0;
   BEGIN
      SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
      v_sal := v_sal + p_raise;
      UPDATE emp SET sal = v_sal WHERE empno = p_empno;
      RETURN v_sal;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
         RETURN -1;
      WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
         DBMS_OUTPUT.PUT_LINE(SQLERRM);
         DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
         DBMS_OUTPUT.PUT_LINE(SQLCODE);
         RETURN -1;
   END;
   --
   --  Procedure that inserts a new employee record into the 'emp' table.
   --
   PROCEDURE hire_emp (
      p_empno         NUMBER,
      p_ename         VARCHAR2,
      p_job           VARCHAR2,
      p_sal           NUMBER,
      p_hiredate      DATE    DEFAULT sysdate,
      p_comm          NUMBER  DEFAULT 0,
      p_mgr           NUMBER,
      p_deptno        NUMBER  DEFAULT 10
   )
   AS
   BEGIN
      INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
         VALUES(p_empno, p_ename, p_job, p_sal,
                p_hiredate, p_comm, p_mgr, p_deptno);
   END;
   --
   --  Procedure that deletes an employee record from the 'emp' table based
   --  on the employee number.
   --
   PROCEDURE fire_emp (
      p_empno         NUMBER
   )
   AS
   BEGIN
      DELETE FROM emp WHERE empno = p_empno;
   END;
END;
```

6.3 Referencing a Package

To reference the types, items and subprograms that are declared within a package specification, we use the dot notation. For example:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
```

To invoke a function from the `emp_admin` package specification, we will execute the following SQL command.

```
SELECT emp_admin.get_dept_name(10) FROM DUAL;
```

Here we are invoking the `get_dept_name` function declared within the package `emp_admin`. We are passing the department number as an argument to the function, which will return the name of the department. Here the value returned should be `ACCOUNTING`, which corresponds to department number `10`.

6.4 Using Packages With User Defined Types

The following example incorporates the various user-defined types discussed in earlier chapters within the context of a package.

The package specification of `emp_rpt` shows the declaration of a record type, `emprec_typ`, and a weakly-typed `REF CURSOR`, `emp_refcur`, as publicly accessible along with two functions and two procedures. Function, `open_emp_by_dept`, returns the `REF CURSOR` type, `EMP_REFCUR`. Procedures, `fetch_emp` and `close_refcur`, both declare a weakly-typed `REF CURSOR` as a formal parameter.

```
CREATE OR REPLACE PACKAGE emp_rpt
IS
    TYPE emprec_typ IS RECORD (
        empno        NUMBER(4),
        ename        VARCHAR(10)
    );
    TYPE emp_refcur IS REF CURSOR;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2;
    FUNCTION open_emp_by_dept (
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR;
    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    );
    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    );
END emp_rpt;
```

The package body shows the declaration of several private variables - a static cursor, `dept_cur`, a table type, `depttab_typ`, a table variable, `t_dept`, an integer variable, `t_dept_max`, and a record variable, `r_emp`.

```
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
    CURSOR dept_cur IS SELECT * FROM dept;
    TYPE depttab_typ IS TABLE of dept%ROWTYPE
        INDEX BY BINARY_INTEGER;
    t_dept          DEPTTAB_TYP;
    t_dept_max      INTEGER := 1;
    r_emp           EMPREC_TYP;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2
    IS
    BEGIN
        FOR i IN 1..t_dept_max LOOP
            IF p_deptno = t_dept(i).deptno THEN
                RETURN t_dept(i).dname;
            END IF;
        END LOOP;
        RETURN 'Unknown';
    END;

    FUNCTION open_emp_by_dept(
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR
    IS
        emp_by_dept EMP_REFCUR;
    BEGIN
        OPEN emp_by_dept FOR SELECT empno, ename FROM emp
            WHERE deptno = p_deptno;
        RETURN emp_by_dept;
    END;

    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
        DBMS_OUTPUT.PUT_LINE('-----    -------');
        LOOP
            FETCH p_refcur INTO r_emp;
            EXIT WHEN p_refcur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' || r_emp.ename);
        END LOOP;
    END;

    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        CLOSE p_refcur;
    END;
BEGIN
    OPEN dept_cur;
    LOOP
        FETCH dept_cur INTO t_dept(t_dept_max);
        EXIT WHEN dept_cur%NOTFOUND;
        t_dept_max := t_dept_max + 1;
    END LOOP;
    CLOSE dept_cur;
    t_dept_max := t_dept_max - 1;
END emp_rpt;
```

This package contains an initialization section that loads the private table variable, `t_dept`, using the private static cursor, `dept_cur`. `t_dept` serves as a department name lookup table in function, `get_dept_name`.

Function, `open_emp_by_dept` returns a `REF CURSOR` variable for a result set of employee numbers and names for a given department. This `REF CURSOR` variable can then be passed to procedure, `fetch_emp`, to retrieve and list the individual rows of the result set. Finally, procedure, `close_refcur`, can be used to close the `REF CURSOR` variable associated with this result set.

The following anonymous block runs the package function and procedures. In the anonymous block's declaration section, note the declaration of cursor variable, `v_emp_cur`, using the package's public `REF CURSOR` type, `EMP_REFCUR`. `v_emp_cur` contains the pointer to the result set that is passed between the package function and procedures.

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    emp_rpt.fetch_emp(v_emp_cur);
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    emp_rpt.close_refcur(v_emp_cur);
END;
```

The following is the result of this anonymous block.

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

The following anonymous block illustrates another means of achieving the same result. Instead of using the package procedures, `fetch_emp` and `close_refcur`, the logic of these programs is coded directly into the anonymous block. In the anonymous block's declaration section, note the addition of record variable, `r_emp`, declared using the package's public record type, `EMPREC_TYP`.

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
    r_emp           emp_rpt.EMPREC_TYP;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH v_emp_cur INTO r_emp;
        EXIT WHEN v_emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' ||
            r_emp.ename);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    CLOSE v_emp_cur;
END;
```

The following is the result of this anonymous block.

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

6.5 Dropping a Package

The syntax for deleting an entire package or just the package body is as follows:

```
DROP PACKAGE [ BODY ] package_name;
```

If the keyword, `BODY`, is omitted, both the package specification and the package body are deleted - i.e., the entire package is dropped. If the keyword, `BODY`, is specified, then only the package body is dropped. The package specification remains intact. *`package_name`* is the identifier of the package to be dropped.

Following statement will destroy only the package body of *`emp_admin`*:

```
DROP PACKAGE BODY emp_admin;
```

The following statement will drop the entire *`emp_admin`* package:

```
DROP PACKAGE emp_admin;
```

7 Built-In Packages

This chapter describes the built-in packages that are provided with Advanced Server. For certain packages, non-superusers must be explicitly granted the `EXECUTE` privilege on the package before using any of the package's functions or procedures. For most of the built-in packages, `EXECUTE` privilege has been granted to `PUBLIC` by default. See the `GRANT` command for granting privileges.

All built-in packages are owned by the special `sys` user which must be specified when granting or revoking privileges on built-in packages:

```
GRANT EXECUTE ON PACKAGE SYS.UTL_FILE TO john;
```

7.1 DBMS_ALERT

The `DBMS_ALERT` package provides the capability to register for, send, and receive alerts.

Table 7-7-1 `DBMS_ALERT` Functions/Procedures

Function/Procedure	Return Type	Description
`REGISTER(`*`name`*`)`	n/a	Register to be able to receive alerts named, *`name`*.
`REMOVE(`*`name`*`)`	n/a	Remove registration for the alert named, *`name`*.
`REMOVEALL`	n/a	Remove registration for all alerts.
`SIGNAL(`*`name`*`, `*`message`*`)`	n/a	Signals the alert named, *`name`*, with *`message`*.
`WAITANY(`*`name`*` OUT, `*`message`*` OUT, `*`status`*` OUT, `*`timeout`*`)`	n/a	Wait for any registered alert to occur.
`WAITONE(`*`name`*`, `*`message`*` OUT, `*`status`*` OUT, `*`timeout`*`)`	n/a	Wait for the specified alert, *`name`*, to occur.

Advanced Server's implementation of `DBMS_ALERT` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

Advanced Server allows a maximum of `500` concurrent alerts. You can use the `dbms_alert.max_alerts` GUC variable (located in the `postgresql.conf` file) to specify the maximum number of concurrent alerts allowed on a system.

To set a value for the dbms_alert.max_alerts variable, open the `postgresql.conf` file (located by default in `/opt/PostgresPlus/9.5AS/data`) with your choice of editor, and edit the `dbms_alert.max_alerts` parameter as shown:

```
dbms_alert.max_alerts = alert_count
```

alert_count

alert_count specifies the maximum number of concurrent alerts. By default, the value of dbms_alert.max_alerts is 100. To disable this feature, set dbms_alert.max_alerts to 0.

For the dbms_alert.max_alerts GUC to function correctly, the custom_variable_classes parameter must contain dbms_alerts:

```
custom_variable_classes = 'dbms_alert, …'
```

After editing the postgresql.conf file parameters, you must restart the server for the changes to take effect.

7.1.1 REGISTER

The REGISTER procedure enables the current session to be notified of the specified alert.

REGISTER(*name* VARCHAR2)

Parameters

name

Name of the alert to be registered.

Examples

The following anonymous block registers for an alert named, alert_test, then waits for the signal.

```
DECLARE
    v_name          VARCHAR2(30) := 'alert_test';
    v_msg           VARCHAR2(80);
    v_status        INTEGER;
    v_timeout       NUMBER(3)  := 120;
BEGIN
    DBMS_ALERT.REGISTER(v_name);
    DBMS_OUTPUT.PUT_LINE('Registered for alert ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
    DBMS_ALERT.WAITONE(v_name,v_msg,v_status,v_timeout);
    DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
    DBMS_ALERT.REMOVE(v_name);
END;

Registered for alert alert_test
Waiting for signal...
```

7.1.2 REMOVE

The `REMOVE` procedure unregisters the session for the named alert.

REMOVE(*name* VARCHAR2)

Parameters

name

Name of the alert to be unregistered.

7.1.3 REMOVEALL

The `REMOVEALL` procedure unregisters the session for all alerts.

REMOVEALL

7.1.4 SIGNAL

The `SIGNAL` procedure signals the occurrence of the named alert.

SIGNAL(*name* VARCHAR2, *message* VARCHAR2)

Parameters

name

Name of the alert.

message

Information to pass with this alert.

Examples

The following anonymous block signals an alert for `alert_test`.

```
DECLARE
    v_name   VARCHAR2(30) := 'alert_test';
BEGIN
    DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;

Issued alert for alert_test
```

7.1.5 WAITANY

The `WAITANY` procedure waits for any of the registered alerts to occur.

```
WAITANY(name OUT VARCHAR2, message OUT VARCHAR2,
  status OUT INTEGER, timeout NUMBER)
```

Parameters

name

Variable receiving the name of the alert.

message

Variable receiving the message sent by the `SIGNAL` procedure.

status

Status code returned by the operation. Possible values are: 0 – alert occurred; 1 – timeout occurred.

timeout

Time to wait for an alert in seconds.

Examples

The following anonymous block uses the `WAITANY` procedure to receive an alert named, `alert_test` or `any_alert`:

```
DECLARE
    v_name          VARCHAR2(30);
    v_msg           VARCHAR2(80);
    v_status        INTEGER;
    v_timeout       NUMBER(3)  := 120;
BEGIN
    DBMS_ALERT.REGISTER('alert_test');
    DBMS_ALERT.REGISTER('any_alert');
    DBMS_OUTPUT.PUT_LINE('Registered for alert alert_test and any_alert');
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
    DBMS_ALERT.WAITANY(v_name,v_msg,v_status,v_timeout);
    DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
    DBMS_ALERT.REMOVEALL;
END;

Registered for alert alert_test and any_alert
```

```
Waiting for signal...
```

An anonymous block in a second session issues a signal for `any_alert`:

```
DECLARE
    v_name   VARCHAR2(30) := 'any_alert';
BEGIN
    DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;

Issued alert for any_alert
```

Control returns to the first anonymous block and the remainder of the code is executed:

```
Registered for alert alert_test and any_alert
Waiting for signal...
Alert name   : any_alert
Alert msg    : This is the message from any_alert
Alert status : 0
Alert timeout: 120 seconds
```

7.1.6 WAITONE

The `WAITONE` procedure waits for the specified registered alert to occur.

```
WAITONE(name VARCHAR2, message OUT VARCHAR2,
  status OUT INTEGER, timeout NUMBER)
```

Parameters

name

Name of the alert.

message

Variable receiving the message sent by the `SIGNAL` procedure.

status

Status code returned by the operation. Possible values are: 0 – alert occurred; 1 – timeout occurred.

timeout

Time to wait for an alert in seconds.

Examples

The following anonymous block is similar to the one used in the WAITANY example except the WAITONE procedure is used to receive the alert named, alert_test.

```
DECLARE
    v_name          VARCHAR2(30) := 'alert_test';
    v_msg           VARCHAR2(80);
    v_status        INTEGER;
    v_timeout       NUMBER(3) := 120;
BEGIN
    DBMS_ALERT.REGISTER(v_name);
    DBMS_OUTPUT.PUT_LINE('Registered for alert ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
    DBMS_ALERT.WAITONE(v_name,v_msg,v_status,v_timeout);
    DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
    DBMS_ALERT.REMOVE(v_name);
END;

Registered for alert alert_test
Waiting for signal...
```

Signal sent for alert_test sent by an anonymous block in a second session:

```
DECLARE
    v_name   VARCHAR2(30) := 'alert_test';
BEGIN
    DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;

Issued alert for alert_test
```

First session is alerted, control returns to the anonymous block, and the remainder of the code is executed:

```
Registered for alert alert_test
Waiting for signal...
Alert name   : alert_test
Alert msg    : This is the message from alert_test
Alert status : 0
Alert timeout: 120 seconds
```

7.1.7 Comprehensive Example

The following example uses two triggers to send alerts when the dept table or the emp table is changed. An anonymous block listens for these alerts and displays messages when an alert is received.

The following are the triggers on the dept and emp tables:

```
CREATE OR REPLACE TRIGGER dept_alert_trig
    AFTER INSERT OR UPDATE OR DELETE ON dept
DECLARE
```

```
    v_action        VARCHAR2(25);
BEGIN
    IF INSERTING THEN
        v_action := ' added department(s) ';
    ELSIF UPDATING THEN
        v_action := ' updated department(s) ';
    ELSIF DELETING THEN
        v_action := ' deleted department(s) ';
    END IF;
    DBMS_ALERT.SIGNAL('dept_alert',USER || v_action || 'on ' ||
        SYSDATE);
END;

CREATE OR REPLACE TRIGGER emp_alert_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action        VARCHAR2(25);
BEGIN
    IF INSERTING THEN
        v_action := ' added employee(s) ';
    ELSIF UPDATING THEN
        v_action := ' updated employee(s) ';
    ELSIF DELETING THEN
        v_action := ' deleted employee(s) ';
    END IF;
    DBMS_ALERT.SIGNAL('emp_alert',USER || v_action || 'on ' ||
        SYSDATE);
END;
```

The following anonymous block is executed in a session while updates to the `dept` and `emp` tables occur in other sessions:

```
DECLARE
    v_dept_alert     VARCHAR2(30) := 'dept_alert';
    v_emp_alert      VARCHAR2(30) := 'emp_alert';
    v_name           VARCHAR2(30);
    v_msg            VARCHAR2(80);
    v_status         INTEGER;
    v_timeout        NUMBER(3) := 60;
BEGIN
    DBMS_ALERT.REGISTER(v_dept_alert);
    DBMS_ALERT.REGISTER(v_emp_alert);
    DBMS_OUTPUT.PUT_LINE('Registered for alerts dept_alert and emp_alert');
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
    LOOP
        DBMS_ALERT.WAITANY(v_name,v_msg,v_status,v_timeout);
        EXIT WHEN v_status != 0;
        DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
        DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
        DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
        DBMS_OUTPUT.PUT_LINE('------------------------------------' ||
            '-------------------------');
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_ALERT.REMOVEALL;
END;

Registered for alerts dept_alert and emp_alert
Waiting for signal...
```

The following changes are made by user, mary:

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

```
INSERT INTO dept VALUES (50,'FINANCE','CHICAGO');
INSERT INTO emp (empno,ename,deptno) VALUES (9001,'JONES',50);
INSERT INTO emp (empno,ename,deptno) VALUES (9002,'ALICE',50);
```

The following change is made by user, john:

```
INSERT INTO dept VALUES (60,'HR','LOS ANGELES');
```

The following is the output displayed by the anonymous block receiving the signals from the triggers:

```
Registered for alerts dept_alert and emp_alert
Waiting for signal...
Alert name   : dept_alert
Alert msg    : mary added department(s) on 25-OCT-07 16:41:01
Alert status : 0
-------------------------------------------------------------
Alert name   : emp_alert
Alert msg    : mary added employee(s) on 25-OCT-07 16:41:02
Alert status : 0
-------------------------------------------------------------
Alert name   : dept_alert
Alert msg    : john added department(s) on 25-OCT-07 16:41:22
Alert status : 0
-------------------------------------------------------------
Alert status : 1
```

7.2 DBMS_CRYPTO

The DBMS_CRYPTO package provides functions and procedures that allow you to encrypt or decrypt RAW, BLOB or CLOB data. You can also use DBMS_CRYPTO functions to generate cryptographically strong random values.

Table 7.7.2 DBMS_CRYPTO Functions and Procedures

Function/Procedure	Return Type	Description
DECRYPT(*src, typ, key, iv*)	RAW	Decrypts RAW data.
DECRYPT(*dst* INOUT, *src, typ, key, iv*)	N/A	Decrypts BLOB data.
DECRYPT(*dst* INOUT, *src, typ, key, iv*)	N/A	Decrypts CLOB data.
ENCRYPT(*src, typ, key, iv*)	RAW	Encrypts RAW data.
ENCRYPT(*dst* INOUT, *src, typ, key, iv*)	N/A	Encrypts BLOB data.
ENCRYPT(*dst* INOUT, *src, typ, key, iv*)	N/A	Encrypts CLOB data.
HASH(*src, typ*)	RAW	Applies a hash algorithm to RAW data.
HASH(*src*)	RAW	Applies a hash algorithm to CLOB data.
MAC(*src, typ, key*)	RAW	Returns the hashed MAC value of the given RAW data using the specified hash algorithm and key.
MAC(*src, typ, key*)	RAW	Returns the hashed MAC value of the given CLOB data using the specified hash algorithm and key.
RANDOMBYTES(*number_bytes*)	RAW	Returns a specified number of cryptographically strong random bytes.
RANDOMINTEGER()	INTEGER	Returns a random INTEGER.
RANDOMNUMBER()	NUMBER	Returns a random NUMBER.

DBMS_CRYPTO functions and procedures support the following error messages:

```
ORA-28239 - DBMS_CRYPTO.KeyNull

ORA-28829 - DBMS_CRYPTO.CipherSuiteNull

ORA-28827 - DBMS_CRYPTO.CipherSuiteInvalid
```

Unlike Oracle, Advanced Server will *not* return error ORA-28233 if you re-encrypt previously encrypted information.

Please note that RAW and BLOB are synonyms for the PostgreSQL BYTEA data type, and CLOB is a synonym for TEXT.

7.2.1 DECRYPT

The `DECRYPT` function or procedure decrypts data using a user-specified cipher algorithm, key and optional initialization vector. The signature of the `DECRYPT` function is:

```
DECRYPT
  (src IN RAW, typ IN INTEGER, key IN RAW, iv IN RAW
   DEFAULT NULL) RETURN RAW
```

The signature of the `DECRYPT` procedure is:

```
DECRYPT
  (dst INOUT BLOB, src IN BLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

or

```
DECRYPT
  (dst INOUT CLOB, src IN CLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

When invoked as a procedure, `DECRYPT` returns `BLOB` or `CLOB` data to a user-specified `BLOB`.

Parameters

`dst`

> *`dst`* specifies the name of a `BLOB` to which the output of the `DECRYPT` procedure will be written. The `DECRYPT` procedure will overwrite any existing data currently in *`dst`*.

`src`

> *`src`* specifies the source data that will be decrypted. If you are invoking `DECRYPT` as a function, specify `RAW` data; if invoking `DECRYPT` as a procedure, specify `BLOB` or `CLOB` data.

`typ`

> *`typ`* specifies the block cipher type and any modifiers. This should match the type specified when the *`src`* was encrypted. Advanced Server supports the following block cipher algorithms, modifiers and cipher suites:

Block Cipher Algorithms	
ENCRYPT_DES	CONSTANT INTEGER := 1;
ENCRYPT_3DES	CONSTANT INTEGER := 3;
ENCRYPT_AES	CONSTANT INTEGER := 4;
ENCRYPT_AES128	CONSTANT INTEGER := 6;
Block Cipher Modifiers	
CHAIN_CBC	CONSTANT INTEGER := 256;
CHAIN_ECB	CONSTANT INTEGER := 768;
Block Cipher Padding Modifiers	
PAD_PKCS5	CONSTANT INTEGER := 4096;
PAD_NONE	CONSTANT INTEGER := 8192;
Block Cipher Suites	
DES_CBC_PKCS5	CONSTANT INTEGER := ENCRYPT_DES + CHAIN_CBC + PAD_PKCS5;
DES3_CBC_PKCS5	CONSTANT INTEGER := ENCRYPT_3DES + CHAIN_CBC + PAD_PKCS5;
AES_CBC_PKCS5	CONSTANT INTEGER := ENCRYPT_AES + CHAIN_CBC + PAD_PKCS5;

key

key specifies the user-defined decryption key. This should match the key specified when the *src* was encrypted.

iv

iv (optional) specifies an initialization vector. If an initialization vector was specified when the *src* was encrypted, you must specify an initialization vector when decrypting the *src*. The default is NULL.

Examples

The following example uses the DBMS_CRYPTO.DECRYPT function to decrypt an encrypted password retrieved from the passwords table:

```
CREATE TABLE passwords
(
  principal  VARCHAR2(90) PRIMARY KEY,  -- username
  ciphertext RAW(9)                     -- encrypted password
);
CREATE FUNCTION get_password(username VARCHAR2) RETURN RAW AS
 typ        INTEGER := DBMS_CRYPTO.DES_CBC_PKCS5;
 key        RAW(128) := 'my secret key';
 iv         RAW(100) := 'my initialization vector';
 password   RAW(2048);
BEGIN

  SELECT ciphertext INTO password FROM passwords WHERE principal = username;

  RETURN dbms_crypto.decrypt(password, typ, key, iv);
END;
```

Note that when calling DECRYPT, you must pass the same cipher type, key value and initialization vector that was used when ENCRYPTING the target.

7.2.2 ENCRYPT

The `ENCRYPT` function or procedure uses a user-specified algorithm, key, and optional initialization vector to encrypt `RAW`, `BLOB` or `CLOB` data. The signature of the `ENCRYPT` function is:

```
ENCRYPT
  (src IN RAW, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL) RETURN RAW
```

The signature of the `ENCRYPT` procedure is:

```
ENCRYPT
  (dst INOUT BLOB, src IN BLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

or

```
ENCRYPT
  (dst INOUT BLOB, src IN CLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

When invoked as a procedure, `ENCRYPT` returns `BLOB` or `CLOB` data to a user-specified `BLOB`.

Parameters

dst

dst specifies the name of a `BLOB` to which the output of the `ENCRYPT` procedure will be written. The `ENCRYPT` procedure will overwrite any existing data currently in *dst*.

src

src specifies the source data that will be encrypted. If you are invoking `ENCRYPT` as a function, specify `RAW` data; if invoking `ENCRYPT` as a procedure, specify `BLOB` or `CLOB` data.

typ

typ specifies the block cipher type that will be used by `ENCRYPT`, and any modifiers. Advanced Server supports the block cipher algorithms, modifiers and cipher suites listed below:

Block Cipher Algorithms	
`ENCRYPT_DES`	`CONSTANT INTEGER := 1;`
`ENCRYPT_3DES`	`CONSTANT INTEGER := 3;`
`ENCRYPT_AES`	`CONSTANT INTEGER := 4;`
`ENCRYPT_AES128`	`CONSTANT INTEGER := 6;`
Block Cipher Modifiers	
`CHAIN_CBC`	`CONSTANT INTEGER := 256;`
`CHAIN_ECB`	`CONSTANT INTEGER := 768;`
Block Cipher Padding Modifiers	
`PAD_PKCS5`	`CONSTANT INTEGER := 4096;`
`PAD_NONE`	`CONSTANT INTEGER := 8192;`
Block Cipher Suites	
`DES_CBC_PKCS5`	`CONSTANT INTEGER := ENCRYPT_DES + CHAIN_CBC + PAD_PKCS5;`
`DES3_CBC_PKCS5`	`CONSTANT INTEGER := ENCRYPT_3DES + CHAIN_CBC + PAD_PKCS5;`
`AES_CBC_PKCS5`	`CONSTANT INTEGER := ENCRYPT_AES + CHAIN_CBC + PAD_PKCS5;`

key

key specifies the encryption key.

iv

iv (optional) specifies an initialization vector. By default, `iv` is `NULL`.

Examples

The following example uses the `DBMS_CRYPTO.DES_CBC_PKCS5` Block Cipher Suite (a pre-defined set of algorithms and modifiers) to encrypt a value retrieved from the `passwords` table:

```
CREATE TABLE passwords
(
  principal  VARCHAR2(90) PRIMARY KEY,  -- username
  ciphertext RAW(9)                     -- encrypted password
);
CREATE PROCEDURE set_password(username VARCHAR2, cleartext RAW) AS
 typ        INTEGER := DBMS_CRYPTO.DES_CBC_PKCS5;
 key        RAW(128)  := 'my secret key';
 iv         RAW(100)  := 'my initialization vector';
 encrypted  RAW(2048);
BEGIN
  encrypted := dbms_crypto.encrypt(cleartext, typ, key, iv);
  UPDATE passwords SET ciphertext = encrypted WHERE principal = username;
END;
```

`ENCRYPT` uses a key value of `my secret key` and an initialization vector of `my initialization vector` when encrypting the `password`; specify the same key and initialization vector when decrypting the `password`.

7.2.3 HASH

The `HASH` function uses a user-specified algorithm to return the hash value of a `RAW` or `CLOB` value. The `HASH` function is available in three forms:

```
HASH
  (src IN RAW, typ IN INTEGER) RETURN RAW

HASH
  (src IN CLOB, typ IN INTEGER) RETURN RAW
```

Parameters

`src`

`src` specifies the value for which the hash value will be generated. You can specify a `RAW`, a `BLOB`, or a `CLOB` value.

`typ`

`typ` specifies the `HASH` function type. Advanced Server supports the `HASH` function types listed below:

HASH Functions	
HASH_MD4	CONSTANT INTEGER := 1;
HASH_MD5	CONSTANT INTEGER := 2;
HASH_SH1	CONSTANT INTEGER := 3;

Examples

The following example uses `DBMS_CRYPTO.HASH` to find the `md5` hash value of the string, `cleartext source`:

```
DECLARE
  typ        INTEGER := DBMS_CRYPTO.HASH_MD5;
  hash_value RAW(100);
BEGIN

  hash_value := DBMS_CRYPTO.HASH('cleartext source', typ);

END;
```

7.2.4 MAC

The `MAC` function uses a user-specified `MAC` function to return the hashed `MAC` value of a `RAW` or `CLOB` value. The `MAC` function is available in three forms:

```
MAC
  (src IN RAW, typ IN INTEGER, key IN RAW) RETURN RAW

MAC
  (src IN CLOB, typ IN INTEGER, key IN RAW) RETURN RAW
```

Parameters

src

src specifies the value for which the MAC value will be generated. Specify a RAW, BLOB, or CLOB value.

typ

typ specifies the MAC function used. Advanced Server supports the MAC functions listed below.

MAC Functions	
HMAC_MD5	CONSTANT INTEGER := 1;
HMAC_SH1	CONSTANT INTEGER := 2;

key

key specifies the key that will be used to calculate the hashed MAC value.

Examples

The following example finds the hashed MAC value of the string cleartext source:

```
DECLARE
  typ       INTEGER := DBMS_CRYPTO.HMAC_MD5;
  key       RAW(100) := 'my secret key';
  mac_value RAW(100);
BEGIN

  mac_value := DBMS_CRYPTO.MAC('cleartext source', typ, key);

END;
```

DBMS_CRYPTO.MAC uses a key value of my secret key when calculating the MAC value of cleartext source.

7.2.5 RANDOMBYTES

The RANDOMBYTES function returns a RAW value of the specified length, containing cryptographically random bytes. The signature is:

```
RANDOMBYTES
  (number_bytes IN INTEGER) RETURNS RAW
```

Parameters

number_bytes

number_bytes specifies the number of random bytes to be returned

Examples

The following example uses RANDOMBYTES to return a value that is 1024 bytes long:

```
DECLARE
  result RAW(1024);
BEGIN
  result := DBMS_CRYPTO.RANDOMBYTES(1024);
END;
```

7.2.6 RANDOMINTEGER

The RANDOMINTEGER() function returns a random INTEGER between 0 and 268,435,455. The signature is:

```
RANDOMINTEGER() RETURNS INTEGER
```

Examples

The following example uses the RANDOMINTEGER function to return a cryptographically strong random INTEGER value:

```
DECLARE
  result INTEGER;
BEGIN
  result := DBMS_CRYPTO.RANDOMINTEGER();
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

7.2.7 RANDOMNUMBER

The `RANDOMNUMBER()` function returns a random `NUMBER` between 0 and `268,435,455`. The signature is:

```
RANDOMNUMBER() RETURNS NUMBER
```

Examples

The following example uses the `RANDOMNUMBER` function to return a cryptographically strong random number:

```
DECLARE
  result NUMBER;
BEGIN
  result := DBMS_CRYPTO.RANDOMNUMBER();
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

7.3 DBMS_JOB

The DBMS_JOB package provides for the creation, scheduling, and managing of jobs. A job runs a stored procedure which has been previously stored in the database. The SUBMIT procedure is used to create and store a job definition. A job identifier is assigned to a job along with its associated stored procedure and the attributes describing when and how often the job is to be run.

This package relies on the pgAgent scheduler. By default, the Advanced Server installer installs pgAgent, but you must start the pgAgent service manually prior to using DBMS_JOB. If you attempt to use this package to schedule a job after un-installing pgAgent, DBMS_JOB will throw an error. DBMS_JOB verifies that pgAgent is installed, but does not verify that the service is running.

Table 7-2 DBMS_JOB Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
BROKEN(*job*, *broken* [, *next_date*])	Procedure	n/a	Specify that a given job is either broken or not broken.
CHANGE(*job*, *what*, *next_date*, *interval*, *instance*, *force*)	Procedure	n/a	Change the job’s parameters.
INTERVAL(*job*, *interval*)	Procedure	n/a	Set the execution frequency by means of a date function that is recalculated each time the job is run. This value becomes the next date/time for execution.
NEXT_DATE(*job*, *next_date*)	Procedure	n/a	Set the next date/time the job is to be run.
REMOVE(*job*)	Procedure	n/a	Delete the job definition from the database.
RUN(*job*)	Procedure	n/a	Forces execution of a job even if it is marked broken.
SUBMIT(*job* OUT, *what* [, *next_date* [, *interval* [, *no_parse*]]])	Procedure	n/a	Creates a job and stores its definition in the database.
WHAT(*job*, *what*)	Procedure	n/a	Change the stored procedure run by a job.

Advanced Server's implementation of DBMS_JOB is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

When and how often a job is run is dependent upon two interacting parameters – *next_date* and *interval*. The *next_date* parameter is a date/time value that specifies the next date/time when the job is to be executed. The *interval* parameter is a string that contains a date function that evaluates to a date/time value.

Just prior to any execution of the job, the expression in the *interval* parameter is evaluated. The resulting value replaces the *next_date* value stored with the job. The job is then executed. In this manner, the expression in *interval* is repeatedly re-evaluated prior to each job execution, supplying the *next_date* date/time for the next execution.

The following examples use the following stored procedure, job_proc, which simply inserts a timestamp into table, jobrun, containing a single VARCHAR2 column.

```
CREATE TABLE jobrun (
    runtime         VARCHAR2(40)
);

CREATE OR REPLACE PROCEDURE job_proc
IS
BEGIN
    INSERT INTO jobrun VALUES ('job_proc run at ' || TO_CHAR(SYSDATE,
        'yyyy-mm-dd hh24:mi:ss'));
END;
```

7.3.1 BROKEN

The BROKEN procedure sets the state of a job to either broken or not broken. A broken job cannot be executed except by using the RUN procedure.

BROKEN(*job* BINARY_INTEGER, *broken* BOOLEAN [, *next_date* DATE])

Parameters

job

Identifier of the job to be set as broken or not broken.

broken

If set to TRUE the job's state is set to broken. If set to FALSE the job's state is set to not broken. Broken jobs cannot be run except by using the RUN procedure.

next_date

Date/time when the job is to be run. The default is SYSDATE.

Examples

Set the state of a job with job identifier 104 to broken:

```
BEGIN
    DBMS_JOB.BROKEN(104,true);
END;
```

Change the state back to not broken:

```
BEGIN
    DBMS_JOB.BROKEN(104,false);
END;
```

7.3.2 CHANGE

The `CHANGE` procedure modifies certain job attributes including the stored procedure to be run, the next date/time the job is to be run, and how often it is to be run.

```
CHANGE(job BINARY_INTEGER what VARCHAR2, next_date DATE,
  interval VARCHAR2, instance BINARY_INTEGER, force BOOLEAN)
```

Parameters

job

Identifier of the job to modify.

what

Stored procedure name. Set this parameter to null if the existing value is to remain unchanged.

next_date

Date/time when the job is to be run next. Set this parameter to null if the existing value is to remain unchanged.

interval

Date function that when evaluated, provides the next date/time the job is to run. Set this parameter to null if the existing value is to remain unchanged.

instance

This argument is ignored, but is included for compatibility.

force

This argument is ignored, but is included for compatibility.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

Examples

Change the job to run next on December 13, 2007. Leave other parameters unchanged.

```
BEGIN
    DBMS_JOB.CHANGE(104,NULL,TO_DATE('13-DEC-07','DD-MON-YY'),NULL, NULL,
    NULL);
END;
```

7.3.3 INTERVAL

The `INTERVAL` procedure sets the frequency of how often a job is to be run.

INTERVAL(*job* BINARY_INTEGER, *interval* VARCHAR2)

Parameters

job

Identifier of the job to modify.

interval

Date function that when evaluated, provides the next date/time the job is to be run.

Examples

Change the job to run once a week:

```
BEGIN
    DBMS_JOB.INTERVAL(104,'SYSDATE + 7');
END;
```

7.3.4 NEXT_DATE

The `NEXT_DATE` procedure sets the date/time of when the job is to be run next.

NEXT_DATE(*job* BINARY_INTEGER, *next_date* DATE)

Parameters

job

Identifier of the job whose next run date is to be set.

next_date

Date/time when the job is to be run next.

Examples

Change the job to run next on December 14, 2007:

```
BEGIN
    DBMS_JOB.NEXT_DATE(104, TO_DATE('14-DEC-07','DD-MON-YY'));
END;
```

7.3.5 REMOVE

The `REMOVE` procedure deletes the specified job from the database. The job must be resubmitted using the `SUBMIT` procedure in order to have it executed again. Note that the stored procedure that was associated with the job is not deleted.

REMOVE(*job* BINARY_INTEGER)

Parameters

job

Identifier of the job that is to be removed from the database.

Examples

Remove a job from the database:

```
BEGIN
    DBMS_JOB.REMOVE(104);
END;
```

7.3.6 RUN

The `RUN` procedure forces the job to be run, even if its state is broken.

RUN(*job* BINARY_INTEGER)

Parameters

job

Identifier of the job to be run.

Examples

Force a job to be run.

```
BEGIN
    DBMS_JOB.RUN(104);
END;
```

7.3.7 SUBMIT

The `SUBMIT` procedure creates a job definition and stores it in the database. A job consists of a job identifier, the stored procedure to be executed, when the job is to be first run, and a date function that calculates the next date/time the job is to be run.

```
SUBMIT(job OUT BINARY_INTEGER, what VARCHAR2
  [, next_date DATE [, interval VARCHAR2 [, no_parse BOOLEAN ]]])
```

Parameters

job

Identifier assigned to the job.

what

Name of the stored procedure to be executed by the job.

next_date

Date/time when the job is to be run next. The default is `SYSDATE`.

interval

Date function that when evaluated, provides the next date/time the job is to run. If *interval* is set to null, then the job is run only once. Null is the default.

no_parse

If set to `TRUE`, do not syntax-check the stored procedure upon job creation – check only when the job first executes. If set to `FALSE`, check the procedure upon job creation. The default is `FALSE`.

Note: The *no_parse* option is not supported in this implementation of `SUBMIT()`. It is included for compatibility only.

Examples

The following example creates a job using stored procedure, `job_proc`. The job will execute immediately and run once a day thereafter as set by the *`interval`* parameter, `SYSDATE + 1`.

```
DECLARE
    jobid           INTEGER;
BEGIN
    DBMS_JOB.SUBMIT(jobid,'job_proc;',SYSDATE,
        'SYSDATE + 1');
    DBMS_OUTPUT.PUT_LINE('jobid: ' || jobid);
END;

jobid: 104
```

The job immediately executes procedure, `job_proc`, populating table, `jobrun`, with a row:

```
SELECT * FROM jobrun;

               runtime
-------------------------------------
 job_proc run at 2007-12-11 11:43:25
(1 row)
```

7.3.8 WHAT

The `WHAT` procedure changes the stored procedure that the job will execute.

`WHAT(`*`job`*` BINARY_INTEGER, `*`what`*` VARCHAR2)`

Parameters

`job`

Identifier of the job for which the stored procedure is to be changed.

`what`

Name of the stored procedure to be executed.

Examples

Change the job to run the `list_emp` procedure:

```
BEGIN
    DBMS_JOB.WHAT(104,'list_emp;');
END;
```

7.4 DBMS_LOB

The `DBMS_LOB` package provides the capability to operate on large objects.

Table 7-3 DBMS_LOB Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
`APPEND(`*`dest_lob`*` IN OUT, `*`src_lob`*`)`	Procedure	n/a	Appends one large object to another.
`COMPARE(`*`lob_1`*`, `*`lob_2`*` [, `*`amount`*` [, `*`offset_1`*` [, `*`offset_2`*` ]]])`	Function	`INTEGER`	Compares two large objects.
`CONVERTOBLOB(`*`dest_lob`*` IN OUT, `*`src_clob`*`, `*`amount`*`, `*`dest_offset`*` IN OUT, `*`src_offset`*` IN OUT, `*`blob_csid`*`, `*`lang_context`*` IN OUT, `*`warning`*` OUT)`	Procedure	n/a	Converts character data to binary.
`CONVERTTOCLOB(`*`dest_lob`*` IN OUT, `*`src_blob`*`, `*`amount`*`, `*`dest_offset`*` IN OUT, `*`src_offset`*` IN OUT, `*`blob_csid`*`, `*`lang_context`*` IN OUT, `*`warning`*` OUT)`	Procedure	n/a	Converts binary data to character.
`COPY(`*`dest_lob`*` IN OUT, `*`src_lob`*`, `*`amount`*` [, `*`dest_offset`*` [, `*`src_offset`*` ]])`	Procedure	n/a	Copies one large object to another.
`ERASE(lob_loc IN OUT, `*`amount`*` IN OUT [, `*`offset`*` ])`	Procedure	n/a	Erase a large object.
`GET_STORAGE_LIMIT(`*`lob_loc`*`)`	Function	`INTEGER`	Get the storage limit for large objects.
`GETLENGTH(`*`lob_loc`*`)`	Function	`INTEGER`	Get the length of the large object.
`INSTR(`*`lob_loc`*`, `*`pattern`*` [, `*`offset`*` [, `*`nth`*` ]])`	Function	`INTEGER`	Get the position of the nth occurrence of a pattern in the large object starting at *`offset`*.
`READ(`*`lob_loc`*`, `*`amount`*` IN OUT, `*`offset`*`, `*`buffer`*` OUT)`	Procedure	n/a	Read a large object.
`SUBSTR(`*`lob_loc`*` [, `*`amount`*` [, `*`offset`*` ]])`	Function	`RAW`, `VARCHAR2`	Get part of a large object.
`TRIM(`*`lob_loc`*` IN OUT, `*`newlen`*`)`	Procedure	n/a	Trim a large object to the specified length.
`WRITE(`*`lob_loc`*` IN OUT, `*`amount`*`, `*`offset`*`, `*`buffer`*`)`	Procedure	n/a	Write data to a large object.
`WRITEAPPEND(`*`lob_loc`*` IN OUT, `*`amount`*`, `*`buffer`*`)`	Procedure	n/a	Write data from the buffer to the end of a large object.

Advanced Server's implementation of `DBMS_LOB` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

The following table lists the public variables available in the package.

Table 7-4 DBMS_LOB Public Variables

Public Variables	Data Type	Value
compress off	INTEGER	0
compress_on	INTEGER	1
deduplicate_off	INTEGER	0
deduplicate_on	INTEGER	4
default_csid	INTEGER	0
default_lang_ctx	INTEGER	0
encrypt_off	INTEGER	0
encrypt_on	INTEGER	1
file_readonly	INTEGER	0
lobmaxsize	INTEGER	1073741823
lob_readonly	INTEGER	0
lob_readwrite	INTEGER	1
no_warning	INTEGER	0
opt_compress	INTEGER	1
opt_deduplicate	INTEGER	4
opt_encrypt	INTEGER	2
warn_inconvertible_char	INTEGER	1

In the following sections, lengths and offsets are measured in bytes if the large objects are `BLOB`s. Lengths and offsets are measured in characters if the large objects are `CLOB`s.

7.4.1 APPEND

The `APPEND` procedure provides the capability to append one large object to another. Both large objects must be of the same type.

```
APPEND(dest_lob IN OUT { BLOB | CLOB }, src_lob { BLOB | CLOB })
```

Parameters

dest_lob

> Large object locator for the destination object. Must be the same data type as *src_lob*.

src_lob

> Large object locator for the source object. Must be the same data type as *dest_lob*.

7.4.2 COMPARE

The `COMPARE` procedure performs an exact byte-by-byte comparison of two large objects for a given length at given offsets. The large objects being compared must be the same data type.

```
status INTEGER COMPARE(lob_1 { BLOB | CLOB },
  lob_2 { BLOB | CLOB }
  [, amount INTEGER [, offset_1 INTEGER [, offset_2 INTEGER ]]])
```

Parameters

lob_1

Large object locator of the first large object to be compared. Must be the same data type as *lob_2*.

lob_2

Large object locator of the second large object to be compared. Must be the same data type as *lob_1*.

amount

If the data type of the large objects is `BLOB`, then the comparison is made for *amount* bytes. If the data type of the large objects is `CLOB`, then the comparison is made for *amount* characters. The default it the maximum size of a large object.

offset_1

Position within the first large object to begin the comparison. The first byte/character is offset 1. The default is 1.

offset_2

Position within the second large object to begin the comparison. The first byte/character is offset 1. The default is 1.

status

Zero if both large objects are exactly the same for the specified length for the specified offsets. Non-zero, if the objects are not the same. *NULL* if *amount*, *offset_1*, or *offset_2* are less than zero.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

7.4.3 CONVERTTOBLOB

The `CONVERTTOBLOB` procedure provides the capability to convert character data to binary.

```
CONVERTTOBLOB(dest_lob IN OUT BLOB, src_clob CLOB,
  amount INTEGER, dest_offset IN OUT INTEGER,
  src_offset IN OUT INTEGER, blob_csid NUMBER,
  lang_context IN OUT INTEGER, warning OUT INTEGER)
```

Parameters

dest_lob

`BLOB` large object locator to which the character data is to be converted.

src_clob

`CLOB` large object locator of the character data to be converted.

amount

Number of characters of *src_clob* to be converted.

dest_offset IN

Position in bytes in the destination `BLOB` where writing of the source `CLOB` should begin. The first byte is offset 1.

dest_offset OUT

Position in bytes in the destination `BLOB` after the write operation completes. The first byte is offset 1.

src_offset IN

Position in characters in the source `CLOB` where conversion to the destination `BLOB` should begin. The first character is offset 1.

src_offset OUT

Position in characters in the source `CLOB` after the conversion operation completes. The first character is offset 1.

blob_csid

Character set ID of the converted, destination `BLOB`.

lang_context IN

Language context for the conversion. The default value of 0 is typically used for this setting.

lang_context OUT

Language context after the conversion completes.

warning

0 if the conversion was successful, 1 if an inconvertible character was encountered.

7.4.4 CONVERTTOCLOB

The CONVERTTOCLOB procedure provides the capability to convert binary data to character.

```
CONVERTTOCLOB(dest_lob IN OUT CLOB, src_blob BLOB,
  amount INTEGER, dest_offset IN OUT INTEGER,
  src_offset IN OUT INTEGER, blob_csid NUMBER,
  lang_context IN OUT INTEGER, warning OUT INTEGER)
```

Parameters

dest_lob

CLOB large object locator to which the binary data is to be converted.

src_blob

BLOB large object locator of the binary data to be converted.

amount

Number of bytes of *src_blob* to be converted.

dest_offset IN

Position in characters in the destination CLOB where writing of the source BLOB should begin. The first character is offset 1.

dest_offset OUT

Position in characters in the destination `CLOB` after the write operation completes. The first character is offset 1.

src_offset IN

Position in bytes in the source `BLOB` where conversion to the destination `CLOB` should begin. The first byte is offset 1.

src_offset OUT

Position in bytes in the source `BLOB` after the conversion operation completes. The first byte is offset 1.

blob_csid

Character set ID of the converted, destination `CLOB`.

lang_context IN

Language context for the conversion. The default value of 0 is typically used for this setting.

lang_context OUT

Language context after the conversion completes.

warning

0 if the conversion was successful, 1 if an inconvertible character was encountered.

7.4.5 COPY

The `COPY` procedure provides the capability to copy one large object to another. The source and destination large objects must be the same data type.

```
COPY(dest_lob IN OUT { BLOB | CLOB }, src_lob
{ BLOB | CLOB },
  amount INTEGER
  [, dest_offset INTEGER [, src_offset INTEGER ]])
```

Parameters

dest_lob

Large object locator of the large object to which *src_lob* is to be copied. Must be the same data type as *src_lob*.

src_lob

Large object locator of the large object to be copied to *dest_lob*. Must be the same data type as *dest_lob*.

amount

Number of bytes/characters of *src_lob* to be copied.

dest_offset

Position in the destination large object where writing of the source large object should begin. The first position is offset 1. The default is 1.

src_offset

Position in the source large object where copying to the destination large object should begin. The first position is offset 1. The default is 1.

7.4.6 ERASE

The `ERASE` procedure provides the capability to erase a portion of a large object. To erase a large object means to replace the specified portion with zero-byte fillers for `BLOB`s or with spaces for `CLOB`s. The actual size of the large object is not altered.

```
ERASE(lob_loc IN OUT { BLOB | CLOB }, amount IN OUT INTEGER
  [, offset INTEGER ])
```

Parameters

lob_loc

Large object locator of the large object to be erased.

amount IN

Number of bytes/characters to be erased.

amount OUT

Number of bytes/characters actually erased. This value can be smaller than the input value if the end of the large object is reached before *amount* bytes/characters have been erased.

offset

Position in the large object where erasing is to begin. The first byte/character is position 1. The default is 1.

7.4.7 GET_STORAGE_LIMIT

The `GET_STORAGE_LIMIT` function returns the limit on the largest allowable large object.

```
size INTEGER GET_STORAGE_LIMIT(lob_loc BLOB)

size INTEGER GET_STORAGE_LIMIT(lob_loc CLOB)
```

Parameters

`size`

Maximum allowable size of a large object in this database.

`lob_loc`

This parameter is ignored, but is included for compatibility.

7.4.8 GETLENGTH

The `GETLENGTH` function returns the length of a large object.

```
amount INTEGER GETLENGTH(lob_loc BLOB)

amount INTEGER GETLENGTH(lob_loc CLOB)
```

Parameters

`lob_loc`

Large object locator of the large object whose length is to be obtained.

`amount`

Length of the large object in bytes for `BLOB`s or characters for `CLOB`s.

7.4.9 INSTR

The `INSTR` function returns the location of the nth occurrence of a given pattern within a large object.

```
position INTEGER INSTR(lob_loc { BLOB | CLOB },
  pattern { RAW | VARCHAR2 } [, offset INTEGER [, nth
INTEGER ]])
```

Parameters

lob_loc

Large object locator of the large object in which to search for pattern.

pattern

Pattern of bytes or characters to match against the large object, `lob`. *pattern* must be `RAW` if *lob_loc* is a `BLOB`. pattern must be `VARCHAR2` if *lob_loc* is a `CLOB`.

offset

Position within *lob_loc* to start search for *pattern*. The first byte/character is position 1. The default is 1.

nth

Search for *pattern*, *nth* number of times starting at the position given by *offset*. The default is 1.

position

Position within the large object where *pattern* appears the nth time specified by *nth* starting from the position given by *offset*.

7.4.10 READ

The READ procedure provides the capability to read a portion of a large object into a buffer.

```
    READ(lob_loc { BLOB | CLOB }, amount IN OUT BINARY_INTEGER,
      offset INTEGER, buffer OUT { RAW | VARCHAR2 })
```

Parameters

lob_loc

Large object locator of the large object to be read.

amount IN

Number of bytes/characters to read.

amount OUT

Number of bytes/characters actually read. If there is no more data to be read, then *amount* returns 0 and a DATA_NOT_FOUND exception is thrown.

offset

Position to begin reading. The first byte/character is position 1.

buffer

Variable to receive the large object. If *lob_loc* is a BLOB, then *buffer* must be RAW. If *lob_loc* is a CLOB, then *buffer* must be VARCHAR2.

7.4.11 SUBSTR

The `SUBSTR` function provides the capability to return a portion of a large object.

```
data { RAW | VARCHAR2 } SUBSTR(lob_loc { BLOB | CLOB }
  [, amount INTEGER [, offset INTEGER ]])
```

Parameters

lob_loc

Large object locator of the large object to be read.

amount

Number of bytes/characters to be returned. Default is 32,767.

offset

Position within the large object to begin returning data. The first byte/character is position 1. The default is 1.

data

Returned portion of the large object to be read. If *lob_loc* is a `BLOB`, the return data type is `RAW`. If *lob_loc* is a `CLOB`, the return data type is `VARCHAR2`.

7.4.12 TRIM

The `TRIM` procedure provides the capability to truncate a large object to the specified length.

```
TRIM(lob_loc IN OUT { BLOB | CLOB }, newlen INTEGER)
```

Parameters

lob_loc

Large object locator of the large object to be trimmed.

newlen

Number of bytes/characters to which the large object is to be trimmed.

7.4.13 WRITE

The `WRITE` procedure provides the capability to write data into a large object. Any existing data in the large object at the specified offset for the given length is overwritten by data given in the buffer.

```
WRITE(lob_loc IN OUT { BLOB | CLOB },
  amount BINARY_INTEGER,
  offset INTEGER, buffer { RAW | VARCHAR2 })
```

Parameters

lob_loc

Large object locator of the large object to be written.

amount

The number of bytes/characters in *buffer* to be written to the large object.

offset

The offset in bytes/characters from the beginning of the large object (origin is 1) for the write operation to begin.

buffer

Contains data to be written to the large object. If *lob_loc* is a `BLOB`, then *buffer* must be `RAW`. If *lob_loc* is a `CLOB`, then *buffer* must be `VARCHAR2`.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

7.4.14 WRITEAPPEND

The `WRITEAPPEND` procedure provides the capability to add data to the end of a large object.

```
WRITEAPPEND(lob_loc IN OUT { BLOB | CLOB },
  amount BINARY_INTEGER, buffer { RAW | VARCHAR2 })
```

Parameters

lob_loc

Large object locator of the large object to which data is to be appended.

amount

Number of bytes/characters from *buffer* to be appended the large object.

buffer

Data to be appended to the large object. If *lob_loc* is a `BLOB`, then *buffer* must be `RAW`. If *lob_loc* is a `CLOB`, then *buffer* must be `VARCHAR2`.

7.5 DBMS_LOCK

Advanced Server provides support for the `DBMS_LOCK.SLEEP` procedure.

Table 7.7.2 DBMS_LOCK Procedure

Function/Procedure	Return Type	Description
`SLEEP(`*`seconds`*`)`	n/a	Suspends a session for the specified number of *`seconds`*.

Advanced Server's implementation of `DBMS_LOCK` is a partial implementation when compared to Oracle's version. Only `DBMS_LOCK.SLEEP` is supported.

7.5.1 SLEEP

The `SLEEP` procedure suspends the current session for the specified number of seconds.

```
SLEEP(seconds NUMBER)
```

Parameters

`seconds`

> `seconds` specifies the number of seconds for which you wish to suspend the session. *`seconds`* can be a fractional value; for example, enter `1.75` to specify one and three-fourths of a second.

7.6 DBMS_MVIEW

Use procedures in the `DBMS_MVIEW` package to manage and refresh materialized views and their dependencies. Advanced Server provides support for the following `DBMS_MVIEW` procedures:

Table 7.7.2 DBMS_MVIEW Procedures

Procedure	Return Type	Description
`GET_MV_DEPENDENCIES(`*`list`*` VARCHAR2, `*`deplist`*` VARCHAR2);`	n/a	The `GET_MV_DEPENDENCIES` procedure returns a list of dependencies for a specified view.
`REFRESH(`*`list`*` VARCHAR2, `*`method`*` VARCHAR2, `*`rollback_seg`*` VARCHAR2 , `*`push_deferred_rpc`*` BOOLEAN, `*`refresh_after_errors`*` BOOLEAN , `*`purge_option`*` NUMBER, `*`parallelism`*` NUMBER, `*`heap_size`*` NUMBER , `*`atomic_refresh`*` BOOLEAN , `*`nested`*` BOOLEAN);`	n/a	This variation of the `REFRESH` procedure refreshes all views named in a comma-separated list of view names.
`REFRESH(`*`tab`*` dbms_utility.uncl_array, `*`method`*` VARCHAR2, `*`rollback_seg`*` VARCHAR2, `*`push_deferred_rpc`*` BOOLEAN, `*`refresh_after_errors`*` BOOLEAN, `*`purge_option`*` NUMBER, `*`parallelism`*` NUMBER, `*`heap_size`*` NUMBER, `*`atomic_refresh`*` BOOLEAN, `*`nested`*` BOOLEAN);`	n/a	This variation of the `REFRESH` procedure refreshes all views named in a table of `dbms_utility.uncl_array` values.
`REFRESH_ALL_MVIEWS(`*`number_of_failures`*` BINARY_INTEGER, `*`method`*` VARCHAR2, `*`rollback_seg`*` VARCHAR2, `*`refresh_after_errors`*` BOOLEAN, `*`atomic_refresh`*` BOOLEAN);`	n/a	The `REFRESH_ALL_MVIEWS` procedure refreshes all materialized views.
`REFRESH_DEPENDENT(`*`number_of_failures`*` BINARY_INTEGER, `*`list`*` VARCHAR2, `*`method`*` VARCHAR2, `*`rollback_seg`*` VARCHAR2, `*`refresh_after_errors`*` BOOLEAN, `*`atomic_refresh`*` BOOLEAN, `*`nested`*` BOOLEAN);`	n/a	This variation of the `REFRESH_DEPENDENT` procedure refreshes all views that are dependent on the views listed in a comma-separated list.
`REFRESH_DEPENDENT(`*`number_of_failures`*` BINARY_INTEGER, `*`tab`*` dbms_utility.uncl_array, `*`method`*` VARCHAR2, `*`rollback_seg`*` VARCHAR2, `*`refresh_after_errors`*` BOOLEAN, `*`atomic_refresh`*` BOOLEAN, `*`nested`*` BOOLEAN);`	n/a	This variation of the `REFRESH_DEPENDENT` procedure refreshes all views that are dependent on the views listed in a table of `dbms_utility.uncl_array` values.

Advanced Server's implementation of `DBMS_MVIEW` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

7.6.1 GET_MV_DEPENDENCIES

When given the name of a materialized view, `GET_MV_DEPENDENCIES` returns a list of items that depend on the specified view. The signature is:

```
GET_MV_DEPENDENCIES(
  list IN VARCHAR2,
  deplist OUT VARCHAR2);
```

Parameters

`list`

> *`list`* specifies the name of a materialized view, or a comma-separated list of materialized view names.

`deplist`

> *`deplist`* is a comma-separated list of schema-qualified dependencies. *`deplist`* is a `VARCHAR2` value.

Examples

The following example:

```
DECLARE
  deplist VARCHAR2(1000);
BEGIN
  DBMS_MVIEW.GET_MV_DEPENDENCIES('public.emp_view', deplist);
  DBMS_OUTPUT.PUT_LINE('deplist: ' || deplist);
END;
```

Displays a list of the dependencies on a materialized view named `public.emp_view`.

7.6.2 REFRESH

Use the `REFRESH` procedure to refresh all views specified in either a comma-separated list of view names, or a table of `DBMS_UTILITY.UNCL_ARRAY` values. The procedure has two signatures; use the first form when specifying a comma-separated list of view names:

```
REFRESH(
  list IN VARCHAR2,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  push_deferred_rpc IN BOOLEAN DEFAULT TRUE,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  purge_option IN NUMBER DEFAULT 1,
  parallelism IN NUMBER DEFAULT 0,
  heap_size IN NUMBER DEFAULT 0,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

Use the second form to specify view names in a table of `DBMS_UTILITY.UNCL_ARRAY` values:

```
REFRESH(
  tab IN OUT DBMS_UTILITY.UNCL_ARRAY,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  push_deferred_rpc IN BOOLEAN DEFAULT TRUE,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  purge_option IN NUMBER DEFAULT 1,
  parallelism IN NUMBER DEFAULT 0,
  heap_size IN NUMBER DEFAULT 0,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

Parameters

`list`

`list` is a `VARCHAR2` value that specifies the name of a materialized view, or a comma-separated list of materialized view names. The names may be schema-qualified.

`tab`

`tab` is a table of `DBMS_UTILITY.UNCL_ARRAY` values that specify the name (or names) of a materialized view.

method

> method is a VARCHAR2 value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is C; this performs a complete refresh of the view.

rollback_seg

> *rollback_seg* is accepted for compatibility and ignored. The default is NULL.

push_deferred_rpc

> *push_deferred_rpc* is accepted for compatibility and ignored. The default is TRUE.

refresh_after_errors

> *refresh_after_errors* is accepted for compatibility and ignored. The default is FALSE.

purge_option

> *purge_option* is accepted for compatibility and ignored. The default is 1.

parallelism

> *parallelism* is accepted for compatibility and ignored. The default is 0.

heap_size IN NUMBER DEFAULT 0,

> *heap_size* is accepted for compatibility and ignored. The default is 0.

atomic_refresh

> *atomic_refresh* is accepted for compatibility and ignored. The default is TRUE.

nested

> *nested* is accepted for compatibility and ignored. The default is FALSE.

Examples

The following example uses DBMS_MVIEW.REFRESH to perform a COMPLETE refresh on the public.emp_view materialized view:

```
EXEC DBMS_MVIEW.REFRESH(list => 'public.emp_view', method => 'C');
```

7.6.3 REFRESH_ALL_MVIEWS

Use the `REFRESH_ALL_MVIEWS` procedure to refresh any materialized views that have not been refreshed since the table or view on which the view depends has been modified. The signature is:

```
REFRESH_ALL_MVIEWS(
  number_of_failures OUT BINARY_INTEGER,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  atomic_refresh IN BOOLEAN DEFAULT TRUE);
```

Parameters

number_of_failures

number_of_failures is a `BINARY_INTEGER` that specifies the number of failures that occurred during the refresh operation.

method

`method` is a `VARCHAR2` value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is `C`; this performs a complete refresh of the view.

rollback_seg

rollback_seg is accepted for compatibility and ignored. The default is `NULL`.

refresh_after_errors

refresh_after_errors is accepted for compatibility and ignored. The default is `FALSE`.

atomic_refresh

atomic_refresh is accepted for compatibility and ignored. The default is `TRUE`.

Examples

The following example performs a `COMPLETE` refresh on all materialized views:

```
DECLARE
  errors INTEGER;
BEGIN
```

```
  DBMS_MVIEW.REFRESH_ALL_MVIEWS(errors, method => 'C');
END;
```

Upon completion, `errors` contains the number of failures.

7.6.4 REFRESH_DEPENDENT

Use the `REFRESH_DEPENDENT` procedure to refresh all material views that are dependent on the views specified in the call to the procedure. You can specify a comma-separated list or provide the view names in a table of `DBMS_UTILITY.UNCL_ARRAY` values.

Use the first form of the procedure to refresh all material views that are dependent on the views specified in a comma-separated list:

```
REFRESH_DEPENDENT(
  number_of_failures OUT BINARY_INTEGER,
  list IN VARCHAR2,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

Use the second form of the procedure to refresh all material views that are dependent on the views specified in a table of `DBMS_UTILITY.UNCL_ARRAY` values:

```
REFRESH_DEPENDENT(
  number_of_failures OUT BINARY_INTEGER,
  tab IN DBMS_UTILITY.UNCL_ARRAY,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

Parameters

`number_of_failures`

`number_of_failures` is a `BINARY_INTEGER` that contains the number of failures that occurred during the refresh operation.

list

> *list* is a VARCHAR2 value that specifies the name of a materialized view, or a comma-separated list of materialized view names. The names may be schema-qualified.

tab

> *tab* is a table of DBMS_UTILITY.UNCL_ARRAY values that specify the name (or names) of a materialized view.

method

> method is a VARCHAR2 value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is C; this performs a complete refresh of the view.

rollback_seg

> *rollback_seg* is accepted for compatibility and ignored. The default is NULL.

refresh_after_errors

> *refresh_after_errors* is accepted for compatibility and ignored. The default is FALSE.

atomic_refresh

> *atomic_refresh* is accepted for compatibility and ignored. The default is TRUE.

nested

> *nested* is accepted for compatibility and ignored. The default is FALSE.

Examples

The following example performs a COMPLETE refresh on all materialized views dependent on a materialized view named emp_view that resides in the public schema:

```
DECLARE
  errors INTEGER;
BEGIN
  DBMS_MVIEW.REFRESH_DEPENDENT(errors, list => 'public.emp_view', method =>
'C');
END;
```

Upon completion, errors contains the number of failures.

7.7 DBMS_OUTPUT

The `DBMS_OUTPUT` package provides the capability to send messages (lines of text) to a message buffer, or get messages from the message buffer. A message buffer is local to a single session. Use the `DBMS_PIPE` package to send messages between sessions.

The procedures and functions available in the `DBMS_OUTPUT` package are listed in the following table.

Table 7-7-5 DBMS_OUTPUT Functions/Procedures

Function/Procedure	Return Type	Description
`DISABLE`	n/a	Disable the capability to send and receive messages.
`ENABLE(`*`buffer_size`*`)`	n/a	Enable the capability to send and receive messages.
`GET_LINE(`*`line`*` OUT, `*`status`*` OUT)`	n/a	Get a line from the message buffer.
`GET_LINES(`*`lines`*` OUT, `*`numlines`*` IN OUT)`	n/a	Get multiple lines from the message buffer.
`NEW_LINE`	n/a	Puts an end-of-line character sequence.
`PUT(`*`item`*`)`	n/a	Puts a partial line without an end-of-line character sequence.
`PUT_LINE(`*`item`*`)`	n/a	Puts a complete line with an end-of-line character sequence.
`SERVEROUTPUT(`*`stdout`*`)`	n/a	Direct messages from `PUT`, `PUT_LINE`, or `NEW_LINE` to either standard output or the message buffer.

The following table lists the public variables available in the `DBMS_OUTPUT` package.

Table 7-7-6 DBMS_OUTPUT Public Variables

Public Variables	Data Type	Value	Description
`chararr`	`TABLE`		For message lines.

7.7.1 CHARARR

The `CHARARR` is for storing multiple message lines.

```
TYPE chararr IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

7.7.2 DISABLE

The DISABLE procedure clears out the message buffer. Any messages in the buffer at the time the DISABLE procedure is executed will no longer be accessible. Any messages subsequently sent with the PUT, PUT_LINE, or NEW_LINE procedures are discarded. No error is returned to the sender when the PUT, PUT_LINE, or NEW_LINE procedures are executed and messages have been disabled.

Use the ENABLE procedure or SERVEROUTPUT(TRUE) procedure to re-enable the sending and receiving of messages.

```
DISABLE
```

Examples

This anonymous block disables the sending and receiving messages in the current session.

```
BEGIN
    DBMS_OUTPUT.DISABLE;
END;
```

7.7.3 ENABLE

The ENABLE procedure enables the capability to send messages to the message buffer or retrieve messages from the message buffer. Running SERVEROUTPUT(TRUE) also implicitly performs the ENABLE procedure.

The destination of a message sent with PUT, PUT_LINE, or NEW_LINE depends upon the state of SERVEROUTPUT.

- If the last state of SERVEROUTPUT is TRUE, the message goes to standard output of the command line.
- If the last state of SERVEROUTPUT is FALSE, the message goes to the message buffer.

```
ENABLE [ (buffer_size INTEGER) ]
```

Parameters

buffer_size

Maximum length of the message buffer in bytes. If a *buffer_size* of less than 2000 is specified, the buffer size is set to 2000.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

Examples

The following anonymous block enables messages. Setting `SERVEROUTPUT(TRUE)` forces them to standard output.

```
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Messages enabled');
END;

Messages enabled
```

The same effect could have been achieved by simply using `SERVEROUTPUT(TRUE)`.

```
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Messages enabled');
END;

Messages enabled
```

The following anonymous block enables messages, but setting `SERVEROUTPUT(FALSE)` directs messages to the message buffer.

```
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    DBMS_OUTPUT.PUT_LINE('Message sent to buffer');
END;
```

7.7.4 GET_LINE

The `GET_LINE` procedure provides the capability to retrieve a line of text from the message buffer. Only text that has been terminated by an end-of-line character sequence is retrieved – that is complete lines generated using `PUT_LINE`, or by a series of `PUT` calls followed by a `NEW_LINE` call.

```
GET_LINE(line OUT VARCHAR2, status OUT INTEGER)
```

Parameters

`line`

Variable receiving the line of text from the message buffer.

`status`

0 if a line was returned from the message buffer, 1 if there was no line to return.

Examples

The following anonymous block writes the emp table out to the message buffer as a comma-delimited string for each row.

```
EXEC DBMS_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE
    v_emprec        VARCHAR2(120);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    DBMS_OUTPUT.ENABLE;
    FOR i IN emp_cur LOOP
        v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||
            NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
            ',' || i.sal || ',' ||
            NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
        DBMS_OUTPUT.PUT_LINE(v_emprec);
    END LOOP;
END;
```

The following anonymous block reads the message buffer and inserts the messages written by the prior example into a table named messages. The rows in messages are then displayed.

```
CREATE TABLE messages (
    status          INTEGER,
    msg             VARCHAR2(100)
);

DECLARE
    v_line          VARCHAR2(100);
    v_status        INTEGER := 0;
BEGIN
    DBMS_OUTPUT.GET_LINE(v_line,v_status);
    WHILE v_status = 0 LOOP
        INSERT INTO messages VALUES(v_status, v_line);
        DBMS_OUTPUT.GET_LINE(v_line,v_status);
    END LOOP;
END;

SELECT msg FROM messages;

                               msg
-----------------------------------------------------------------
 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)
```

7.7.5 GET_LINES

The GET_LINES procedure provides the capability to retrieve one or more lines of text from the message buffer into a collection. Only text that has been terminated by an end-of-line character sequence is retrieved – that is complete lines generated using PUT_LINE, or by a series of PUT calls followed by a NEW_LINE call.

```
GET_LINES(lines OUT CHARARR, numlines IN OUT INTEGER)
```

Parameters

lines

Table receiving the lines of text from the message buffer. See CHARARR for a description of *lines*.

numlines IN

Number of lines to be retrieved from the message buffer.

numlines OUT

Actual number of lines retrieved from the message buffer. If the output value of *numlines* is less than the input value, then there are no more lines left in the message buffer.

Examples

The following example uses the GET_LINES procedure to store all rows from the emp table that were placed on the message buffer, into an array.

```
EXEC DBMS_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE
    v_emprec        VARCHAR2(120);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    DBMS_OUTPUT.ENABLE;
    FOR i IN emp_cur LOOP
        v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||
            NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
            ',' || i.sal || ',' ||
            NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
        DBMS_OUTPUT.PUT_LINE(v_emprec);
    END LOOP;
END;

DECLARE
```

```
    v_lines         DBMS_OUTPUT.CHARARR;
    v_numlines      INTEGER := 14;
    v_status        INTEGER := 0;
BEGIN
    DBMS_OUTPUT.GET_LINES(v_lines,v_numlines);
    FOR i IN 1..v_numlines LOOP
        INSERT INTO messages VALUES(v_numlines, v_lines(i));
    END LOOP;
END;

SELECT msg FROM messages;

                               msg
-----------------------------------------------------------------
 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)
```

7.7.6 NEW_LINE

The `NEW_LINE` procedure writes an end-of-line character sequence in the message buffer.

```
NEW_LINE
```

Parameters

The `NEW_LINE` procedure expects no parameters.

7.7.7 PUT

The `PUT` procedure writes a string to the message buffer. No end-of-line character sequence is written at the end of the string. Use the `NEW_LINE` procedure to add an end-of-line character sequence.

PUT(*item* VARCHAR2)

Parameters

item

Text written to the message buffer.

Examples

The following example uses the `PUT` procedure to display a comma-delimited list of employees from the `emp` table.

```
DECLARE
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    FOR i IN emp_cur LOOP
        DBMS_OUTPUT.PUT(i.empno);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.ename);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.job);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.mgr);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.hiredate);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.sal);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.comm);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.deptno);
        DBMS_OUTPUT.NEW_LINE;
    END LOOP;
END;

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
```

7.7.8 PUT_LINE

The `PUT_LINE` procedure writes a single line to the message buffer including an end-of-line character sequence.

```
PUT_LINE(item VARCHAR2)
```

Parameters

item

Text to be written to the message buffer.

Examples

The following example uses the `PUT_LINE` procedure to display a comma-delimited list of employees from the `emp` table.

```
DECLARE
    v_emprec        VARCHAR2(120);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    FOR i IN emp_cur LOOP
        v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||
            NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
            ',' || i.sal || ',' ||
            NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
        DBMS_OUTPUT.PUT_LINE(v_emprec);
    END LOOP;
END;

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
```

7.7.9 SERVEROUTPUT

The SERVEROUTPUT procedure provides the capability to direct messages to standard output of the command line or to the message buffer. Setting SERVEROUTPUT(TRUE) also performs an implicit execution of ENABLE.

The default setting of SERVEROUTPUT is implementation dependent. For example, in Oracle SQL*Plus, SERVEROUTPUT(FALSE) is the default. In PSQL, SERVEROUTPUT(TRUE) is the default. Also note that in Oracle SQL*Plus, this setting is controlled using the SQL*Plus SET command, not by a stored procedure as implemented in Advanced Server.

```
SERVEROUTPUT(stdout BOOLEAN)
```

Parameters

stdout

Set to TRUE if subsequent PUT, PUT_LINE, or NEW_LINE commands are to send text directly to standard output of the command line. Set to FALSE if text is to be sent to the message buffer.

Examples

The following anonymous block sends the first message to the command line and the second message to the message buffer.

```
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('This message goes to the command line');
    DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    DBMS_OUTPUT.PUT_LINE('This message goes to the message buffer');
END;

This message goes to the command line
```

If within the same session, the following anonymous block is executed, the message stored in the message buffer from the prior example is flushed and displayed on the command line as well as the new message.

```
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Flush messages from the buffer');
END;

This message goes to the message buffer
Flush messages from the buffer
```

7.8 DBMS_PIPE

The `DBMS_PIPE` package provides the capability to send messages through a pipe within or between sessions connected to the same database cluster.

The procedures and functions available in the `DBMS_PIPE` package are listed in the following table.

Table 7-7-7 DBMS_PIPE Functions/Procedures

Function/Procedure	Return Type	Description
`CREATE_PIPE(`*`pipename`*` [, `*`maxpipesize`*` ] [, `*`private`*` ])`	`INTEGER`	Explicitly create a private pipe if *`private`* is "true" (the default) or a public pipe if *`private`* is "false".
`NEXT_ITEM_TYPE`	`INTEGER`	Determine the data type of the next item in a received message.
`PACK_MESSAGE(`*`item`*`)`	n/a	Place *`item`* in the session's local message buffer.
`PURGE(`*`pipename`*`)`	n/a	Remove unreceived messages from the specified pipe.
`RECEIVE_MESSAGE(`*`pipename`*` [, `*`timeout`*` ])`	`INTEGER`	Get a message from a specified pipe.
`REMOVE_PIPE(`*`pipename`*`)`	`INTEGER`	Delete an explicitly created pipe.
`RESET_BUFFER`	n/a	Reset the local message buffer.
`SEND_MESSAGE(`*`pipename`*` [, `*`timeout`*` ] [, `*`maxpipesize`*` ])`	`INTEGER`	Send a message on a pipe.
`UNIQUE_SESSION_NAME`	`VARCHAR2`	Obtain a unique session name.
`UNPACK_MESSAGE(`*`item`*` OUT)`	n/a	Retrieve the next data item from a message into a type-compatible variable, *`item`*.

Pipes are categorized as implicit or explicit. An *implicit pipe* is created if a reference is made to a pipe name that was not previously created by the `CREATE_PIPE` function. For example, if the `SEND_MESSAGE` function is executed using a non-existent pipe name, a new implicit pipe is created with that name. An *explicit pipe* is created using the `CREATE_PIPE` function whereby the first parameter specifies the pipe name for the new pipe.

Pipes are also categorized as private or public. A *private pipe* can only be accessed by the user who created the pipe. Even a superuser cannot access a private pipe that was created by another user. A *public pipe* can be accessed by any user who has access to the `DBMS_PIPE` package.

A public pipe can only be created by using the `CREATE_PIPE` function with the third parameter set to `FALSE`. The `CREATE_PIPE` function can be used to create a private pipe by setting the third parameter to `TRUE` or by omitting the third parameter. All implicit pipes are private.

The individual data items or "lines" of a message are first built-in a *local message buffer*, unique to the current session. The `PACK_MESSAGE` procedure builds the message in the session's local message buffer. The `SEND_MESSAGE` function is then used to send the message through the pipe.

Receipt of a message involves the reverse operation. The `RECEIVE_MESSAGE` function is used to get a message from the specified pipe. The message is written to the session's local message buffer. The `UNPACK_MESSAGE` procedure is then used to transfer the message data items from the message buffer to program variables. If a pipe contains multiple messages, `RECEIVE_MESSAGE` gets the messages in *FIFO* (first-in-first-out) order.

Each session maintains separate message buffers for messages created with the `PACK_MESSAGE` procedure and messages retrieved by the `RECEIVE_MESSAGE` function. Thus messages can be both built and received in the same session. However, if consecutive `RECEIVE_MESSAGE` calls are made, only the message from the last `RECEIVE_MESSAGE` call will be preserved in the local message buffer.

7.8.1 CREATE_PIPE

The `CREATE_PIPE` function creates an explicit public pipe or an explicit private pipe with a specified name.

```
status INTEGER CREATE_PIPE(pipename VARCHAR2
  [, maxpipesize INTEGER ] [, private BOOLEAN ])
```

Parameters

pipename

Name of the pipe.

maxpipesize

Maximum capacity of the pipe in bytes. Default is 8192 bytes.

private

Create a public pipe if set to `FALSE`. Create a private pipe if set to `TRUE`. This is the default.

status

Status code returned by the operation. 0 indicates successful creation.

Examples

The following example creates a private pipe named `messages`:

```
DECLARE
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.CREATE_PIPE('messages');
    DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END;
CREATE_PIPE status: 0
```

The following example creates a public pipe named `mailbox`:

```
DECLARE
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.CREATE_PIPE('mailbox',8192,FALSE);
    DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END;
CREATE_PIPE status: 0
```

7.8.2 NEXT_ITEM_TYPE

The `NEXT_ITEM_TYPE` function returns an integer code identifying the data type of the next data item in a message that has been retrieved into the session's local message buffer. As each item is moved off of the local message buffer with the `UNPACK_MESSAGE` procedure, the `NEXT_ITEM_TYPE` function will return the data type code for the next available item. A code of 0 is returned when there are no more items left in the message.

```
typecode INTEGER NEXT_ITEM_TYPE
```

Parameters

`typecode`

Code identifying the data type of the next data item as shown in Table 7-7-8.

Table 7-7-8 NEXT_ITEM_TYPE Data Type Codes

Type Code	Data Type
0	No more data items
9	NUMBER
11	VARCHAR2
13	DATE
23	RAW

Note: The type codes list in the table are not compatible with Oracle databases. Oracle assigns a different numbering sequence to the data types.

Examples

The following example shows a pipe packed with a `NUMBER` item, a `VARCHAR2` item, a `DATE` item, and a `RAW` item. A second anonymous block then uses the `NEXT_ITEM_TYPE` function to display the type code of each item.

```
DECLARE
    v_number        NUMBER := 123;
    v_varchar       VARCHAR2(20) := 'Character data';
    v_date          DATE := SYSDATE;
    v_raw           RAW(4) := '21222324';
    v_status        INTEGER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE(v_number);
    DBMS_PIPE.PACK_MESSAGE(v_varchar);
    DBMS_PIPE.PACK_MESSAGE(v_date);
    DBMS_PIPE.PACK_MESSAGE(v_raw);
    v_status := DBMS_PIPE.SEND_MESSAGE('datatypes');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

SEND_MESSAGE status: 0

DECLARE
    v_number        NUMBER;
    v_varchar       VARCHAR2(20);
    v_date          DATE;
    v_timestamp     TIMESTAMP;
    v_raw           RAW(4);
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('datatypes');
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
    DBMS_OUTPUT.PUT_LINE('----------------------------------');

    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_number);
    DBMS_OUTPUT.PUT_LINE('NUMBER Item   : ' || v_number);
    DBMS_OUTPUT.PUT_LINE('----------------------------------');

    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_varchar);
    DBMS_OUTPUT.PUT_LINE('VARCHAR2 Item : ' || v_varchar);
    DBMS_OUTPUT.PUT_LINE('----------------------------------');

    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_date);
    DBMS_OUTPUT.PUT_LINE('DATE Item     : ' || v_date);
    DBMS_OUTPUT.PUT_LINE('----------------------------------');

    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_raw);
    DBMS_OUTPUT.PUT_LINE('RAW Item      : ' || v_raw);
    DBMS_OUTPUT.PUT_LINE('----------------------------------');

    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_OUTPUT.PUT_LINE('---------------------------------');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

RECEIVE_MESSAGE status: 0
----------------------------------
NEXT_ITEM_TYPE: 9
NUMBER Item   : 123
----------------------------------
NEXT_ITEM_TYPE: 11
VARCHAR2 Item : Character data
----------------------------------
NEXT_ITEM_TYPE: 13
DATE Item     : 02-OCT-07 11:11:43
----------------------------------
NEXT_ITEM_TYPE: 23
RAW Item      : 21222324
----------------------------------
NEXT_ITEM_TYPE: 0
```

7.8.3 PACK_MESSAGE

The `PACK_MESSAGE` procedure places an item of data in the session's local message buffer. `PACK_MESSAGE` must be executed at least once before issuing a `SEND_MESSAGE` call.

```
PACK_MESSAGE(item { DATE | NUMBER | VARCHAR2 | RAW })
```

Use the `UNPACK_MESSAGE` procedure to obtain data items once the message is retrieved using a `RECEIVE_MESSAGE` call.

Parameters

`item`

An expression evaluating to any of the acceptable parameter data types. The value is added to the session's local message buffer.

7.8.4 PURGE

The `PURGE` procedure removes the unreceived messages from a specified implicit pipe.

```
PURGE(pipename VARCHAR2)
```

Use the REMOVE_PIPE function to delete an explicit pipe.

Parameters

pipename

Name of the pipe.

Examples

Two messages are sent on a pipe:

```
DECLARE
    v_status        INTEGER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE('Message #1');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);

    DBMS_PIPE.PACK_MESSAGE('Message #2');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END;

SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

Receive the first message and unpack it:

```
DECLARE
    v_item          VARCHAR2(80);
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_item);
    DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END;

RECEIVE_MESSAGE status: 0
Item: Message #1
```

Purge the pipe:

```
EXEC DBMS_PIPE.PURGE('pipe');
```

Try to retrieve the next message. The RECEIVE_MESSAGE call returns status code 1 indicating it timed out because no message was available.

```
DECLARE
    v_item          VARCHAR2(80);
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
```

```
END;

RECEIVE_MESSAGE status: 1
```

7.8.5 RECEIVE_MESSAGE

The `RECEIVE_MESSAGE` function obtains a message from a specified pipe.

```
status INTEGER RECEIVE_MESSAGE(pipename VARCHAR2
  [, timeout INTEGER ])
```

Parameters

pipename

Name of the pipe.

timeout

Wait time (seconds). Default is 86400000 (1000 days).

status

Status code returned by the operation.

The possible status codes are:

Table 7-7-9 RECEIVE_MESSAGE Status Codes

Status Code	Description
0	Success
1	Time out
2	Message too large .for the buffer

7.8.6 REMOVE_PIPE

The `REMOVE_PIPE` function deletes an explicit private or explicit public pipe.

```
status INTEGER REMOVE_PIPE(pipename VARCHAR2)
```

Use the `REMOVE_PIPE` function to delete explicitly created pipes – i.e., pipes created with the `CREATE_PIPE` function.

Parameters

pipename

Name of the pipe.

status

Status code returned by the operation. A status code of 0 is returned even if the named pipe is non-existent.

Examples

Two messages are sent on a pipe:

```
DECLARE
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.CREATE_PIPE('pipe');
    DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status : ' || v_status);

    DBMS_PIPE.PACK_MESSAGE('Message #1');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);

    DBMS_PIPE.PACK_MESSAGE('Message #2');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END;

CREATE_PIPE status : 0
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

Receive the first message and unpack it:

```
DECLARE
    v_item          VARCHAR2(80);
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_item);
    DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END;

RECEIVE_MESSAGE status: 0
Item: Message #1
```

Remove the pipe:

```
SELECT DBMS_PIPE.REMOVE_PIPE('pipe') FROM DUAL;

remove_pipe
-------------
           0
```

```
(1 row)
```

Try to retrieve the next message. The `RECEIVE_MESSAGE` call returns status code 1 indicating it timed out because the pipe had been deleted.

```
DECLARE
    v_item          VARCHAR2(80);
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END;

RECEIVE_MESSAGE status: 1
```

7.8.7 RESET_BUFFER

The `RESET_BUFFER` procedure resets a "pointer" to the session's local message buffer back to the beginning of the buffer. This has the effect of causing subsequent `PACK_MESSAGE` calls to overwrite any data items that existed in the message buffer prior to the `RESET_BUFFER` call.

```
RESET_BUFFER
```

Examples

A message to John is written to the local message buffer. It is replaced by a message to Bob by calling `RESET_BUFFER`. The message is sent on the pipe.

```
DECLARE
    v_status        INTEGER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE('Hi, John');
    DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 3:00, today?');
    DBMS_PIPE.PACK_MESSAGE('If not, is tomorrow at 8:30 ok with you?');
    DBMS_PIPE.RESET_BUFFER;
    DBMS_PIPE.PACK_MESSAGE('Hi, Bob');
    DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 9:30, tomorrow?');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END;

SEND_MESSAGE status: 0
```

The message to Bob is in the received message.

```
DECLARE
    v_item          VARCHAR2(80);
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_item);
    DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
    DBMS_PIPE.UNPACK_MESSAGE(v_item);
    DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END;

RECEIVE_MESSAGE status: 0
Item: Hi, Bob
Item: Can you attend a meeting at 9:30, tomorrow?
```

7.8.8 SEND_MESSAGE

The `SEND_MESSAGE` function sends a message from the session's local message buffer to the specified pipe.

```
status SEND_MESSAGE(pipename VARCHAR2 [, timeout INTEGER ]
  [, maxpipesize INTEGER ])
```

Parameters

pipename

Name of the pipe.

timeout

Wait time (seconds). Default is 86400000 (1000 days).

maxpipesize

Maximum capacity of the pipe in bytes. Default is 8192 bytes.

status

Status code returned by the operation.

The possible status codes are:

Table 7-7-10 SEND_MESSAGE Status Codes

Status Code	Description
0	Success
1	Time out
3	Function interrupted

7.8.9 UNIQUE_SESSION_NAME

The `UNIQUE_SESSION_NAME` function returns a name, unique to the current session.

```
name VARCHAR2 UNIQUE_SESSION_NAME
```

Parameters

`name`

Unique session name.

Examples

The following anonymous block retrieves and displays a unique session name.

```
DECLARE
    v_session       VARCHAR2(30);
BEGIN
    v_session := DBMS_PIPE.UNIQUE_SESSION_NAME;
    DBMS_OUTPUT.PUT_LINE('Session Name: ' || v_session);
END;

Session Name: PG$PIPE$5$2752
```

7.8.10 UNPACK_MESSAGE

The `UNPACK_MESSAGE` procedure copies the data items of a message from the local message buffer to a specified program variable. The message must be placed in the local message buffer with the `RECEIVE_MESSAGE` function before using `UNPACK_MESSAGE`.

```
UNPACK_MESSAGE(item OUT { DATE | NUMBER | VARCHAR2 | RAW })
```

Parameters

`item`

Type-compatible variable that receives a data item from the local message buffer.

7.8.11 Comprehensive Example

The following example uses a pipe as a "mailbox". The procedures to create the mailbox, add a multi-item message to the mailbox (up to three items), and display the full contents of the mailbox are enclosed in a package named, `mailbox`.

```
CREATE OR REPLACE PACKAGE mailbox
IS
    PROCEDURE create_mailbox;
    PROCEDURE add_message (
        p_mailbox   VARCHAR2,
        p_item_1    VARCHAR2,
        p_item_2    VARCHAR2 DEFAULT 'END',
        p_item_3    VARCHAR2 DEFAULT 'END'
    );
    PROCEDURE empty_mailbox (
        p_mailbox   VARCHAR2,
        p_waittime  INTEGER DEFAULT 10
    );
END mailbox;

CREATE OR REPLACE PACKAGE BODY mailbox
IS
    PROCEDURE create_mailbox
    IS
        v_mailbox   VARCHAR2(30);
        v_status    INTEGER;
    BEGIN
        v_mailbox := DBMS_PIPE.UNIQUE_SESSION_NAME;
        v_status := DBMS_PIPE.CREATE_PIPE(v_mailbox,1000,FALSE);
        IF v_status = 0 THEN
            DBMS_OUTPUT.PUT_LINE('Created mailbox: ' || v_mailbox);
        ELSE
            DBMS_OUTPUT.PUT_LINE('CREATE_PIPE failed - status: ' ||
                v_status);
        END IF;
    END create_mailbox;

    PROCEDURE add_message (
        p_mailbox   VARCHAR2,
        p_item_1    VARCHAR2,
        p_item_2    VARCHAR2 DEFAULT 'END',
        p_item_3    VARCHAR2 DEFAULT 'END'
    )
    IS
        v_item_cnt  INTEGER := 0;
        v_status    INTEGER;
    BEGIN
        DBMS_PIPE.PACK_MESSAGE(p_item_1);
        v_item_cnt := 1;
        IF p_item_2 != 'END' THEN
            DBMS_PIPE.PACK_MESSAGE(p_item_2);
            v_item_cnt := v_item_cnt + 1;
        END IF;
        IF p_item_3 != 'END' THEN
            DBMS_PIPE.PACK_MESSAGE(p_item_3);
            v_item_cnt := v_item_cnt + 1;
        END IF;
        v_status := DBMS_PIPE.SEND_MESSAGE(p_mailbox);
        IF v_status = 0 THEN
            DBMS_OUTPUT.PUT_LINE('Added message with ' || v_item_cnt ||
                ' item(s) to mailbox ' || p_mailbox);
        ELSE
            DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE in add_message failed - ' ||
                'status: ' || v_status);
        END IF;
    END add_message;

    PROCEDURE empty_mailbox (
        p_mailbox    VARCHAR2,
        p_waittime   INTEGER DEFAULT 10
    )
    IS
        v_msgno      INTEGER DEFAULT 0;
        v_itemno     INTEGER DEFAULT 0;
        v_item       VARCHAR2(100);
        v_status     INTEGER;
    BEGIN
        v_status := DBMS_PIPE.RECEIVE_MESSAGE(p_mailbox,p_waittime);
        WHILE v_status = 0 LOOP
            v_msgno := v_msgno + 1;
            DBMS_OUTPUT.PUT_LINE('****** Start message #' || v_msgno ||
                ' ******');
            BEGIN
                LOOP
                    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
                    EXIT WHEN v_status = 0;
                    DBMS_PIPE.UNPACK_MESSAGE(v_item);
                    v_itemno := v_itemno + 1;
                    DBMS_OUTPUT.PUT_LINE('Item #' || v_itemno || ': ' ||
                        v_item);
                END LOOP;
                DBMS_OUTPUT.PUT_LINE('******* End message #' || v_msgno ||
                    ' *******');
                DBMS_OUTPUT.PUT_LINE('*');
                v_itemno := 0;
                v_status := DBMS_PIPE.RECEIVE_MESSAGE(p_mailbox,1);
            END;
        END LOOP;
        DBMS_OUTPUT.PUT_LINE('Number of messages received: ' || v_msgno);
        v_status := DBMS_PIPE.REMOVE_PIPE(p_mailbox);
        IF v_status = 0 THEN
            DBMS_OUTPUT.PUT_LINE('Deleted mailbox ' || p_mailbox);
        ELSE
            DBMS_OUTPUT.PUT_LINE('Could not delete mailbox - status: '
                || v_status);
        END IF;
    END empty_mailbox;
END mailbox;
```

The following demonstrates the execution of the procedures in `mailbox`. The first procedure creates a public pipe using a name generated by the `UNIQUE_SESSION_NAME` function.

```
EXEC mailbox.create_mailbox;

Created mailbox: PG$PIPE$13$3940
```

Using the mailbox name, any user in the same database with access to the `mailbox` package and `DBMS_PIPE` package can add messages:

```
EXEC mailbox.add_message('PG$PIPE$13$3940','Hi, John','Can you attend a
meeting at 3:00, today?','-- Mary');

Added message with 3 item(s) to mailbox PG$PIPE$13$3940

EXEC mailbox.add_message('PG$PIPE$13$3940','Don''t forget to submit your
report','Thanks,','-- Joe');

Added message with 3 item(s) to mailbox PG$PIPE$13$3940
```

Finally, the contents of the mailbox can be emptied:

```
EXEC mailbox.empty_mailbox('PG$PIPE$13$3940');

****** Start message #1 ******
Item #1: Hi, John
Item #2: Can you attend a meeting at 3:00, today?
Item #3: -- Mary
******* End message #1 *******
*
****** Start message #2 ******
Item #1: Don't forget to submit your report
Item #2: Thanks,
Item #3: Joe
******* End message #2 *******
*
Number of messages received: 2
Deleted mailbox PG$PIPE$13$3940
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

7.9 DBMS_PROFILER

The `DBMS_PROFILER` package collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a performance profiling session; use the functions and procedures listed below to control the profiling tool.

For more information about the `DBMS_PROFILER` built-in package (including usage examples and a reference guide to the `DBMS_PROFILER` tables and views), see the Advanced Server Performance Features Guide.

Table 7-11 DBMS_PROFILER Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
`FLUSH_DATA`	Both	Status Code or Exception	Flushes performance data collected in the current session without terminating the session (profiling continues).
`GET_VERSION(`*`major`*` OUT, `*`minor`*` OUT)`	Procedure	n/a	Returns the version number of this package.
`INTERNAL_VERSION_CHECK`	Function	Status Code	Confirms that the current version of the profiler will work with the current database.
`PAUSE_PROFILER`	Both	Status Code or Exception	Pause data collection.
`RESUME_PROFILER`	Both	Status Code or Exception	Resume data collection.
`START_PROFILER(`*`run_comment`*`, `*`run_comment1`*` [, `*`run_number`*` OUT ])`	Both	Status Code or Exception	Start data collection.
`STOP_PROFILER`	Both	Status Code or Exception	Stop data collection and flush performance data to the `PLSQL_PROFILER_RAWDATA` table.

The functions within the `DBMS_PROFILER` package return a status code to indicate success or failure; the `DBMS_PROFILER` procedures raise an exception only if they encounter a failure. The status codes and messages returned by the functions, and the exceptions raised by the procedures are listed in the table below.

Table 7-12 DBMS_PROFILER Status Codes and Exceptions

Status Code	Message	Exception	Description
`-1`	`error version`	`version_mismatch`	The profiler version and the database are incompatible.
`0`	`success`	n/a	The operation completed successfully.
`1`	`error_param`	`profiler_error`	The operation received an incorrect parameter.
`2`	`error_io`	`profiler_error`	The data flush operation has failed.

7.9.1 FLUSH_DATA

The `FLUSH_DATA` function/procedure flushes the data collected in the current session without terminating the profiler session. The data is flushed to the tables described in the Advanced Server Performance Features Guide. The function and procedure signatures are:

```
status INTEGER FLUSH_DATA

FLUSH_DATA
```

Parameters

status

Status code returned by the operation.

7.9.2 GET_VERSION

The `GET_VERSION` procedure returns the version of `DBMS_PROFILER`. The procedure signature is:

```
GET_VERSION(major OUT INTEGER, minor OUT INTEGER)
```

Parameters

major

The major version number of `DBMS_PROFILER`.

minor

The minor version number of `DBMS_PROFILER`.

7.9.3 INTERNAL_VERSION_CHECK

The `INTERNAL_VERSION_CHECK` function confirms that the current version of `DBMS_PROFILER` will work with the current database. The function signature is:

```
status INTEGER INTERNAL_VERSION_CHECK
```

Parameters

status

Status code returned by the operation.

7.9.4 PAUSE_PROFILER

The `PAUSE_PROFILER` function/procedure pauses a profiling session. The function and procedure signatures are:

```
status INTEGER PAUSE_PROFILER

PAUSE_PROFILER
```

Parameters

status

Status code returned by the operation.

7.9.5 RESUME_PROFILER

The `RESUME_PROFILER` function/procedure pauses a profiling session. The function and procedure signatures are:

```
status INTEGER RESUME_PROFILER

RESUME_PROFILER
```

Parameters

status

Status code returned by the operation.

7.9.6 START_PROFILER

The `START_PROFILER` function/procedure starts a data collection session. The function and procedure signatures are:

```
status INTEGER START_PROFILER(run_comment TEXT := SYSDATE,
  run_comment1 TEXT := '' [, run_number OUT INTEGER ])

START_PROFILER(run_comment TEXT := SYSDATE,
  run_comment1 TEXT := '' [, run_number OUT INTEGER ])
```

Parameters

run_comment

A user-defined comment for the profiler session. The default value is `SYSDATE`.

run_comment1

An additional user-defined comment for the profiler session. The default value is `''`.

run_number

The session number of the profiler session.

status

Status code returned by the operation.

7.9.7 STOP_PROFILER

The `STOP_PROFILER` function/procedure stops a profiling session and flushes the performance information to the `DBMS_PROFILER` tables and view. The function and procedure signatures are:

```
status INTEGER STOP_PROFILER

STOP_PROFILER
```

Parameters

status

Status code returned by the operation.

7.10 DBMS_RANDOM

The `DBMS_RANDOM` package provides a number of methods to generate random values. The procedures and functions available in the `DBMS_RANDOM` package are listed in the following table.

Table 7. DBMS_RANDOM Functions/Procedures

Function/Procedure	Return Type	Description
`INITIALIZE(`*`val`*`)`	n/a	Initializes the `DBMS_RANDOM` package with the specified seed *`value`*. Deprecated, but supported for backward compatibility.
`NORMAL()`	`NUMBER`	Returns a random `NUMBER.`
`RANDOM`	`INTEGER`	Returns a random `INTEGER` with a value greater than or equal to -2^31 and less than 2^31. Deprecated, but supported for backward compatibility.
`SEED(`*`val`*`)`	n/a	Resets the seed with the specified *`value`*.
`SEED(`*`val`*`)`	n/a	Resets the seed with the specified *`value`*.
`STRING(`*`opt, len`*`)`	`VARCHAR2`	Returns a random string.
`TERMINATE`	n/a	`TERMINATE` has no effect. Deprecated, but supported for backward compatibility.
`VALUE`	`NUMBER`	Returns a random number with a value greater than or equal to `0` and less than `1`, with 38 digit precision.
`VALUE(`*`low, high`*`)`	`NUMBER`	Returns a random number with a value greater than or equal to *`low`* and less than *`high`*.

7.10.1 INITIALIZE

The `INITIALIZE` procedure initializes the `DBMS_RANDOM` package with a seed value. The signature is:

```
INITIALIZE(val IN INTEGER)
```

This procedure should be considered deprecated; it is included for backward compatibility only.

Parameters

`val`

`val` is the seed value used by the `DBMS_RANDOM` package algorithm.

Example

The following code snippet demonstrates a call to the `INITIALIZE` procedure that initializes the `DBMS_RANDOM` package with the seed value, `6475`.

```
DBMS_RANDOM.INITIALIZE(6475);
```

7.10.2 NORMAL

The `NORMAL` function returns a random number of type `NUMBER`. The signature is:

```
result NUMBER NORMAL()
```

Parameters

`result`

`result` is a random value of type `NUMBER`.

Example

The following code snippet demonstrates a call to the `NORMAL` function:

```
x:= DBMS_RANDOM.NORMAL();
```

7.10.3 RANDOM

The `RANDOM` function returns a random `INTEGER` value that is greater than or equal to -2 ^31 and less than 2 ^31. The signature is:

```
result INTEGER RANDOM()
```

This function should be considered deprecated; it is included for backward compatibility only.

Parameters

`result`

`result` is a random value of type `INTEGER`.

Example

The following code snippet demonstrates a call to the `RANDOM` function. The call returns a random number:

```
x := DBMS_RANDOM.RANDOM();
```

7.10.4 SEED

The first form of the `SEED` procedure resets the seed value for the `DBMS_RANDOM` package with an `INTEGER` value. The `SEED` procedure is available in two forms; the signature of the first form is:

```
SEED(val IN INTEGER)
```

Parameters

val

val is the seed value used by the `DBMS_RANDOM` package algorithm.

Example

The following code snippet demonstrates a call to the `SEED` procedure; the call sets the seed value at `8495`.

```
DBMS_RANDOM.SEED(8495);
```

7.10.5 SEED

The second form of the `SEED` procedure resets the seed value for the `DBMS_RANDOM` package with a string value. The `SEED` procedure is available in two forms; the signature of the second form is:

```
SEED(val IN VARCHAR2)
```

Parameters

val

val is the seed value used by the DBMS_RANDOM package algorithm.

Example

The following code snippet demonstrates a call to the SEED procedure; the call sets the seed value to abc123.

```
DBMS_RANDOM.SEED('abc123');
```

7.10.6 STRING

The STRING function returns a random VARCHAR2 string in a user-specified format. The signature of the STRING function is:

```
result VARCHAR2 STRING(opt IN CHAR, len IN NUMBER)
```

Parameters

opt

Formatting option for the returned string. *option* may be:

Option	**Specifies Formatting Option**
u or U	Uppercase alpha string
l or L	Lowercase alpha string
a or A	Mixed case string
x or X	Uppercase alpha-numeric string
p or P	Any printable characters

len

The length of the returned string.

result

result is a random value of type VARCHAR2.

Example

The following code snippet demonstrates a call to the STRING function; the call returns a random alpha-numeric character string that is 10 characters long.

```
x := DBMS_RANDOM.STRING('X', 10);
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

7.10.7 TERMINATE

The `TERMINATE` procedure has no effect. The signature is:

```
TERMINATE
```

The `TERMINATE` procedure should be considered deprecated; the procedure is supported for compatibility only.

7.10.8 VALUE

The `VALUE` function returns a random `NUMBER` that is greater than or equal to 0, and less than 1, with 38 digit precision. The `VALUE` function has two forms; the signature of the first form is:

```
result NUMBER VALUE()
```

Parameters

result

result is a random value of type `NUMBER`.

Example

The following code snippet demonstrates a call to the `VALUE` function. The call returns a random `NUMBER`:

```
x := DBMS_RANDOM.VALUE();
```

7.10.9 VALUE

The `VALUE` function returns a random `NUMBER` with a value that is between user-specified boundaries. The `VALUE` function has two forms; the signature of the second form is:

```
result NUMBER VALUE(low IN NUMBER, high IN NUMBER)
```

Parameters

low

> *low* specifies the lower boundary for the random value. The random value may be equal to *low*.

high

> *high* specifies the upper boundary for the random value; the random value will be less than *high*.

result

> *result* is a random value of type `NUMBER`.

Example

The following code snippet demonstrates a call to the `VALUE` function. The call returns a random `NUMBER` with a value that is greater than or equal to 1 and less than 100:

```
x := DBMS_RANDOM.VALUE(1, 100);
```

7.11 DBMS_RLS

The `DBMS_RLS` package enables the implementation of Virtual Private Database on certain Advanced Server database objects.

Table 7-13 DBMS_RLS Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
`ADD_POLICY(`*`object_schema, object_name, policy_name, function_schema, policy_function`*` [, `*`statement_types`*` [, `*`update_check`*` [, `*`enable`*` [, `*`static_policy`*` [, `*`policy_type`*` [, `*`long_predicate`*` [, `*`sec_relevant_cols`*` [, `*`sec_relevant_cols_opt`*` ]]]]]]]])`	Procedure	n/a	Add a security policy to a database object.
`DROP_POLICY(`*`object_schema, object_name, policy_name`*`)`	Procedure	n/a	Remove a security policy from a database object.
`ENABLE_POLICY(`*`object_schema, object_name, policy_name, enable`*`)`	Procedure	n/a	Enable or disable a security policy.

Advanced Server's implementation of `DBMS_RLS` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

Virtual Private Database is a type of fine-grained access control using security policies. *Fine-grained access control* in Virtual Private Database means that access to data can be controlled down to specific rows as defined by the security policy.

The rules that encode a security policy are defined in a *policy function*, which is an SPL function with certain input parameters and return value. The *security policy* is the named association of the policy function to a particular database object, typically a table.

Note: In Advanced Server, the policy function can be written in any language supported by Advanced Server such as SQL, PL/pgSQL and SPL.

Note: The database objects currently supported by Advanced Server Virtual Private Database are tables. Policies cannot be applied to views or synonyms.

The advantages of using Virtual Private Database are the following:

- Provides a fine-grained level of security. Database object level privileges given by the `GRANT` command determine access privileges to the entire instance of a database object, while Virtual Private Database provides access control for the individual rows of a database object instance.

- A different security policy can be applied depending upon the type of SQL command (`INSERT`, `UPDATE`, `DELETE`, or `SELECT`).
- The security policy can vary dynamically for each applicable SQL command affecting the database object depending upon factors such as the session user of the application accessing the database object.
- Invocation of the security policy is transparent to all applications that access the database object and thus, individual applications do not have to be modified to apply the security policy.
- Once a security policy is enabled, it is not possible for any application (including new applications) to circumvent the security policy except by the system privilege noted by the following.
- Even superusers cannot circumvent the security policy except by the system privilege noted by the following.

Note: The only way security policies can be circumvented is if the `EXEMPT ACCESS POLICY` system privilege has been granted to a user. The `EXEMPT ACCESS POLICY` privilege should be granted with extreme care as a user with this privilege is exempted from all policies in the database. See the `GRANT` command in Section 3.3.55 or the `ALTER ROLE` command in Section 3.3.3 for additional information.

The `DBMS_RLS` package provides procedures to create policies, remove policies, enable policies, and disable policies.

The process for implementing Virtual Private Database is as follows:

- Create a policy function. The function must have two input parameters of type `VARCHAR2`. The first input parameter is for the schema containing the database object to which the policy is to apply and the second input parameter is for the name of that database object. The function must have a `VARCHAR2` return type. The function must return a string in the form of a `WHERE` clause predicate. This predicate is dynamically appended as an `AND` condition to the SQL command that acts upon the database object. Thus, rows that do not satisfy the policy function predicate are filtered out from the SQL command result set.
- Use the `ADD_POLICY` procedure to define a new policy, which is the association of a policy function with a database object. With the `ADD_POLICY` procedure, you can also specify the types of SQL commands (`INSERT`, `UPDATE`, `DELETE`, or `SELECT`) to which the policy is to apply, whether or not to enable the policy at the time of its creation, and if the policy should apply to newly inserted rows or the modified image of updated rows.
- Use the `ENABLE_POLICY` procedure to disable or enable an existing policy.
- Use the `DROP_POLICY` procedure to remove an existing policy. The `DROP_POLICY` procedure does not drop the policy function or the associated database object.

Once policies are created, they can be viewed in the catalog views, compatible with Oracle databases: `ALL_POLICIES` (see Section 10.11), `DBA_POLICIES` (see Section 10.35), or `USER_POLICIES` (see Section 10.62).

The `SYS_CONTEXT` function is often used with `DBMS_RLS`. The signature is:

```
SYS_CONTEXT(namespace, attribute)
```

Where:

> *`namespace`* is a `VARCHAR2`; the only accepted value is `USERENV`. Any other value will return `NULL`.
>
> *`attribute`* is a `VARCHAR2`. *`attribute`* may be:

attribute Value	Equivalent Value
`SESSION_USER`	`pg_catalog.session_user`
`CURRENT_USER`	`pg_catalog.current_user`
`CURRENT_SCHEMA`	`pg_catalog.current_schema`
`HOST`	`pg_catalog.inet_host`
`IP_ADDRESS`	`pg_catalog.inet_client_addr`
`SERVER_HOST`	`pg_catalog.inet_server_addr`

Note: The examples used to illustrate the `DBMS_RLS` package are based on a modified copy of the sample `emp` table provided with Advanced Server along with a role named `salesmgr` that is granted all privileges on the table. You can create the modified copy of the `emp` table named `vpemp` and the `salesmgr` role as shown by the following:

```
CREATE TABLE public.vpemp AS SELECT empno, ename, job, sal, comm, deptno FROM
emp;
ALTER TABLE vpemp ADD authid VARCHAR2(12);
UPDATE vpemp SET authid = 'researchmgr' WHERE deptno = 20;
UPDATE vpemp SET authid = 'salesmgr' WHERE deptno = 30;
SELECT * FROM vpemp;

empno | ename  |    job    |   sal    |  comm    | deptno |   authid
-------+--------+-----------+---------+---------+--------+-------------
  7782 | CLARK  | MANAGER   | 2450.00 |         |     10 |
  7839 | KING   | PRESIDENT | 5000.00 |         |     10 |
  7934 | MILLER | CLERK     | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK     |  800.00 |         |     20 | researchmgr
  7566 | JONES  | MANAGER   | 2975.00 |         |     20 | researchmgr
  7788 | SCOTT  | ANALYST   | 3000.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK     | 1100.00 |         |     20 | researchmgr
  7902 | FORD   | ANALYST   | 3000.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  | 1600.00 |  300.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN  | 1250.00 |  500.00 |     30 | salesmgr
  7654 | MARTIN | SALESMAN  | 1250.00 | 1400.00 |     30 | salesmgr
  7698 | BLAKE  | MANAGER   | 2850.00 |         |     30 | salesmgr
  7844 | TURNER | SALESMAN  | 1500.00 |    0.00 |     30 | salesmgr
  7900 | JAMES  | CLERK     |  950.00 |         |     30 | salesmgr
(14 rows)
```

```
CREATE ROLE salesmgr WITH LOGIN PASSWORD 'password';
GRANT ALL ON vpemp TO salesmgr;
```

7.11.1 ADD_POLICY

The `ADD_POLICY` procedure creates a new policy by associating a policy function with a database object.

You must be a superuser to execute this procedure.

```
ADD_POLICY(object_schema VARCHAR2, object_name VARCHAR2,
  policy_name VARCHAR2, function_schema VARCHAR2,
  policy_function VARCHAR2
  [, statement_types VARCHAR2
  [, update_check BOOLEAN
  [, enable BOOLEAN
  [, static_policy BOOLEAN
  [, policy_type INTEGER
  [, long_predicate BOOLEAN
  [, sec_relevant_cols VARCHAR2
  [, sec_relevant_cols_opt INTEGER ]]]]]]]])
```

Parameters

object_schema

Name of the schema containing the database object to which the policy is to be applied.

object_name

Name of the database object to which the policy is to be applied. A given database object may have more than one policy applied to it.

policy_name

Name assigned to the policy. The combination of database object (identified by *object_schema* and *object_name*) and policy name must be unique within the database.

function_schema

Name of the schema containing the policy function.

Note: The policy function may belong to a package in which case `function_schema` must contain the name of the schema in which the package is defined.

`policy_function`

Name of the SPL function that defines the rules of the security policy. The same function may be specified in more than one policy.

Note: The policy function may belong to a package in which case *`policy_function`* must also contain the package name in dot notation (that is, *`package_name.function_name`*).

`statement_types`

Comma-separated list of SQL commands to which the policy applies. Valid SQL commands are `INSERT`, `UPDATE`, `DELETE`, and `SELECT`. The default is `INSERT,UPDATE,DELETE,SELECT`.

Note: Advanced Server accepts `INDEX` as a statement type, but it is ignored. Policies are not applied to index operations in Advanced Server.

`update_check`

Applies to `INSERT` and `UPDATE` SQL commands only.

When set to `TRUE`, the policy is applied to newly inserted rows and to the modified image of updated rows. If any of the new or modified rows do not qualify according to the policy function predicate, then the `INSERT` or `UPDATE` command throws an exception and no rows are inserted or modified by the `INSERT` or `UPDATE` command.

When set to `FALSE`, the policy is not applied to newly inserted rows or the modified image of updated rows. Thus, a newly inserted row may not appear in the result set of a subsequent SQL command that invokes the same policy. Similarly, rows which qualified according to the policy prior to an `UPDATE` command may not appear in the result set of a subsequent SQL command that invokes the same policy.

The default is `FALSE`.

`enable`

When set to `TRUE`, the policy is enabled and applied to the SQL commands given by the *`statement_types`* parameter. When set to `FALSE` the policy is disabled and not applied to any SQL commands. The policy can be enabled using the `ENABLE_POLICY` procedure. The default is `TRUE`.

`static_policy`

In Oracle, when set to `TRUE`, the policy is *static*, which means the policy function is evaluated once per database object the first time it is invoked by a policy on that database object. The resulting policy function predicate string is saved in memory and reused for all invocations of that policy on that database object while the database server instance is running.

When set to `FALSE`, the policy is *dynamic*, which means the policy function is re-evaluated and the policy function predicate string regenerated for all invocations of the policy.

The default is `FALSE`.

Note: In Oracle 10g, the *`policy_type`* parameter was introduced, which is intended to replace the *`static_policy`* parameter. In Oracle, if the *`policy_type`* parameter is not set to its default value of `NULL`, the *`policy_type`* parameter setting overrides the *`static_policy`* setting.

Note: The setting of *`static_policy`* is ignored by Advanced Server. Advanced Server implements only the dynamic policy, regardless of the setting of the *`static_policy`* parameter.

`policy_type`

In Oracle, determines when the policy function is re-evaluated, and hence, if and when the predicate string returned by the policy function changes. The default is `NULL`.

Note: The setting of this parameter is ignored by Advanced Server. Advanced Server always assumes a dynamic policy.

`long_predicate`

In Oracle, allows predicates up to 32K bytes if set to `TRUE`, otherwise predicates are limited to 4000 bytes. The default is `FALSE`.

Note: The setting of this parameter is ignored by Advanced Server. An Advanced Server policy function can return a predicate of unlimited length for all practical purposes.

sec_relevant_cols

Comma-separated list of columns of *object_name*. Provides *column-level Virtual Private Database* for the listed columns. The policy is enforced if any of the listed columns are referenced in a SQL command of a type listed in *statement_types*. The policy is not enforced if no such columns are referenced.

The default is NULL, which has the same effect as if all of the database object's columns were included in *sec_relevant_cols*.

sec_relevant_cols_opt

In Oracle, if *sec_relevant_cols_opt* is set to DBMS_RLS.ALL_ROWS (INTEGER constant of value 1), then the columns listed in *sec_relevant_cols* return NULL on all rows where the applied policy predicate is false. (If *sec_relevant_cols_opt* is not set to DBMS_RLS.ALL_ROWS, these rows would not be returned at all in the result set.) The default is NULL.

Note: Advanced Server does not support the DBMS_RLS.ALL_ROWS functionality. Advanced Server throws an error if sec_relevant_cols_opt is set to DBMS_RLS.ALL_ROWS (INTEGER value of 1).

Examples

This example uses the following policy function:

```
CREATE OR REPLACE FUNCTION verify_session_user (
    p_schema        VARCHAR2,
    p_object        VARCHAR2
)
RETURN VARCHAR2
IS
BEGIN
    RETURN 'authid = SYS_CONTEXT(''USERENV'', ''SESSION_USER'')';
END;
```

This function generates the predicate authid = SYS_CONTEXT('USERENV', 'SESSION_USER'), which is added to the WHERE clause of any SQL command of the type specified in the ADD_POLICY procedure.

This limits the effect of the SQL command to those rows where the content of the authid column is the same as the session user.

Note: This example uses the SYS_CONTEXT function to return the login user name. In Oracle the SYS_CONTEXT function is used to return attributes of an *application context*. The first parameter of the SYS_CONTEXT function is the name of an application context while the second parameter is the name of an attribute set within the application context.

USERENV is a special built-in namespace that describes the current session. Advanced Server does not support application contexts, but only this specific usage of the SYS_CONTEXT function.

The following anonymous block calls the ADD_POLICY procedure to create a policy named secure_update to be applied to the vpemp table using function verify_session_user whenever an INSERT, UPDATE, or DELETE SQL command is given referencing the vpemp table.

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_update';
    v_function_schema       VARCHAR2(30) := 'enterprisedb';
    v_policy_function       VARCHAR2(30) := 'verify_session_user';
    v_statement_types       VARCHAR2(30) := 'INSERT,UPDATE,DELETE';
    v_update_check          BOOLEAN      := TRUE;
    v_enable                BOOLEAN      := TRUE;
BEGIN
    DBMS_RLS.ADD_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name,
        v_function_schema,
        v_policy_function,
        v_statement_types,
        v_update_check,
        v_enable
    );
END;
```

After successful creation of the policy, a terminal session is started by user salesmgr. The following query shows the content of the vpemp table:

```
edb=# \c edb salesmgr
Password for user salesmgr:
You are now connected to database "edb" as user "salesmgr".
edb=> SELECT * FROM vpemp;
 empno | ename  |    job    |   sal   |  comm   | deptno |   authid
-------+--------+-----------+---------+---------+--------+-------------
  7782 | CLARK  | MANAGER   | 2450.00 |         |     10 |
  7839 | KING   | PRESIDENT | 5000.00 |         |     10 |
  7934 | MILLER | CLERK     | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK     |  800.00 |         |     20 | researchmgr
  7566 | JONES  | MANAGER   | 2975.00 |         |     20 | researchmgr
  7788 | SCOTT  | ANALYST   | 3000.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK     | 1100.00 |         |     20 | researchmgr
  7902 | FORD   | ANALYST   | 3000.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  | 1600.00 |  300.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN  | 1250.00 |  500.00 |     30 | salesmgr
  7654 | MARTIN | SALESMAN  | 1250.00 | 1400.00 |     30 | salesmgr
  7698 | BLAKE  | MANAGER   | 2850.00 |         |     30 | salesmgr
  7844 | TURNER | SALESMAN  | 1500.00 |    0.00 |     30 | salesmgr
  7900 | JAMES  | CLERK     |  950.00 |         |     30 | salesmgr
(14 rows)
```

An unqualified UPDATE command (no WHERE clause) is issued by the salesmgr user:

```
edb=> UPDATE vpemp SET comm = sal * .75;
UPDATE 6
```

Instead of updating all rows in the table, the policy restricts the effect of the update to only those rows where the authid column contains the value salesmgr as specified by the policy function predicate authid = SYS_CONTEXT('USERENV', 'SESSION_USER').

The following query shows that the comm column has been changed only for those rows where authid contains salesmgr. All other rows are unchanged.

```
edb=> SELECT * FROM vpemp;
 empno | ename  |    job    |   sal   |  comm   | deptno |   authid
-------+--------+-----------+---------+---------+--------+-------------
  7782 | CLARK  | MANAGER   | 2450.00 |         |     10 |
  7839 | KING   | PRESIDENT | 5000.00 |         |     10 |
  7934 | MILLER | CLERK     | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK     |  800.00 |         |     20 | researchmgr
  7566 | JONES  | MANAGER   | 2975.00 |         |     20 | researchmgr
  7788 | SCOTT  | ANALYST   | 3000.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK     | 1100.00 |         |     20 | researchmgr
  7902 | FORD   | ANALYST   | 3000.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  | 1600.00 | 1200.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN  | 1250.00 |  937.50 |     30 | salesmgr
  7654 | MARTIN | SALESMAN  | 1250.00 |  937.50 |     30 | salesmgr
  7698 | BLAKE  | MANAGER   | 2850.00 | 2137.50 |     30 | salesmgr
  7844 | TURNER | SALESMAN  | 1500.00 | 1125.00 |     30 | salesmgr
  7900 | JAMES  | CLERK     |  950.00 |  712.50 |     30 | salesmgr
(14 rows)
```

Furthermore, since the *update_check* parameter was set to TRUE in the ADD_POLICY procedure, the following INSERT command throws an exception since the value given for the authid column, researchmgr, does not match the session user, which is salesmgr, and hence, fails the policy.

```
edb=> INSERT INTO vpemp VALUES (9001,'SMITH','ANALYST',3200.00,NULL,20,
'researchmgr');
ERROR:  policy with check option violation
DETAIL:  Policy predicate was evaluated to FALSE with the updated values
```

If *update_check* was set to FALSE, the preceding INSERT command would have succeeded.

The following example illustrates the use of the *sec_relevant_cols* parameter to apply a policy only when certain columns are referenced in the SQL command. The following policy function is used for this example, which selects rows where the employee salary is less than 2000.

```
CREATE OR REPLACE FUNCTION sal_lt_2000 (
    p_schema        VARCHAR2,
    p_object        VARCHAR2
)
RETURN VARCHAR2
IS
BEGIN
    RETURN 'sal < 2000';
END;
```

The policy is created so that it is enforced only if a `SELECT` command includes columns `sal` or `comm`:

```
DECLARE
    v_object_schema         VARCHAR2(30)  := 'public';
    v_object_name           VARCHAR2(30)  := 'vpemp';
    v_policy_name           VARCHAR2(30)  := 'secure_salary';
    v_function_schema       VARCHAR2(30)  := 'enterprisedb';
    v_policy_function       VARCHAR2(30)  := 'sal_lt_2000';
    v_statement_types       VARCHAR2(30)  := 'SELECT';
    v_sec_relevant_cols     VARCHAR2(30)  := 'sal,comm';
BEGIN
    DBMS_RLS.ADD_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name,
        v_function_schema,
        v_policy_function,
        v_statement_types,
        sec_relevant_cols => v_sec_relevant_cols
    );
END;
```

If a query does not reference columns `sal` or `comm`, then the policy is not applied. The following query returns all 14 rows of table `vpemp`:

```
edb=# SELECT empno, ename, job, deptno, authid FROM vpemp;
 empno | ename  |    job    | deptno |   authid
-------+--------+-----------+--------+-------------
  7782 | CLARK  | MANAGER   |     10 |
  7839 | KING   | PRESIDENT |     10 |
  7934 | MILLER | CLERK     |     10 |
  7369 | SMITH  | CLERK     |     20 | researchmgr
  7566 | JONES  | MANAGER   |     20 | researchmgr
  7788 | SCOTT  | ANALYST   |     20 | researchmgr
  7876 | ADAMS  | CLERK     |     20 | researchmgr
  7902 | FORD   | ANALYST   |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  |     30 | salesmgr
  7521 | WARD   | SALESMAN  |     30 | salesmgr
  7654 | MARTIN | SALESMAN  |     30 | salesmgr
  7698 | BLAKE  | MANAGER   |     30 | salesmgr
  7844 | TURNER | SALESMAN  |     30 | salesmgr
  7900 | JAMES  | CLERK     |     30 | salesmgr
(14 rows)
```

If the query references the `sal` or `comm` columns, then the policy is applied to the query eliminating any rows where `sal` is greater than or equal to `2000` as shown by the following:

```
edb=# SELECT empno, ename, job, sal, comm, deptno, authid FROM vpemp;
 empno | ename  |   job    |   sal    |  comm    | deptno |   authid
```

```
-------+--------+----------+---------+---------+--------+-------------
  7934 | MILLER | CLERK    | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK    |  800.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK    | 1100.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN | 1600.00 | 1200.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN | 1250.00 |  937.50 |     30 | salesmgr
  7654 | MARTIN | SALESMAN | 1250.00 |  937.50 |     30 | salesmgr
  7844 | TURNER | SALESMAN | 1500.00 | 1125.00 |     30 | salesmgr
  7900 | JAMES  | CLERK    |  950.00 |  712.50 |     30 | salesmgr
(8 rows)
```

7.11.2 DROP_POLICY

The `DROP_POLICY` procedure deletes an existing policy. The policy function and database object associated with the policy are not deleted by the `DROP_POLICY` procedure.

You must be a superuser to execute this procedure.

```
DROP_POLICY(object_schema VARCHAR2, object_name VARCHAR2,
  policy_name VARCHAR2)
```

Parameters

object_schema

Name of the schema containing the database object to which the policy applies.

object_name

Name of the database object to which the policy applies.

policy_name

Name of the policy to be deleted.

Examples

The following example deletes policy `secure_update` on table `public.vpemp`:

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_update';
BEGIN
    DBMS_RLS.DROP_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name
    );
END;
```

7.11.3 ENABLE_POLICY

The `ENABLE_POLICY` procedure enables or disables an existing policy on the specified database object.

You must be a superuser to execute this procedure.

```
ENABLE_POLICY(object_schema VARCHAR2, object_name VARCHAR2,
  policy_name VARCHAR2, enable BOOLEAN)
```

Parameters

object_schema

Name of the schema containing the database object to which the policy applies.

object_name

Name of the database object to which the policy applies.

policy_name

Name of the policy to be enabled or disabled.

enable

When set to `TRUE`, the policy is enabled. When set to `FALSE`, the policy is disabled.

Examples

The following example disables policy `secure_update` on table `public.vpemp`:

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_update';
    v_enable                BOOLEAN := FALSE;
BEGIN
    DBMS_RLS.ENABLE_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name,
        v_enable
    );
END;
```

7.12 DBMS_SCHEDULER

The `DBMS_SCHEDULER` package provides a way to create and manage Oracle-styled jobs, programs and job schedules.

Table 7.7.2 DBMS_SCHEDULER Functions and Procedures

Function/Procedure	Return Type	Description
`CREATE_JOB(`*`job_name, job_type, job_action, number_of_arguments, start_date, repeat_interval, end_date, job_class, enabled, auto_drop, comments`*`)`	n/a	Use the first form of the `CREATE_JOB` procedure to create a job, specifying program and schedule details by means of parameters.
`CREATE_JOB(`*`job_name, program_name, schedule_name, job_class, enabled, auto_drop, comments`*`)`	n/a	Use the second form of `CREATE_JOB` to create a job that uses a named program and named schedule.
`CREATE_PROGRAM(`*`program_name, program_type, program_action, number_of_arguments, enabled, comments`*`)`	n/a	Use `CREATE_PROGRAM` to create a program.
`CREATE_SCHEDULE(`*`schedule_name, start_date, repeat_interval, end_date, comments`*`)`	n/a	Use the `CREATE_SCHEDULE` procedure to create a schedule.
`DEFINE_PROGRAM_ARGUMENT(`*`program_name, argument_position, argument_name, argument_type, default_value, out_argument`*`)`	n/a	Use the first form of the `DEFINE_PROGRAM_ARGUMENT` procedure to define a program argument that has a default value.
`DEFINE_PROGRAM_ARGUMENT(`*`program_name, argument_position, argument_name, argument_type, out_argument`*`)`	n/a	Use the first form of the `DEFINE_PROGRAM_ARGUMENT` procedure to define a program argument that does not have a default value.
`DISABLE(`*`name, force, commit_semantics`*`)`	n/a	Use the `DISABLE` procedure to disable a job or program.
`DROP_JOB(`*`job_name, force, defer, commit_semantics`*`)`	n/a	Use the `DROP_JOB` procedure to drop a job.
`DROP_PROGRAM(`*`program_name, force`*`)`	n/a	Use the `DROP_PROGRAM` procedure to drop a program.
`DROP_PROGRAM_ARGUMENT(`*`program_name, argument_position`*`)`	n/a	Use the first form of `DROP_PROGRAM_ARGUMENT` to drop a program argument by specifying the argument position.
`DROP_PROGRAM_ARGUMENT(`*`program_name, argument_name`*`)`	n/a	Use the second form of `DROP_PROGRAM_ARGUMENT` to drop a program argument by specifying the argument name.
`DROP_SCHEDULE(`*`schedule_name, force`*`)`	n/a	Use the `DROP SCHEDULE` procedure to drop a schedule.
`ENABLE(`*`name, commit_semantics`*`)`	n/a	Use the `ENABLE` command to enable a program or job.
`EVALUATE_CALENDAR_STRING(`	n/a	Use `EVALUATE_CALENDAR_STRING` to review the

Function/Procedure	Return Type	Description
calendar_string, *start_date*, *return_date_after*, *next_run_date*)		execution date described by a user-defined calendar schedule.
RUN_JOB(*job_name*, *use_current_session*, *manually*)	n/a	Use the RUN_JOB procedure to execute a job immediately.
SET_JOB_ARGUMENT_VALUE(*job_name*, *argument_position*, *argument_value*)	n/a	Use the first form of SET_JOB_ARGUMENT value to set the value of a job argument described by the argument's position.
SET_JOB_ARGUMENT_VALUE(*job_name*, *argument_name*, *argument_value*)	n/a	Use the second form of SET_JOB_ARGUMENT value to set the value of a job argument described by the argument's name.

Advanced Server's implementation of DBMS_SCHEDULER is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

The DBMS_SCHEDULER package is dependent on the pgAgent service; you must have a pgAgent service installed and running on your server before using DBMS_SCHEDULER.

Before using DBMS_SCHEDULER, a database superuser must create the catalog tables in which the DBMS_SCHEDULER programs, schedules and jobs are stored. Use the psql client to connect to the database, and invoke the command:

```
CREATE EXTENSION dbms_scheduler;
```

By default, the dbms_scheduler extension resides in the contrib/dbms_scheduler_ext subdirectory (under the Advanced Server installation).

Note that after creating the DBMS_SCHEDULER tables, only a superuser will be able to perform a dump or reload of the database.

7.12.1 Using Calendar Syntax to Specify a Repeating Interval

The CREATE_JOB and CREATE_SCHEDULE procedures use Oracle-styled calendar syntax to define the interval with which a job or schedule is repeated. You should provide the scheduling information in the *repeat_interval* parameter of each procedure.

repeat_interval is a value (or series of values) that define the interval between the executions of the scheduled job. Each value is composed of a token, followed by an equal sign, followed by the unit (or units) on which the schedule will execute. Multiple token values must be separated by a semi-colon (;).

For example, the following value:

```
FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;BYMINUTE=45
```

Defines a schedule that is executed each weeknight at 5:45.

The token types and syntax described in the table below are supported by Advanced Server:

Token type	**Syntax**	**Valid Values**
`FREQ`	`FREQ=`*`predefined_interval`*	Where *`predefined_interval`* is one of the following: `YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY`. The `SECONDLY` keyword is not supported.
`BYMONTH`	`BYMONTH=`*`month`*`(, `*`month`*`)...`	Where *`month`* is the three-letter abbreviation of the month name: `JAN | FEB | MAR | APR | MAY | JUN | JUL | AUG | SEP | OCT | NOV | DEC`
`BYMONTH`	`BYMONTH=`*`month`*`(, `*`month`*`)...`	Where *`month`* is the numeric value representing the month: `1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12`
`BYMONTHDAY`	`BYMONTHDAY=`*`day_of_month`*	Where *`day_of_month`* is a value from `1` through `31`
`BYDAY`	`BYDAY=`*`weekday`*	Where *`weekday`* is a three-letter abbreviation or single-digit value representing the day of the week. Monday | `MON` | `1` Tuesday | `TUE` | `2` Wednesday | `WED` | `3` Thursday | `THU` | `4` Friday | `FRI` | `5` Saturday | `SAT` | `6` Sunday | `SUN` | `7`
`BYDATE`	`BYDATE=`*`date`*`(, `*`date`*`)...`	Where *`date`* is *YYYYMMDD*. `YYYY` is a four-digit year representation of the year, `MM` is a two-digit representation of the month, and `DD` is a two-digit day representation of the day.
`BYDATE`	`BYDATE=`*`date`*`(, `*`date`*`)...`	Where *`date`* is *MMDD*. `MM` is a two-digit representation of the month, and `DD` is a two-digit day representation of the day
`BYHOUR`	`BYHOUR=`*`hour`*	Where *`hour`* is a value from 0 through 23.
`BYMINUTE`	`BYMINUTE=`*`minute`*	Where *`minute`* is a value from 0 through 59.

7.12.2 CREATE_JOB

Use the `CREATE_JOB` procedure to create a job. The procedure comes in two forms; the first form of the procedure specifies a schedule within the job definition, as well as a job action that will be invoked when the job executes:

```
CREATE_JOB(
  job_name IN VARCHAR2,
  job_type IN VARCHAR2,
  job_action IN VARCHAR2,
  number_of_arguments IN PLS_INTEGER DEFAULT 0,
  start_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
  repeat_interval IN VARCHAR2 DEFAULT NULL,
  end_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
  job_class IN VARCHAR2 DEFAULT 'DEFAULT_JOB_CLASS',
  enabled IN BOOLEAN DEFAULT FALSE,
  auto_drop IN BOOLEAN DEFAULT TRUE,
  comments IN VARCHAR2 DEFAULT NULL)
```

The second form uses a job schedule to specify the schedule on which the job will execute, and specifies the name of a program that will execute when the job runs:

```
CREATE_JOB(
  job_name IN VARCHAR2,
  program_name IN VARCHAR2,
  schedule_name IN VARCHAR2,
  job_class IN VARCHAR2 DEFAULT 'DEFAULT_JOB_CLASS',
  enabled IN BOOLEAN DEFAULT FALSE,
  auto_drop IN BOOLEAN DEFAULT TRUE,
  comments IN VARCHAR2 DEFAULT NULL)
```

Parameters

job_name

> *job_name* specifies the optionally schema-qualified name of the job being created.

job_type

> *job_type* specifies the type of job. The current implementation of `CREATE_JOB` supports a job type of `PLSQL_BLOCK` or `STORED_PROCEDURE`.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

job_action

If *job_type* is PLSQL_BLOCK, *job_action* specifies the content of the PL/SQL block that will be invoked when the job executes. The block must be terminated with a semi-colon (;).

If *job_type* is STORED_PROCEDURE, *job_action* specifies the optionally schema-qualified name of the procedure.

number_of_arguments

number_of_arguments is an INTEGER value that specifies the number of arguments expected by the job. The default is 0.

start_date

start_date is a TIMESTAMP WITH TIME ZONE value that specifies the first time that the job is scheduled to execute. The default value is NULL, indicating that the job should be scheduled to execute when the job is enabled.

repeat_interval

repeat_interval is a VARCHAR2 value that specifies how often the job will repeat. If a *repeat_interval* is not specified, the job will execute only once. The default value is NULL.

For information about defining a repeating schedule for a job, see Section 7.12.1.

end_date

end_date is a TIMESTAMP WITH TIME ZONE value that specifies a time after which the job will no longer execute. If a date is specified, the *end_date* must be after *start_date*. The default value is NULL.

Please note that if an *end_date* is not specified and a *repeat_interval* is specified, the job will repeat indefinitely until it is disabled.

program_name

program_name is the name of a program that will be executed by the job.

schedule_name

schedule_name is the name of the schedule associated with the job.

job_class

job_class is accepted for compatibility and ignored.

enabled

enabled is a BOOLEAN value that specifies if the job is enabled when created. By default, a job is created in a disabled state, with *enabled* set to FALSE. To enable a job, specify a value of TRUE when creating the job, or enable the job with the DBMS_SCHEDULER.ENABLE procedure.

auto_drop

The *auto_drop* parameter is accepted for compatibility and is ignored. By default, a job's status will be changed to DISABLED after the time specified in *end_date*.

comments

Use the *comments* parameter to specify a comment about the job.

Example

The following example demonstrates a call to the CREATE_JOB procedure:

```
EXEC
  DBMS_SCHEDULER.CREATE_JOB (
    job_name        => 'update_log',
    job_type        => 'PLSQL_BLOCK',
    job_action      => 'BEGIN INSERT INTO my_log VALUES(current_timestamp);
                        END;',
    start_date      => '01-JUN-15 09:00:00.000000',
    repeat_interval => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',
    end_date        => NULL,
    enabled         => TRUE,
    comments        => 'This job adds a row to the my_log table.');
```

The code fragment creates a job named update_log that executes each weeknight at 5:00. The job executes a PL/SQL block that inserts the current timestamp into a logfile (my_log). Since no end_date is specified, the job will execute until it is disabled by the DBMS_SCHEDULER.DISABLE procedure.

7.12.3 CREATE_PROGRAM

Use the `CREATE_PROGRAM` procedure to create a `DBMS_SCHEDULER` program. The signature is:

```
CREATE_PROGRAM(
  program_name IN VARCHAR2,
  program_type IN VARCHAR2,
  program_action IN VARCHAR2,
  number_of_arguments IN PLS_INTEGER DEFAULT 0,
  enabled IN BOOLEAN DEFAULT FALSE,
  comments IN VARCHAR2 DEFAULT NULL)
```

Parameters

program_name

program_name specifies the name of the program that is being created.

program_type

program_type specifies the type of program. The current implementation of `CREATE_PROGRAM` supports a *program_type* of `PLSQL_BLOCK` or `PROCEDURE`.

program_action

If *program_type* is `PLSQL_BLOCK`, *program_action* contains the PL/SQL block that will execute when the program is invoked. The PL/SQL block must be terminated with a semi-colon (`;`).

If *program_type* is `PROCEDURE`, *program_action* contains the name of the stored procedure.

number_of_arguments

If *program_type* is `PLSQL_BLOCK`, this argument is ignored.

If *program_type* is `PROCEDURE`, *number_of_arguments* specifies the number of arguments required by the procedure. The default value is `0`.

enabled

enabled specifies if the program is created enabled or disabled:

- If *enabled* is `TRUE`, the program is created enabled.

- If *enabled* is FALSE, the program is created disabled; use the DBMS_SCHEDULER.ENABLE program to enable a disabled program.

The default value is FALSE.

comments

Use the *comments* parameter to specify a comment about the program; by default, this parameter is NULL.

Example

The following call to the CREATE_PROGRAM procedure creates a program named update_log:

```
EXEC
  DBMS_SCHEDULER.CREATE_PROGRAM (
    program_name     => 'update_log',
    program_type     => 'PLSQL_BLOCK',
    program_action   => 'BEGIN INSERT INTO my_log VALUES(current_timestamp);
                         END;',
    enabled          => TRUE,
    comment          => 'This program adds a row to the my_log table.');
```

update_log is a PL/SQL block that adds a row containing the current date and time to the my_log table. The program will be enabled when the CREATE_PROGRAM procedure executes.

7.12.4 CREATE_SCHEDULE

Use the `CREATE_SCHEDULE` procedure to create a job schedule. The signature of the `CREATE_SCHEDULE` procedure is:

```
CREATE_SCHEDULE(
  schedule_name IN VARCHAR2,
  start_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
  repeat_interval IN VARCHAR2,
  end_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
  comments IN VARCHAR2 DEFAULT NULL)
```

Parameters

schedule_name

schedule_name specifies the name of the schedule.

start_date

start_date is a `TIMESTAMP WITH TIME ZONE` value that specifies the date and time that the schedule is eligible to execute. If a *start_date* is not specified, the date that the job is enabled is used as the *start_date*. By default, *start_date* is `NULL`.

repeat_interval

repeat_interval is a `VARCHAR2` value that specifies how often the job will repeat. If a *repeat_interval* is not specified, the job will execute only once, on the date specified by *start_date*.

For information about defining a repeating schedule for a job, see Section 7.12.1 .

Please note: you must provide a value for either *start_date* or *repeat_interval*; if both *start_date* and *repeat_interval* are `NULL`, the server will return an error.

end_date `IN TIMESTAMP WITH TIME ZONE DEFAULT NULL`

end_date is a `TIMESTAMP WITH TIME ZONE` value that specifies a time after which the schedule will no longer execute. If a date is specified, the *end_date* must be after the *start_date*. The default value is `NULL`.

Please note that if a *repeat_interval* is specified and an *end_date* is not specified, the schedule will repeat indefinitely until it is disabled.

`comments IN VARCHAR2 DEFAULT NULL)`

Use the `comments` parameter to specify a comment about the schedule; by default, this parameter is `NULL`.

Example

The following code fragment calls `CREATE_SCHEDULE` to create a schedule named `weeknights_at_5`:

```
EXEC
  DBMS_SCHEDULER.CREATE_SCHEDULE (
    schedule_name    => 'weeknights_at_5',
    start_date       => '01-JUN-13 09:00:00.000000'
    repeat_interval  => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',
    comments         => 'This schedule executes each weeknight at 5:00');
```

The schedule executes each weeknight, at 5:00 pm, effective after June 1, 2013. Since no `end_date` is specified, the schedule will execute indefinitely until it is disabled with `DBMS_SCHEDULER.DISABLE`.

7.12.5 DEFINE_PROGRAM_ARGUMENT

Use the `DEFINE_PROGRAM_ARGUMENT` procedure to define a program argument. The `DEFINE_PROGRAM_ARGUMENT` procedure comes in two forms; the first form defines an argument with a default value:

```
DEFINE_PROGRAM_ARGUMENT(
  program_name IN VARCHAR2,
  argument_position IN PLS_INTEGER,
  argument_name IN VARCHAR2 DEFAULT NULL,
  argument_type IN VARCHAR2,
  default_value IN VARCHAR2,
  out_argument IN BOOLEAN DEFAULT FALSE)
```

The second form defines an argument without a default value:

```
DEFINE_PROGRAM_ARGUMENT(
  program_name IN VARCHAR2,
  argument_position IN PLS_INTEGER,
  argument_name IN VARCHAR2 DEFAULT NULL,
  argument_type IN VARCHAR2,
  out_argument IN BOOLEAN DEFAULT FALSE)
```

Parameters

program_name

> *program_name* is the name of the program to which the arguments belong.

argument_position

> *argument_position* specifies the position of the argument as it is passed to the program.

argument_name

> *argument_name* specifies the optional name of the argument. By default, *argument_name* is `NULL`.

argument_type `IN VARCHAR2`

> *argument_type* specifies the data type of the argument.

default_value

default_value specifies the default value assigned to the argument. *default_value* will be overridden by a value specified by the job when the job executes.

out_argument IN BOOLEAN DEFAULT FALSE

out_argument is not currently used; if specified, the value must be FALSE.

Example

The following code fragment uses the DEFINE_PROGRAM_ARGUMENT procedure to define the first and second arguments in a program named add_emp:

```
EXEC
  DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
    program_name          => 'add_emp',
    argument_position     => 1,
    argument_name         => 'dept_no',
    argument_type         => 'INTEGER,
    default_value         => '20');
EXEC
  DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
    program_name          => 'add_emp',
    argument_position     => 2,
    argument_name         => 'emp_name',
    argument_type         => 'VARCHAR2');
```

The first argument is an INTEGER value named dept_no that has a default value of 20. The second argument is a VARCHAR2 value named emp_name; the second argument does not have a default value.

7.12.6 DISABLE

Use the `DISABLE` procedure to disable a program or a job. The signature of the `DISABLE` procedure is:

```
DISABLE(
  name IN VARCHAR2,
  force IN BOOLEAN DEFAULT FALSE,
  commit_semantics IN VARCHAR2 DEFAULT 'STOP_ON_FIRST_ERROR')
```

Parameters

name

name specifies the name of the program or job that is being disabled.

force

force is accepted for compatibility, and ignored.

commit_semantics

commit_semantics instructs the server how to handle an error encountered while disabling a program or job. By default, *commit_semantics* is set to `STOP_ON_FIRST_ERROR`, instructing the server to stop when it encounters an error. Any programs or jobs that were successfully disabled prior to the error will be committed to disk.

The `TRANSACTIONAL` and `ABSORB_ERRORS` keywords are accepted for compatibility, and ignored.

Example

The following call to the `DISABLE` procedure disables a program named `update_emp`:

```
DBMS_SCHEDULER.DISABLE('update_emp');
```

7.12.7 DROP_JOB

Use the `DROP_JOB` procedure to `DROP` a job, `DROP` any arguments that belong to the job, and eliminate any future job executions. The signature of the procedure is:

```
DROP_JOB(
  job_name IN VARCHAR2,
  force IN BOOLEAN DEFAULT FALSE,
  defer IN BOOLEAN DEFAULT FALSE,
  commit_semantics IN VARCHAR2 DEFAULT 'STOP_ON_FIRST_ERROR')
```

Parameters

job_name

job_name specifies the name of the job that is being dropped.

force

force is accepted for compatibility, and ignored.

defer

defer is accepted for compatibility, and ignored.

commit_semantics

commit_semantics instructs the server how to handle an error encountered while dropping a program or job. By default, *commit_semantics* is set to `STOP_ON_FIRST_ERROR`, instructing the server to stop when it encounters an error.

The `TRANSACTIONAL` and `ABSORB_ERRORS` keywords are accepted for compatibility, and ignored.

Example

The following call to `DROP_JOB` drops a job named `update_log`:

```
DBMS_SCHEDULER.DROP_JOB('update_log');
```

7.12.8 DROP_PROGRAM

The `DROP_PROGRAM` procedure

The signature of the `DROP_PROGRAM` procedure is:

```
DROP_PROGRAM(
  program_name IN VARCHAR2,
  force IN BOOLEAN DEFAULT FALSE)
```

Parameters

program_name

program_name specifies the name of the program that is being dropped.

force

force is a `BOOLEAN` value that instructs the server how to handle programs with dependent jobs.

Specify `FALSE` to instruct the server to return an error if the program is referenced by a job.

Specify `TRUE` to instruct the server to disable any jobs that reference the program before dropping the program.

The default value is `FALSE`.

Example

The following call to `DROP_PROGRAM` drops a job named `update_emp`:

```
DBMS_SCHEDULER.DROP_PROGRAM('update_emp');
```

7.12.9 DROP_PROGRAM_ARGUMENT

Use the `DROP_PROGRAM_ARGUMENT` procedure to drop a program argument. The `DROP_PROGRAM_ARGUMENT` procedure comes in two forms; the first form uses an argument position to specify which argument to drop:

```
DROP_PROGRAM_ARGUMENT(
  program_name IN VARCHAR2,
  argument_position IN PLS_INTEGER)
```

The second form takes the argument name:

```
DROP_PROGRAM_ARGUMENT(
  program_name IN VARCHAR2,
  argument_name IN VARCHAR2)
```

Parameters

program_name

program_name specifies the name of the program that is being modified.

argument_position

argument_position specifies the position of the argument that is being dropped.

argument_name

argument_name specifies the name of the argument that is being dropped.

Examples

The following call to `DROP_PROGRAM_ARGUMENT` drops the first argument in the `update_emp` program:

```
DBMS_SCHEDULER.DROP_PROGRAM_ARGUMENT('update_emp', 1);
```

The following call to `DROP_PROGRAM_ARGUMENT` drops an argument named `emp_name`:

```
DBMS_SCHEDULER.DROP_PROGRAM_ARGUMENT(update_emp', 'emp_name');
```

7.12.10 DROP_SCHEDULE

Use the `DROP_SCHEDULE` procedure to drop a schedule. The signature is:

```
DROP_SCHEDULE(
  schedule_name IN VARCHAR2,
  force IN BOOLEAN DEFAULT FALSE)
```

Parameters

schedule_name

schedule_name specifies the name of the schedule that is being dropped.

force

force specifies the behavior of the server if the specified schedule is referenced by any job:

- Specify `FALSE` to instruct the server to return an error if the specified schedule is referenced by a job. This is the default behavior.
- Specify `TRUE` to instruct the server to disable to any jobs that use the specified schedule before dropping the schedule. Any running jobs will be allowed to complete before the schedule is dropped.

Example

The following call to `DROP_SCHEDULE` drops a schedule named `weeknights_at_5`:

```
DBMS_SCHEDULER.DROP_SCHEDULE('weeknights_at_5', TRUE);
```

The server will disable any jobs that use the schedule before dropping the schedule.

7.12.11 ENABLE

Use the `ENABLE` procedure to enable a disabled program or job.

The signature of the `ENABLE` procedure is:

```
ENABLE(
  name IN VARCHAR2,
  commit_semantics IN VARCHAR2 DEFAULT 'STOP_ON_FIRST_ERROR')
```

Parameters

name

name specifies the name of the program or job that is being enabled.

commit_semantics

commit_semantics instructs the server how to handle an error encountered while enabling a program or job. By default, *commit_semantics* is set to STOP_ON_FIRST_ERROR, instructing the server to stop when it encounters an error.

The TRANSACTIONAL and ABSORB_ERRORS keywords are accepted for compatibility, and ignored.

Example

The following call to DBMS_SCHEDULER.ENABLE enables the update_emp program:

```
DBMS_SCHEDULER.ENABLE('update_emp');
```

7.12.12 EVALUATE_CALENDAR_STRING

Use the EVALUATE_CALENDAR_STRING procedure to evaluate the *repeat_interval* value specified when creating a schedule with the CREATE_SCHEDULE procedure. The EVALUATE_CALENDAR_STRING procedure will return the date and time that a specified schedule will execute without actually scheduling the job.

The signature of the EVALUATE_CALENDAR_STRING procedure is:

```
EVALUATE_CALENDAR_STRING(
  calendar_string IN VARCHAR2,
  start_date IN TIMESTAMP WITH TIME ZONE,
  return_date_after IN TIMESTAMP WITH TIME ZONE,
  next_run_date OUT TIMESTAMP WITH TIME ZONE)
```

Parameters

calendar_string

calendar_string is the calendar string that describes a *repeat_interval* (see Section 7.12.1) that is being evaluated.

start_date IN TIMESTAMP WITH TIME ZONE

> *start_date* is the date and time after which the *repeat_interval* will become valid.

return_date_after

> Use the *return_date_after parameter* to specify the date and time that EVALUATE_CALENDAR_STRING should use as a starting date when evaluating the *repeat_interval*.
>
> For example, if you specify a *return_date_after* value of 01-APR-13 09.00.00.000000, EVALUATE_CALENDAR_STRING will return the date and time of the first iteration of the schedule after April 1st, 2013.

next_run_date OUT TIMESTAMP WITH TIME ZONE

> *next_run_date* is an OUT parameter that will contain the first occurrence of the schedule after the date specified by the *return_date_after* parameter.

Example

The following example evaluates a calendar string and returns the first date and time that the schedule will be executed after June 15, 2013:

```
DECLARE
  result     TIMESTAMP;
BEGIN

  DBMS_SCHEDULER.EVALUATE_CALENDAR_STRING
  (
    'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',
    '15-JUN-2013', NULL, result
  );

    DBMS_OUTPUT.PUT_LINE('next_run_date: ' || result);
END;
/

next_run_date: 17-JUN-13 05.00.00.000000 PM
```

June 15, 2013 is a Saturday; the schedule will not execute until Monday, June 17, 2013 at 5:00 pm.

7.12.13 RUN_JOB

Use the `RUN_JOB` procedure to execute a job immediately. The signature of the `RUN_JOB` procedure is:

```
RUN_JOB(
  job_name IN VARCHAR2,
  use_current_session IN BOOLEAN DEFAULT TRUE
```

Parameters

job_name

job_name specifies the name of the job that will execute.

use_current_session

By default, the job will execute in the current session. If specified, *use_current_session* must be set to `TRUE` ; if *use_current_session* is set to `FALSE`, Advanced Server will return an error.

Example

The following call to `RUN_JOB` executes a job named `update_log`:

```
DBMS_SCHEDULER.RUN_JOB('update_log', TRUE);
```

Passing a value of `TRUE` as the second argument instructs the server to invoke the job in the current session.

7.12.14 SET_JOB_ARGUMENT_VALUE

Use the `SET_JOB_ARGUMENT_VALUE` procedure to specify a value for an argument. The `SET_JOB_ARGUMENT_VALUE` procedure comes in two forms; the first form specifies which argument should be modified by position:

```
SET_JOB_ARGUMENT_VALUE(
  job_name IN VARCHAR2,
  argument_position IN PLS_INTEGER,
  argument_value IN VARCHAR2)
```

The second form uses an argument name to specify which argument to modify:

```
SET_JOB_ARGUMENT_VALUE(
  job_name IN VARCHAR2,
  argument_name IN VARCHAR2,
  argument_value IN VARCHAR2)
```

Argument values set by the `SET_JOB_ARGUMENT_VALUE` procedure override any values set by default.

Parameters

`job_name`

> *`job_name`* specifies the name of the job to which the modified argument belongs.

`argument_position`

> Use *`argument_position`* to specify the argument position for which the value will be set.

`argument_name`

> Use *`argument_name`* to specify the argument by name for which the value will be set.

`argument_value`

> *`argument_value`* specifies the new value of the argument.

Examples

The following example assigns a value of `30` to the first argument in the `update_emp` job:

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE('update_emp', 1, '30');
```

The following example sets the `emp_name` argument to `SMITH`:

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE('update_emp', 'emp_name', 'SMITH');
```

7.13 DBMS_SESSION

Advanced Server provides support for the `DBMS_SESSION.SET_ROLE` procedure.

Table 7.7.2 DBMS_SESSION Procedure

Function/Procedure	Return Type	Description
`SET_ROLE(`*`role_cmd`*`)`	n/a	Executes a `SET ROLE` statement followed by the string value specified in *`role_cmd`*.

Advanced Server's implementation of `DBMS_SESSION` is a partial implementation when compared to Oracle's version. Only `DBMS_SESSION.SET_ROLE` is supported.

7.13.1 SET_ROLE

The `SET_ROLE` procedure sets the current session user to the role specified in *`role_cmd`*. After invoking the `SET_ROLE` procedure, the current session will use the permissions assigned to the specified role. The signature of the procedure is:

```
SET_ROLE(role_cmd)
```

The `SET_ROLE` procedure appends the value specified for *`role_cmd`* to the `SET ROLE` statement, and then invokes the statement.

Parameters

`role_cmd`

`role_cmd` specifies a role name in the form of a string value.

Example

The following call to the `SET_ROLE` procedure invokes the `SET ROLE` command to set the identity of the current session user to `manager`:

```
edb=# exec DBMS_SESSION.SET_ROLE('manager');
```

7.14 DBMS_SQL

The DBMS_SQL package provides an application interface compatible with Oracle databases to the EnterpriseDB dynamic SQL functionality. With DBMS_SQL you can construct queries and other commands at run time (rather than when you write the application). EnterpriseDB Advanced Server offers native support for dynamic SQL; DBMS_SQL provides a way to use dynamic SQL in a fashion compatible with Oracle databases without modifying your application.

DBMS_SQL assumes the privileges of the current user when executing dynamic SQL statements.

Table 7-14 DBMS_SQL Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
BIND_VARIABLE(*c*, *name*, *value* [, *out_value_size*])	Procedure	n/a	Bind a value to a variable.
BIND_VARIABLE_CHAR(*c*, *name*, *value* [, *out_value_size*])	Procedure	n/a	Bind a CHAR value to a variable.
BIND_VARIABLE_RAW(*c*, *name*, *value* [, *out_value_size*])	Procedure	n/a	Bind a RAW value to a variable.
CLOSE_CURSOR(*c* IN OUT)	Procedure	n/a	Close a cursor.
COLUMN_VALUE(*c*, *position*, *value* OUT [, *column_error* OUT [, *actual_length* OUT]])	Procedure	n/a	Return a column value into a variable.
COLUMN_VALUE_CHAR(*c*, *position*, *value* OUT [, *column_error* OUT [, *actual_length* OUT]])	Procedure	n/a	Return a CHAR column value into a variable.
COLUMN_VALUE_RAW(*c*, *position*, *value* OUT [, *column_error* OUT [, *actual_length* OUT]])	Procedure	n/a	Return a RAW column value into a variable.
DEFINE_COLUMN(*c*, *position*, *column* [, *column_size*])	Procedure	n/a	Define a column in the SELECT list.
DEFINE_COLUMN_CHAR(*c*, *position*, *column*, *column_size*)	Procedure	n/a	Define a CHAR column in the SELECT list.
DEFINE_COLUMN_RAW(*c*, *position*, *column*, *column_size*)	Procedure	n/a	Define a RAW column in the SELECT list.
DESCRIBE_COLUMNS	Procedure	n/a	Defines columns to hold a cursor result set.
EXECUTE(*c*)	Function	INTEGER	Execute a cursor.
EXECUTE_AND_FETCH(*c* [, *exact*])	Function	INTEGER	Execute a cursor and fetch a single row.
FETCH_ROWS(*c*)	Function	INTEGER	Fetch rows from the cursor.
IS_OPEN(*c*)	Function	BOOLEAN	Check if a cursor is open.
LAST_ROW_COUNT	Function	INTEGER	Return cumulative number of rows fetched.
OPEN_CURSOR	Function	INTEGER	Open a cursor.
PARSE(*c*, *statement*, *language_flag*)	Procedure	n/a	Parse a statement.

Advanced Server's implementation of `DBMS_SQL` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

The following table lists the public variable available in the `DBMS_SQL` package.

Table 7-15 DBMS_SQL Public Variables

Public Variables	Data Type	Value	Description
`native`	`INTEGER`	1	Provided for compatibility with Oracle syntax. See DBMS_SQL.PARSE for more information.
`V6`	`INTEGER`	2	Provided for compatibility with Oracle syntax. See DBMS_SQL.PARSE for more information.
`V7`	`INTEGER`	3	Provided for compatibility with Oracle syntax. See DBMS_SQL.PARSE for more information

7.14.1 BIND_VARIABLE

The `BIND_VARIABLE` procedure provides the capability to associate a value with an `IN` or `IN OUT` bind variable in a SQL command.

```
BIND_VARIABLE(c INTEGER, name VARCHAR2,
  value { BLOB | CLOB | DATE | FLOAT | INTEGER | NUMBER |
          TIMESTAMP | VARCHAR2 }
  [, out_value_size INTEGER ])
```

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value to be assigned.

out_value_size

If *name* is an `IN OUT` variable, defines the maximum length of the output value. If not specified, the length of *value* is assumed.

Examples

The following anonymous block uses bind variables to insert a row into the `emp` table.

```
DECLARE
    curid           INTEGER;
    v_sql           VARCHAR2(150) := 'INSERT INTO emp VALUES ' ||
                        '(:p_empno, :p_ename, :p_job, :p_mgr, ' ||
                        ':p_hiredate, :p_sal, :p_comm, :p_deptno)';
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_mgr           emp.mgr%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    v_empno    := 9001;
    v_ename    := 'JONES';
    v_job      := 'SALESMAN';
    v_mgr      := 7369;
    v_hiredate := TO_DATE('13-DEC-07','DD-MON-YY');
    v_sal      := 8500.00;
    v_comm     := 1500.00;
    v_deptno   := 40;
    DBMS_SQL.BIND_VARIABLE(curid,':p_empno',v_empno);
    DBMS_SQL.BIND_VARIABLE(curid,':p_ename',v_ename);
    DBMS_SQL.BIND_VARIABLE(curid,':p_job',v_job);
    DBMS_SQL.BIND_VARIABLE(curid,':p_mgr',v_mgr);
    DBMS_SQL.BIND_VARIABLE(curid,':p_hiredate',v_hiredate);
    DBMS_SQL.BIND_VARIABLE(curid,':p_sal',v_sal);
    DBMS_SQL.BIND_VARIABLE(curid,':p_comm',v_comm);
    DBMS_SQL.BIND_VARIABLE(curid,':p_deptno',v_deptno);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;

Number of rows processed: 1
```

7.14.2 BIND_VARIABLE_CHAR

The `BIND_VARIABLE_CHAR` procedure provides the capability to associate a `CHAR` value with an `IN` or `IN OUT` bind variable in a SQL command.

```
BIND_VARIABLE_CHAR(c INTEGER, name VARCHAR2, value CHAR
  [, out_value_size INTEGER ])
```

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value of type `CHAR` to be assigned.

out_value_size

If *name* is an `IN OUT` variable, defines the maximum length of the output value. If not specified, the length of *value* is assumed.

7.14.3 BIND VARIABLE RAW

The `BIND_VARIABLE_RAW` procedure provides the capability to associate a `RAW` value with an `IN` or `IN OUT` bind variable in a SQL command.

```
BIND_VARIABLE_RAW(c INTEGER, name VARCHAR2, value RAW
  [, out_value_size INTEGER ])
```

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value of type `RAW` to be assigned.

out_value_size

If *name* is an `IN OUT` variable, defines the maximum length of the output value. If not specified, the length of *value* is assumed.

7.14.4 CLOSE_CURSOR

The `CLOSE_CURSOR` procedure closes an open cursor. The resources allocated to the cursor are released and it can no longer be used.

```
CLOSE_CURSOR(c IN OUT INTEGER)
```

Parameters

c

Cursor ID of the cursor to be closed.

Examples

The following example closes a previously opened cursor:

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
            .
            .
            .
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

7.14.5 COLUMN_VALUE

The `COLUMN_VALUE` procedure defines a variable to receive a value from a cursor.

```
COLUMN_VALUE(c INTEGER, position INTEGER, value OUT { BLOB |
  CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }
  [, column_error OUT NUMBER [, actual_length OUT INTEGER ]])
```

Parameters

c

Cursor id of the cursor returning data to the variable being defined.

position

Position within the cursor of the returned data. The first value in the cursor is position 1.

value

Variable receiving the data returned in the cursor by a prior fetch call.

column_error

Error number associated with the column, if any.

actual_length

Actual length of the data prior to any truncation.

Examples

The following example shows the portion of an anonymous block that receives the values from a cursor using the `COLUMN_VALUE` procedure.

```
DECLARE
    curid           INTEGER;
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                    'comm FROM emp';
    v_status        INTEGER;
BEGIN
            .
            .
            .
    LOOP
        v_status := DBMS_SQL.FETCH_ROWS(curid);
        EXIT WHEN v_status = 0;
        DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
        DBMS_SQL.COLUMN_VALUE(curid,2,v_ename);
        DBMS_SQL.COLUMN_VALUE(curid,3,v_hiredate);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,5,v_comm);
        DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,10) || '  ' ||
            TO_CHAR(v_hiredate,'yyyy-mm-dd') || ' ' ||
            TO_CHAR(v_sal,'9,999.99') || ' ' ||
            TO_CHAR(NVL(v_comm,0),'9,999.99'));
    END LOOP;
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

7.14.6 COLUMN_VALUE_CHAR

The COLUMN_VALUE_CHAR procedure defines a variable to receive a CHAR value from a cursor.

```
COLUMN_VALUE_CHAR(c INTEGER, position INTEGER, value OUT CHAR
  [, column_error OUT NUMBER [, actual_length OUT INTEGER ]])
```

Parameters

c

Cursor id of the cursor returning data to the variable being defined.

position

Position within the cursor of the returned data. The first value in the cursor is position 1.

value

Variable of data type CHAR receiving the data returned in the cursor by a prior fetch call.

column_error

Error number associated with the column, if any.

actual_length

Actual length of the data prior to any truncation.

7.14.7 COLUMN VALUE RAW

The `COLUMN_VALUE_RAW` procedure defines a variable to receive a `RAW` value from a cursor.

```
COLUMN_VALUE_RAW(c INTEGER, position INTEGER, value OUT RAW
  [, column_error OUT NUMBER [, actual_length OUT INTEGER ]])
```

Parameters

c

Cursor id of the cursor returning data to the variable being defined.

position

Position within the cursor of the returned data. The first value in the cursor is position 1.

value

Variable of data type `RAW` receiving the data returned in the cursor by a prior fetch call.

column_error

Error number associated with the column, if any.

actual_length

Actual length of the data prior to any truncation.

7.14.8 DEFINE_COLUMN

The `DEFINE_COLUMN` procedure defines a column or expression in the `SELECT` list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN(c INTEGER, position INTEGER, column { BLOB |
  CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }
  [, column_size INTEGER ])
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

Parameters

c

Cursor id of the cursor associated with the SELECT command.

position

Position of the column or expression in the SELECT list that is being defined.

column

A variable that is of the same data type as the column or expression in position *position* of the SELECT list.

column_size

The maximum length of the returned data. *column_size* must be specified only if *column* is VARCHAR2. Returned data exceeding *column_size* is truncated to *column_size* characters.

Examples

The following shows how the empno, ename, hiredate, sal, and comm columns of the emp table are defined with the DEFINE_COLUMN procedure.

```
DECLARE
    curid           INTEGER;
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                    'comm FROM emp';
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_ename,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_hiredate);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_sal);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_comm);
            .
            .
            .
END;
```

The following shows an alternative to the prior example that produces the exact same results. Note that the lengths of the data types are irrelevant – the empno, sal, and comm columns will still return data equivalent to NUMBER(4) and NUMBER(7,2), respectively, even though `v_num` is defined as `NUMBER(1)` (assuming the declarations in the `COLUMN_VALUE` procedure are of the appropriate maximum sizes). The `ename` column will return data up to ten characters in length as defined by the *`length`* parameter in the `DEFINE_COLUMN` call, not by the data type declaration, `VARCHAR2(1)` declared for `v_varchar`. The actual size of the returned data is dictated by the `COLUMN_VALUE` procedure.

```
DECLARE
    curid           INTEGER;
    v_num           NUMBER(1);
    v_varchar       VARCHAR2(1);
    v_date          DATE;
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                    'comm FROM emp';
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_num);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_varchar,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_date);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_num);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_num);
            .
            .
            .
END;
```

7.14.9 DEFINE_COLUMN_CHAR

The `DEFINE_COLUMN_CHAR` procedure defines a `CHAR` column or expression in the `SELECT` list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN_CHAR(c INTEGER, position INTEGER, column
CHAR, column_size INTEGER)
```

Parameters

`c`

Cursor id of the cursor associated with the `SELECT` command.

`position`

Position of the column or expression in the `SELECT` list that is being defined.

`column`

A `CHAR` variable.

column_size

The maximum length of the returned data. Returned data exceeding *column_size* is truncated to *column_size* characters.

7.14.10 DEFINE COLUMN RAW

The DEFINE_COLUMN_RAW procedure defines a RAW column or expression in the SELECT list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN_RAW(c INTEGER, position INTEGER, column RAW,
  column_size INTEGER)
```

Parameters

c

Cursor id of the cursor associated with the SELECT command.

position

Position of the column or expression in the SELECT list that is being defined.

column

A RAW variable.

column_size

The maximum length of the returned data. Returned data exceeding *column_size* is truncated to *column_size* characters.

7.14.11 DESCRIBE COLUMNS

The DESCRIBE_COLUMNS procedure describes the columns returned by a cursor.

```
DESCRIBE_COLUMNS(c INTEGER, col_cnt OUT INTEGER, desc_t OUT
  DESC_TAB);
```

Parameters

c

The cursor ID of the cursor.

col_cnt

The number of columns in cursor result set.

desc_tab

The table that contains a description of each column returned by the cursor. The descriptions are of type `DESC_REC`, and contain the following values:

Column Name	Type
`col_type`	`INTEGER`
`col_max_len`	`INTEGER`
`col_name`	`VARCHAR2(128)`
`col_name_len`	`INTEGER`
`col_schema_name`	`VARCHAR2(128)`
`col_schema_name_len`	`INTEGER`
`col_precision`	`INTEGER`
`col_scale`	`INTEGER`
`col_charsetid`	`INTEGER`
`col_charsetform`	`INTEGER`
`col_null_ok`	`BOOLEAN`

7.14.12 EXECUTE

The `EXECUTE` function executes a parsed SQL command or SPL block.

```
status INTEGER EXECUTE(c INTEGER)
```

Parameters

c

Cursor ID of the parsed SQL command or SPL block to be executed.

status

Number of rows processed if the SQL command was `DELETE`, `INSERT`, or `UPDATE`. *status* is meaningless for all other commands.

Examples

The following anonymous block inserts a row into the `dept` table.

```
DECLARE
    curid           INTEGER;
    v_sql           VARCHAR2(50);
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    v_sql := 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

7.14.13 EXECUTE_AND_FETCH

Function `EXECUTE_AND_FETCH` executes a parsed `SELECT` command and fetches one row.

```
status INTEGER EXECUTE_AND_FETCH(c INTEGER
  [, exact BOOLEAN ])
```

Parameters

c

Cursor id of the cursor for the `SELECT` command to be executed.

exact

If set to `TRUE`, an exception is thrown if the number of rows in the result set is not exactly equal to 1. If set to `FALSE`, no exception is thrown. The default is `FALSE`. A `NO_DATA_FOUND` exception is thrown if *exact* is `TRUE` and there are no rows in the result set. A `TOO_MANY_ROWS` exception is thrown if *exact* is `TRUE` and there is more than one row in the result set.

status

Returns 1 if a row was successfully fetched, 0 if no rows to fetch. If an exception is thrown, no value is returned.

Examples

The following stored procedure uses the `EXECUTE_AND_FETCH` function to retrieve one employee using the employee's name. An exception will be thrown if the employee is not found, or there is more than one employee with the same name.

```
CREATE OR REPLACE PROCEDURE select_by_name(
    p_ename         emp.ename%TYPE
)
IS
```

```
    curid           INTEGER;
    v_empno         emp.empno%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_dname         dept.dname%TYPE;
    v_disp_date     VARCHAR2(10);
    v_sql           VARCHAR2(120) := 'SELECT empno, hiredate, sal, ' ||
                                     'NVL(comm, 0), dname ' ||
                                     'FROM emp e, dept d ' ||
                                     'WHERE ename = :p_ename ' ||
                                     'AND e.deptno = d.deptno';
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.BIND_VARIABLE(curid,':p_ename',UPPER(p_ename));
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_hiredate);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_sal);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_comm);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_dname,14);
    v_status := DBMS_SQL.EXECUTE_AND_FETCH(curid,TRUE);
    DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
    DBMS_SQL.COLUMN_VALUE(curid,2,v_hiredate);
    DBMS_SQL.COLUMN_VALUE(curid,3,v_sal);
    DBMS_SQL.COLUMN_VALUE(curid,4,v_comm);
    DBMS_SQL.COLUMN_VALUE(curid,5,v_dname);
    v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
    DBMS_OUTPUT.PUT_LINE('Number    : ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name      : ' || UPPER(p_ename));
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
    DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
    DBMS_OUTPUT.PUT_LINE('Department: ' || v_dname);
    DBMS_SQL.CLOSE_CURSOR(curid);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_ename || ' not found');
        DBMS_SQL.CLOSE_CURSOR(curid);
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('Too many employees named, ' ||
            p_ename || ', found');
        DBMS_SQL.CLOSE_CURSOR(curid);
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        DBMS_SQL.CLOSE_CURSOR(curid);
END;

EXEC select_by_name('MARTIN')

Number    : 7654
Name      : MARTIN
Hire Date : 09/28/1981
Salary    : 1250
Commission: 1400
Department: SALES
```

7.14.14 FETCH_ROWS

The `FETCH_ROWS` function retrieves a row from a cursor.

```
status INTEGER FETCH_ROWS(c INTEGER)
```

Parameters

c

Cursor ID of the cursor from which to fetch a row.

status

Returns `1` if a row was successfully fetched, `0` if no more rows to fetch.

Examples

The following examples fetches the rows from the `emp` table and displays the results.

```
DECLARE
    curid           INTEGER;
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                    'comm FROM emp';
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_ename,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_hiredate);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_sal);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_comm);

    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      HIREDATE    SAL       COMM');
    DBMS_OUTPUT.PUT_LINE('-----  ---------- ----------  --------  ' ||
        '--------');
    LOOP
        v_status := DBMS_SQL.FETCH_ROWS(curid);
        EXIT WHEN v_status = 0;
        DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
        DBMS_SQL.COLUMN_VALUE(curid,2,v_ename);
        DBMS_SQL.COLUMN_VALUE(curid,3,v_hiredate);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,5,v_comm);
        DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,10) || '  ' ||
            TO_CHAR(v_hiredate,'yyyy-mm-dd') || ' ' ||
            TO_CHAR(v_sal,'9,999.99') || ' ' ||
            TO_CHAR(NVL(v_comm,0),'9,999.99'));
    END LOOP;
    DBMS_SQL.CLOSE_CURSOR(curid);
END;

EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00       .00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00       .00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
7698   BLAKE       1981-05-01  2,850.00       .00
7782   CLARK       1981-06-09  2,450.00       .00
7788   SCOTT       1987-04-19  3,000.00       .00
7839   KING        1981-11-17  5,000.00       .00
7844   TURNER      1981-09-08  1,500.00       .00
7876   ADAMS       1987-05-23  1,100.00       .00
7900   JAMES       1981-12-03    950.00       .00
7902   FORD        1981-12-03  3,000.00       .00
7934   MILLER      1982-01-23  1,300.00       .00
```

7.14.15 IS_OPEN

The `IS_OPEN` function provides the capability to test if the given cursor is open.

```
status BOOLEAN IS_OPEN(c INTEGER)
```

Parameters

c

Cursor ID of the cursor to be tested.

status

Set to `TRUE` if the cursor is open, set to `FALSE` if the cursor is not open.

7.14.16 LAST_ROW_COUNT

The `LAST_ROW_COUNT` function returns the number of rows that have been currently fetched.

```
rowcnt INTEGER LAST_ROW_COUNT
```

Parameters

rowcnt

Number of row fetched thus far.

Examples

The following example uses the `LAST_ROW_COUNT` function to display the total number of rows fetched in the query.

```
DECLARE
    curid           INTEGER;
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                    'comm FROM emp';
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_ename,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_hiredate);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_sal);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_comm);

    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL       COMM');
    DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------  ' ||
        '--------');
    LOOP
        v_status := DBMS_SQL.FETCH_ROWS(curid);
        EXIT WHEN v_status = 0;
        DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
        DBMS_SQL.COLUMN_VALUE(curid,2,v_ename);
        DBMS_SQL.COLUMN_VALUE(curid,3,v_hiredate);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,5,v_comm);
        DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,10) || '  ' ||
            TO_CHAR(v_hiredate,'yyyy-mm-dd') || ' ' ||
            TO_CHAR(v_sal,'9,999.99') || ' ' ||
            TO_CHAR(NVL(v_comm,0),'9,999.99'));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Number of rows: ' || DBMS_SQL.LAST_ROW_COUNT);
    DBMS_SQL.CLOSE_CURSOR(curid);
```

```
END;

EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00       .00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00       .00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
7698   BLAKE       1981-05-01  2,850.00       .00
7782   CLARK       1981-06-09  2,450.00       .00
7788   SCOTT       1987-04-19  3,000.00       .00
7839   KING        1981-11-17  5,000.00       .00
7844   TURNER      1981-09-08  1,500.00       .00
7876   ADAMS       1987-05-23  1,100.00       .00
7900   JAMES       1981-12-03    950.00       .00
7902   FORD        1981-12-03  3,000.00       .00
7934   MILLER      1982-01-23  1,300.00       .00
Number of rows: 14
```

7.14.17 OPEN_CURSOR

The `OPEN_CURSOR` function creates a new cursor. A cursor must be used to parse and execute any dynamic SQL statement. Once a cursor has been opened, it can be re-used with the same or different SQL statements. The cursor does not have to be closed and re-opened in order to be re-used.

```
c INTEGER OPEN_CURSOR
```

Parameters

c

Cursor ID number associated with the newly created cursor.

Examples

The following example creates a new cursor:

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
            .
            .
            .
END;
```

7.14.18 PARSE

The `PARSE` procedure parses a SQL command or SPL block. If the SQL command is a DDL command, it is immediately executed and does not require running the `EXECUTE` function.

```
PARSE(c INTEGER, statement VARCHAR2, language_flag INTEGER)
```

Parameters

c

Cursor ID of an open cursor.

statement

SQL command or SPL block to be parsed. A SQL command must not end with the semi-colon terminator, however an SPL block does require the semi-colon terminator.

language_flag

Language flag provided for compatibility with Oracle syntax. Use `DBMS_SQL.V6`, `DBMS_SQL.V7` or `DBMS_SQL.native`. This flag is ignored, and all syntax is assumed to be in EnterpriseDB Advanced Server form.

Examples

The following anonymous block creates a table named, `job`. Note that DDL statements are executed immediately by the `PARSE` procedure and do not require a separate `EXECUTE` step.

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid, 'CREATE TABLE job (jobno NUMBER(3), ' ||
        'jname VARCHAR2(9))',DBMS_SQL.native);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

The following inserts two rows into the `job` table.

```
DECLARE
    curid           INTEGER;
    v_sql           VARCHAR2(50);
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    v_sql := 'INSERT INTO job VALUES (100, ''ANALYST'')';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    v_sql := 'INSERT INTO job VALUES (200, ''CLERK'')';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;

Number of rows processed: 1
Number of rows processed: 1
```

The following anonymous block uses the `DBMS_SQL` package to execute a block containing two `INSERT` statements. Note that the end of the block contains a terminating semi-colon, while in the prior example, each individual `INSERT` statement does not have a terminating semi-colon.

```
DECLARE
    curid           INTEGER;
    v_sql           VARCHAR2(100);
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    v_sql := 'BEGIN ' ||
               'INSERT INTO job VALUES (300, ''MANAGER''); '  ||
               'INSERT INTO job VALUES (400, ''SALESMAN''); ' ||
             'END;';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

7.15 DBMS_UTILITY

The `DBMS_UTILITY` package provides various utility programs.

Table 7-16 DBMS_UTILITY Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
`ANALYZE_DATABASE(`*`method`*` [, `*`estimate_rows`*` [, `*`estimate_percent`*` [, `*`method_opt`*` ]]])`	Procedure	n/a	Analyze database tables.
`ANALYZE_PART_OBJECT(`*`schema`*`, `*`object_name`*` [, `*`object_type`*` [, `*`command_type`*` [, `*`command_opt`*` [, `*`sample_clause`*` ]]]])`	Procedure	n/a	Analyze a partitioned table.
`ANALYZE_SCHEMA(`*`schema`*`, `*`method`*` [, `*`estimate_rows`*` [, `*`estimate_percent`*` [, `*`method_opt`*` ]]])`	Procedure	n/a	Analyze schema tables.
`CANONICALIZE(`*`name`*`, `*`canon_name`*` OUT, `*`canon_len`*`)`	Procedure	n/a	Canonicalizes a string – e.g., strips off white space.
`COMMA_TO_TABLE(`*`list`*`, `*`tablen`*` OUT, `*`tab`*` OUT)`	Procedure	n/a	Convert a comma-delimited list of names to a table of names.
`DB_VERSION(`*`version`*` OUT, `*`compatibility`*` OUT)`	Procedure	n/a	Get the database version.
`EXEC_DDL_STATEMENT(`*`parse_string`*`)`	Procedure	n/a	Execute a DDL statement.
`FORMAT_CALL_STACK`	Function	`TEXT`	Formats the current call stack.
`GET_CPU_TIME`	Function	`NUMBER`	Get the current CPU time.
`GET_DEPENDENCY(`*`type`*`, `*`schema`*`, `*`name`*`)`	Procedure	n/a	Get objects that are dependent upon the given object..
`GET_HASH_VALUE(`*`name`*`, `*`base`*`, `*`hash_size`*`)`	Function	`NUMBER`	Compute a hash value.
`GET_PARAMETER_VALUE(`*`parnam`*`, `*`intval`*` OUT, `*`strval`*` OUT)`	Procedure	`BINARY_INTEGER`	Get database initialization parameter settings.
`GET_TIME`	Function	`NUMBER`	Get the current time.
`NAME_TOKENIZE(`*`name`*`, `*`a`*` OUT, `*`b`*` OUT, `*`c`*` OUT, `*`dblink`*` OUT, `*`nextpos`*` OUT)`	Procedure	n/a	Parse the given name into its component parts.
`TABLE_TO_COMMA(`*`tab`*`, `*`tablen`*` OUT, `*`list`*` OUT)`	Procedure	n/a	Convert a table of names to a comma-delimited list.

Advanced Server's implementation of `DBMS_UTILITY` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

The following table lists the public variables available in the `DBMS_UTILITY` package.

Table 7-17 DBMS_UTILITY Public Variables

Public Variables	Data Type	Value	Description
`inv_error_on_restrictions`	`PLS_INTEGER`	`1`	Used by the `INVALIDATE` procedure.

Public Variables	Data Type	Value	Description
lname_array	TABLE		For lists of long names.
uncl_array	TABLE		For lists of users and names.

7.15.1 LNAME_ARRAY

The LNAME_ARRAY is for storing lists of long names including fully-qualified names.

```
TYPE lname_array IS TABLE OF VARCHAR2(4000) INDEX BY BINARY_INTEGER;
```

7.15.2 UNCL_ARRAY

The UNCL_ARRAY is for storing lists of users and names.

```
TYPE uncl_array IS TABLE OF VARCHAR2(227) INDEX BY BINARY_INTEGER;
```

7.15.3 ANALYZE_DATABASE, ANALYZE SCHEMA and ANALYZE PART_OBJECT

The ANALYZE_DATABASE(), ANALYZE_SCHEMA() and ANALYZE_PART_OBJECT() procedures provide the capability to gather statistics on tables in the database. When you execute the ANALYZE statement, Postgres samples the data in a table and records distribution statistics in the pg_statistics system table.

ANALYZE_DATABASE, ANALYZE_SCHEMA, and ANALYZE_PART_OBJECT differ primarily in the number of tables that are processed:

- ANALYZE_DATABASE analyzes all tables in all schemas within the current database.
- ANALYZE_SCHEMA analyzes all tables in a given schema (within the current database).
- ANALYZE_PART_OBJECT analyzes a single table.

The syntax for the ANALYZE commands are:

```
ANALYZE_DATABASE(method VARCHAR2 [, estimate_rows NUMBER
  [, estimate_percent NUMBER [, method_opt VARCHAR2 ]]])

ANALYZE_SCHEMA(schema VARCHAR2, method VARCHAR2
  [, estimate_rows NUMBER [, estimate_percent NUMBER
  [, method_opt VARCHAR2 ]]])

ANALYZE_PART_OBJECT(schema VARCHAR2, object_name VARCHAR2
  [, object_type CHAR [, command_type CHAR
  [, command_opt VARCHAR2 [, sample_clause ]]]])
```

Parameters - `ANALYZE_DATABASE` and `ANALYZE_SCHEMA`

`method`

> `method` determines whether the `ANALYZE` procedure populates the `pg_statistics` table or removes entries from the `pg_statistics` table. If you specify a method of `DELETE`, the `ANALYZE` procedure removes the relevant rows from `pg_statistics`. If you specify a method of `COMPUTE` or `ESTIMATE`, the `ANALYZE` procedure analyzes a table (or multiple tables) and records the distribution information in `pg_statistics`. There is no difference between `COMPUTE` and `ESTIMATE;` both methods execute the Postgres `ANALYZE` statement. All other parameters are validated and then ignored.

`estimate_rows`

> Number of rows upon which to base estimated statistics. One of *`estimate_rows`* or *`estimate_percent`* must be specified if method is `ESTIMATE`.
>
> This argument is ignored, but is included for compatibility.

`estimate_percent`

> Percentage of rows upon which to base estimated statistics. One of *`estimate_rows`* or *`estimate_percent`* must be specified if method is `ESTIMATE`.
>
> This argument is ignored, but is included for compatibility.

`method_opt`

> Object types to be analyzed. Any combination of the following:

```
[ FOR TABLE ]
[ FOR ALL [ INDEXED ] COLUMNS ] [ SIZE n ]
[ FOR ALL INDEXES ]
```

> This argument is ignored, but is included for compatibility.

Parameters - `ANALYZE_PART_OBJECT`

`schema`

> Name of the schema whose objects are to be analyzed.

object_name

Name of the partitioned object to be analyzed.

object_type

Type of object to be analyzed. Valid values are: `T` – table, `I` – index.

This argument is ignored, but is included for compatibility.

command_type

Type of analyze functionality to perform. Valid values are: `E` - gather estimated statistics based upon on a specified number of rows or a percentage of rows in the *sample_clause* clause; `C` - compute exact statistics; or `V` – validate the structure and integrity of the partitions.

This argument is ignored, but is included for compatibility.

command_opt

For *command_type* `C` or `E`, can be any combination of:

```
[ FOR TABLE ]
[ FOR ALL COLUMNS ]
[ FOR ALL LOCAL INDEXES ]
```

For *command_type* `V`, can be `CASCADE` if *object_type* is `T`.

This argument is ignored, but is included for compatibility.

sample_clause

If *command_type* is `E`, contains the following clause to specify the number of rows or percentage or rows on which to base the estimate.

```
SAMPLE n { ROWS | PERCENT }
```

This argument is ignored, but is included for compatibility.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

7.15.4 CANONICALIZE

The `CANONICALIZE` procedure performs the following operations on an input string:

- If the string is not double-quoted, verifies that it uses the characters of a legal identifier. If not, an exception is thrown. If the string is double-quoted, all characters are allowed.
- If the string is not double-quoted and does not contain periods, uppercases all alphabetic characters and eliminates leading and trailing spaces.
- If the string is double-quoted and does not contain periods, strips off the double quotes.
- If the string contains periods and no portion of the string is double-quoted, uppercases each portion of the string and encloses each portion in double quotes.
- If the string contains periods and portions of the string are double-quoted, returns the double-quoted portions unchanged including the double quotes and returns the non-double-quoted portions uppercased and enclosed in double quotes.

```
CANONICALIZE(name VARCHAR2, canon_name OUT VARCHAR2,
  canon_len BINARY_INTEGER)
```

Parameters

name

String to be canonicalized.

canon_name

The canonicalized string.

canon_len

Number of bytes in *name* to canonicalize starting from the first character.

Examples

The following procedure applies the `CANONICALIZE` procedure on its input parameter and displays the results.

```
CREATE OR REPLACE PROCEDURE canonicalize (
    p_name      VARCHAR2,
    p_length    BINARY_INTEGER DEFAULT 30
)
```

```
IS
    v_canon     VARCHAR2(100);
BEGIN
    DBMS_UTILITY.CANONICALIZE(p_name,v_canon,p_length);
    DBMS_OUTPUT.PUT_LINE('Canonicalized name ==>' || v_canon || '<==');
    DBMS_OUTPUT.PUT_LINE('Length: ' || LENGTH(v_canon));
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

EXEC canonicalize('Identifier')
Canonicalized name ==>IDENTIFIER<==
Length: 10

EXEC canonicalize('"Identifier"')
Canonicalized name ==>Identifier<==
Length: 10

EXEC canonicalize('"_+142%"')
Canonicalized name ==>_+142%<==
Length: 6

EXEC canonicalize('abc.def.ghi')
Canonicalized name ==>"ABC"."DEF"."GHI"<==
Length: 17

EXEC canonicalize('"abc.def.ghi"')
Canonicalized name ==>abc.def.ghi<==
Length: 11

EXEC canonicalize('"abc".def."ghi"')
Canonicalized name ==>"abc"."DEF"."ghi"<==
Length: 17

EXEC canonicalize('"abc.def".ghi')
Canonicalized name ==>"abc.def"."GHI"<==
Length: 15
```

7.15.5 COMMA_TO_TABLE

The `COMMA_TO_TABLE` procedure converts a comma-delimited list of names into a table of names. Each entry in the list becomes a table entry. The names must be formatted as valid identifiers.

```
COMMA_TO_TABLE(list VARCHAR2, tablen OUT BINARY_INTEGER,
  tab OUT { LNAME_ARRAY | UNCL_ARRAY })
```

Parameters

list

Comma-delimited list of names.

tablen

Number of entries in *tab*.

tab

Table containing the individual names in *list*.

LNAME_ARRAY

A DBMS_UTILITY LNAME_ARRAY (as described in Section 7.9.1).

UNCL_ARRAY

A DBMS_UTILITY UNCL_ARRAY (as described in Section 7.9.2).

Examples

The following procedure uses the `COMMA_TO_TABLE` procedure to convert a list of names to a table. The table entries are then displayed.

```
CREATE OR REPLACE PROCEDURE comma_to_table (
    p_list      VARCHAR2
)
IS
    r_lname     DBMS_UTILITY.LNAME_ARRAY;
    v_length    BINARY_INTEGER;
BEGIN
    DBMS_UTILITY.COMMA_TO_TABLE(p_list,v_length,r_lname);
    FOR i IN 1..v_length LOOP
        DBMS_OUTPUT.PUT_LINE(r_lname(i));
    END LOOP;
END;
```

```
EXEC comma_to_table('edb.dept, edb.emp, edb.jobhist')

edb.dept
edb.emp
edb.jobhist
```

7.15.6 DB_VERSION

The `DB_VERSION` procedure returns the version number of the database.

```
DB_VERSION(version OUT VARCHAR2, compatibility OUT
VARCHAR2)
```

Parameters

`version`

Database version number.

`compatibility`

Compatibility setting of the database. (To be implementation-defined as to its meaning.)

Examples

The following anonymous block displays the database version information.

```
DECLARE
    v_version       VARCHAR2(150);
    v_compat        VARCHAR2(150);
BEGIN
    DBMS_UTILITY.DB_VERSION(v_version,v_compat);
    DBMS_OUTPUT.PUT_LINE('Version: '       || v_version);
    DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END;

Version: EnterpriseDB 9.5.0.0 on i686-pc-linux-gnu, compiled by GCC gcc (GCC)
4.1.2 20080704 (Red Hat 4.1.2-48), 32-bit
Compatibility: EnterpriseDB 9.5.0.0 on i686-pc-linux-gnu, compiled by GCC gcc
(GCC) 4.1.220080704 (Red Hat 4.1.2-48), 32-bit
```

7.15.7 EXEC_DDL_STATEMENT

The `EXEC_DDL_STATEMENT` provides the capability to execute a DDL command.

```
EXEC_DDL_STATEMENT(parse_string VARCHAR2)
```

Parameters

parse_string

The DDL command to be executed.

Examples

The following anonymous block creates the `job` table.

```
BEGIN
    DBMS_UTILITY.EXEC_DDL_STATEMENT(
        'CREATE TABLE job (' ||
          'jobno NUMBER(3),' ||
          'jname VARCHAR2(9))'
    );
END;
```

If the *parse_string* does not include a valid DDL statement, Advanced Server returns the following error:

```
edb=#  exec dbms_utility.exec_ddl_statement('select rownum from dual');
ERROR:  EDB-20001: 'parse_string' must be a valid DDL statement
```

In this case, Advanced Server's behavior differs from Oracle's; Oracle accepts the invalid *parse_string* without complaint.

7.15.8 FORMAT_CALL_STACK

The `FORMAT_CALL_STACK` function returns the formatted contents of the current call stack.

```
DBMS_UTILITY.FORMAT_CALL_STACK
return VARCHAR2
```

This function can be used in a stored procedure, function or package to return the current call stack in a readable format. This function is useful for debugging purposes.

7.15.9 GET_CPU_TIME

The `GET_CPU_TIME` function returns the CPU time in hundredths of a second from some arbitrary point in time.

```
cputime NUMBER GET_CPU_TIME
```

Parameters

`cputime`

Number of hundredths of a second of CPU time.

Examples

The following `SELECT` command retrieves the current CPU time, which is 603 hundredths of a second or .0603 seconds.

```
SELECT DBMS_UTILITY.GET_CPU_TIME FROM DUAL;

get_cpu_time
--------------
          603
```

7.15.10 GET_DEPENDENCY

The `GET_DEPENDENCY` procedure provides the capability to list the objects that are dependent upon the specified object. `GET_DEPENDENCY` does not show dependencies for functions or procedures.

```
GET_DEPENDENCY(type VARCHAR2, schema VARCHAR2,
  name VARCHAR2)
```

Parameters

`type`

The object type of *`name`*. Valid values are `INDEX`, `PACKAGE`, `PACKAGE BODY`, `SEQUENCE`, `TABLE`, `TRIGGER`, `TYPE` and `VIEW`.

`schema`

Name of the schema in which *`name`* exists.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

name

Name of the object for which dependencies are to be obtained.

Examples

The following anonymous block finds dependencies on the `EMP` table.

```
BEGIN
    DBMS_UTILITY.GET_DEPENDENCY('TABLE','public','EMP');
END;

DEPENDENCIES ON public.EMP
------------------------------------------------------------------
*TABLE public.EMP()
*   CONSTRAINT c public.emp()
*   CONSTRAINT f public.emp()
*   CONSTRAINT p public.emp()
*   TYPE public.emp()
*   CONSTRAINT c public.emp()
*   CONSTRAINT f public.jobhist()
*   VIEW .empname_view()
```

7.15.11 GET_HASH_VALUE

The `GET_HASH_VALUE` function provides the capability to compute a hash value for a given string.

```
hash NUMBER GET_HASH_VALUE(name VARCHAR2, base NUMBER,
  hash_size NUMBER)
```

Parameters

name

The string for which a hash value is to be computed.

base

Starting value at which hash values are to be generated.

hash_size

The number of hash values for the desired hash table.

hash

The generated hash value.

Examples

The following anonymous block creates a table of hash values using the `ename` column of the `emp` table and then displays the key along with the hash value. The hash values start at 100 with a maximum of 1024 distinct values.

```
DECLARE
    v_hash          NUMBER;
    TYPE hash_tab IS TABLE OF NUMBER INDEX BY VARCHAR2(10);
    r_hash          HASH_TAB;
    CURSOR emp_cur IS SELECT ename FROM emp;
BEGIN
    FOR r_emp IN emp_cur LOOP
        r_hash(r_emp.ename) :=
            DBMS_UTILITY.GET_HASH_VALUE(r_emp.ename,100,1024);
    END LOOP;
    FOR r_emp IN emp_cur LOOP
        DBMS_OUTPUT.PUT_LINE(RPAD(r_emp.ename,10) || ' ' ||
            r_hash(r_emp.ename));
    END LOOP;
END;

SMITH      377
ALLEN      740
WARD       718
JONES      131
MARTIN     176
BLAKE      568
CLARK      621
SCOTT      1097
KING       235
TURNER     850
ADAMS      156
JAMES      942
FORD       775
MILLER     148
```

7.15.12 GET_PARAMETER_VALUE

The `GET_PARAMETER_VALUE` procedure provides the capability to retrieve database initialization parameter settings.

```
status BINARY_INTEGER GET_PARAMETER_VALUE(parnam VARCHAR2,
  intval OUT INTEGER, strval OUT VARCHAR2)
```

Parameters

parnam

Name of the parameter whose value is to be returned. The parameters are listed in the `pg_settings` system view.

intval

Value of an integer parameter or the length of *strval*.

strval

Value of a string parameter.

status

Returns 0 if the parameter value is `INTEGER` or `BOOLEAN`. Returns 1 if the parameter value is a string.

Examples

The following anonymous block shows the values of two initialization parameters.

```
DECLARE
    v_intval        INTEGER;
    v_strval        VARCHAR2(80);
BEGIN
    DBMS_UTILITY.GET_PARAMETER_VALUE('max_fsm_pages', v_intval, v_strval);
    DBMS_OUTPUT.PUT_LINE('max_fsm_pages' || ': ' || v_intval);
    DBMS_UTILITY.GET_PARAMETER_VALUE('client_encoding', v_intval, v_strval);
    DBMS_OUTPUT.PUT_LINE('client_encoding' || ': ' || v_strval);
END;

max_fsm_pages: 72625
client_encoding: SQL_ASCII
```

7.15.13 GET_TIME

The `GET_TIME` function provides the capability to return the current time in hundredths of a second.

```
time NUMBER GET_TIME
```

Parameters

time

Number of hundredths of a second from the time in which the program is started.

Examples

The following example shows calls to the `GET_TIME` function.

```
SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
----------
  1555860

SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
----------
  1556037
```

7.15.14 NAME_TOKENIZE

The `NAME_TOKENIZE` procedure parses a name into its component parts. Names without double quotes are uppercased. The double quotes are stripped from names with double quotes.

```
NAME_TOKENIZE(name VARCHAR2, a OUT VARCHAR2,
  b OUT VARCHAR2,c OUT VARCHAR2, dblink OUT VARCHAR2,
  nextpos OUT BINARY_INTEGER)
```

Parameters

name

String containing a name in the following format:

```
a[.b[.c]][@dblink ]
```

a

Returns the leftmost component.

b

Returns the second component, if any.

c

Returns the third component, if any.

dblink

Returns the database link name.

nextpos

Position of the last character parsed in name.

Examples

The following stored procedure is used to display the returned parameter values of the `NAME_TOKENIZE` procedure for various names.

```
CREATE OR REPLACE PROCEDURE name_tokenize (
    p_name          VARCHAR2
)
```

```
IS
    v_a               VARCHAR2(30);
    v_b               VARCHAR2(30);
    v_c               VARCHAR2(30);
    v_dblink          VARCHAR2(30);
    v_nextpos         BINARY_INTEGER;
BEGIN
    DBMS_UTILITY.NAME_TOKENIZE(p_name,v_a,v_b,v_c,v_dblink,v_nextpos);
    DBMS_OUTPUT.PUT_LINE('name    : ' || p_name);
    DBMS_OUTPUT.PUT_LINE('a       : ' || v_a);
    DBMS_OUTPUT.PUT_LINE('b       : ' || v_b);
    DBMS_OUTPUT.PUT_LINE('c       : ' || v_c);
    DBMS_OUTPUT.PUT_LINE('dblink  : ' || v_dblink);
    DBMS_OUTPUT.PUT_LINE('nextpos: ' || v_nextpos);
END;
```

Tokenize the name, `emp`:

```
BEGIN
    name_tokenize('emp');
END;

name    : emp
a       : EMP
b       :
c       :
dblink  :
nextpos: 3
```

Tokenize the name, `edb.list_emp`:

```
BEGIN
    name_tokenize('edb.list_emp');
END;

name    : edb.list_emp
a       : EDB
b       : LIST_EMP
c       :
dblink  :
nextpos: 12
```

Tokenize the name, `"edb"."Emp_Admin".update_emp_sal`:

```
BEGIN
    name_tokenize('"edb"."Emp_Admin".update_emp_sal');
END;

name    : "edb"."Emp_Admin".update_emp_sal
a       : edb
b       : Emp_Admin
c       : UPDATE_EMP_SAL
dblink  :
nextpos: 32
```

Tokenize the name `edb.emp@edb_dblink`:

```
BEGIN
```

```
    name_tokenize('edb.emp@edb_dblink');
END;

name    : edb.emp@edb_dblink
a       : EDB
b       : EMP
c       :
dblink  : EDB_DBLINK
nextpos: 18
```

7.15.15 TABLE_TO_COMMA

The `TABLE_TO_COMMA` procedure converts table of names into a comma-delimited list of names. Each table entry becomes a list entry. The names must be formatted as valid identifiers.

```
TABLE_TO_COMMA(tab { LNAME_ARRAY | UNCL_ARRAY },
  tablen OUT BINARY_INTEGER, list OUT VARCHAR2)
```

Parameters

tab

Table containing names.

LNAME_ARRAY

A DBMS_UTILITY LNAME_ARRAY (as described in Section 7.9.1).

UNCL_ARRAY

A DBMS_UTILITY UNCL_ARRAY (as described in Section 7.9.2).

tablen

Number of entries in *list*.

list

Comma-delimited list of names from *tab*.

Examples

The following example first uses the `COMMA_TO_TABLE` procedure to convert a comma-delimited list to a table. The `TABLE_TO_COMMA` procedure then converts the table back to a comma-delimited list that is displayed.

```
CREATE OR REPLACE PROCEDURE table_to_comma (
    p_list      VARCHAR2
)
IS
    r_lname     DBMS_UTILITY.LNAME_ARRAY;
    v_length    BINARY_INTEGER;
    v_listlen   BINARY_INTEGER;
    v_list      VARCHAR2(80);
BEGIN
    DBMS_UTILITY.COMMA_TO_TABLE(p_list,v_length,r_lname);
    DBMS_OUTPUT.PUT_LINE('Table Entries');
    DBMS_OUTPUT.PUT_LINE('-------------');
    FOR i IN 1..v_length LOOP
        DBMS_OUTPUT.PUT_LINE(r_lname(i));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('-------------');
    DBMS_UTILITY.TABLE_TO_COMMA(r_lname,v_listlen,v_list);
    DBMS_OUTPUT.PUT_LINE('Comma-Delimited List: ' || v_list);
END;

EXEC table_to_comma('edb.dept, edb.emp, edb.jobhist')

Table Entries
-------------
edb.dept
edb.emp
edb.jobhist
-------------
Comma-Delimited List: edb.dept, edb.emp, edb.jobhist
```

7.16 UTL_ENCODE

The `UTL_ENCODE` package provides a way to encode and decode data.

Table 7.7.2 UTL_ENCODE Functions and Procedures

Function/Procedure	Return Type	Description
`BASE64_DECODE(`*`r`*`)`	`RAW`	Use the `BASE64_DECODE` function to translate a Base64 encoded string to the original `RAW` value.
`BASE64_ENCODE(`*`r`*`)`	`RAW`	Use the `BASE64_ENCODE` function to translate a `RAW` string to an encoded Base64 value.
`BASE64_ENCODE(`*`loid`*`)`	`TEXT`	Use the `BASE64_ENCODE` function to translate a `TEXT` string to an encoded Base64 value.
`MIMEHEADER_DECODE(`*`buf`*`)`	`VARCHAR2`	Use the `MIMEHEADER_DECODE` function to translate an encoded `MIMEHEADER` formatted string to it's original value.
`MIMEHEADER_ENCODE(`*`buf`*`, `*`encode_charset`*`, `*`encoding`*`)`	`VARCHAR2`	Use the `MIMEHEADER_ENCODE` function to convert and encode a string in `MIMEHEADER` format.
`QUOTED_PRINTABLE_DECODE(`*`r`*`)`	`RAW`	Use the `QUOTED_PRINTABLE_DECODE` function to translate an encoded string to a `RAW` value.
`QUOTED_PRINTABLE_ENCODE(`*`r`*`)`	`RAW`	Use the `QUOTED_PRINTABLE_ENCODE` function to translate an input string to a quoted-printable formatted `RAW` value.
`TEXT_DECODE(`*`buf`*`, `*`encode_charset`*`, `*`encoding`*`)`	`VARCHAR2`	Use the `TEXT_DECODE` function to decode a string encoded by `TEXT_ENCODE`.
`TEXT_ENCODE(`*`buf`*`, `*`encode_charset`*`, `*`encoding`*`)`	`VARCHAR2`	Use the `TEXT_ENCODE` function to translate a string to a user-specified character set, and then encode the string.
`UUDECODE(`*`r`*`)`	`RAW`	Use the `UUDECODE` function to translate a uuencode encoded string to a `RAW` value.
`UUENCODE(`*`r`*`, `*`type`*`, `*`filename`*`, `*`permission`*`)`	`RAW`	Use the `UUENCODE` function to translate a `RAW` string to an encoded uuencode value.

7.16.1 BASE64_DECODE

Use the `BASE64_DECODE` function to translate a Base64 encoded string to the original value originally encoded by `BASE64_ENCODE`. The signature is:

```
BASE64_DECODE(r IN RAW)
```

This function returns a `RAW` value.

Parameters

`r`

`r` is the string that contains the Base64 encoded data that will be translated to RAW form.

Examples

Note: Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

This command instructs the server to escape any non-printable characters, and to display `BYTEA` or `RAW` values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

http://www.postgresql.org/docs/9.5/static/datatype-binary.html

The following example first encodes (using `BASE64_ENCODE`), and then decodes (using `BASE64_DECODE`) a string that contains the text `abc`:

```
edb=# SELECT UTL_ENCODE.BASE64_ENCODE(CAST ('abc' AS RAW));
 base64_encode
---------------
 YWJj
(1 row)

edb=# SELECT UTL_ENCODE.BASE64_DECODE(CAST ('YWJj' AS RAW));
 base64_decode
---------------
 abc
(1 row)
```

7.16.2 BASE64_ENCODE

Use the BASE64_ENCODE function to translate and encode a string in Base64 format (as described in RFC 4648). This function can be useful when composing MIME email that you intend to send using the UTL_SMTP package. The BASE64_ENCODE function has two signatures:

```
BASE64_ENCODE(r IN RAW)
```

and

```
BASE64_ENCODE(loid IN OID)
```

This function returns a RAW value or an OID.

Parameters

r

r specifies the RAW string that will be translated to Base64.

loid

loid specifies the object ID of a large object that will be translated to Base64.

Examples

Note: Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

This command instructs the server to escape any non-printable characters, and to display BYTEA or RAW values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

http://www.postgresql.org/docs/9.5/static/datatype-binary.html

The following example first encodes (using BASE64_ENCODE), and then decodes (using BASE64_DECODE) a string that contains the text abc:

```
edb=# SELECT UTL_ENCODE.BASE64_ENCODE(CAST ('abc' AS RAW));
 base64_encode
---------------
 YWJj
(1 row)

edb=# SELECT UTL_ENCODE.BASE64_DECODE(CAST ('YWJj' AS RAW));
 base64_decode
---------------
```

```
 abc
(1 row)
```

7.16.3 MIMEHEADER_DECODE

Use the `MIMEHEADER_DECODE` function to decode values that are encoded by the `MIMEHEADER_ENCODE` function. The signature is:

```
MIMEHEADER_DECODE(buf IN VARCHAR2)
```

This function returns a `VARCHAR2` value.

Parameters

buf

buf contains the value (encoded by `MIMEHEADER_ENCODE`) that will be decoded.

Examples

The following examples use the `MIMEHEADER_ENCODE` and `MIMEHEADER_DECODE` functions to first encode, and then decode a string:

```
edb=# SELECT UTL_ENCODE.MIMEHEADER_ENCODE('What is the date?') FROM DUAL;
      mimeheader_encode
------------------------------
 =?UTF8?Q?What is the date??=
(1 row)

edb=# SELECT UTL_ENCODE.MIMEHEADER_DECODE('=?UTF8?Q?What is the date??=')
FROM DUAL;
 mimeheader_decode
-------------------
 What is the date?
(1 row)
```

7.16.4 MIMEHEADER_ENCODE

Use the `MIMEHEADER_ENCODE` function to convert a string into mime header format, and then encode the string. The signature is:

```
MIMEHEADER_ENCODE(buf IN VARCHAR2, encode_charset IN
VARCHAR2 DEFAULT NULL, encoding IN INTEGER DEFAULT NULL)
```

This function returns a `VARCHAR2` value.

Parameters

buf

> *buf* contains the string that will be formatted and encoded. The string is a `VARCHAR2` value.

encode_charset

> *encode_charset* specifies the character set to which the string will be converted before being formatted and encoded. The default value is `NULL`.

encoding

> *encoding* specifies the encoding type used when encoding the string. You can specify:
>
> - `Q` to enable quoted-printable encoding. If you do not specify a value, `MIMEHEADER_ENCODE` will use quoted-printable encoding.
> - `B` to enable base-64 encoding.

Examples

The following examples use the `MIMEHEADER_ENCODE` and `MIMEHEADER_DECODE` functions to first encode, and then decode a string:

```
edb=# SELECT UTL_ENCODE.MIMEHEADER_ENCODE('What is the date?') FROM DUAL;
      mimeheader_encode
------------------------------
 =?UTF8?Q?What is the date??=
(1 row)

edb=# SELECT UTL_ENCODE.MIMEHEADER_DECODE('=?UTF8?Q?What is the date??=')
FROM DUAL;
 mimeheader_decode
-------------------
 What is the date?
(1 row)
```

7.16.5 QUOTED_PRINTABLE_DECODE

Use the `QUOTED_PRINTABLE_DECODE` function to translate an encoded quoted-printable string into a decoded RAW string.

The signature is:

```
QUOTED_PRINTABLE_DECODE(r IN RAW)
```

This function returns a `RAW` value.

Parameters

r

r contains the encoded string that will be decoded. The string is a RAW value, encoded by `QUOTED_PRINTABLE_ENCODE`.

Examples

Note: Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

This command instructs the server to escape any non-printable characters, and to display `BYTEA` or `RAW` values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

http://www.postgresql.org/docs/9.5/static/datatype-binary.html

The following example first encodes and then decodes a string:

```
edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_ENCODE('E=mc2') FROM DUAL;
quoted_printable_encode
-------------------------
 E=3Dmc2
(1 row)

edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_DECODE('E=3Dmc2') FROM DUAL;
 quoted_printable_decode
-------------------------
 E=mc2
(1 row)
```

7.16.6 QUOTED_PRINTABLE_ENCODE

Use the `QUOTED_PRINTABLE_ENCODE` function to translate and encode a string in quoted-printable format. The signature is:

```
QUOTED_PRINTABLE_ENCODE(r IN RAW)
```

This function returns a `RAW` value.

Parameters

r

> *r* contains the string (a RAW value) that will be encoded in a quoted-printable format.

Examples

Note: Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

This command instructs the server to escape any non-printable characters, and to display `BYTEA` or `RAW` values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

http://www.postgresql.org/docs/9.5/static/datatype-binary.html

The following example first encodes and then decodes a string:

```
edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_ENCODE('E=mc2') FROM DUAL;
quoted_printable_encode
-------------------------
 E=3Dmc2
(1 row)

edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_DECODE('E=3Dmc2') FROM DUAL;
 quoted_printable_decode
-------------------------
 E=mc2
(1 row)
```

7.16.7 TEXT_DECODE

Use the `TEXT_DECODE` function to translate and decode an encoded string to the `VARCHAR2` value that was originally encoded by the `TEXT_ENCODE` function. The signature is:

```
TEXT_DECODE(buf IN VARCHAR2, encode_charset IN VARCHAR2
DEFAULT NULL, encoding IN PLS_INTEGER DEFAULT NULL)
```

This function returns a `VARCHAR2` value.

Parameters

buf

buf contains the encoded string that will be translated to the original value encoded by `TEXT_ENCODE`.

encode_charset

encode_charset specifies the character set to which the string will be translated before encoding. The default value is `NULL`.

encoding

encoding specifies the encoding type used by `TEXT_DECODE`. Specify:

- `UTL_ENCODE.BASE64` to specify base-64 encoding.
- `UTL_ENCODE.QUOTED_PRINTABLE` to specify quoted printable encoding. This is the default.

Examples

The following example uses the `TEXT_ENCODE` and `TEXT_DECODE` functions to first encode, and then decode a string:

```
edb=# SELECT UTL_ENCODE.TEXT_ENCODE('What is the date?', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
       text_encode
--------------------------
 V2hhdCBpcyB0aGUgZGF0ZT8=
(1 row)

edb=# SELECT UTL_ENCODE.TEXT_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
    text_decode
-------------------
 What is the date?
(1 row)
```

7.16.8 TEXT_ENCODE

Use the `TEXT_ENCODE` function to translate a string to a user-specified character set, and then encode the string. The signature is:

```
TEXT_DECODE(buf IN VARCHAR2, encode_charset IN VARCHAR2
DEFAULT NULL, encoding IN PLS_INTEGER DEFAULT NULL)
```

This function returns a `VARCHAR2` value.

Parameters

buf

buf contains the encoded string that will be translated to the specified character set and encoded by `TEXT_ENCODE`.

encode_charset

encode_charset specifies the character set to which the value will be translated before encoding. The default value is `NULL`.

encoding

encoding specifies the encoding type used by `TEXT_ENCODE`. Specify:

- `UTL_ENCODE.BASE64` to specify base-64 encoding.
- `UTL_ENCODE.QUOTED_PRINTABLE` to specify quoted printable encoding. This is the default.

Examples

The following example uses the `TEXT_ENCODE` and `TEXT_DECODE` functions to first encode, and then decode a string:

```
edb=# SELECT UTL_ENCODE.TEXT_ENCODE('What is the date?', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
       text_encode
--------------------------
 V2hhdCBpcyB0aGUgZGF0ZT8=
(1 row)

edb=# SELECT UTL_ENCODE.TEXT_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
    text_decode
-------------------
 What is the date?
(1 row)
```

7.16.9 UUDECODE

Use the `UUDECODE` function to translate and decode a uuencode encoded string to the `RAW` value that was originally encoded by the `UUENCODE` function. The signature is:

```
UUDECODE(r IN RAW)
```

This function returns a `RAW` value.

Note: If you are using the Advanced Server `UUDECODE` function to decode uuencoded data that was created by the Oracle implementation of the `UTL_ENCODE.UUENCODE` function, then you must first set the Advanced Server configuration parameter `utl_encode.uudecode_redwood` to `TRUE` before invoking the Advanced Server `UUDECODE` function on the Oracle-created data. (For example, this situation may occur if you migrated Oracle tables containing uuencoded data to an Advanced Server database.)

The uuencoded data created by the Oracle version of the `UUENCODE` function results in a format that differs from the uuencoded data created by the Advanced Server `UUENCODE` function. As a result, attempting to use the Advanced Server `UUDECODE` function on the Oracle uuencoded data results in an error unless the configuration parameter `utl_encode.uudecode_redwood` is set to `TRUE`.

However, if you are using the Advanced Server `UUDECODE` function on uuencoded data created by the Advanced Server `UUENCODE` function, then `utl_encode.uudecode_redwood` must be set to `FALSE`, which is the default setting.

Parameters

r

r contains the uuencoded string that will be translated to `RAW`.

Examples

Note: Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

This command instructs the server to escape any non-printable characters, and to display `BYTEA` or `RAW` values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

http://www.postgresql.org/docs/9.5/static/datatype-binary.html

The following example uses `UUENCODE` and `UUDECODE` to first encode and then decode a string:

```
edb=# SET bytea_output = escape;
SET
edb=# SELECT UTL_ENCODE.UUENCODE('What is the date?') FROM DUAL;
                              uuencode
--------------------------------------------------------------------
 begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012
(1 row)

edb=# SELECT UTL_ENCODE.UUDECODE
edb-# ('begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012')
edb-# FROM DUAL;
     uudecode
-------------------
 What is the date?
(1 row)
```

7.16.10 UUENCODE

Use the `UUENCODE` function to translate `RAW` data into a uuencode formatted encoded string. The signature is:

> `UUENCODE(`*`r`*` IN RAW, `*`type`*` IN INTEGER DEFAULT 1, `*`filename`*` IN VARCHAR2 DEFAULT NULL, `*`permission`*` IN VARCHAR2 DEFAULT NULL)`

This function returns a `RAW` value.

Parameters

`r`

> *`r`* contains the `RAW` string that will be translated to uuencode format.

`type`

> *`type`* is an `INTEGER` value or constant that specifies the type of uuencoded string that will be returned; the default value is `1`. The possible values are:

Value	Constant
1	complete
2	header_piece
3	middle_piece
4	end_piece

`filename`

> *`filename`* is a `VARCHAR2` value that specifies the file name that you want to embed in the encoded form; if you do not specify a file name, `UUENCODE` will include a filename of `uuencode.txt` in the encoded form.

permission

permission is a VARCHAR2 that specifies the permission mode; the default value is NULL.

Examples

Note: Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

This command instructs the server to escape any non-printable characters, and to display BYTEA or RAW values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

http://www.postgresql.org/docs/9.5/static/datatype-binary.html

The following example uses UUENCODE and UUDECODE to first encode and then decode a string:

```
edb=# SET bytea_output = escape;
SET
edb=# SELECT UTL_ENCODE.UUENCODE('What is the date?') FROM DUAL;
                              uuencode
--------------------------------------------------------------------
 begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012
(1 row)

edb=# SELECT UTL_ENCODE.UUDECODE
edb-# ('begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012')
edb-# FROM DUAL;
     uudecode
-------------------
 What is the date?
(1 row)
```

7.17 UTL_FILE

The `UTL_FILE` package provides the capability to read from, and write to files on the operating system's file system. Non-superusers must be granted `EXECUTE` privilege on the `UTL_FILE` package by a superuser before using any of the functions or procedures in the package. For example the following command grants the privilege to user `mary`:

```
GRANT EXECUTE ON PACKAGE SYS.UTL_FILE TO mary;
```

Also, the operating system username, `enterprisedb`, must have the appropriate read and/or write permissions on the directories and files to be accessed using the `UTL_FILE` functions and procedures. If the required file permissions are not in place, an exception is thrown in the `UTL_FILE` function or procedure.

A handle to the file to be written to, or read from is used to reference the file. The *file handle* is defined by a public variable in the `UTL_FILE` package named, `UTL_FILE.FILE_TYPE`. A variable of type `FILE_TYPE` must be declared to receive the file handle returned by calling the `FOPEN` function. The file handle is then used for all subsequent operations on the file.

References to directories on the file system are done using the directory name or alias that is assigned to the directory using the `CREATE DIRECTORY` command. The procedures and functions available in the `UTL_FILE` package are listed in the following table.

Table 7-7-18 UTL_FILE Functions/Procedures

Function/Procedure	Return Type	Description
`FCLOSE(`*`file`*` IN OUT)`	n/a	Closes the specified file identified by *`file`*.
`FCLOSE_ALL`	n/a	Closes all open files.
`FCOPY(`*`location`*`, `*`filename`*`, `*`dest_dir`*`, `*`dest_file`*` [, `*`start_line`*` [, `*`end_line`*` ] ])`	n/a	Copies *`filename`* in the directory identified by *`location`* to file, *`dest_file`*, in directory, *`dest_dir`*, starting from line, *`start_line`*, to line, *`end_line`*.
`FFLUSH(`*`file`*`)`	n/a	Forces data in the buffer to be written to disk in the file identified by *`file`*.
`FOPEN(`*`location`*`, `*`filename`*`, `*`open_mode`*` [, `*`max_linesize`*` ])`	`FILE_TYPE`	Opens file, *`filename`*, in the directory identified by *`location`*.
`FREMOVE(`*`location`*`, `*`filename`*`)`	n/a	Removes the specified file from the file system.
`FRENAME(`*`location`*`, `*`filename`*`, `*`dest_dir`*`, `*`dest_file`*` [, `*`overwrite`*` ])`	n/a	Renames the specified file.
`GET_LINE(`*`file`*`, `*`buffer`*` OUT)`	n/a	Reads a line of text into variable, *`buffer`*, from the file identified by *`file`*.
`IS_OPEN(`*`file`*`)`	`BOOLEAN`	Determines whether or not the given file is open.

Function/Procedure	Return Type	Description
`NEW_LINE(file [, lines ])`	n/a	Writes an end-of-line character sequence into the file.
`PUT(file, buffer)`	n/a	Writes `buffer` to the given file. `PUT` does not write an end-of-line character sequence.
`PUT_LINE(file, buffer)`	n/a	Writes `buffer` to the given file. An end-of-line character sequence is added by the `PUT_LINE` procedure.
`PUTF(file, format [, arg1 ] [, ...])`	n/a	Writes a formatted string to the given file. Up to five substitution parameters, *arg1*,...*arg5* may be specified for replacement in *format*.

Advanced Server's implementation of `UTL_FILE` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

UTL_FILE Exception Codes

If a call to a `UTL_FILE` procedure or function raises an exception, you can use the condition name to catch the exception. For more information about error handling, see Section 4.5.7, *Exception Handling*.

The `UTL_FILE` package reports the following exception codes compatible with Oracle databases:

Exception Code	Condition name
-29283	invalid_operation
-29285	write_error
-29284	read_error
-29282	invalid_filehandle
-29287	invalid_maxlinesize
-29281	invalid_mode
-29280	invalid_path

7.17.1 Setting File Permissions with utl_file.umask

When a `UTL_FILE` function or procedure creates a file, there are default file permissions as shown by the following.

```
-rw------- 1 enterprisedb enterprisedb 21 Jul 24 16:08 utlfile
```

Note that all permissions are denied on users belonging to the `enterprisedb` group as well as all other users. Only the `enterprisedb` user has read and write permissions on the created file.

If you wish to have a different set of file permissions on files created by the UTL_FILE functions and procedures, you can accomplish this by setting the utl_file.umask configuration parameter.

The utl_file.umask parameter sets the *file mode creation mask* or simply, the *mask*, in a manner similar to the Linux umask command. This is for usage only within the Advanced Server UTL_FILE package.

Note: The utl_file.umask parameter is not supported on Windows systems.

The value specified for utl_file.umask is a 3 or 4-character octal string that would be valid for the Linux umask command. The setting determines the permissions on files created by the UTL_FILE functions and procedures. (Refer to any information source regarding Linux or Unix systems for information on file permissions and the usage of the umask command.)

The following is an example of setting the file permissions with utl_file.umask.

First, set up the directory in the file system to be used by the UTL_FILE package. Be sure the operating system account, enterprisedb or postgres, whichever is applicable, can read and write in the directory.

```
mkdir /tmp/utldir
chmod 777 /tmp/utldir
```

The CREATE DIRECTORY command is issued in psql to create the directory database object using the file system directory created in the preceding step.

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

Set the utl_file.umask configuration parameter. The following setting allows the file owner any permission. Group users and other users are permitted any permission except for the execute permission.

```
SET utl_file.umask TO '0011';
```

In the same session during which the utl_file.umask parameter is set to the desired value, run the UTL_FILE functions and procedures.

```
DECLARE
    v_utlfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'utldir';
    v_filename      VARCHAR2(20) := 'utlfile';
BEGIN
    v_utlfile := UTL_FILE.FOPEN(v_directory, v_filename, 'w');
    UTL_FILE.PUT_LINE(v_utlfile, 'Simple one-line file');
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_utlfile);
END;
```

The permission settings on the resulting file show that group users and other users have read and write permissions on the file as well as the file owner.

```
$ pwd
/tmp/utldir
$ ls -l
total 4
-rw-rw-rw- 1 enterprisedb enterprisedb 21 Jul 24 16:04 utlfile
```

This parameter can also be set on a per role basis with the `ALTER ROLE` command, on a per database basis with the `ALTER DATABASE` command, or for the entire database server instance by setting it in the `postgresql.conf` file.

7.17.2 FCLOSE

The `FCLOSE` procedure closes an open file.

```
FCLOSE(file IN OUT FILE_TYPE)
```

Parameters

`file`

Variable of type `FILE_TYPE` containing a file handle of the file to be closed.

7.17.3 FCLOSE_ALL

The `FLCLOSE_ALL` procedures closes all open files. The procedure executes successfully even if there are no open files to close.

```
FCLOSE_ALL
```

7.17.4 FCOPY

The `FCOPY` procedure copies text from one file to another.

```
FCOPY(location VARCHAR2, filename VARCHAR2,
  dest_dir VARCHAR2, dest_file VARCHAR2
  [, start_line PLS_INTEGER [, end_line PLS_INTEGER ] ])
```

Parameters

location

> Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory containing the file to be copied.

filename

> Name of the source file to be copied.

dest_dir

> Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory to which the file is to be copied.

dest_file

> Name of the destination file.

start_line

> Line number in the source file from which copying will begin. The default is 1.

end_line

> Line number of the last line in the source file to be copied. If omitted or null, copying will go to the last line of the file.

Examples

The following makes a copy of a file, `C:\TEMP\EMPDIR\empfile.csv`, containing a comma-delimited list of employees from the `emp` table. The copy, `empcopy.csv`, is then listed.

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

DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_src_dir       VARCHAR2(50)  := 'empdir';
    v_src_file      VARCHAR2(20)  := 'empfile.csv';
    v_dest_dir      VARCHAR2(50)  := 'empdir';
    v_dest_file     VARCHAR2(20)  := 'empcopy.csv';
    v_emprec        VARCHAR2(120);
    v_count         INTEGER := 0;
BEGIN
    UTL_FILE.FCOPY(v_src_dir,v_src_file,v_dest_dir,v_dest_file);
    v_empfile := UTL_FILE.FOPEN(v_dest_dir,v_dest_file,'r');
    DBMS_OUTPUT.PUT_LINE('The following is the destination file, ''' ||
        v_dest_file || '''');
    LOOP
```

```
        UTL_FILE.GET_LINE(v_empfile,v_emprec);
        DBMS_OUTPUT.PUT_LINE(v_emprec);
        v_count := v_count + 1;
    END LOOP;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            UTL_FILE.FCLOSE(v_empfile);
            DBMS_OUTPUT.PUT_LINE(v_count || ' records retrieved');
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
            DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

The following is the destination file, 'empcopy.csv'
7369,SMITH,CLERK,7902,17-DEC-80,800,,20
7499,ALLEN,SALESMAN,7698,20-FEB-81,1600,300,30
7521,WARD,SALESMAN,7698,22-FEB-81,1250,500,30
7566,JONES,MANAGER,7839,02-APR-81,2975,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81,1250,1400,30
7698,BLAKE,MANAGER,7839,01-MAY-81,2850,,30
7782,CLARK,MANAGER,7839,09-JUN-81,2450,,10
7788,SCOTT,ANALYST,7566,19-APR-87,3000,,20
7839,KING,PRESIDENT,,17-NOV-81,5000,,10
7844,TURNER,SALESMAN,7698,08-SEP-81,1500,0,30
7876,ADAMS,CLERK,7788,23-MAY-87,1100,,20
7900,JAMES,CLERK,7698,03-DEC-81,950,,30
7902,FORD,ANALYST,7566,03-DEC-81,3000,,20
7934,MILLER,CLERK,7782,23-JAN-82,1300,,10
14 records retrieved
```

7.17.5 FFLUSH

The `FFLUSH` procedure flushes unwritten data from the write buffer to the file.

```
FFLUSH(file FILE_TYPE)
```

Parameters

file

Variable of type `FILE_TYPE` containing a file handle.

Examples

Each line is flushed after the `NEW_LINE` procedure is called.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
    FOR i IN emp_cur LOOP
```

```
        UTL_FILE.PUT(v_empfile,i.empno);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.ename);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.job);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.mgr);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.hiredate);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.sal);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.comm);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.deptno);
        UTL_FILE.NEW_LINE(v_empfile);
        UTL_FILE.FFLUSH(v_empfile);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
END;
```

7.17.6 FOPEN

The FOPEN function opens a file for I/O.

```
filetype FILE_TYPE FOPEN(location VARCHAR2,
  filename VARCHAR2,open_mode VARCHAR2
  [, max_linesize BINARY_INTEGER ])
```

Parameters

location

Directory name, as stored in pg_catalog.edb_dir.dirname, of the directory containing the file to be opened.

filename

Name of the file to be opened.

open_mode

Mode in which the file will be opened. Modes are: a - append to file; r - read from file; w - write to file.

max_linesize

Maximum size of a line in characters. In read mode, an exception is thrown if an attempt is made to read a line exceeding *max_linesize*. In write and append modes, an exception is thrown if an attempt is made to write a line exceeding `max_linesize`. The end-of-line character(s) are not included in determining if the maximum line size is exceeded. This behavior is not compatible with Oracle databases; Oracle does count the end-of-line character(s).

`filetype`

Variable of type `FILE_TYPE` containing the file handle of the opened file.

7.17.7 FREMOVE

The `FREMOVE` procedure removes a file from the system.

```
FREMOVE(location VARCHAR2, filename VARCHAR2)
```

An exception is thrown if the file to be removed does not exist.

Parameters

`location`

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory containing the file to be removed.

`filename`

Name of the file to be removed.

Examples

The following removes file `empfile.csv`.

```
DECLARE
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
BEGIN
    UTL_FILE.FREMOVE(v_directory,v_filename);
    DBMS_OUTPUT.PUT_LINE('Removed file: ' || v_filename);
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
            DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

Removed file: empfile.csv
```

7.17.8 FRENAME

The `FRENAME` procedure renames a given file. This effectively moves a file from one location to another.

```
FRENAME(location VARCHAR2, filename VARCHAR2,
  dest_dir VARCHAR2, dest_file VARCHAR2,
  [ overwrite BOOLEAN ])
```

Parameters

location

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory containing the file to be renamed.

filename

Name of the source file to be renamed.

dest_dir

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory to which the renamed file is to exist.

dest_file

New name of the original file.

overwrite

Replaces any existing file named *dest_file* in *dest_dir* if set to `TRUE`, otherwise an exception is thrown if set to FALSE. This is the default.

Examples

The following renames a file, `C:\TEMP\EMPDIR\empfile.csv`, containing a comma-delimited list of employees from the `emp` table. The renamed file, `C:\TEMP\NEWDIR\newemp.csv`, is then listed.

```
CREATE DIRECTORY "newdir" AS 'C:/TEMP/NEWDIR';

DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_src_dir       VARCHAR2(50)  := 'empdir';
    v_src_file      VARCHAR2(20)  := 'empfile.csv';
    v_dest_dir      VARCHAR2(50)  := 'newdir';
    v_dest_file     VARCHAR2(50)  := 'newemp.csv';
    v_replace       BOOLEAN := FALSE;
    v_emprec        VARCHAR2(120);
    v_count         INTEGER := 0;
BEGIN
    UTL_FILE.FRENAME(v_src_dir,v_src_file,v_dest_dir,
        v_dest_file,v_replace);
    v_empfile := UTL_FILE.FOPEN(v_dest_dir,v_dest_file,'r');
    DBMS_OUTPUT.PUT_LINE('The following is the renamed file, ''' ||
        v_dest_file || '''');
    LOOP
        UTL_FILE.GET_LINE(v_empfile,v_emprec);
        DBMS_OUTPUT.PUT_LINE(v_emprec);
        v_count := v_count + 1;
    END LOOP;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            UTL_FILE.FCLOSE(v_empfile);
            DBMS_OUTPUT.PUT_LINE(v_count || ' records retrieved');
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
            DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

The following is the renamed file, 'newemp.csv'
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 records retrieved
```

7.17.9 GET_LINE

The `GET_LINE` procedure reads a line of text from a given file up to, but not including the end-of-line terminator. A `NO_DATA_FOUND` exception is thrown when there are no more lines to read.

```
GET_LINE(file FILE_TYPE, buffer OUT VARCHAR2)
```

Parameters

`file`

Variable of type `FILE_TYPE` containing the file handle of the opened file.

buffer

Variable to receive a line from the file.

Examples

The following anonymous block reads through and displays the records in file `empfile.csv`.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50)  := 'empdir';
    v_filename      VARCHAR2(20)  := 'empfile.csv';
    v_emprec        VARCHAR2(120);
    v_count         INTEGER := 0;
BEGIN
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'r');
    LOOP
        UTL_FILE.GET_LINE(v_empfile,v_emprec);
        DBMS_OUTPUT.PUT_LINE(v_emprec);
        v_count := v_count + 1;
    END LOOP;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            UTL_FILE.FCLOSE(v_empfile);
            DBMS_OUTPUT.PUT_LINE('End of file ' || v_filename || ' - ' ||
                v_count || ' records retrieved');
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
            DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

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
End of file empfile.csv - 14 records retrieved
```

7.17.10 IS_OPEN

The `IS_OPEN` function determines whether or not the given file is open.

```
status BOOLEAN IS_OPEN(file FILE_TYPE)
```

Parameters

file

Variable of type `FILE_TYPE` containing the file handle of the file to be tested.

status

`TRUE` if the given file is open, `FALSE` otherwise.

7.17.11 NEW_LINE

The `NEW_LINE` procedure writes an end-of-line character sequence in the file.

```
NEW_LINE(file FILE_TYPE [, lines INTEGER ])
```

Parameters

file

Variable of type `FILE_TYPE` containing the file handle of the file to which end-of-line character sequences are to be written.

lines

Number of end-of-line character sequences to be written. The default is one.

Examples

A file containing a double-spaced list of employee records is written.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
    FOR i IN emp_cur LOOP
        UTL_FILE.PUT(v_empfile,i.empno);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.ename);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.job);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.mgr);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.hiredate);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.sal);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.comm);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.deptno);
        UTL_FILE.NEW_LINE(v_empfile,2);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
END;

Created file: empfile.csv
```

This file is then displayed:

```
C:\TEMP\EMPDIR>TYPE empfile.csv

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
```

7.17.12 PUT

The `PUT` procedure writes a string to the given file. No end-of-line character sequence is written at the end of the string. Use the `NEW_LINE` procedure to add an end-of-line character sequence.

```
PUT(file FILE_TYPE, buffer { DATE | NUMBER | TIMESTAMP |
  VARCHAR2 })
```

Parameters

file

Variable of type `FILE_TYPE` containing the file handle of the file to which the given string is to be written.

buffer

Text to be written to the specified file.

Examples

The following example uses the `PUT` procedure to create a comma-delimited file of employees from the `emp` table.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
    FOR i IN emp_cur LOOP
        UTL_FILE.PUT(v_empfile,i.empno);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.ename);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.job);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.mgr);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.hiredate);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.sal);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.comm);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.deptno);
        UTL_FILE.NEW_LINE(v_empfile);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
END;

Created file: empfile.csv
```

The following is the contents of `empfile.csv` created above:

```
C:\TEMP\EMPDIR>TYPE empfile.csv

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
```

7.17.13 PUT_LINE

The `PUT_LINE` procedure writes a single line to the given file including an end-of-line character sequence.

```
PUT_LINE(file FILE_TYPE,
  buffer {DATE|NUMBER|TIMESTAMP|VARCHAR2})
```

Parameters

file

Variable of type `FILE_TYPE` containing the file handle of the file to which the given line is to be written.

buffer

Text to be written to the specified file.

Examples

The following example uses the `PUT_LINE` procedure to create a comma-delimited file of employees from the `emp` table.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
    v_emprec        VARCHAR2(120);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
    FOR i IN emp_cur LOOP
        v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||
            NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
            ',' || i.sal || ',' ||
            NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
        UTL_FILE.PUT_LINE(v_empfile,v_emprec);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
```

```
END;
```

The following is the contents of `empfile.csv` created above:

```
C:\TEMP\EMPDIR>TYPE empfile.csv

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
```

7.17.14 PUTF

The `PUTF` procedure writes a formatted string to the given file.

```
PUTF(file FILE_TYPE, format VARCHAR2 [, arg1 VARCHAR2]
  [, ...])
```

Parameters

file

Variable of type `FILE_TYPE` containing the file handle of the file to which the formatted line is to be written.

format

String to format the text written to the file. The special character sequence, `%s`, is substituted by the value of arg. The special character sequence, `\n`, indicates a new line. Note, however, in Advanced Server, a new line character must be specified with two consecutive backslashes instead of one - `\\n`. This characteristic is not compatible with Oracle databases.

arg1

Up to five arguments, *arg1*,...*arg5*, to be substituted in the format string for each occurrence of `%s`. The first arg is substituted for the first occurrence of `%s`, the second arg is substituted for the second occurrence of `%s`, etc.

Examples

The following anonymous block produces formatted output containing data from the `emp` table. Note the use of the E literal syntax and double backslashes for the new line character sequence in the format string which are not compatible with Oracle databases.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50)  := 'empdir';
    v_filename      VARCHAR2(20)  := 'empfile.csv';
    v_format        VARCHAR2(200);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    v_format := E'%s %s, %s\\nSalary: $%s Commission: $%s\\n\\n';
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
    FOR i IN emp_cur LOOP
        UTL_FILE.PUTF(v_empfile,v_format,i.empno,i.ename,i.job,i.sal,
            NVL(i.comm,0));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

Created file: empfile.csv
```

The following is the contents of `empfile.csv` created above:

```
C:\TEMP\EMPDIR>TYPE empfile.csv
7369 SMITH, CLERK
Salary: $800.00 Commission: $0
7499 ALLEN, SALESMAN
Salary: $1600.00 Commission: $300.00
7521 WARD, SALESMAN
Salary: $1250.00 Commission: $500.00
7566 JONES, MANAGER
Salary: $2975.00 Commission: $0
7654 MARTIN, SALESMAN
Salary: $1250.00 Commission: $1400.00
7698 BLAKE, MANAGER
Salary: $2850.00 Commission: $0
7782 CLARK, MANAGER
Salary: $2450.00 Commission: $0
7788 SCOTT, ANALYST
Salary: $3000.00 Commission: $0
7839 KING, PRESIDENT
Salary: $5000.00 Commission: $0
7844 TURNER, SALESMAN
Salary: $1500.00 Commission: $0.00
7876 ADAMS, CLERK
Salary: $1100.00 Commission: $0
7900 JAMES, CLERK
Salary: $950.00 Commission: $0
7902 FORD, ANALYST
Salary: $3000.00 Commission: $0
7934 MILLER, CLERK
Salary: $1300.00 Commission: $0
```

7.18 UTL_HTTP

The `UTL_HTTP` package provides a way to use the HTTP or HTTPS protocol to retrieve information found at an URL.

Table 7.7.2 UTL_HTTP Functions and Procedures

Function/Procedure	Return Type	Description
`BEGIN_REQUEST(`*`url`*`, `*`method`*`, `*`http_version`*`)`	`UTL_HTTP.REQ`	Initiates a new HTTP request.
`END_REQUEST(`*`r`*` IN OUT)`	n/a	Ends an HTTP request before allowing it to complete.
`END_RESPONSE(`*`r`*` IN OUT)`	n/a	Ends the HTTP response.
`GET_BODY_CHARSET`	`VARCHAR2`	Returns the default character set of the body of future HTTP requests.
`GET_BODY_CHARSET(`*`charset`*` OUT)`	n/a	Returns the default character set of the body of future HTTP requests.
`GET_FOLLOW_REDIRECT(`*`max_redirects`*` OUT)`	n/a	Current setting for the maximum number of redirections allowed.
`GET_HEADER(`*`r`*` IN OUT, `*`n`*`, `*`name`*` OUT, `*`value`*` OUT)`	n/a	Returns the *n*th header of the HTTP response.
`GET_HEADER_BY_NAME(`*`r`*` IN OUT, `*`name`*`, `*`value`*` OUT, `*`n`*`)`	n/a	Returns the HTTP response header for the specified name.
`GET_HEADER_COUNT(`*`r`*` IN OUT)`	`INTEGER`	Returns the number of HTTP response headers.
`GET_RESPONSE(`*`r`*` IN OUT)`	`UTL_HTTP.RESP`	Returns the HTTP response.
`GET_RESPONSE_ERROR_CHECK(`*`enable`*` OUT)`	n/a	Returns whether or not response error check is set.
`GET_TRANSFER_TIMEOUT(`*`timeout`*` OUT)`	n/a	Returns the transfer timeout setting for HTTP requests.
`READ_LINE(`*`r`*` IN OUT, `*`data`*` OUT, `*`remove_crlf`*`)`	n/a	Returns the HTTP response body in text form until the end of line.
`READ_RAW(`*`r`*` IN OUT, `*`data`*` OUT, `*`len`*`)`	n/a	Returns the HTTP response body in binary form for a specified number of bytes.
`READ_TEXT(`*`r`*` IN OUT, `*`data`*` OUT, `*`len`*`)`	n/a	Returns the HTTP response body in text form for a specified number of characters.
`REQUEST(`*`url`*`)`	`VARCHAR2`	Returns the content of a web page.
`REQUEST_PIECES(`*`url`*`, `*`max_pieces`*`)`	`UTL_HTTP. HTML_PIECES`	Returns a table of 2000-byte segments retrieved from an URL.
`SET_BODY_CHARSET(`*`charset`*`)`	n/a	Sets the default character set of the body of future HTTP requests.
`SET_FOLLOW_REDIRECT(`*`max_redirects`*`)`	n/a	Sets the maximum number of times to follow the redirect instruction.
`SET_FOLLOW_REDIRECT(`*`r`*` IN OUT, `*`max_redirects`*`)`	n/a	Sets the maximum number of times to follow the redirect instruction for an individual request.
`SET_HEADER(`*`r`*` IN OUT, `*`name`*`, `*`value`*`)`	n/a	Sets the HTTP request header.
`SET_RESPONSE_ERROR_CHECK(`*`enable`*`)`	n/a	Determines whether or not HTTP 4xx and

Function/Procedure	Return Type	Description
		5xx status codes are to be treated as errors.
`SET_TRANSFER_TIMEOUT(`*`timeout`*`)`	n/a	Sets the default, transfer timeout value for HTTP requests.
`SET_TRANSFER_TIMEOUT(`*`r`*` IN OUT, `*`timeout`*`)`	n/a	Sets the transfer timeout value for an individual HTTP request.
`WRITE_LINE(`*`r`*` IN OUT, `*`data`*`)`	n/a	Writes CRLF terminated data to the HTTP request body in TEXT form.
`WRITE_RAW(`*`r`*` IN OUT, `*`data`*`)`	n/a	Writes data to the HTTP request body in BINARY form.
`WRITE_TEXT(`*`r`*` IN OUT, `*`data`*`)`	n/a	Writes data to the HTTP request body in TEXT form.

Advanced Server's implementation of `UTL_HTTP` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

Please Note:

In Advanced Server, an `HTTP 4xx` or `HTTP 5xx` response produces a database error; in Oracle, this is configurable but `FALSE` by default.

In Advanced Server, the `UTL_HTTP` text interfaces expect the downloaded data to be in the database encoding. All currently-available interfaces are text interfaces. In Oracle, the encoding is detected from HTTP headers; in the absence of the header, the default is configurable and defaults to `ISO-8859-1`.

Advanced Server ignores all cookies it receives.

The `UTL_HTTP` exceptions that can be raised in Oracle are not recognized by Advanced Server. In addition, the error codes returned by Advanced Server are not the same as those returned by Oracle.

There are various public constants available with `UTL_HTTP`. These are listed in the following tables.

The following table contains `UTL_HTTP` public constants defining HTTP versions and port assignments.

HTTP VERSIONS	
`HTTP_VERSION_1_0`	`CONSTANT VARCHAR2(64) := 'HTTP/1.0';`
`HTTP_VERSION_1_1`	`CONSTANT VARCHAR2(64) := 'HTTP/1.1';`
STANDARD PORT ASSIGNMENTS	
`DEFAULT_HTTP_PORT`	`CONSTANT INTEGER := 80;`
`DEFAULT_HTTPS_PORT`	`CONSTANT INTEGER := 443;`

The following table contains `UTL_HTTP` public status code constants.

1XX INFORMATIONAL	
`HTTP_CONTINUE`	`CONSTANT INTEGER := 100;`
`HTTP_SWITCHING_PROTOCOLS`	`CONSTANT INTEGER := 101;`
`HTTP_PROCESSING`	`CONSTANT INTEGER := 102;`
2XX SUCCESS	
`HTTP_OK`	`CONSTANT INTEGER := 200;`
`HTTP_CREATED`	`CONSTANT INTEGER := 201;`
`HTTP_ACCEPTED`	`CONSTANT INTEGER := 202;`
`HTTP_NON_AUTHORITATIVE_INFO`	`CONSTANT INTEGER := 203;`
`HTTP_NO_CONTENT`	`CONSTANT INTEGER := 204;`
`HTTP_RESET_CONTENT`	`CONSTANT INTEGER := 205;`
`HTTP_PARTIAL_CONTENT`	`CONSTANT INTEGER := 206;`
`HTTP_MULTI_STATUS`	`CONSTANT INTEGER := 207;`
`HTTP_ALREADY_REPORTED`	`CONSTANT INTEGER := 208;`
`HTTP_IM_USED`	`CONSTANT INTEGER := 226;`
3XX REDIRECTION	
`HTTP_MULTIPLE_CHOICES`	`CONSTANT INTEGER := 300;`
`HTTP_MOVED_PERMANENTLY`	`CONSTANT INTEGER := 301;`
`HTTP_FOUND`	`CONSTANT INTEGER := 302;`
`HTTP_SEE_OTHER`	`CONSTANT INTEGER := 303;`
`HTTP_NOT_MODIFIED`	`CONSTANT INTEGER := 304;`
`HTTP_USE_PROXY`	`CONSTANT INTEGER := 305;`
`HTTP_SWITCH_PROXY`	`CONSTANT INTEGER := 306;`
`HTTP_TEMPORARY_REDIRECT`	`CONSTANT INTEGER := 307;`
`HTTP_PERMANENT_REDIRECT`	`CONSTANT INTEGER := 308;`
4XX CLIENT ERROR	
`HTTP_BAD_REQUEST`	`CONSTANT INTEGER := 400;`
`HTTP_UNAUTHORIZED`	`CONSTANT INTEGER := 401;`
`HTTP_PAYMENT_REQUIRED`	`CONSTANT INTEGER := 402;`
`HTTP_FORBIDDEN`	`CONSTANT INTEGER := 403;`
`HTTP_NOT_FOUND`	`CONSTANT INTEGER := 404;`
`HTTP_METHOD_NOT_ALLOWED`	`CONSTANT INTEGER := 405;`
`HTTP_NOT_ACCEPTABLE`	`CONSTANT INTEGER := 406;`
`HTTP_PROXY_AUTH_REQUIRED`	`CONSTANT INTEGER := 407;`
`HTTP_REQUEST_TIME_OUT`	`CONSTANT INTEGER := 408;`
`HTTP_CONFLICT`	`CONSTANT INTEGER := 409;`
`HTTP_GONE`	`CONSTANT INTEGER := 410;`
`HTTP_LENGTH_REQUIRED`	`CONSTANT INTEGER := 411;`
`HTTP_PRECONDITION_FAILED`	`CONSTANT INTEGER := 412;`
`HTTP_REQUEST_ENTITY_TOO_LARGE`	`CONSTANT INTEGER := 413;`
`HTTP_REQUEST_URI_TOO_LARGE`	`CONSTANT INTEGER := 414;`
`HTTP_UNSUPPORTED_MEDIA_TYPE`	`CONSTANT INTEGER := 415;`
`HTTP_REQ_RANGE_NOT_SATISFIABLE`	`CONSTANT INTEGER := 416;`
`HTTP_EXPECTATION_FAILED`	`CONSTANT INTEGER := 417;`
`HTTP_I_AM_A_TEAPOT`	`CONSTANT INTEGER := 418;`
`HTTP_AUTHENTICATION_TIME_OUT`	`CONSTANT INTEGER := 419;`
`HTTP_ENHANCE_YOUR_CALM`	`CONSTANT INTEGER := 420;`
`HTTP_UNPROCESSABLE_ENTITY`	`CONSTANT INTEGER := 422;`
`HTTP_LOCKED`	`CONSTANT INTEGER := 423;`
`HTTP_FAILED_DEPENDENCY`	`CONSTANT INTEGER := 424;`
`HTTP_UNORDERED_COLLECTION`	`CONSTANT INTEGER := 425;`
`HTTP_UPGRADE_REQUIRED`	`CONSTANT INTEGER := 426;`
`HTTP_PRECONDITION_REQUIRED`	`CONSTANT INTEGER := 428;`
`HTTP_TOO_MANY_REQUESTS`	`CONSTANT INTEGER := 429;`
`HTTP_REQUEST_HEADER_FIELDS_TOO_LARGE`	`CONSTANT INTEGER := 431;`
`HTTP_NO_RESPONSE`	`CONSTANT INTEGER := 444;`
`HTTP_RETRY_WITH`	`CONSTANT INTEGER := 449;`

HTTP_BLOCKED_BY_WINDOWS_PARENTAL_CONTROLS	CONSTANT INTEGER := 450;
HTTP_REDIRECT	CONSTANT INTEGER := 451;
HTTP_REQUEST_HEADER_TOO_LARGE	CONSTANT INTEGER := 494;
HTTP_CERT_ERROR	CONSTANT INTEGER := 495;
HTTP_NO_CERT	CONSTANT INTEGER := 496;
HTTP_HTTP_TO_HTTPS	CONSTANT INTEGER := 497;
HTTP_CLIENT_CLOSED_REQUEST	CONSTANT INTEGER := 499;
5XX SERVER ERROR	
HTTP_INTERNAL_SERVER_ERROR	CONSTANT INTEGER := 500;
HTTP_NOT_IMPLEMENTED	CONSTANT INTEGER := 501;
HTTP_BAD_GATEWAY	CONSTANT INTEGER := 502;
HTTP_SERVICE_UNAVAILABLE	CONSTANT INTEGER := 503;
HTTP_GATEWAY_TIME_OUT	CONSTANT INTEGER := 504;
HTTP_VERSION_NOT_SUPPORTED	CONSTANT INTEGER := 505;
HTTP_VARIANT_ALSO_NEGOTIATES	CONSTANT INTEGER := 506;
HTTP_INSUFFICIENT_STORAGE	CONSTANT INTEGER := 507;
HTTP_LOOP_DETECTED	CONSTANT INTEGER := 508;
HTTP_BANDWIDTH_LIMIT_EXCEEDED	CONSTANT INTEGER := 509;
HTTP_NOT_EXTENDED	CONSTANT INTEGER := 510;
HTTP_NETWORK_AUTHENTICATION_REQUIRED	CONSTANT INTEGER := 511;
HTTP_NETWORK_READ_TIME_OUT_ERROR	CONSTANT INTEGER := 598;
HTTP_NETWORK_CONNECT_TIME_OUT_ERROR	CONSTANT INTEGER := 599;

7.18.1 HTML_PIECES

The `UTL_HTTP` package declares a type named `HTML_PIECES`, which is a table of type `VARCHAR2 (2000)` indexed by `BINARY INTEGER`. A value of this type is returned by the `REQUEST_PIECES` function.

```
TYPE html_pieces IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
```

7.18.2 REQ

The `REQ` record type holds information about each HTTP request.

```
TYPE req IS RECORD (
    url             VARCHAR2(32767),    -- URL to be accessed
    method          VARCHAR2(64),       -- HTTP method
    http_version    VARCHAR2(64),       -- HTTP version
    private_hndl    INTEGER             -- Holds handle for this request
);
```

7.18.3 RESP

The `RESP` record type holds information about the response from each HTTP request.

```
TYPE resp IS RECORD (
    status_code     INTEGER,            -- HTTP status code
    reason_phrase   VARCHAR2(256),      -- HTTP response reason phrase
    http_version    VARCHAR2(64),       -- HTTP version
    private_hndl    INTEGER             -- Holds handle for this response
);
```

7.18.4 BEGIN_REQUEST

The `BEGIN_REQUEST` function initiates a new HTTP request. A network connection is established to the web server with the specified URL. The signature is:

```
BEGIN_REQUEST(url IN VARCHAR2, method IN VARCHAR2 DEFAULT
'GET ', http_version IN VARCHAR2 DEFAULT NULL) RETURN
UTL_HTTP.REQ
```

The `BEGIN_REQUEST` function returns a record of type `UTL_HTTP.REQ`.

Parameters

`url`

`url` is the Uniform Resource Locator from which `UTL_HTTP` will return content.

`method`

`method` is the HTTP method to be used. The default is `GET`.

`http_version`

`http_version` is the HTTP protocol version sending the request. The specified values should be either `HTTP/1.0` or `HTTP/1.1`. The default is null in which case the latest HTTP protocol version supported by the `UTL_HTTP` package is used which is 1.1.

7.18.5 END_REQUEST

The `END_REQUEST` procedure terminates an HTTP request. Use the `END_REQUEST` procedure to terminate an HTTP request without completing it and waiting for the response. The normal process is to begin the request, get the response, then close the response. The signature is:

```
END_REQUEST(r IN OUT UTL_HTTP.REQ)
```

Parameters

r

r is the HTTP request record.

7.18.6 END_RESPONSE

The `END_RESPONSE` procedure terminates the HTTP response. The `END_RESPONSE` procedure completes the HTTP request and response. This is the normal method to end the request and response process. The signature is:

```
END_RESPONSE(r IN OUT UTL_HTTP.RESP)
```

Parameters

r

r is the `HTTP` response record.

7.18.7 GET_BODY_CHARSET

The `GET_BODY_CHARSET` program is available in the form of both a procedure and a function. A call to `GET_BODY_CHARSET` returns the default character set of the body of future HTTP requests.

The procedure signature is:

```
GET_BODY_CHARSET(charset OUT VARCHAR2)
```

The function signature is:

```
GET_BODY_CHARSET() RETURN VARCHAR2
```

This function returns a `VARCHAR2` value.

Parameters

charset

charset is the character set of the body.

Examples

The following is an example of the `GET_BODY_CHARSET` function.

```
edb=# SELECT UTL_HTTP.GET_BODY_CHARSET() FROM DUAL;
 get_body_charset
-------------------
 ISO-8859-1
(1 row)
```

7.18.8 GET_FOLLOW_REDIRECT

The `GET_FOLLOW_REDIRECT` procedure returns the current setting for the maximum number of redirections allowed. The signature is:

> `GET_FOLLOW_REDIRECT(`*`max_redirects`*` OUT INTEGER)`

Parameters

`max_redirects`

> *`max_redirects`* is maximum number of redirections allowed.

7.18.9 GET_HEADER

The `GET_HEADER` procedure returns the *`n`*th header of the HTTP response. The signature is:

> `GET_HEADER(`*`r`*` IN OUT UTL_HTTP.RESP, `*`n`*` INTEGER, `*`name`*` OUT VARCHAR2, `*`value`*` OUT VARCHAR2)`

Parameters

`r`

> *`r`* is the HTTP response record.

`n`

> *`n`* is the *`n`*th header of the HTTP response record to retrieve.

`name`

> *`name`* is the name of the response header.

value

value is the value of the response header.

Examples

The following example retrieves the header count, then the headers.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_name          VARCHAR2(30);
    v_value         VARCHAR2(200);
    v_header_cnt    INTEGER;
BEGIN
 -- Initiate request and get response
    v_req := UTL_HTTP.BEGIN_REQUEST('www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);

 -- Get header count
    v_header_cnt := UTL_HTTP.GET_HEADER_COUNT(v_resp);
    DBMS_OUTPUT.PUT_LINE('Header Count: ' || v_header_cnt);

 -- Get all headers
    FOR i IN 1 .. v_header_cnt LOOP
        UTL_HTTP.GET_HEADER(v_resp, i, v_name, v_value);
        DBMS_OUTPUT.PUT_LINE(v_name || ': ' || v_value);
    END LOOP;

 -- Terminate request
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```

The following is the output from the example.

```
Header Count: 23
Age: 570
Cache-Control: must-revalidate
Content-Type: text/html; charset=utf-8
Date: Wed, 30 Apr 2015 14:57:52 GMT
ETag: "aab02f2bd2d696eed817ca89ef411dda"
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Last-Modified: Wed, 30 Apr 2015 14:15:49 GMT
RTSS: 1-1307-3
Server: Apache/2.2.3 (Red Hat)
Set-Cookie: SESS2771d0952de2a1a84d322a262e0c173c=jn1u1j1etmdi5gg41h8hakvs01;
expires=Fri, 23-May-2015 18:21:43 GMT; path=/; domain=.enterprisedb.com
Vary: Accept-Encoding
Via: 1.1 varnish
X-EDB-Backend: ec
X-EDB-Cache: HIT
X-EDB-Cache-Address: 10.31.162.212
X-EDB-Cache-Server: ip-10-31-162-212
X-EDB-Cache-TTL: 600.000
X-EDB-Cacheable: MAYBE: The user has a cookie of some sort. Maybe it's double
choc-chip!
X-EDB-Do-GZIP: false
X-Powered-By: PHP/5.2.17
X-Varnish: 484508634 484506789
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

```
transfer-encoding: chunked
Connection: keep-alive
```

7.18.10 GET_HEADER_BY_NAME

The `GET_HEADER_BY_NAME` procedure returns the header of the HTTP response according to the specified name. The signature is:

```
GET_HEADER_BY_NAME(r IN OUT UTL_HTTP.RESP, name VARCHAR2,
value OUT VARCHAR2, n INTEGER DEFAULT 1)
```

Parameters

r

r is the HTTP response record.

name

name is the name of the response header to retrieve.

value

value is the value of the response header.

n

n is the *n*th header of the HTTP response record to retrieve according to the values specified by *name*. The default is 1.

Examples

The following example retrieves the header for Content-Type.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_name          VARCHAR2(30) := 'Content-Type';
    v_value         VARCHAR2(200);
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    UTL_HTTP.GET_HEADER_BY_NAME(v_resp, v_name, v_value);
    DBMS_OUTPUT.PUT_LINE(v_name || ': ' || v_value);
    UTL_HTTP.END_RESPONSE(v_resp);
END;

Content-Type: text/html; charset=utf-8
```

7.18.11 GET_HEADER_COUNT

The `GET_HEADER_COUNT` function returns the number of HTTP response headers. The signature is:

```
GET_HEADER_COUNT(r IN OUT UTL_HTTP.RESP) RETURN INTEGER
```

This function returns an `INTEGER` value.

Parameters

r

r is the HTTP response record.

7.18.12 GET_RESPONSE

The `GET_RESPONSE` function sends the network request and returns any HTTP response. The signature is:

```
GET_RESPONSE(r IN OUT UTL_HTTP.REQ) RETURN UTL_HTTP.RESP
```

This function returns a `UTL_HTTP.RESP` record.

Parameters

r

r is the HTTP request record.

7.18.13 GET_RESPONSE_ERROR_CHECK

The `GET_RESPONSE_ERROR_CHECK` procedure returns whether or not response error check is set. The signature is:

```
GET_RESPONSE_ERROR_CHECK(enable OUT BOOLEAN)
```

Parameters

enable

enable returns TRUE if response error check is set, otherwise it returns FALSE.

7.18.14 GET_TRANSFER_TIMEOUT

The GET_TRANSFER_TIMEOUT procedure returns the current, default transfer timeout setting for HTTP requests. The signature is:

```
GET_TRANSFER_TIMEOUT(timeout OUT INTEGER)
```

Parameters

timeout

timeout is the transfer timeout setting in seconds.

7.18.15 READ_LINE

The READ_LINE procedure returns the data from the HTTP response body in text form until the end of line is reached. A CR character, a LF character, a CR LF sequence, or the end of the response body constitutes the end of line. The signature is:

```
READ_LINE(r IN OUT UTL_HTTP.RESP, data OUT VARCHAR2,
remove_crlf BOOLEAN DEFAULT FALSE)
```

Parameters

r

r is the HTTP response record.

data

data is the response body in text form.

remove_crlf

Set *remove_crlf* to TRUE to remove new line characters, otherwise set to FALSE. The default is FALSE.

Examples

The following example retrieves and displays the body of the specified website.

```
DECLARE
    v_req               UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_value         VARCHAR2(1024);
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('http://www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    LOOP
        UTL_HTTP.READ_LINE(v_resp, v_value, TRUE);
        DBMS_OUTPUT.PUT_LINE(v_value);
    END LOOP;
    EXCEPTION
        WHEN OTHERS THEN
            UTL_HTTP.END_RESPONSE(v_resp);
END;
```

The following is the output.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" dir="ltr">

  <!-- ___________________________ HEAD ___________________________ -->

  <head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />


    <title>EnterpriseDB | The Postgres Database Company</title>

    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta name="keywords" content="postgres, postgresql, postgresql installer,
mysql migration, open source database, training, replication" />
<meta name="description" content="The leader in open source database
products, services, support, training and expertise based on PostgreSQL. Free
downloads, documentation, and tutorials." />
<meta name="abstract" content="The Enterprise PostgreSQL Company" />
<link rel="EditURI" type="application/rsd+xml" title="RSD"
href="http://www.enterprisedb.com/blogapi/rsd" />
<link rel="alternate" type="application/rss+xml" title="EnterpriseDB RSS"
href="http://www.enterprisedb.com/rss.xml" />
<link rel="shortcut icon"
href="/sites/all/themes/edb_pixelcrayons/favicon.ico" type="image/x-icon" />
    <link type="text/css" rel="stylesheet" media="all"
href="/sites/default/files/css/css_db11adabae0aed6b79a2c3c52def4754.css" />
<!--[if IE 6]>
<link type="text/css" rel="stylesheet" media="all"
href="/sites/all/themes/oho_basic/css/ie6.css?g" />
<![endif]-->
<!--[if IE 7]>
<link type="text/css" rel="stylesheet" media="all"
href="/sites/all/themes/oho_basic/css/ie7.css?g" />
<![endif]-->
    <script type="text/javascript"
src="/sites/default/files/js/js_74d97b1176812e2fd6e43d62503a5204.js"></script
>
<script type="text/javascript">
<!--//--><![CDATA[//><!--
```

7.18.16 READ_RAW

The `READ_RAW` procedure returns the data from the HTTP response body in binary form. The number of bytes returned is specified by the *`len`* parameter. The signature is:

```
READ_RAW(r IN OUT UTL_HTTP.RESP, data OUT RAW, len INTEGER)
```

Parameters

`r`

`r` is the HTTP response record.

`data`

`data` is the response body in binary form.

`len`

Set *`len`* to the number of bytes of data to be returned.

Examples

The following example retrieves and displays the first 150 bytes in binary form.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_data          RAW;
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('http://www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    UTL_HTTP.READ_RAW(v_resp, v_data, 150);
    DBMS_OUTPUT.PUT_LINE(v_data);
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```

The following is the output from the example.

```
\x3c21444f43545950452068746d6c205055424c494320222d2f2f5733432f2f4454442058485
44d4c20312e30205374726963742f2f454e220d0a202022687474703a2f2f7777772e77332e6f
72672f54522f7868746d6c312f4454442f7868746d6c312d7374726963742e647464223e0d0a3
c68746d6c20786d6c6e733d22687474703a2f2f7777772e77332e6f72672f313939392f
```

7.18.17 READ_TEXT

The `READ_TEXT` procedure returns the data from the HTTP response body in text form. The maximum number of characters returned is specified by the *`len`* parameter. The signature is:

```
READ_TEXT(r IN OUT UTL_HTTP.RESP, data OUT VARCHAR2, len
INTEGER)
```

Parameters

`r`

`r` is the HTTP response record.

`data`

`data` is the response body in text form.

`len`

Set *`len`* to the maximum number of characters to be returned.

Examples

The following example retrieves the first 150 characters.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_data          VARCHAR2(150);
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('http://www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    UTL_HTTP.READ_TEXT(v_resp, v_data, 150);
    DBMS_OUTPUT.PUT_LINE(v_data);
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```

The following is the output.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/
```

7.18.18 REQUEST

The `REQUEST` function returns the first 2000 bytes retrieved from a user-specified URL. The signature is:

```
REQUEST(url IN VARCHAR2) RETURN VARCHAR2
```

If the data found at the given URL is longer than 2000 bytes, the remainder will be discarded. If the data found at the given URL is shorter than 2000 bytes, the result will be shorter than 2000 bytes.

Parameters

url

url is the Uniform Resource Locator from which `UTL_HTTP` will return content.

Example

The following command returns the first 2000 bytes retrieved from the EnterpriseDB website:

```
SELECT UTL_HTTP.REQUEST('http://www.enterprisedb.com/') FROM DUAL;
```

7.18.19 REQUEST_PIECES

The `REQUEST_PIECES` function returns a table of 2000-byte segments retrieved from an URL. The signature is:

```
REQUEST_PIECES(url IN VARCHAR2, max_pieces NUMBER IN
DEFAULT 32767) RETURN UTL_HTTP.HTML_PIECES
```

Parameters

url

url is the Uniform Resource Locator from which `UTL_HTTP` will return content.

max_pieces

max_pieces specifies the maximum number of 2000-byte segments that the `REQUEST_PIECES` function will return. If *max_pieces* specifies more units than are available at the specified *url*, the final unit will contain fewer bytes.

Example

The following example returns the first four 2000 byte segments retrieved from the EnterpriseDB website:

```
DECLARE
    result UTL_HTTP.HTML_PIECES;
BEGIN
result := UTL_HTTP.REQUEST_PIECES('http://www.enterprisedb.com/', 4);
END;
```

7.18.20 SET_BODY_CHARSET

The `SET_BODY_CHARSET` procedure sets the default character set of the body of future HTTP requests. The signature is:

```
SET_BODY_CHARSET(charset VARCHAR2 DEFAULT NULL)
```

Parameters

`charset`

> *`charset`* is the character set of the body of future requests. The default is null in which case the database character set is assumed.

7.18.21 SET_FOLLOW_REDIRECT

The `SET_FOLLOW_REDIRECT` procedure sets the maximum number of times the HTTP redirect instruction is to be followed in the response to this request or future requests. This procedures has two signatures:

```
SET_FOLLOW_REDIRECT(max_redirects IN INTEGER DEFAULT 3)
```

and

```
SET_FOLLOW_REDIRECT(r IN OUT UTL_HTTP.REQ, max_redirects IN
INTEGER DEFAULT 3)
```

Use the second form to change the maximum number of redirections for an individual request that a request inherits from the session default settings.

Parameters

`r`

> *`r`* is the HTTP request record.

max_redirects

max_redirects is maximum number of redirections allowed. Set to 0 to disable redirections. The default is 3.

7.18.22 SET_HEADER

The `SET_HEADER` procedure sets the HTTP request header. The signature is:

```
SET_HEADER(r IN OUT UTL_HTTP.REQ, name IN VARCHAR2, value
IN VARCHAR2 DEFAULT NULL)
```

Parameters

r

r is the HTTP request record.

name

name is the name of the request header.

value

value is the value of the request header. The default is null.

7.18.23 SET_RESPONSE_ERROR_CHECK

The `SET_RESPONSE_ERROR_CHECK` procedure determines whether or not HTTP 4xx and 5xx status codes returned by the `GET_RESPONSE` function should be interpreted as errors. The signature is:

```
SET_RESPONSE_ERROR_CHECK(enable IN BOOLEAN DEFAULT FALSE)
```

Parameters

enable

Set *enable* to `TRUE` if HTTP 4xx and 5xx status codes are to be treated as errors, otherwise set to `FALSE`. The default is `FALSE`.

7.18.24 SET_TRANSFER_TIMEOUT

The `SET_TRANSFER_TIMEOUT` procedure sets the default, transfer timeout setting for waiting for a response from an HTTP request. This procedure has two signatures:

```
SET_TRANSFER_TIMEOUT(timeout IN INTEGER DEFAULT 60)
```

and

```
SET_TRANSFER_TIMEOUT(r IN OUT UTL_HTTP.REQ, timeout IN
INTEGER DEFAULT 60)
```

Use the second form to change the transfer timeout setting for an individual request that a request inherits from the session default settings.

Parameters

r

r is the HTTP request record.

timeout

timeout is the transfer timeout setting in seconds for HTTP requests. The default is 60 seconds.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

7.18.25 WRITE_LINE

The `WRITE_LINE` procedure writes data to the HTTP request body in text form; the text is terminated with a CRLF character pair. The signature is:

```
WRITE_LINE(r IN OUT UTL_HTTP.REQ, data IN VARCHAR2)
```

Parameters

r

r is the HTTP request record.

data

data is the request body in `TEXT` form.

Example

The following example writes data (`Account balance $500.00`) in text form to the request body to be sent using the HTTP `POST` method. The data is sent to a hypothetical web application (`post.php`) that accepts and processes data.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('http://www.example.com/post.php',
       'POST');
    UTL_HTTP.SET_HEADER(v_req, 'Content-Length', '23');
    UTL_HTTP.WRITE_LINE(v_req, 'Account balance $500.00');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    DBMS_OUTPUT.PUT_LINE('Status Code: ' || v_resp.status_code);
    DBMS_OUTPUT.PUT_LINE('Reason Phrase: ' || v_resp.reason_phrase);
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```

Assuming the web application successfully processed the `POST` method, the following output would be displayed:

```
Status Code: 200
Reason Phrase: OK
```

7.18.26 WRITE_RAW

The `WRITE_RAW` procedure writes data to the HTTP request body in binary form. The signature is:

```
WRITE_RAW(r IN OUT UTL_HTTP.REQ, data IN RAW)
```

Parameters

r

r is the HTTP request record.

data

data is the request body in binary form.

Example

The following example writes data in binary form to the request body to be sent using the HTTP `POST` method to a hypothetical web application that accepts and processes such data.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('http://www.example.com/post.php',
       'POST');
    UTL_HTTP.SET_HEADER(v_req, 'Content-Length', '23');
    UTL_HTTP.WRITE_RAW(v_req, HEXTORAW
('54657374696e6720504f5354206d6574686f6420696e20485454502072657175657374'));
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    DBMS_OUTPUT.PUT_LINE('Status Code: ' || v_resp.status_code);
    DBMS_OUTPUT.PUT_LINE('Reason Phrase: ' || v_resp.reason_phrase);
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```

The text string shown in the `HEXTORAW` function is the hexadecimal translation of the text `Testing POST method in HTTP request`.

Assuming the web application successfully processed the `POST` method, the following output would be displayed:

```
Status Code: 200
Reason Phrase: OK
```

7.18.27 WRITE_TEXT

The `WRITE_TEXT` procedure writes data to the HTTP request body in text form. The signature is:

```
WRITE_TEXT(r IN OUT UTL_HTTP.REQ, data IN VARCHAR2)
```

Parameters

r

r is the HTTP request record.

data

data is the request body in text form.

Example

The following example writes data (`Account balance $500.00`) in text form to the request body to be sent using the HTTP `POST` method. The data is sent to a hypothetical web application (`post.php`) that accepts and processes data.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('http://www.example.com/post.php',
       'POST');
    UTL_HTTP.SET_HEADER(v_req, 'Content-Length', '23');
    UTL_HTTP.WRITE_TEXT(v_req, 'Account balance $500.00');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    DBMS_OUTPUT.PUT_LINE('Status Code: ' || v_resp.status_code);
    DBMS_OUTPUT.PUT_LINE('Reason Phrase: ' || v_resp.reason_phrase);
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```

Assuming the web application successfully processed the `POST` method, the following output would be displayed:

```
Status Code: 200
Reason Phrase: OK
```

7.19 UTL_MAIL

The `UTL_MAIL` package provides the capability to manage e-mail.

Note: An administrator must grant execute privileges to each user or group before they can use this package.

Table 7-19 UTL_MAIL Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
`SEND(`*sender*`,` *recipients*`,` *cc*`,` *bcc*`,` *subject*`,` *message* `[,` *mime_type* `[,` *priority* `]])`	Procedure	n/a	Packages and sends an e-mail to an SMTP server.
`SEND_ATTACH_RAW(`*sender*`,` *recipients*`,` *cc*`,` *bcc*`,` *subject*`,` *message*`,` *mime_type*`,` *priority*`,` *attachment* `[,` *att_inline* `[,` *att_mime_type* `[,` *att_filename* `]]])`	Procedure	n/a	Same as the `SEND` procedure, but with `BYTEA` or large object attachments.
`SEND_ATTACH_VARCHAR2(`*sender*`,` *recipients*`,` *cc*`,` *bcc*`,` *subject*`,` *message*`,` *mime_type*`,` *priority*`,` *attachment* `[,` *att_inline* `[,` *att_mime_type* `[,` *att_filename* `]]])`	Procedure	n/a	Same as the `SEND` procedure, but with `VARCHAR2` attachments.

7.19.1 SEND

The `SEND` procedure provides the capability to send an e-mail to an SMTP server.

```
SEND(sender VARCHAR2, recipients VARCHAR2, cc VARCHAR2,
  bcc VARCHAR2, subject VARCHAR2, message VARCHAR2
  [, mime_type VARCHAR2 [, priority PLS_INTEGER ]])
```

Parameters

sender

E-mail address of the sender.

recipients

Comma-separated e-mail addresses of the recipients.

cc

Comma-separated e-mail addresses of copy recipients.

bcc

Comma-separated e-mail addresses of blind copy recipients.

subject

Subject line of the e-mail.

message

Body of the e-mail.

mime_type

Mime type of the message. The default is `text/plain; charset=us-ascii`.

priority

Priority of the e-mail The default is 3.

Examples

The following anonymous block sends a simple e-mail message.

```
DECLARE
    v_sender        VARCHAR2(30);
    v_recipients    VARCHAR2(60);
    v_subj          VARCHAR2(20);
    v_msg           VARCHAR2(200);
BEGIN
    v_sender := 'jsmith@enterprisedb.com';
    v_recipients := 'ajones@enterprisedb.com,rrogers@enterprisedb.com';
    v_subj := 'Holiday Party';
    v_msg := 'This year''s party is scheduled for Friday, Dec. 21 at ' ||
             '6:00 PM. Please RSVP by Dec. 15th.';
    UTL_MAIL.SEND(v_sender,v_recipients,NULL,NULL,v_subj,v_msg);
END;
```

7.19.2 SEND_ATTACH_RAW

The `SEND_ATTACH_RAW` procedure provides the capability to send an e-mail to an SMTP server with an attachment containing either `BYTEA` data or a large object (identified by the large object's `OID`). The call to `SEND_ATTACH_RAW` can be written in two ways:

```
SEND_ATTACH_RAW(sender VARCHAR2, recipients VARCHAR2,
  cc VARCHAR2, bcc VARCHAR2, subject VARCHAR2, message
VARCHAR2,
  mime_type VARCHAR2, priority PLS_INTEGER,
  attachment BYTEA[, att_inline BOOLEAN
  [, att_mime_type VARCHAR2[, att_filename VARCHAR2 ]]])

SEND_ATTACH_RAW(sender VARCHAR2, recipients VARCHAR2,
  cc VARCHAR2, bcc VARCHAR2, subject VARCHAR2, message
VARCHAR2,
  mime_type VARCHAR2, priority PLS_INTEGER, attachment OID
  [, att_inline BOOLEAN [, att_mime_type VARCHAR2
  [, att_filename VARCHAR2 ]]])
```

Parameters

sender

E-mail address of the sender.

recipients

Comma-separated e-mail addresses of the recipients.

cc

Comma-separated e-mail addresses of copy recipients.

bcc

Comma-separated e-mail addresses of blind copy recipients.

subject

Subject line of the e-mail.

message

Body of the e-mail.

mime_type

Mime type of the message. The default is `text/plain; charset=us-ascii`.

priority

Priority of the e-mail. The default is `3`.

attachment

The attachment.

att_inline

If set to TRUE, then the attachment is viewable inline, FALSE otherwise. The default is TRUE.

att_mime_type

Mime type of the attachment. The default is application/octet.

att_filename

The file name containing the attachment. The default is NULL.

7.19.3 SEND_ATTACH_VARCHAR2

The SEND_ATTACH_VARCHAR2 procedure provides the capability to send an e-mail to an SMTP server with a text attachment.

```
SEND_ATTACH_VARCHAR2(sender VARCHAR2, recipients VARCHAR2,
cc VARCHAR2, bcc VARCHAR2, subject VARCHAR2, message
VARCHAR2, mime_type VARCHAR2, priority PLS_INTEGER,
attachment VARCHAR2 [, att_inline BOOLEAN [, att_mime_type
VARCHAR2 [, att_filename VARCHAR2 ]]])
```

Parameters

sender

E-mail address of the sender.

recipients

Comma-separated e-mail addresses of the recipients.

cc

Comma-separated e-mail addresses of copy recipients.

bcc

Comma-separated e-mail addresses of blind copy recipients.

subject

Subject line of the e-mail.

message

Body of the e-mail.

mime_type

Mime type of the message. The default is `text/plain; charset=us-ascii`.

priority

Priority of the e-mail The default is 3.

attachment

The `VARCHAR2` attachment.

att_inline

If set to `TRUE`, then the attachment is viewable inline, `FALSE` otherwise. The default is `TRUE`.

att_mime_type

Mime type of the attachment. The default is `text/plain; charset=us-ascii`.

att_filename

The file name containing the attachment. The default is `NULL`.

7.20 UTL_RAW

The `UTL_RAW` package allows you to manipulate or retrieve the length of raw data types.

Note: An administrator must grant execute privileges to each user or group before they can use this package.

Function/Procedure	Function or Procedure	Return Type	Description
`CAST_TO_RAW(`*`c`*` IN VARCHAR2)`	Function	`RAW`	Converts a `VARCHAR2` string to a `RAW` value.
`CAST_TO_VARCHAR2(`*`r`*` IN RAW)`	Function	`VARCHAR2`	Converts a `RAW` value to a `VARCHAR2` string.
`CONCAT(`*`r1`*` IN RAW, `*`r2`*` IN RAW, `*`r3`*` IN RAW,…)`	Function	`RAW`	Concatenate multiple `RAW` values into a single `RAW` value.
`CONVERT(`*`r`*` IN RAW, `*`to_charset`*` IN VARCHAR2, `*`from_charset`*` IN VARCHAR2`	Function	`RAW`	Converts encoded data from one encoding to another, and returns the result as a `RAW` value.
`LENGTH(`*`r`*` IN RAW)`	Function	`NUMBER`	Returns the length of a `RAW` value.
`SUBSTR(`*`r`*` IN RAW, `*`pos`*` IN INTEGER, `*`len`*` IN INTEGER)`	Function	`RAW`	Returns a portion of a `RAW` value.

Advanced Server's implementation of `UTL_RAW` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

7.20.1 CAST_TO_RAW

The `CAST_TO_RAW` function converts a `VARCHAR2` string to a `RAW` value. The signature is:

```
CAST_TO_RAW(c VARCHAR2)
```

The function returns a `RAW` value if you pass a non-`NULL` value; if you pass a `NULL` value, the function will return `NULL`.

Parameters

c

The `VARCHAR2` value that will be converted to `RAW`.

Example

The following example uses the CAST_TO_RAW function to convert a VARCHAR2 string to a RAW value:

```
DECLARE
  v VARCHAR2;
  r RAW;
BEGIN
  v := 'Accounts';
  dbms_output.put_line(v);
  r := UTL_RAW.CAST_TO_RAW(v);
  dbms_output.put_line(r);
END;
```

The result set includes the content of the original string and the converted RAW value:

```
Accounts
\x4163636f756e7473
```

7.20.2 CAST_TO_VARCHAR2

The CAST_TO_VARCHAR2 function converts RAW data to VARCHAR2 data. The signature is:

```
CAST_TO_VARCHAR2(r RAW)
```

The function returns a VARCHAR2 value if you pass a non-NULL value; if you pass a NULL value, the function will return NULL.

Parameters

r

The RAW value that will be converted to a VARCHAR2 value.

Example

The following example uses the CAST_TO_VARCHAR2 function to convert a RAW value to a VARCHAR2 string:

```
DECLARE
  r RAW;
  v VARCHAR2;
BEGIN
  r := '\x4163636f756e7473'
  dbms_output.put_line(v);
  v := UTL_RAW.CAST_TO_VARCHAR2(r);
  dbms_output.put_line(r);
END;
```

The result set includes the content of the original string and the converted `RAW` value:

```
\x4163636f756e7473
Accounts
```

7.20.3 CONCAT

The `CONCAT` function concatenates multiple `RAW` values into a single `RAW` value. The signature is:

`CONCAT(`*`r1`*` RAW, `*`r2`*` RAW, `*`r3`*` RAW,…)`

The function returns a `RAW` value. Unlike the Oracle implementation, the Advanced Server implementation is a variadic function, and does not place a restriction on the number of values that can be concatenated.

Parameters

`r1, r2, r3,…`

The `RAW` values that `CONCAT` will concatenate.

Example

The following example uses the `CONCAT` function to concatenate multiple `RAW` values into a single `RAW` value:

```
SELECT UTL_RAW.CAST_TO_VARCHAR2(UTL_RAW.CONCAT('\x61', '\x62', '\x63')) FROM
DUAL;
 concat
--------
 abc
(1 row)
```

The result (the concatenated values) is then converted to `VARCHAR2` format by the `CAST_TO_VARCHAR2` function.

7.20.4 CONVERT

The `CONVERT` function converts a string from one encoding to another encoding and returns the result as a `RAW` value. The signature is:

```
CONVERT(r RAW, to_charset VARCHAR2, from_charset VARCHAR2)
```

The function returns a `RAW` value.

Parameters

`r`

The `RAW` value that will be converted.

`to_charset`

The name of the encoding to which *`r`* will be converted.

`from_charset`

The name of the encoding from which *`r`* will be converted.

Example

The following example uses the `UTL_RAW.CAST_TO_RAW` function to convert a `VARCHAR2` string (`Accounts`) to a raw value, and then convert the value from `UTF8` to `LATIN7`, and then from `LATIN7` to `UTF8`:

```
DECLARE
  r RAW;
  v VARCHAR2;
BEGIN
  v:= 'Accounts';
  dbms_output.put_line(v);
  r:= UTL_RAW.CAST_TO_RAW(v);
  dbms_output.put_line(r);
  r:= UTL_RAW.CONVERT(r, 'UTF8', 'LATIN7');
  dbms_output.put_line(r);
  r:= UTL_RAW.CONVERT(r, 'LATIN7', 'UTF8');
  dbms_output.put_line(r);
```

The example returns the `VARCHAR2` value, the `RAW` value, and the converted values:

```
Accounts
\x4163636f756e7473
\x4163636f756e7473
\x4163636f756e7473
```

7.20.5 LENGTH

The `LENGTH` function returns the length of a `RAW` value. The signature is:

```
LENGTH(r RAW)
```

The function returns a `RAW` value.

Parameters

`r`

The `RAW` value that `LENGTH` will evaluate.

Example

The following example uses the `LENGTH` function to return the length of a `RAW` value:

```
SELECT UTL_RAW.LENGTH(UTL_RAW.CAST_TO_RAW('Accounts')) FROM DUAL;
 length
--------
8
(1 row)
```

The following example uses the `LENGTH` function to return the length of a `RAW` value that includes multi-byte characters:

```
SELECT UTL_RAW.LENGTH(UTL_RAW.CAST_TO_RAW('独孤求败'));
 length
--------
     12
(1 row)
```

7.20.6 SUBSTR

The `SUBSTR` function returns a substring of a `RAW` value. The signature is:

```
SUBSTR (r RAW, pos INTEGER, len INTEGER)
```

This function returns a `RAW` value.

Parameters

r

The RAW value from which the substring will be returned.

pos

The position within the RAW value of the first byte of the returned substring.

- If *pos* is 0 or 1, the substring begins at the first byte of the RAW value.
- If *pos* is greater than one, the substring begins at the first byte specified by *pos*. For example, if *pos* is 3, the substring begins at the third byte of the value.
- If *pos* is negative, the substring begins at *pos* bytes from the end of the source value. For example, if *pos* is -3, the substring begins at the third byte from the end of the value.

len

The maximum number of bytes that will be returned.

Example

The following example uses the SUBSTR function to select a substring that begins 3 bytes from the start of a RAW value:

```
SELECT UTL_RAW.SUBSTR(UTL_RAW.CAST_TO_RAW('Accounts'), 3, 5) FROM DUAL;
 substr
--------
 count
(1 row)
```

The following example uses the SUBSTR function to select a substring that starts 5 bytes from the end of a RAW value:

```
SELECT UTL_RAW.SUBSTR(UTL_RAW.CAST_TO_RAW('Accounts'), -5 , 3) FROM DUAL;
 substr
--------
 oun
(1 row)
```

7.21 UTL_SMTP

The `UTL_SMTP` package provides the capability to send e-mails over the Simple Mail Transfer Protocol (SMTP).

Note: An administrator must grant execute privileges to each user or group before they can use this package.

Table 7-20 UTL_SMTP Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
`CLOSE_DATA(`*`c`*` IN OUT)`	Procedure	`n/a`	Ends an e-mail message.
`COMMAND(`*`c`*` IN OUT, `*`cmd`*` [, `*`arg`*` ])`	Both	`REPLY`	Execute an SMTP command.
`COMMAND_REPLIES(`*`c`*` IN OUT, `*`cmd`*` [, `*`arg`*` ])`	Function	`REPLIES`	Execute an SMTP command where multiple reply lines are expected.
`DATA(`*`c`*` IN OUT, `*`body`*` VARCHAR2)`	Procedure	`n/a`	Specify the body of an e-mail message.
`EHLO(`*`c`*` IN OUT, `*`domain`*`)`	Procedure	`n/a`	Perform initial handshaking with an SMTP server and return extended information.
`HELO(`*`c`*` IN OUT, `*`domain`*`)`	Procedure	`n/a`	Perform initial handshaking with an SMTP server
`HELP(c IN OUT [, command ])`	Function	`REPLIES`	Send the `HELP` command.
`MAIL(`*`c`*` IN OUT, `*`sender`*` [, `*`parameters`*` ])`	Procedure	`n/a`	Start a mail transaction.
`NOOP(`*`c`*` IN OUT)`	Both	`REPLY`	Send the null command.
`OPEN_CONNECTION(`*`host`*` [, `*`port`*` [, `*`tx_timeout`*` ]])`	Function	`CONNECTION`	Open a connection.
`OPEN_DATA(`*`c`*` IN OUT)`	Both	`REPLY`	Send the `DATA` command.
`QUIT(`*`c`*` IN OUT)`	Procedure	`n/a`	Terminate the SMTP session and disconnect.
`RCPT(`*`c`*` IN OUT, `*`recipient`*` [, `*`parameters`*` ])`	Procedure	`n/a`	Specify the recipient of an e-mail message.
`RSET(`*`c`*` IN OUT)`	Procedure	`n/a`	Terminate the current mail transaction.
`VRFY(`*`c`*` IN OUT, `*`recipient`*`)`	Function	`REPLY`	Validate an e-mail address.
`WRITE_DATA(`*`c`*` IN OUT, `*`data`*`)`	Procedure	n/a	Write a portion of the e-mail message.

Advanced Server's implementation of `UTL_SMTP` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

The following table lists the public variables available in the `UTL_SMTP` package.

Table 7-21 UTL_SMTP Public Variables

Public Variables	Data Type	Value	Description
`connection`	`RECORD`		Description of an SMTP connection.

Public Variables	Data Type	Value	Description
reply	RECORD		SMTP reply line.

7.21.1 CONNECTION

The CONNECTION record type provides a description of an SMTP connection.

```
TYPE connection IS RECORD (
    host             VARCHAR2(255),
    port             PLS_INTEGER,
    tx_timeout       PLS_INTEGER
);
```

7.21.2 REPLY/REPLIES

The REPLY record type provides a description of an SMTP reply line. REPLIES is a table of multiple SMTP reply lines.

```
TYPE reply IS RECORD (
    code             INTEGER,
    text             VARCHAR2(508)
);
TYPE replies IS TABLE OF reply INDEX BY BINARY_INTEGER;
```

7.21.3 CLOSE_DATA

The CLOSE_DATA procedure terminates an e-mail message by sending the following sequence:

    <CR><LF>.<CR><LF>

This is a single period at the beginning of a line.

CLOSE_DATA(*c* IN OUT CONNECTION)

Parameters

c

The SMTP connection to be closed.

7.21.4 COMMAND

The `COMMAND` procedure provides the capability to execute an SMTP command. If you are expecting multiple reply lines, use `COMMAND_REPLIES`.

```
reply REPLY COMMAND(c IN OUT CONNECTION, cmd VARCHAR2
  [, arg VARCHAR2 ])

COMMAND(c IN OUT CONNECTION, cmd VARCHAR2 [, arg VARCHAR2
])
```

Parameters

c

The SMTP connection to which the command is to be sent.

cmd

The SMTP command to be processed.

arg

An argument to the SMTP command. The default is null.

reply

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in *reply*.

See Section 7.21.2 for a description of `REPLY` and `REPLIES`.

7.21.5 COMMAND_REPLIES

The `COMMAND_REPLIES` function processes an SMTP command that returns multiple reply lines. Use `COMMAND` if only a single reply line is expected.

```
replies REPLIES COMMAND(c IN OUT CONNECTION, cmd VARCHAR2
  [, arg VARCHAR2 ])
```

Parameters

c

The SMTP connection to which the command is to be sent.

cmd

The SMTP command to be processed.

arg

An argument to the SMTP command. The default is null.

replies

SMTP reply lines to the command. See Section 7.21.2 for a description of `REPLY` and `REPLIES`.

7.21.6 DATA

The `DATA` procedure provides the capability to specify the body of the e-mail message. The message is terminated with a `<CR><LF>.<CR><LF>` sequence.

```
DATA(c IN OUT CONNECTION, body VARCHAR2)
```

Parameters

c

The SMTP connection to which the command is to be sent.

body

Body of the e-mail message to be sent.

7.21.7 EHLO

The `EHLO` procedure performs initial handshaking with the SMTP server after establishing the connection. The `EHLO` procedure allows the client to identify itself to the SMTP server according to RFC 821. RFC 1869 specifies the format of the information returned in the server's reply. The HELO procedure performs the equivalent functionality, but returns less information about the server.

```
EHLO(c IN OUT CONNECTION, domain VARCHAR2)
```

Parameters

c

The connection to the SMTP server over which to perform handshaking.

domain

Domain name of the sending host.

7.21.8 HELO

The `HELO` procedure performs initial handshaking with the SMTP server after establishing the connection. The `HELO` procedure allows the client to identify itself to the SMTP server according to RFC 821. The EHLO procedure performs the equivalent functionality, but returns more information about the server.

```
HELO(c IN OUT, domain VARCHAR2)
```

Parameters

c

The connection to the SMTP server over which to perform handshaking.

domain

Domain name of the sending host.

7.21.9 HELP

The `HELP` function provides the capability to send the `HELP` command to the SMTP server.

```
replies REPLIES HELP(c IN OUT CONNECTION [, command
VARCHAR2 ])
```

Parameters

c

The SMTP connection to which the command is to be sent.

command

Command on which help is requested.

replies

SMTP reply lines to the command. See Section 7.21.2 for a description of `REPLY` and `REPLIES`.

7.21.10 MAIL

The `MAIL` procedure initiates a mail transaction.

```
MAIL(c IN OUT CONNECTION, sender VARCHAR2
  [, parameters VARCHAR2 ])
```

Parameters

c

Connection to SMTP server on which to start a mail transaction.

sender

The sender's e-mail address.

parameters

Mail command parameters in the format, `key=value` as defined in RFC 1869, Section 6.

7.21.11 NOOP

The `NOOP` function/procedure sends the null command to the SMTP server. The `NOOP` has no effect upon the server except to obtain a successful response.

```
reply REPLY NOOP(c IN OUT CONNECTION)

NOOP(c IN OUT CONNECTION)
```

Parameters

c

The SMTP connection on which to send the command.

reply

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in *reply*. See Section 7.21.2 for a description of `REPLY` and `REPLIES`.

7.21.12 OPEN_CONNECTION

The `OPEN_CONNECTION` functions open a connection to an SMTP server.

```
c CONNECTION OPEN_CONNECTION(host VARCHAR2 [, port
PLS_INTEGER [, tx_timeout PLS_INTEGER DEFAULT NULL]])
```

Parameters

host

Name of the SMTP server.

port

Port number on which the SMTP server is listening. The default is 25.

tx_timeout

Time out value in seconds. Do not wait is indicated by specifying 0. Wait indefinitely is indicated by setting timeout to null. The default is null.

c

Connection handle returned by the SMTP server.

7.21.13 OPEN_DATA

The `OPEN_DATA` procedure sends the `DATA` command to the SMTP server.

```
OPEN_DATA(c IN OUT CONNECTION)
```

Parameters

c

SMTP connection on which to send the command.

7.21.14 QUIT

The `QUIT` procedure closes the session with an SMTP server.

```
QUIT(c IN OUT CONNECTION)
```

Parameters

c

SMTP connection to be terminated.

7.21.15 RCPT

The `RCPT` procedure provides the e-mail address of the recipient. To schedule multiple recipients, invoke `RCPT` multiple times.

```
RCPT(c IN OUT CONNECTION, recipient VARCHAR2
  [, parameters VARCHAR2 ])
```

Parameters

c

Connection to SMTP server on which to add a recipient.

recipient

The recipient's e-mail address.

parameters

Mail command parameters in the format, `key=value` as defined in RFC 1869, Section 6.

7.21.16 RSET

The `RSET` procedure provides the capability to terminate the current mail transaction.

```
RSET(c IN OUT CONNECTION)
```

Parameters

c

SMTP connection on which to cancel the mail transaction.

7.21.17 VRFY

The `VRFY` function provides the capability to validate and verify the recipient's e-mail address. If valid, the recipient's full name and fully qualified mailbox is returned.

```
reply REPLY VRFY(c IN OUT CONNECTION, recipient VARCHAR2)
```

Parameters

c

The SMTP connection on which to verify the e-mail address.

recipient

The recipient's e-mail address to be verified.

reply

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in *reply*. See Section 7.21.2 for a description of `REPLY` and `REPLIES`.

7.21.18 WRITE_DATA

The `WRITE_DATA` procedure provides the capability to add `VARCHAR2` data to an e-mail message. The `WRITE_DATA` procedure may be repetitively called to add data.

```
WRITE_DATA(c IN OUT CONNECTION, data VARCHAR2)
```

Parameters

c

The SMTP connection on which to add data.

data

Data to be added to the e-mail message. The data must conform to the RFC 822 specification.

7.21.19 Comprehensive Example

The following procedure constructs and sends a text e-mail message using the `UTL_SMTP` package.

```
CREATE OR REPLACE PROCEDURE send_mail (
    p_sender        VARCHAR2,
    p_recipient     VARCHAR2,
    p_subj          VARCHAR2,
    p_msg           VARCHAR2,
    p_mailhost      VARCHAR2
)
IS
    v_conn          UTL_SMTP.CONNECTION;
    v_crlf          CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);
    v_port          CONSTANT PLS_INTEGER := 25;
BEGIN
    v_conn := UTL_SMTP.OPEN_CONNECTION(p_mailhost,v_port);
    UTL_SMTP.HELO(v_conn,p_mailhost);
    UTL_SMTP.MAIL(v_conn,p_sender);
    UTL_SMTP.RCPT(v_conn,p_recipient);
    UTL_SMTP.DATA(v_conn, SUBSTR(
        'Date: ' || TO_CHAR(SYSDATE,
        'Dy, DD Mon YYYY HH24:MI:SS') || v_crlf
        || 'From: ' || p_sender || v_crlf
        || 'To: ' || p_recipient || v_crlf
        || 'Subject: ' || p_subj || v_crlf
        || p_msg
        , 1, 32767));
    UTL_SMTP.QUIT(v_conn);
END;

EXEC send_mail('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday
Party','Are you planning to attend?','smtp.enterprisedb.com');
```

The following example uses the `OPEN_DATA`, `WRITE_DATA`, and `CLOSE_DATA` procedures instead of the `DATA` procedure.

```
CREATE OR REPLACE PROCEDURE send_mail_2 (
    p_sender        VARCHAR2,
    p_recipient     VARCHAR2,
    p_subj          VARCHAR2,
    p_msg           VARCHAR2,
    p_mailhost      VARCHAR2
)
IS
    v_conn          UTL_SMTP.CONNECTION;
    v_crlf          CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);
    v_port          CONSTANT PLS_INTEGER := 25;
BEGIN
    v_conn := UTL_SMTP.OPEN_CONNECTION(p_mailhost,v_port);
    UTL_SMTP.HELO(v_conn,p_mailhost);
    UTL_SMTP.MAIL(v_conn,p_sender);
    UTL_SMTP.RCPT(v_conn,p_recipient);
    UTL_SMTP.OPEN_DATA(v_conn);
    UTL_SMTP.WRITE_DATA(v_conn,'From: ' || p_sender || v_crlf);
    UTL_SMTP.WRITE_DATA(v_conn,'To: ' || p_recipient || v_crlf);
    UTL_SMTP.WRITE_DATA(v_conn,'Subject: ' || p_subj || v_crlf);
    UTL_SMTP.WRITE_DATA(v_conn,v_crlf || p_msg);
    UTL_SMTP.CLOSE_DATA(v_conn);
    UTL_SMTP.QUIT(v_conn);
END;

EXEC send_mail_2('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday
Party','Are you planning to attend?','smtp.enterprisedb.com');
```

7.22 UTL_URL

The UTL_URL package provides a way to escape illegal and reserved characters within an URL.

Table 7.7.2 UTL_HTTP Functions and Procedures

Function/Procedure	Return Type	Description
`ESCAPE(url, escape_reserved_chars, url_charset)`	VARCHAR2	Use the ESCAPE function to escape any illegal and reserved characters in a URL.
`UNESCAPE(url, url_charset)`	VARCHAR2	The UNESCAPE function to convert an URL to it's original form.

The UTL_URL package will return the BAD_URL exception if the call to a function includes an incorrectly-formed URL.

7.22.1 ESCAPE

Use the ESCAPE function to escape illegal and reserved characters within an URL. The signature is:

```
ESCAPE(url VARCHAR2, escape_reserved_chars BOOLEAN,
url_charset VARCHAR2)
```

Reserved characters are replaced with a percent sign, followed by the two-digit hex code of the ascii value for the escaped character.

Parameters

url

url specifies the Uniform Resource Locator that UTL_URL will escape.

escape_reserved_chars

escape_reserved_chars is a BOOLEAN value that instructs the ESCAPE function to escape reserved characters as well as illegal characters:

- If *escaped_reserved_*chars is FALSE, ESCAPE will escape only the illegal characters in the specified URL.
- If *escape_reserved_chars* is TRUE, ESCAPE will escape both the illegal characters and the reserved characters in the specified URL.

By default, *escape_reserved_chars* is FALSE.

Within an URL, legal characters are:

Uppercase A through Z	Lowercase a through z	0 through 9
asterisk (*)	exclamation point (!)	hyphen (-)
left parenthesis (()	period (.)	right parenthesis ())
single-quote (')	tilde (~)	underscore (_)

Some characters are legal in some parts of an URL, while illegal in others; to review comprehensive rules about illegal characters, please refer to RFC 2396. Some *examples* of characters that are considered illegal in any part of an URL are:

Illegal Character	Escape Sequence
a blank space ( )	`%20`
curly braces ({ or })	`%7b` and `%7d`
hash mark (#)	`%23`

The `ESCAPE` function considers the following characters to be reserved, and will escape them if `escape_reserved_chars` is set to `TRUE`:

Reserved Character	Escape Sequence
ampersand (&)	`%5C`
at sign (@)	`%25`
colon (:)	`%3a`
comma (,)	`%2c`
dollar sign ($)	`%24`
equal sign (=)	`%3d`
plus sign (+)	`%2b`
question mark (?)	`%3f`
semi-colon (;)	`%3b`
slash (/)	`%2f`

`url_charset`

`url_charset` specifies a character set to which a given character will be converted before it is escaped. If `url_`*`charset`* is `NULL`, the character will not be converted. The default value of *`url_charset`* is `ISO-8859-1`.

Examples

The following anonymous block uses the `ESCAPE` function to escape the blank spaces in the URL:

```
DECLARE
  result varchar2(400);
BEGIN
 result := UTL_URL.ESCAPE('http://www.example.com/Using the ESCAPE
function.html');
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

The resulting (escaped) URL is:

```
http://www.example.com/Using%20the%20ESCAPE%20function.html
```

If you include a value of `TRUE` for the *`escape_reserved_chars`* parameter when invoking the function:

```
DECLARE
  result varchar2(400);
BEGIN
 result := UTL_URL.ESCAPE('http://www.example.com/Using the ESCAPE
function.html', TRUE);
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

The `ESCAPE` function escapes the reserved characters as well as the illegal characters in the URL:

```
http%3A%2F%2Fwww.example.com%2FUsing%20the%20ESCAPE%20function.html
```

7.22.2 UNESCAPE

The `UNESCAPE` function removes escape characters added to an URL by the `ESCAPE` function, converting the URL to it's original form.

The signature is:

UNESCAPE(*url* VARCHAR2, *url_charset* VARCHAR2)

Parameters

url

url specifies the Uniform Resource Locator that `UTL_URL` will unescape.

url_charset

After unescaping a character, the character is assumed to be in *url_charset* encoding, and will be converted from that encoding to database encoding before being returned. If `url_`*charset* is `NULL`, the character will not be converted. The default value of *url_charset* is `ISO-8859-1`.

Examples

The following anonymous block uses the `ESCAPE` function to escape the blank spaces in the URL:

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

```
DECLARE
  result varchar2(400);
BEGIN
 result :=
UTL_URL.UNESCAPE('http://www.example.com/Using%20the%20UNESCAPE%20function.ht
ml');
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

The resulting (unescaped) URL is:

```
http://www.example.com/Using the UNESCAPE function.html
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

8 Object Types and Objects

This chapter discusses how object-oriented programming techniques can be exploited in SPL. Object-oriented programming as seen in programming languages such as Java and C++ centers on the concept of objects. An *object* is a representation of a real-world entity such as a person, place, or thing. The generic description or definition of a particular object such as a person for example, is called an *object type*. Specific people such as "Joe" or "Sally" are said to be *objects of object type*, person, or equivalently, *instances* of the object type, person, or simply, person objects.

Note: The terms "database objects" and "objects" that have been used in this document up to this point should not be confused with an object type and object as used in this chapter. The previous usage of these terms relates to the entities that can be created in a database such as tables, views, indexes, users, etc. Within the context of this chapter, object type and object refer to specific data structures supported by the SPL programming language to implement object-oriented concepts.

Note: In Oracle, the term *abstract data type* (ADT) is used to describe object types in PL/SQL. The SPL implementation of object types is intended to be compatible with Oracle abstract data types.

Note: Advanced Server has not yet implemented support for some features of object-oriented programming languages. This chapter documents only those features that have been implemented.

8.1 Basic Object Concepts

An object type is a description or definition of some entity. This definition of an object type is characterized by two components:

- *Attributes* – fields that describe particular characteristics of an object instance. For a person object, examples might be name, address, gender, date of birth, height, weight, eye color, occupation, etc.
- *Methods* – programs that perform some type of function or operation on, or related to an object. For a person object, examples might be calculating the person's age, displaying the person's attributes, changing the values assigned to the person's attributes, etc.

The following sections elaborate on some basic object concepts.

8.1.1 Attributes

Every object type must contain at least one attribute. The data type of an attribute can be any of the following:

- A base data type such as `NUMBER`, `VARCHAR2`, etc.
- Another object type
- A globally defined collection type (created by the `CREATE TYPE` command) such as a nested table or varray

An attribute gets its initial value (which may be null) when an object instance is initially created. Each object instance has its own set of attribute values.

8.1.2 Methods

Methods are SPL procedures or functions defined within an object type. Methods can be categorized into three general types:

- *Member Methods* – procedures or functions that operate within the context of an object instance. Member methods have access to, and can change the attributes of the object instance on which they are operating.
- *Static Methods* – procedures or functions that operate independently of any particular object instance. Static methods do not have access to, and cannot change the attributes of an object instance.
- *Constructor Methods* – functions used to create an instance of an object type. A default constructor method is always provided when an object type is defined.

8.1.3 Overloading Methods

In an object type it is permissible to define two or more identically named methods of the same type (this is, either a procedure or function), but with different signatures. Such methods are referred to as *overloaded* methods.

A method's signature consists of the number of formal parameters, the data types of its formal parameters, and their order.

8.2 Object Type Components

Object types are created and stored in the database by using the following two constructs of the SPL language:

- The *object type specification* - This is the public interface specifying the attributes and method signatures of the object type.
- The *object type body* - This contains the implementation of the methods specified in the object type specification.

The following sections describe the commands used to create the object type specification and the object type body.

8.2.1 Object Type Specification Syntax

The following is the syntax of the object type specification:

```
CREATE [ OR REPLACE ] TYPE name
  [ AUTHID { DEFINER | CURRENT_USER } ]
  { IS | AS } OBJECT
( { attribute { datatype | objtype | collecttype } }
    [, ...]
  [ method_spec ] [, ...]
  [ constructor ] [, ...]
) [ [ NOT ] { FINAL | INSTANTIABLE } ] ...;
```

where `method_spec` is the following:

```
[ [ NOT ] { FINAL | INSTANTIABLE } ] ...
[ OVERRIDING ]
  subprogram_spec
```

where `subprogram_spec` is the following:

```
{ MEMBER | STATIC }
{ PROCEDURE proc_name
    [ ( [  SELF [ IN | IN OUT ] name ]
        [, parm1 [ IN | IN OUT | OUT ] datatype1
                 [ DEFAULT value1 ] ]
        [, parm2 [ IN | IN OUT | OUT ] datatype2
                 [ DEFAULT value2 ]
        ] ...)
    ]
|
  FUNCTION func_name
    [ ( [  SELF [ IN | IN OUT ] name ]
        [, parm1 [ IN | IN OUT | OUT ] datatype1
                 [ DEFAULT value1 ] ]
```

```
          [, parm2 [ IN | IN OUT | OUT ] datatype2
                   [ DEFAULT value2 ]
          ] ...)
      ]
    RETURN return_type
  }
```

where *constructor* is the following:

```
    CONSTRUCTOR func_name
      [ ( [  SELF [ IN | IN OUT ] name ]
          [, parm1 [ IN | IN OUT | OUT ] datatype1
                   [ DEFAULT value1 ] ]
          [, parm2 [ IN | IN OUT | OUT ] datatype2
                   [ DEFAULT value2 ]
          ] ...)
      ]
    RETURN self AS RESULT
```

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.

`name` is an identifier (optionally schema-qualified) assigned to the object type.

If the `AUTHID` clause is omitted or `DEFINER` is specified, the rights of the object type owner are used to determine access privileges to database objects. If `CURRENT_USER` is specified, the rights of the current user executing a method in the object are used to determine access privileges.

`attribute` is an identifier assigned to an attribute of the object type.

`datatype` is a base data type.

`objtype` is a previously defined object type.

`collecttype` is a previously defined collection type.

Following the closing parenthesis of the `CREATE TYPE` definition, `[ NOT ] FINAL` specifies whether or not a subtype can be derived from this object type. `FINAL`, which is the default, means that no subtypes can be derived from this object type. Specify NOT FINAL if you want to allow subtypes to be defined under this object type.

Note: Even though the specification of NOT FINAL is accepted in the CREATE TYPE command, SPL does not currently support the creation of subtypes.

Following the closing parenthesis of the CREATE TYPE definition, [NOT] INSTANTIABLE specifies whether or not an object instance can be created of this object type. INSTANTIABLE, which is the default, means that an instance of this object type can be created. Specify NOT INSTANTIABLE if this object type is to be used only as a parent "template" from which other specialized subtypes are to be defined. If NOT INSTANTIABLE is specified, then NOT FINAL must be specified as well. If 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.

Note: Even though the specification of NOT INSTANTIABLE is accepted in the CREATE TYPE command, SPL does not currently support the creation of subtypes.

method_spec denotes the specification of a member method or static method.

Prior to the definition of a method, [NOT] FINAL specifies whether or not the method can be overridden in a subtype. NOT FINAL is the default meaning the method can be overridden in a subtype.

Prior to the definition of a method specify OVERRIDING if the method overrides an identically named method in a supertype. The overriding method must have the same number of identically named method parameters with the same data types and parameter modes, in the same order, and the same return type (if the method is a function) as defined in the supertype.

Prior to the definition of a method, [NOT] INSTANTIABLE specifies whether or not the object type definition provides an implementation for the method. If INSTANTIABLE is specified, then the CREATE TYPE BODY command for the object type must specify the implementation of the method. If NOT INSTANTIABLE is specified, then the CREATE TYPE BODY command for the object type must not contain the implementation of the method. In this latter case, it is assumed a subtype contains the implementation of the method, overriding the method in this object type. If there are any NOT INSTANTIABLE methods in the object type, then the object type definition itself, must specify NOT INSTANTIABLE and NOT FINAL following the closing parenthesis of the object type specification. The default is INSTANTIABLE.

subprogram_spec denotes the specification of a procedure or function and begins with the specification of either MEMBER or STATIC. A member subprogram must be invoked with respect to a particular object instance while a static subprogram is not invoked with respect to any object instance.

`proc_name` is an identifier of a procedure. If the `SELF` parameter is specified, *`name`* is the object type name given in the `CREATE TYPE` command. If specified, *`parm1`*, *`parm2`*, … are the formal parameters of the procedure. *`datatype1`*, *`datatype2`*, … are the data types of *`parm1`*, *`parm2`*, … respectively. `IN`, `IN OUT`, and `OUT` are the possible parameter modes for each formal parameter. If none are specified, the default is `IN`. *`value1`*, *`value2`*, … are default values that may be specified for `IN` parameters.

Include the `CONSTRUCTOR` keyword and function definition to define a constructor function.

`func_name` is an identifier of a function. If the `SELF` parameter is specified, *`name`* is the object type name given in the `CREATE TYPE` command. If specified, *`parm1`*, *`parm2`*, … are the formal parameters of the function. *`datatype1`*, *`datatype2`*, … are the data types of *`parm1`*, *`parm2`*, … respectively. `IN`, `IN OUT`, and `OUT` are the possible parameter modes for each formal parameter. If none are specified, the default is `IN`. *`value1`*, *`value2`*, … are default values that may be specified for `IN` parameters. *`return_type`* is the data type of the value the function returns.

The following points should be noted about an object type specification:

- There must be at least one attribute defined in the object type.

- There may be none, one, or more methods defined in the object type.

- For each 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 (for example as `MEMBER FUNCTION (SELF IN OUT` *`object_type`* `...))`.

  If `SELF` is 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.

- A static method cannot be overridden (`OVERRIDING` and `STATIC` cannot be specified together in *`method_spec`*).

- A static method must be instantiable (`NOT INSTANTIABLE` and `STATIC` cannot be specified together in *`method_spec`*).

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

8.2.2 Object Type Body Syntax

The following is the syntax of the object type body:

```
CREATE [ OR REPLACE ] TYPE BODY name
  { IS | AS }
  method_spec [...]
  [constructor] [...]
END;
```

where `method_spec` is the following:

```
  subprogram_spec
```

and `subprogram_spec` is the following:

```
{ MEMBER | STATIC }
{ PROCEDURE proc_name
    [ ( [  SELF [ IN | IN OUT ] name ]
        [, parm1 [ IN | IN OUT | OUT ] datatype1
                 [ DEFAULT value1 ] ]
        [, parm2 [ IN | IN OUT | OUT ] datatype2
                 [ DEFAULT value2 ]
        ] ...)
    ]
{ IS | AS }
[ variable_declaration; ] ...
  BEGIN
    statement; ...
[ EXCEPTION
    WHEN ... THEN
      statement; ...]
  END;
|
  FUNCTION func_name
    [ ( [  SELF [ IN | IN OUT ] name ]
        [, parm1 [ IN | IN OUT | OUT ] datatype1
                 [ DEFAULT value1 ] ]
        [, parm2 [ IN | IN OUT | OUT ] datatype2
                 [ DEFAULT value2 ]
        ] ...)
    ]
  RETURN return_type
{ IS | AS }
[ variable_declaration; ] ...
  BEGIN
  statement; ...
[ EXCEPTION
    WHEN ... THEN
      statement; ...]
      END;
```

where *constructor* is:

```
      CONSTRUCTOR func_name
        [ ( [  SELF [ IN | IN OUT ] name ]
            [, parm1 [ IN | IN OUT | OUT ] datatype1
                     [ DEFAULT value1 ] ]
            [, parm2 [ IN | IN OUT | OUT ] datatype2
                     [ DEFAULT value2 ]
            ] ...)
        ]
      RETURN self AS RESULT
    { IS | AS }
    [ variable_declaration; ] ...
      BEGIN
      statement; ...
    [ EXCEPTION
        WHEN ... THEN
          statement; ...]
      END;
```

name is an identifier (optionally schema-qualified) assigned to the object type.

method_spec denotes the implementation of an instantiable method that was specified in the CREATE TYPE command.

If INSTANTIABLE was specified or omitted in *method_spec* of the CREATE TYPE command, then there must be a *method_spec* for this method in the CREATE TYPE BODY command.

If NOT INSTANTIABLE was specified in *method_spec* of the CREATE TYPE command, then there must be no *method_spec* for this method in the CREATE TYPE BODY command.

subprogram_spec denotes the specification of a procedure or function and begins with the specification of either MEMBER or STATIC. The same qualifier must be used as was specified in *subprogram_spec* of the CREATE TYPE command.

proc_name is an identifier of a procedure specified in the CREATE TYPE command. The parameter declarations have the same meaning as described for the CREATE TYPE command, and must be specified in the CREATE TYPE BODY command in the same manner as specified in the CREATE TYPE command.

Include the CONSTRUCTOR keyword and function definition to define a constructor function.

func_name is an identifier of a function specified in the `CREATE TYPE` command. The parameter declarations have the same meaning as described for the `CREATE TYPE` command, and must be specified in the `CREATE TYPE BODY` command in the same manner as specified in the `CREATE TYPE` command. *return_type* is the data type of the value the function returns and must match *return_type* given in the `CREATE TYPE` command.

variable_declaration is a declaration of a variable, which can be accessed only from within the subprogram. There can be none, one, or more variable declarations. *statement* is an SPL program statement.

8.3 *Creating Object Types*

You can use the CREATE TYPE command to create an object type specification, and the CREATE TYPE BODY command to create an object type body. This section provides some examples using the CREATE TYPE and CREATE TYPE BODY commands.

The first example creates the addr_object_type object type that contains only attributes and no methods:

```
CREATE OR REPLACE TYPE addr_object_type AS OBJECT
(
    street          VARCHAR2(30),
    city            VARCHAR2(20),
    state           CHAR(2),
    zip             NUMBER(5)
);
```

Since there are no methods in this object type, an object type body is not required. This example creates a composite type, that allows you to treat related objects as a single attribute.

8.3.1 Member Methods

A member method is a function or procedure that is defined within an object type and can only be invoked through an instance of that type. Member methods have access to, and can change the attributes of, the object instance on which they are operating.

The following object type specification creates the emp_obj_typ object type:

```
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)
);
```

Object type emp_obj_typ contains a member method named display_emp. display_emp uses a SELF parameter, that passes the object instance on which the method is invoked.

A SELF parameter is a parameter whose data type is that of the object type being defined. SELF always refers to the instance that is invoking the method. A SELF parameter is the first parameter in a member procedure or function *regardless* of whether it is explicitly declared in the parameter list.

The following code snippet defines an object type body for `emp_obj_typ`:

```
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;
```

You can also use the `SELF` parameter in an object type body. To illustrate how the `SELF` parameter would be used in the `CREATE TYPE BODY` command, the preceding object type body could be written as follows:

```
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    : ' || SELF.empno);
        DBMS_OUTPUT.PUT_LINE('Name           : ' || SELF.ename);
        DBMS_OUTPUT.PUT_LINE('Street         : ' || SELF.addr.street);
        DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || SELF.addr.city || ', ' ||
            SELF.addr.state || ' ' || LPAD(SELF.addr.zip,5,'0'));
    END;
END;
```

Both versions of the `emp_obj_typ` body are completely equivalent.

8.3.2 Static Methods

Like a member method, a static method belongs to a type. A static method, however, is invoked not by an *instance* of the type, but by using the *name* of the type. For example, to invoke a static function named `get_count`, defined within the `emp_obj_type` type, you would write:

```
emp_obj_type.get_count();
```

A static method does not have access to, and cannot change the attributes of an object instance, and does not typically work with an instance of the type.

The following object type specification includes a static function `get_dname` and a member procedure `display_dept`:

```
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
);
```

The object type body for dept_obj_typ defines a static function named get_dname and a member procedure named display_dept:

```
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 := 'ACCOUNTING';
            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;
```

Within the static function get_dname, there can be no references to SELF. Since a static function is invoked independently of any object instance, it has no implicit access to any object attribute.

Member procedure display_dept can access the deptno attribute of the object instance passed in the SELF parameter. It is not necessary to explicitly declare the SELF parameter in the display_dept parameter list.

The last DBMS_OUTPUT.PUT_LINE statement in the display_dept procedure includes a call to the static function get_dname (qualified by its object type name dept_obj_typ).

8.3.3 Constructor Methods

A constructor method is a function that creates an instance of an object type, typically by assigning values to the members of the object. An object type may define several constructors to accomplish different tasks. A constructor method is a member function (invoked with a SELF parameter) whose name matches the name of the type.

For example, if you define a type named address, each constructor is named address. You may overload a constructor by creating one or more different constructor functions with the same name, but with different argument types.

The SPL compiler will provide a default constructor for each object type. The default constructor is a member function whose name matches the name of the type and whose argument list matches the type members (in order). For example, given an object type such as:

```
CREATE TYPE address AS OBJECT
(
  street_address VARCHAR2(40),
  postal_code    VARCHAR2(10),
  city           VARCHAR2(40),
  state          VARCHAR2(2)
)
```

The SPL compiler will provide a default constructor with the following signature:

```
CONSTRUCTOR FUNCTION address
(
  street_address VARCHAR2(40),
  postal_code    VARCHAR2(10),
  city           VARCHAR2(40),
  state          VARCHAR2(2)
)
```

The body of the default constructor simply sets each member to `NULL`.

To create a custom constructor, declare the constructor function (using the keyword constructor) in the `CREATE TYPE` command and define the construction function in the `CREATE TYPE BODY` command. For example, you may wish to create a custom constructor for the `address` type which computes the city and state given a `street_address` and `postal_code`:

```
CREATE TYPE address AS OBJECT
(
  street_address VARCHAR2(40),
  postal_code    VARCHAR2(10),
  city           VARCHAR2(40),
  state          VARCHAR2(2),

  CONSTRUCTOR FUNCTION address
    (
      street_address VARCHAR2,
      postal_code VARCHAR2
    ) RETURN self AS RESULT
)
 CREATE TYPE BODY address AS
  CONSTRUCTOR FUNCTION address
    (
      street_address VARCHAR2,
      postal_code VARCHAR2
    ) RETURN self AS RESULT
  IS
    BEGIN
      self.street_address := street_address;
      self.postal_code := postal_code;
      self.city := postal_code_to_city(postal_code);
      self.state := postal_code_to_state(postal_code);
      RETURN;
    END;
END;
```

To create an instance of an object type, you invoke one of the constructor methods for that type. For example:

```
DECLARE
  cust_addr address := address('100 Main Street', 02203');
BEGIN
  DBMS_OUTPUT.PUT_LINE(cust_addr.city);  -- displays Boston
  DBMS_OUTPUT.PUT_LINE(cust_addr.state); -- displays MA
END;
```

Custom constructor functions are typically used to compute member values when given incomplete information. The preceding example computes the values for `city` and `state` when given a postal code.

Custom constructor functions are also used to enforce business rules that restrict the state of an object. For example, if you define an object type to represent a `payment`, you can use a custom constructor to ensure that no object of type `payment` can be created with an `amount` that is `NULL`, negative, or zero. The default constructor would set `payment.amount` to `NULL` so you must create a custom constructor (whose signature matches the default constructor) to prohibit `NULL` amounts.

8.4 Creating Object Instances

To create an instance of an object type, you must first declare a variable of the object type, and then initialize the declared object variable. The syntax for declaring an object variable is:

```
    object obj_type
```

object is an identifier assigned to the object variable.

obj_type is the identifier of a previously defined object type.

After declaring the object variable, you must invoke a *constructor method* to initialize the object with values. Use the following syntax to invoke the constructor method:

```
    [NEW] obj_type ({expr1 | NULL} [, {expr2 | NULL} ] [, ...])
```

obj_type is the identifier of the object type's constructor method; the constructor method has the same name as the previously declared object type.

expr1, *expr2*, … are expressions that are type-compatible with the first attribute of the object type, the second attribute of the object type, etc. If an attribute is of an object type, then the corresponding expression can be NULL, an object initialization expression, or any expression that returns that object type.

The following anonymous block declares and initializes a variable:

```
DECLARE
    v_emp           EMP_OBJ_TYP;
BEGIN
    v_emp := emp_obj_typ (9001,'JONES',
        addr_obj_typ('123 MAIN STREET','EDISON','NJ',08817));
END;
```

The variable (v_emp) is declared with a previously defined object type named EMP_OBJ_TYPE. The body of the block initializes the variable using the emp_obj_typ and addr_obj_type constructors.

You can include the NEW keyword when creating a new instance of an object in the body of a block. The NEW keyword invokes the object constructor whose signature matches the arguments provided.

The following example declares two variables, named mgr and emp. The variables are both of EMP_OBJ_TYPE. The mgr object is initialized in the declaration, while the emp object is initialized to NULL in the declaration, and assigned a value in the body.

```
DECLARE
    mgr   EMP_OBJ_TYPE := (9002,'SMITH');
    emp  EMP_OBJ_TYPE;
BEGIN
    emp := NEW EMP_OBJ_TYPE (9003,'RAY');
END;
```

Note: In Advanced Server, the following alternate syntax can be used in place of the constructor method.

```
[ ROW ] ({ expr1 | NULL } [, { expr2 | NULL } ] [, ...])
```

`ROW` is an optional keyword if two or more terms are specified within the parenthesis-enclosed, comma-delimited list. If only one term is specified, then specification of the `ROW` keyword is mandatory.

8.5 Referencing an Object

Once an object variable is created and initialized, individual attributes can be referenced using dot notation of the form:

```
object.attribute
```

`object` is the identifier assigned to the object variable. *`attribute`* is the identifier of an object type attribute.

If *`attribute`*, itself, is of an object type, then the reference must take the form:

```
object.attribute.attribute_inner
```

`attribute_inner` is an identifier belonging to the object type to which *`attribute`* references in its definition of *`object`*.

The following example expands upon the previous anonymous block to display the values assigned to the `emp_obj_typ` object.

```
DECLARE
    v_emp          EMP_OBJ_TYP;
BEGIN
    v_emp := emp_obj_typ(9001,'JONES',
        addr_obj_typ('123 MAIN STREET','EDISON','NJ',08817));
    DBMS_OUTPUT.PUT_LINE('Employee No    : ' || v_emp.empno);
    DBMS_OUTPUT.PUT_LINE('Name           : ' || v_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Street         : ' || v_emp.addr.street);
    DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || v_emp.addr.city || ', ' ||
        v_emp.addr.state || ' ' || LPAD(v_emp.addr.zip,5,'0'));
END;
```

The following is the output from this anonymous block.

```
Employee No   : 9001
Name          : JONES
Street        : 123 MAIN STREET
City/State/Zip: EDISON, NJ 08817
```

Methods are called in a similar manner as attributes.

Once an object variable is created and initialized, member procedures or functions are called using dot notation of the form:

```
object.prog_name
```

object is the identifier assigned to the object variable. *prog_name* is the identifier of the procedure or function.

Static procedures or functions are not called utilizing an object variable. Instead the procedure or function is called utilizing the object type name:

```
object_type.prog_name
```

object_type is the identifier assigned to the object type. *prog_name* is the identifier of the procedure or function.

The results of the previous anonymous block can be duplicated by calling the member procedure `display_emp`:

```
DECLARE
    v_emp          EMP_OBJ_TYP;
BEGIN
    v_emp := emp_obj_typ(9001,'JONES',
        addr_obj_typ('123 MAIN STREET','EDISON','NJ',08817));
    v_emp.display_emp;
END;
```

The following is the output from this anonymous block.

```
Employee No   : 9001
Name          : JONES
Street        : 123 MAIN STREET
City/State/Zip: EDISON, NJ 08817
```

The following anonymous block creates an instance of `dept_obj_typ` and calls the member procedure `display_dept`:

```
DECLARE
    v_dept          DEPT_OBJ_TYP := dept_obj_typ (20);
BEGIN
    v_dept.display_dept;
END;
```

The following is the output from this anonymous block.

```
Dept No     : 20
Dept Name   : RESEARCH
```

The static function defined in `dept_obj_typ` can be called directly by qualifying it by the object type name as follows:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(dept_obj_typ.get_dname(20));
END;

RESEARCH
```

8.6 Dropping an Object Type

The syntax for deleting an object type is as follows.

```
DROP TYPE objtype;
```

`objtype` is the identifier of the object type to be dropped. If the definition of *`objtype`* contains attributes that are themselves object types or collection types, these nested object types or collection types must be dropped last.

If an object type body is defined for the object type, the `DROP TYPE` command deletes the object type body as well as the object type specification. In order to recreate the complete object type, both the `CREATE TYPE` and `CREATE TYPE BODY` commands must be reissued.

The following example drops the `emp_obj_typ` and the `addr_obj_typ` object types created earlier in this chapter. `emp_obj_typ` must be dropped first since it contains `addr_obj_typ` within its definition as an attribute.

```
DROP TYPE emp_obj_typ;
DROP TYPE addr_obj_typ;
```

The syntax for deleting an object type body, but not the object type specification is as follows.

```
DROP TYPE BODY objtype;
```

The object type body can be recreated by issuing the `CREATE TYPE BODY` command.

The following example drops only the object type body of the `dept_obj_typ`.

```
DROP TYPE BODY dept_obj_typ;
```

9 Open Client Library

The Open Client Library provides application interoperability with the Oracle Call Interface – an application that was formerly "locked in" can now work with either an EDB Postgres Advanced Server or an Oracle database with minimal to no changes to the application code. The EnterpriseDB implementation of the Open Client Library is written in C.

Please note: EnterpriseDB does not support use of the Open Client Library with Oracle Real Application Clusters (RAC) and Oracle Exadata; the aforementioned Oracle products have not been evaluated nor certified with this EnterpriseDB product.

9.1 Comparison with Oracle Call Interface

The following diagram compares the Open Client Library and Oracle Call Interface application stacks.

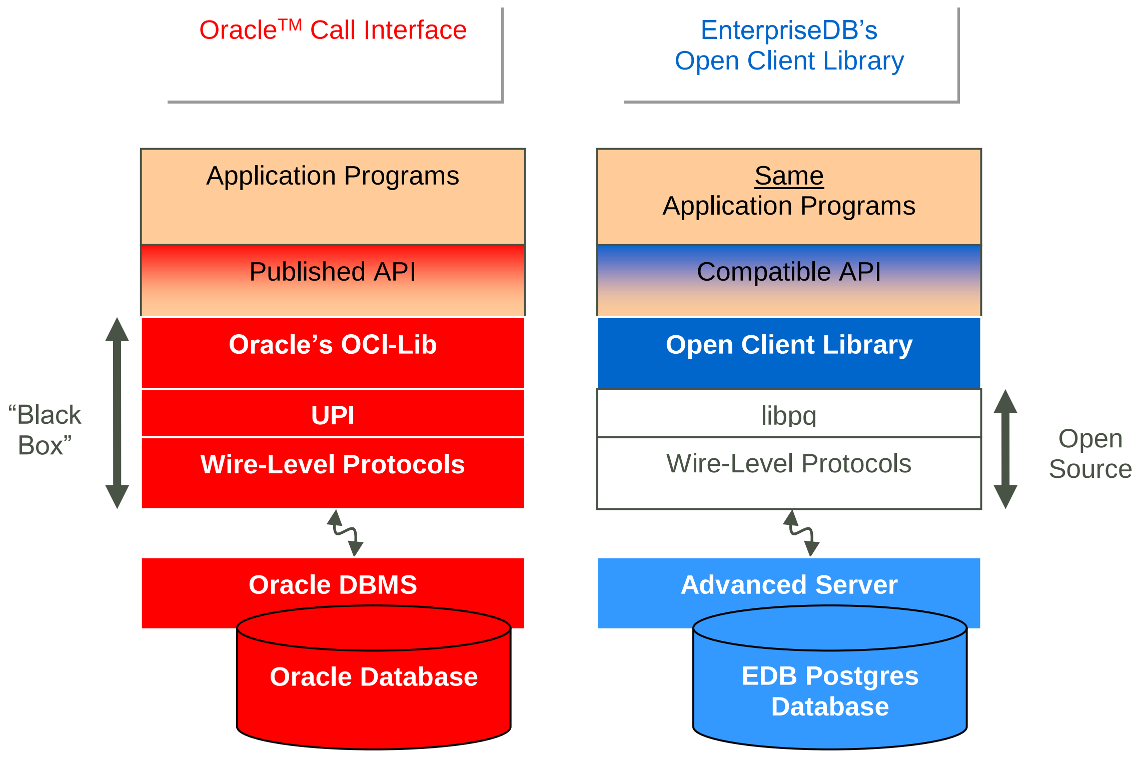


Figure 6 Open Client Library

9.2 Compiling and Linking a Program

The EnterpriseDB Open Client Library allows applications written using the Oracle Call Interface API to connect to and access an EnterpriseDB database with minimal changes to the C source code. The EnterpriseDB Open Client Library files are named:

On Linux:

```
libedboci.so
```

On Windows:

```
edboci.dll
```

The files are installed in the `connectors/edb-oci/lib` subdirectory.

Compiling and Linking a Sample Program

The following example compiles and links the sample program `edb_demo.c` in a Linux environment. The `edb_demo.c` is located in the `connectors/edb-oci/samples` subdirectory.

1. Set the `ORACLE_HOME` and `EDB_HOME` environment variables.

   Set `ORACLE_HOME` to the complete pathname of the Oracle home directory.

   For example:

   ```
   export
   ORACLE_HOME=/usr/lib/oracle/xe/app/oracle/product/10.2.0/server
   ```

   Set `EDB_HOME` to the complete pathname of the home directory.

   For example:

   ```
   export EDB_HOME=/opt/PostgresPlus
   ```

2. Set `LD_LIBRARY_PATH` to the complete path of `libpthread.so`. By default, `libpthread.so` is located in `/usr/lib`.

   ```
   export LD_LIBRARY_PATH=/usr/lib:$LD_LIBRARY_PATH
   ```

3. Set `LD_LIBRARY_PATH` to include the Advanced Server Open Client library. By default, `libiconv.so.2` is located in `$EDB_HOME/connectors/edb-oci/lib`.

```
export
LD_LIBRARY_PATH=$EDB_HOME/connectors/edb-oci:$EDB_HOME/
connectors/edb-oci/lib:$LD_LIBRARY_PATH
```

4. Then, compile and link the OCI API program.

```
cd $EDB_HOME/connectors/edb-oci/samples

make
```

9.3 Ref Cursor Support

The Advanced Server Open Client Library supports the use of `REF CURSOR`'s as `OUT` parameters in PL/SQL procedures that are compatible with Oracle. Support is provided through the following API's:

- `OCIBindByName`
- `OCIBindByPos`
- `OCIBindDynamic`
- `OCIStmtPrepare`
- `OCIStmtExecute`
- `OCIStmtFetch`
- `OCIAttrGet`

OCL also supports the `SQLT_RSET` data type.

The following example demonstrates how to invoke a stored procedure that opens a cursor and returns a `REF CURSOR` as an output parameter. The code sample assumes that a PL/SQL procedure named `openCursor` (with an `OUT` parameter of type `REF CURSOR`) has been created on the database server, and that the required handles have been allocated:

```
char * openCursor =
  "begin \
     openCursor(:cmdRefCursor); \
   end;";
OCIStmt *stmtOpenRefCursor;
OCIStmt *stmtUseRefCursor;
```

Allocate handles for executing a stored procedure to open and use the `REF CURSOR`:

```
  /* Handle for the stored procedure to open the ref cursor */
  OCIHandleAlloc((dvoid *) envhp,
                 (dvoid **) &stmtOpenRefCursor,
                 OCI_HTYPE_STMT,
                 0,
                 (dvoid **) NULL));
```

```
  /* Handle for using the Ref Cursor */
  OCIHandleAlloc((dvoid *) envhp,
                  (dvoid **) &stmtUseRefCursor,
                  OCI_HTYPE_STMT,
                  0,
                  (dvoid **) NULL));
```

Then, prepare the PL/SQL block that is used to open the `REF CURSOR`:

```
  OCIStmtPrepare(stmtOpenRefCursor,
                  errhp,
                  (text *) openCursor,
                  (ub4) strlen(openCursor),
                  OCI_NTV_SYNTAX,
                  OCI_DEFAULT));
```

Bind the PL/SQL `openCursor` `OUT` parameter:

```
  OCIBindByPos(stmtOpenRefCursor,
                  &bndplrc1,
                  errhp,
                  1,
                  (dvoid*) &stmtUseRefCursor,
                          /* the returned ref cursor */
                  0,
                  SQLT_RSET,
                      /* SQLT_RSET type representing cursor
*/
                  (dvoid *) 0,
                  (ub2 *) 0,
                  (ub2) 0,
                  (ub4) 0,
                  (ub4 *) 0,
                  OCI_DEFAULT));
```

Use the `stmtOpenRefCursor` statement handle to call the `openCursor` procedure:

```
  OCIStmtExecute(svchp,
                  stmtOpenRefCursor,
                  errhp,
                  1,
                  0,
                  0,
                  0,
                  OCI_DEFAULT);
```

At this point, the `stmtUseRefCursor` statement handle contains the reference to the cursor. To obtain the information, define output variables for the ref cursor:

```
/* Define the output variables for the ref cursor */
  OCIDefineByPos(stmtUseRefCursor,
                  &defnEmpNo,
                  errhp,
                 (ub4) 1,
                 (dvoid *) &empNo,
                 (sb4) sizeof(empNo),
                 SQLT_INT,
                 (dvoid *) 0,
                 (ub2 *)0,
                 (ub2 *)0,
                 (ub4) OCI_DEFAULT));
```

Then, fetch the first row of the result set into the target variables:

```
/* Fetch the cursor data */
  OCIStmtFetch(stmtUseRefCursor,
                 errhp,
                 (ub4) 1,
                 (ub4) OCI_FETCH_NEXT,
                 (ub4) OCI_DEFAULT))
```

9.4 OCL Function Reference

The following tables list the functions supported in the Open Client Library. Note that any and all header files must be supplied by the user. Advanced Server does not supply any such files.

9.4.1 Connect, Authorize and Initialize Functions

Table 9-9-1 Connect, Authorize, Terminate and Initialize Functions

Function	Description
`OCIBreak`	Aborts the specified OCI function.
`OCIEnvCreate`	Create an OCI environment.
`OCIEnvInit`	Initialize an OCI environment handle.
`OCIInitialize`	Initialize the OCI environment.
`OCILogoff`	Release a session.
`OCILogon`	Create a logon connection.
`OCILogon2`	Create a logon session in various modes.
`OCIReset`	Resets the current operation/protocol.
`OCIServerAttach`	Establish an access path to a data source. For information about using the `tnsnames.ora` file, see Section 9.6.
`OCIServerDetach`	Remove access to a data source.
`OCISessionBegin`	Create a user session.
`OCISessionEnd`	End a user session.
`OCISessionGet`	Get session from session pool.
`OCISessionRelease`	Release a session.
`OCITerminate`	Detach from shared memory subsystem.

9.4.1.1 Using the tnsnames.ora File

The `OCIServerAttach` method uses a connection descriptor specified in the `dblink` parameter of the `tnsnames.ora` file. Use the `tnsnames.ora` file, compatible with Oracle databases, to specify database connection addresses. Advanced Server searches the user's home directory for a file named `tnsnames.ora`. If Advanced Server doesn't find the `tnsnames.ora` file in the user's home directory, it searches the path specified by `TNS_ADMIN`.

The sample `tnsnames.ora` file contains:

```
 EDBX =
(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 5444))
  (CONNECT_DATA = (SERVER = DEDICATED)(SID = edb))
)
```

Any parameters not included in the sample, are ignored by the Open Client Library. In the sample, SID refers to the database named edb, in the cluster running on server 'localhost' at port 5444.

A C program call to OCIServerAttach that uses the tnsnames.ora file will look like:

```
static text *username = (text *) "enterprisedb";
static text *password = (text *) "edb";
static text *attach_str = "EDBX";
OCIServerAttach( srvhp, errhp, attach_str, strlen(attach_str),
0);
```

If you don't have a tnsnames.ora file, supply the connection string parameter in the form //localhost:5444/edbx.

9.4.2 Handle and Descriptor Functions

Table 9-1 Handle and Descriptor Functions

Function	Description
OCIAttrGet	Get handle attributes. Advanced server supports the following handle attributes: OCI_ATTR_USERNAME, OCI_ATTR_PASSWORD, OCI_ATTR_SERVER, OCI_ATTR_ENV, OCI_ATTR_SESSION, OCI_ATTR_ROW_COUNT, OCI_ATTR_CHARSET_FORM, OCI_ATTR_CHARSET_ID, EDB_ATTR_STMT_LEVEL_TX, OCI_ATTR_MODULE
OCIAttrSet	Set handle attributes. Advanced server supports the following handle attributes: OCI_ATTR_USERNAME, OCI_ATTR_PASSWORD, OCI_ATTR_SERVER, OCI_ATTR_ENV, OCI_ATTR_SESSION, OCI_ATTR_ROW_COUNT, OCI_ATTR_CHARSET_FORM, OCI_ATTR_CHARSET_ID, EDB_ATTR_STMT_LEVEL_TX, OCI_ATTR_MODULE
OCIDescriptorAlloc	Allocate and initialize a descriptor.
OCIDescriptorFree	Free an allocated descriptor.
OCIHandleAlloc	Allocate and initialize a handle.
OCIHandleFree	Free an allocated handle.
OCIParamGet	Get a parameter descriptor.
OCIParamSet	Set a parameter descriptor.

9.4.2.1 EDB_ATTR_EMPTY_STRINGS

By default, Advanced Server will treat an empty string as a NULL value. You can use the EDB_ATTR_EMPTY_STRINGS environment attribute to control the behavior of the OCL when mapping empty strings. To modify the mapping behavior, use the OCIAttrSet() function to set EDB_ATTR_EMPTY_STRINGS to one of the following:

Value	Description
OCI_DEFAULT	Treat an empty string as a NULL value.
EDB_EMPTY_STRINGS_NULL	Treat an empty string as a NULL value.
EDB_EMPTY_STRINGS_EMPTY	Treat an empty string as a string of zero length.

To find the value of EDB_ATTR_EMPTY_STRINGS, query OCIAttrGet().

9.4.2.2 EDB_ATTR_HOLDABLE

Advanced Server supports statements that execute as WITH HOLD cursors. The EDB_ATTR_HOLDABLE attribute specifies which statements execute as WITH HOLD cursors. The EDB_ATTR_HOLDABLE attribute can be set to any of the following three values:

- EDB_WITH_HOLD - execute as a WITH HOLD cursor
- EDB_WITHOUT_HOLD - execute using a protocol-level prepared statement
- OCI_DEFAULT - see the definition that follows

You can set the attribute in an OCIStmt handle or an OCIServer handle. When you create an OCIServer handle or an OCIStmt handle, the EDB_ATTR_HOLDABLE attribute for that handle is set to OCI_DEFAULT.

You can change the EDB_ATTR_HOLDABLE attribute for a handle by calling OCIAttrSet() and retrieve the attribute by calling OCIAttrGet().

When Advanced Server executes a SELECT statement, it examines the EDB_ATTR_HOLDABLE attribute in the OCIServer handle. If that attribute is set to EDB_WITH_HOLD, the query is executed as a WITH HOLD cursor.

If the EDB_ATTR_HOLDABLE attribute in the OCIServer handle is set to EDB_WITHOUT_HOLD, the query is executed as a normal prepared statement.

If the EDB_ATTR_HOLDABLE attribute in the OCIServer handle is set to OCI_DEFAULT, Advanced Server uses the value of the EDB_ATTR_HOLDABLE attribute in the OCIServer handle (if the EDB_ATTR_HOLDABLE attribute in the OCIServer is set to EDB_WITH_HOLD, the query executes as a WITH HOLD cursor, otherwise, the query executes as a protocol-prepared statement).

9.4.2.3 EDB_ATTR_STMT_LVL_TX

Unless otherwise instructed, the OCL library will ROLLBACK the current transaction whenever the server reports an error. If you choose, you can override the automatic ROLLBACK with the edb_stmt_level_tx parameter, which preserves modifications within a transaction, even if one (or several) statements raise an error within the transaction. For more information about edb_stmt_level_tx, see Section 1.3.4.

You can use the OCIServer attribute with OCIAttrSet() and OCIAttrGet() to enable or disable EDB_ATTR_STMT_LEVEL_TX. By default, edb_stmt_level_tx is disabled. To enable edb_stmt_level_tx, the client application must call OCIAttrSet():

```
OCIServer *server  = myServer;
ub1        enabled = 1;

OCIAttrSet(server, OCI_HTYPE_SERVER, &enabled,
  sizeof(enabled), EDB_ATTR_STMT_LEVEL_TX, err);
```

To disable edb_stmt_level_tx:

```
OCIServer *server  = myServer;
ub1        enabled = 0;

OCIAttrSet(server, OCI_HTYPE_SERVER, &enabled,
  sizeof(enabled), EDB_ATTR_STMT_LEVEL_TX, err);
```

9.4.3 Bind, Define and Describe Functions

Table 9-2 Bind, Define, and Describe Functions

Function	Description
OCIBindByName	Bind by name.
OCIBindByPos	Bind by position.
OCIBindDynamic	Set additional attributes after bind.
OCIBindArrayOfStruct	Bind an array of structures for bulk operations.
OCIDefineArrayOfStruct	Specify the attributes of an array.
OCIDefineByPos	Define an output variable association.
OCIDefineDynamic	Set additional attributes for define.
OCIDescribeAny	Describe existing schema objects.
OCIStmtGetBindInfo	Get bind and indicator variable names and handle.
OCIUserCallbackRegister	Define a user-defined callback.

9.4.4 Statement Functions

Table 9-3 Statement Functions

Function	Description
`OCIStmtExecute`	Execute a prepared SQL statement.
`OCIStmtFetch`	Fetch rows of data (deprecated).
`OCIStmtFetch2`	Fetch rows of data.
`OCIStmtPrepare`	Prepare a SQL statement.
`OCIStmtPrepare2`	Prepare a SQL statement.
`OCIStmtRelease`	Release a statement handle.

9.4.5 Transaction Functions

Table 9-4 Transaction Functions

Function	Description
`OCITransCommit`	Commit a transaction.
`OCITransRollback`	Roll back a transaction.

9.4.6 XA Functions

Table 9-5 XA Functions

Function	Description
`xaoEnv`	Returns OCL environment handle.
`xaoSvcCtx`	Returns OCL service context.

9.4.6.1 xaoSvcCtx

In order to use the `xaoSvcCtx` function, extensions in the `xaoSvcCtx` or `xa_open` connection string format must be provided as follows:

```
    Oracle_XA{+required_fields ...}
```

Where *`required_fields`* are the following:

`HostName=`*`host_ip_address`* specifies the IP address of the Advanced Server database.

`PortNumber=`*`host_port_number`* specifies the port number on which Advanced Server is running.

`SqlNet=`*`dbname`* specifies the database name.

`Acc=P/`*`username`*`/`*`password`* specifies the database username and password. *`password`* may be omitted in which case the field is specified as `Acc=P/username/`.

`AppName=`*`app_id`* specifies a number that identifies the application.

The following is an example of the connection string:

```
Oracle_XA+HostName=192.168.1.1+PortNumber=1533+SqlNet=XE+Acc=P/
user/password+AppName=1234
```

9.4.7 Date and Datetime Functions

Table 9-6 Date and Datetime Functions

Function	Description
`OCIDateAddDays`	Add or subtract a number of days.
`OCIDateAddMonths`	Add or subtract a number of months.
`OCIDateAssign`	Assign a date.
`OCIDateCheck`	Check if the given date is valid.
`OCIDateCompare`	Compare two dates.
`OCIDateDaysBetween`	Find the number of days between two dates.
`OCIDateFromText`	Convert a string to a date.
`OCIDateGetDate`	Get the date portion of a date.
`OCIDateGetTime`	Get the time portion of a date.
`OCIDateLastDay`	Get the date of the last day of the month.
`OCIDateNextDay`	Get the date of the next day.
`OCIDateSetDate`	Set the date portion of a date.
`OCIDateSetTime`	Set the time portion of a date.
`OCIDateSysDate`	Get the current system date and time.
`OCIDateToText`	Convert a date to a string.
`OCIDateTimeAssign`	Perform datetime assignment.
`OCIDateTimeCheck`	Check if the date is valid.
`OCIDateTimeCompare`	Compare two datetime values.
`OCIDateTimeConstruct`	Construct a datetime descriptor.
`OCIDateTimeConvert`	Convert one datetime type to another.
`OCIDateTimeFromArray`	Convert an array of size `OCI_DT_ARRAYLEN` to an `OCIDateTime` descriptor.
`OCIDateTimeFromText`	Convert the given string to Oracle datetime type in the `OCIDateTime` descriptor according to the specified format.
`OCIDateTimeGetDate`	Get the date portion of a datetime value.
`OCIDateTimeGetTime`	Get the time portion of a datetime value.
`OCIDateTimeGetTimeZoneName`	Get the time zone name portion of a datetime value.
`OCIDateTimeGetTimeZoneOffset`	Get the time zone (hour, minute) portion of a datetime value.
`OCIDateTimeSubtract`	Take two datetime values as input and return their difference as an interval.
`OCIDateTimeSysTimeStamp`	Get the system current date and time as a timestamp with time zone.

Function	Description
`OCIDateTimeToArray`	Convert an `OCIDateTime` descriptor to an array.
`OCIDateTimeToText`	Convert the given date to a string according to the specified format.

9.4.8 Interval Functions

Table 9-7 Interval Functions

Function	Description
`OCIIntervalAdd`	Adds two interval values.
`OCIIntervalAssign`	Copies one interval value into another interval value.
`OCIIntervalCompare`	Compares two interval values.
`OCIIntervalGetDaySecond`	Extracts days, hours, minutes, seconds and fractional seconds from an interval.
`OCIIntervalSetDaySecond`	Modifies days, hours, minutes, seconds and fractional seconds in an interval.
`OCIIntervalGetYearMonth`	Extracts year and month values from an interval.
`OCIIntervalSetYearMonth`	Modifies year and month values in an interval.
`OCIIntervalDivide`	Implements division of `OCIInterval` values by `OCINumber` values.
`OCIIntervalMultiply`	Implements multiplication of `OCIInterval` values by `OCINumber` values.
`OCIIntervalSubtract`	Subtracts one interval value from another interval value.
`OCIIntervalToText`	Extrapolates a character string from an interval.
`OCIIntervalCheck`	Verifies the validity of an interval value.
`OCIIntervalToNumber`	Converts an `OCIInterval` value into a `OCINumber` value.
`OCIIntervalFromNumber`	Converts a `OCINumber` value into an `OCIInterval` value.
`OCIDateTimeIntervalAdd`	Adds an `OCIInterval` value to an `OCIDatetime` value, resulting in an `OCIDatetime` value.
`OCIDateTimeIntervalSub`	Subtracts an `OCIInterval` value from an `OCIDatetime` value, resulting in an `OCIDatetime` value.
`OCIIntervalFromText`	Converts a text string into an interval.
`OCIIntervalFromTZ`	Converts a time zone specification into an interval value.

9.4.9 Number Functions

Table 9-8 Number Functions

Function	Description
`OCINumberAbs`	Compute the absolute value.
`OCINumberAdd`	Adds `NUMBER`s.
`OCINumberArcCos`	Compute the arc cosine.
`OCINumberArcSin`	Compute the arc sine.
`OCINumberArcTan`	Compute the arc tangent.
`OCINumberArcTan2`	Compute the arc tangent of two `NUMBER`s.

Function	Description
`OCINumberAssign`	Assign one `NUMBER` to another.
`OCINumberCeil`	Compute the ceiling of `NUMBER`.
`OCINumberCmp`	Compare `NUMBER`s.
`OCINumberCos`	Compute the cosine.
`OCINumberDec`	Decrement a `NUMBER`.
`OCINumberDiv`	Divide two `NUMBER`s.
`OCINumberExp`	Raise e to the specified `NUMBER` power.
`OCINumberFloor`	Compute the floor of a `NUMBER`.
`OCINumberFromInt`	Convert an integer to an Oracle `NUMBER`.
`OCINumberFromReal`	Convert a real to an Oracle `NUMBER`.
`OCINumberFromText`	Convert a string to an Oracle `NUMBER`.
`OCINumberHypCos`	Compute the hyperbolic cosine.
`OCINumberHypSin`	Compute the hyperbolic sine.
`OCINumberHypTan`	Compute the hyperbolic tangent.
`OCINumberInc`	Increments a `NUMBER`.
`OCINumberIntPower`	Raise a given base to an integer power.
`OCINumberIsInt`	Test if a `NUMBER` is an integer.
`OCINumberIsZero`	Test if a `NUMBER` is zero.
`OCINumberLn`	Compute the natural logarithm.
`OCINumberLog`	Compute the logarithm to an arbitrary base.
`OCINumberMod`	Modulo division.
`OCINumberMul`	Multiply `NUMBER`s.
`OCINumberNeg`	Negate a `NUMBER`.
`OCINumberPower`	Exponentiation to base e.
`OCINumberPrec`	Round a `NUMBER` to a specified number of decimal places.
`OCINumberRound`	Round a `NUMBER` to a specified decimal place.
`OCINumberSetPi`	Initialize a `NUMBER` to Pi.
`OCINumberSetZero`	Initialize a `NUMBER` to zero.
`OCINumberShift`	Multiply by 10, shifting specified number of decimal places.
`OCINumberSign`	Obtain the sign of a `NUMBER`.
`OCINumberSin`	Compute the sine.
`OCINumberSqrt`	Compute the square root of a `NUMBER`.
`OCINumberSub`	Subtract `NUMBER`s.
`OCINumberTan`	Compute the tangent.
`OCINumberToInt`	Convert a `NUMBER` to an integer.
`OCINumberToReal`	Convert a `NUMBER` to a real.
`OCINumberToRealArray`	Convert an array of `NUMBER` to a real array.
`OCINumberToText`	Converts a `NUMBER` to a string.
`OCINumberTrunc`	Truncate a `NUMBER` at a specified decimal place.

9.4.10 String Functions

Table 9-9 String Functions

Function	Description
`OCIStringAllocSize`	Get allocated size of string memory in bytes.
`OCIStringAssign`	Assign string to a string.
`OCIStringAssignText`	Assign text string to a string.
`OCIStringPtr`	Get string pointer.
`OCIStringResize`	Resize string memory.
`OCIStringSize`	Get string size.

9.4.11 Cartridge Services and File I/O Interface Functions

Table 9-10 Cartridge Services and File I/O Interface Functions

Function	Description
`OCIFileClose`	Close an open file.
`OCIFileExists`	Test to see if the file exists.
`OCIFileFlush`	Write buffered data to a file.
`OCIFileGetLength`	Get the length of a file.
`OCIFileInit`	Initialize the `OCIFile` package.
`OCIFileOpen`	Open a file.
`OCIFileRead`	Read from a file into a buffer.
`OCIFileSeek`	Change the current position in a file.
`OCIFileTerm`	Terminate the `OCIFile` package.
`OCIFileWrite`	Write `buflen` bytes into the file.

9.4.12 LOB Functions

Table 9-11 LOB Functions

Function	Description
`OCILobRead`	Returns a LOB value (or a portion of a LOB value).
`OCILOBWriteAppend`	Adds data to a LOB value.
`OCILobGetLength`	Returns the length of a LOB value.
`OCILobTrim`	Trims data from the end of a LOB value.
`OCILobOpen`	Opens a LOB value for use by other LOB functions.
`OCILobClose`	Closes a LOB value.

9.4.13 Miscellaneous Functions

Table 9-12 Miscellaneous Functions

Function	Description
`OCIClientVersion`	Return client library version.
`OCIErrorGet`	Return error message.

Function	Description
`OCIPGErrorGet`	Return native error messages reported by libpq or the server. The signature is: `sword OCIPGErrorGet(dvoid *hndlp, ub4 recordno, OraText *errcodep,ub4 errbufsiz, OraText *bufp, ub4 bufsiz, ub4 type)`
`OCIPasswordChange`	Change password.
`OCIPing`	Confirm that the connection and server are active.
`OCIServerVersion`	Get the Oracle version string.

9.4.14 Supported Data Types

Table 9-13 Supported Data Types

Function	Description
`ANSI_DATE`	ANSI date
`SQLT_AFC`	ANSI fixed character
`SQLT_AVC`	ANSI variable character
`SQLT_BDOUBLE`	Binary double
`SQLT_BIN`	Binary data
`SQLT_BFLOAT`	Binary float
`SQLT_CHR`	Character string
`SQLT_DAT`	Oracle date
`SQLT_DATE`	ANSI date
`SQLT_FLT`	Float
`SQLT_INT`	Integer
`SQLT_LBI`	Long binary
`SQLT_LNG`	Long
`SQLT_LVB`	Longer long binary
`SQLT_LVC`	Longer longs (character)
`SQLT_NUM`	Oracle numeric
`SQLT_ODT`	OCI date type
`SQLT_STR`	Zero-terminated string
`SQLT_TIMESTAMP`	Timestamp
`SQLT_TIMESTAMP_TZ`	Timestamp with time zone
`SQLT_TIMESTAMP_LTZ`	Timestamp with local time zone
`SQLT_UIN`	Unsigned integer
`SQLT_VBI`	VCS format binary
`SQLT_VCS`	Variable character
`SQLT_VNU`	Number with preceding length byte
`SQLT_VST`	OCI string type

10 Oracle Catalog Views

The Oracle Catalog Views provide information about database objects in a manner compatible with the Oracle data dictionary views.

10.1 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.

10.2 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.

10.3 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.

10.4 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.

10.5 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.

10.6 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`.
`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`.

10.7 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.

10.8 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.

10.9 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.

10.10 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

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

10.11 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.

10.12 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.

10.13 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.

10.14 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.

10.15 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.

10.16 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

10.17 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.

10.18 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.

10.19 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.

10.20 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.

10.21 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.

10.22 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`.

10.23 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.

10.24 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.

10.25 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.

10.26 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.

10.27 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.

10.28 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.

10.29 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.

10.30 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.
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.

10.31 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.

10.32 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.

10.33 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.

10.34 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`

10.35 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`.

10.36 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.

10.37 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.

10.38 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`.

10.39 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.

10.40 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.

10.41 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.

10.42 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.

10.43 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.

10.44 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

10.45 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.

10.46 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.

10.47 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.

10.48 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.

10.49 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`.

10.50 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.

10.51 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.

10.52 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.

10.53 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.

10.54 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.

10.55 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.

10.56 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.

10.57 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`.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

10.58 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.

10.59 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.

10.60 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

10.61 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`

10.62 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`.

10.63 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.

10.64 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.

10.65 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.

10.66 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.

10.67 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

10.68 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.

10.69 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.

10.70 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

10.71 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.

10.72 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.

10.73 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

10.74 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`.

10.75 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.

10.76 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.

10.77 V$VERSION

The V$VERSION view provides information about product compatibility.

Name	Type	Description
banner	TEXT	Displays product compatibility information.

10.78 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`.

11 Utilities

The sections in this chapter describe various utility programs. These include:

- EDB*Plus
- EDB*Loader
- EDB*Wrap
- Dynamic Runtime Instrumentation

*11.1 EDB*Plus*

EDB*Plus is a utility program that provides a command line user interface to the EDB Postgres Advanced Server. EDB*Plus accepts SQL commands, SPL anonymous blocks, and EDB*Plus commands. EDB*Plus commands are compatible with Oracle SQL*Plus commands and provide various capabilities including:

- Querying certain database objects
- Executing stored procedures
- Formatting output from SQL commands
- Executing batch scripts
- Executing OS commands
- Recording output

The following section describes how to connect to an Advanced Server database using EDB*Plus. The final section provides a summary of the EDB*Plus commands.

11.1.1 Starting EDB*Plus

To open an EDB*Plus command line, navigate through the `Applications` (or `Start`) menu to the `Advanced Server` menu, to the `Run SQL Command Line` menu, and select the `EDB*Plus` option. You can also invoke EDB*Plus from the operating system command line with the following command:

```
edbplus [ -S[ILENT ] ] [ login | /NOLOG ] [ @scriptfile[.ext ] ]
```

`-SILENT`

If specified, the EDB*Plus sign-on banner is suppressed along with all prompts.

`login`

Login information for connecting to the database server and database. *`login`* takes the following format. (There must be no white space within the login information.)

```
username[/password][@{connectstring | variable } ]
```

Where:

`username` is a database username with which to connect to the database.

`password` is the password associated with the specified *`username`*. If a *`password`* is not provided, but a password is required for authentication, EDB*Plus will prompt for the password.

`connectstring` is the database connection string.

`variable` is a variable defined in the `login.sql` file that contains a database connection string. The `login.sql` file can be found in the `edbplus` subdirectory of the Advanced Server home directory.

```
host[:port][/dbname ] ]
```

`host` is the hostname on which the database server resides. If neither `@`*`connectstring`* nor `@`*`variable`* nor `/NOLOG` is specified, the default host is assumed to be the localhost. *`port`* is the port number receiving connections on the database server. If not specified, the default is 5444. *`dbname`* is the name of the database to connect to. If not specified the default is `edb`.

/NOLOG

Specify /NOLOG to start EDB*Plus without establishing a database connection. SQL commands and EDB*Plus commands that require a database connection cannot be used in this mode. The CONNECT command can be subsequently given to connect to a database after starting EDB*Plus with the /NOLOG option.

scriptfile[.*ext*]

scriptfile is the name of a file residing in the current working directory, containing SQL and/or EDB*Plus commands that will be automatically executed after startup of EDB*Plus. *ext* is the filename extension. If the filename extension is sql, then the .sql extension may be omitted when specifying *scriptfile*. When creating a script file, always name the file with an extension, otherwise it will not be accessible by EDB*Plus. (EDB*Plus will always assume a .sql extension on filenames that are specified with no extension.)

The following example shows user enterprisedb with password, password, connecting to database edb running on a database server on the localhost at port 5444.

```
C:\Program Files (x86)\PostgresPlus\9.5AS\edbplus>edbplus
enterprisedb/password
Connected to EnterpriseDB 9.5.0.0 (localhost:5444/edb) AS enterprisedb

EDB*Plus: Release 9.5
Copyright (c) 2008-2015, EnterpriseDB Corporation.  All rights reserved.

SQL>
```

The following example shows user enterprisedb with password, password, connecting to database edb running on a database server on the localhost at port 5445.

```
C:\Program Files (x86)\PostgresPlus\9.5AS\edbplus>edbplus
enterprisedb/password@localhost:5445/edb
Connected to EnterpriseDB 9.5.0.0 (localhost:5445/edb) AS enterprisedb

EDB*Plus: Release 9.5
Copyright (c) 2008-2015, EnterpriseDB Corporation.  All rights reserved.

SQL>
```

Using variable hr_5445 in the login.sql file, the following illustrates how it is used to connect to database hr on localhost at port 5445.

```
C:\Program Files (x86)\PostgresPlus\9.5AS\edbplus>edbplus
enterprisedb/password@hr_5445
Connected to EnterpriseDB 9.5.0.0 (localhost:5445/hr) AS enterprisedb

EDB*Plus: Release 9.5 (Build 28)
Copyright (c) 2008-2015, EnterpriseDB Corporation.  All rights reserved.

SQL>
```

The following is the content of the `login.sql` file used in the previous example.

```
define edb="localhost:5445/edb"
define hr_5445="localhost:5445/hr"
```

The following example executes a script file, `dept_query.sql` after connecting to database `edb` on server localhost at port 5444.

```
C:\Program Files (x86)\PostgresPlus\9.5AS\edbplus>edbplus
enterprisedb/password @dept_query
Connected to EnterpriseDB 9.5.0.0 (localhost:5444/edb) AS enterprisedb

SQL> SELECT * FROM dept;

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

SQL> EXIT
Disconnected from EnterpriseDB Database.
```

The following is the content of file `dept_query.sql` used in the previous example.

```
SET PAGESIZE 9999
SET ECHO ON
SELECT * FROM dept;
EXIT
```

11.1.2 Command Summary

This section contains a summary of EDB*Plus commands.

11.1.2.1 ACCEPT

The ACCEPT command displays a prompt and waits for the user's keyboard input. The value input by the user is placed in the specified variable.

```
ACC[EPT ] variable
```

The following example creates a new variable named my_name, accepts a value of John Smith, then displays the value using the DEFINE command.

```
SQL> ACCEPT my_name
Enter value for my_name: John Smith
SQL> DEFINE my_name
DEFINE MY_NAME = "John Smith"
```

11.1.2.2 APPEND

APPEND is a line editor command that appends the given text to the end of the current line in the SQL buffer.

```
A[PPEND ] text
```

In the following example, a SELECT command is built-in the SQL buffer using the APPEND command. Note that two spaces are placed between the APPEND command and the WHERE clause in order to separate dept and WHERE by one space in the SQL buffer.

```
SQL> APPEND SELECT * FROM dept
SQL> LIST
  1* SELECT * FROM dept
SQL> APPEND  WHERE deptno = 10
SQL> LIST
  1* SELECT * FROM dept WHERE deptno = 10
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

11.1.2.3 CHANGE

`CHANGE` is a line editor command performs a search-and-replace on the current line in the SQL buffer.

```
C[HANGE ] /from/[to/ ]
```

If *`to/`* is specified, the first occurrence of text *`from`* in the current line is changed to text *`to`*. If *`to/`* is omitted, the first occurrence of text *`from`* in the current line is deleted.

The following sequence of commands makes line 3 the current line, then changes the department number in the `WHERE` clause from 20 to 30.

```
SQL> LIST
  1  SELECT empno, ename, job, sal, comm
  2  FROM emp
  3  WHERE deptno = 20
  4* ORDER BY empno
SQL> 3
  3* WHERE deptno = 20
SQL> CHANGE /20/30/
  3* WHERE deptno = 30
SQL> LIST
  1  SELECT empno, ename, job, sal, comm
  2  FROM emp
  3  WHERE deptno = 30
  4* ORDER BY empno
```

11.1.2.4 CLEAR

The `CLEAR` command removes the contents of the SQL buffer, deletes all column definitions set with the `COLUMN` command, or clears the screen.

```
CL[EAR ] [ BUFF[ER ] | SQL | COL[UMNS ] | SCR[EEN ] ]
```

`BUFFER | SQL`

Clears the SQL buffer.

`COLUMNS`

Removes column definitions.

`SCREEN`

Clears the screen. This is the default if no options are specified.

11.1.2.5 COLUMN

The `COLUMN` command controls output formatting. The formatting attributes set by using the `COLUMN` command remain in effect only for the duration of the current session.

```
COL[UMN ]
  [ column
    { CLE[AR ] |
      { FOR[MAT ] spec |
        HEA[DING ] text |
        { OFF | ON }
      } [...]
    }
  ]
```

If the `COLUMN` command is specified with no subsequent options, formatting options for current columns in effect for the session are displayed.

If the `COLUMN` command is followed by a column name, then the column name may be followed by one of the following:

1. No other options
2. `CLEAR`
3. Any combination of `FORMAT`, `HEADING`, and one of `OFF` or `ON`

`column`

Name of a column in a table to which subsequent column formatting options are to apply. If no other options follow *`column`*, then the current column formatting options if any, of *`column`* are displayed.

`CLEAR`

The `CLEAR` option reverts all formatting options back to their defaults for *`column`*. If the `CLEAR` option is specified, it must be the only option specified.

`spec`

Format specification to be applied to *`column`*. For character columns, *`spec`* takes the following format:

`n`

`n` is a positive integer that specifies the column width in characters within which to display the data. Data in excess of *`n`* will wrap around with the specified column width.

For numeric columns, *spec* is comprised of the following elements.

Table 10-11-1 Numeric Column Format Elements

Element	Description
`$`	Display a leading dollar sign.
`,`	Display a comma in the indicated position.
`.`	Marks the location of the decimal point.
`0`	Display leading zeros.
`9`	Number of significant digits to display.

If loss of significant digits occurs due to overflow of the format, then all `#`'s are displayed.

text

Text to be used for the column heading of *column*.

`OFF | ON`

If `OFF` is specified, formatting options are reverted back to their defaults, but are still available within the session. If `ON` is specified, the formatting options specified by previous `COLUMN` commands for *column* within the session are re-activated.

The following example shows the effect of changing the display width of the `job` column.

```
SQL> SET PAGESIZE 9999
SQL> COLUMN job FORMAT A5
SQL> COLUMN job
COLUMN   JOB  ON
FORMAT   A5
wrapped
SQL> SELECT empno, ename, job FROM emp;

EMPNO ENAME      JOB
----- ---------- -----
 7369 SMITH      CLERK
 7499 ALLEN      SALES
                 MAN

 7521 WARD       SALES
                 MAN

 7566 JONES      MANAG
                 ER

 7654 MARTIN     SALES
                 MAN

 7698 BLAKE      MANAG
                 ER
 7782 CLARK      MANAG
                 ER

 7788 SCOTT      ANALY
                 ST

 7839 KING       PRESI
                 DENT

 7844 TURNER     SALES
                 MAN

 7876 ADAMS      CLERK
 7900 JAMES      CLERK
 7902 FORD       ANALY
                 ST

 7934 MILLER     CLERK

14 rows retrieved.
```

The following example applies a format to the `sal` column.

```
SQL> COLUMN sal FORMAT $99,999.00
SQL> COLUMN
COLUMN   JOB  ON
FORMAT   A5
wrapped

COLUMN   SAL  ON
FORMAT   $99,999.00
wrapped
SQL> SELECT empno, ename, job, sal FROM emp;

EMPNO ENAME      JOB           SAL
----- ---------- ----- -----------
 7369 SMITH      CLERK     $800.00
 7499 ALLEN      SALES   $1,600.00
                 MAN

 7521 WARD       SALES   $1,250.00
                 MAN

 7566 JONES      MANAG   $2,975.00
                 ER

 7654 MARTIN     SALES   $1,250.00
                 MAN

 7698 BLAKE      MANAG   $2,850.00
                 ER

 7782 CLARK      MANAG   $2,450.00
                 ER

 7788 SCOTT      ANALY   $3,000.00
                 ST

 7839 KING       PRESI   $5,000.00
                 DENT

 7844 TURNER     SALES   $1,500.00
```

```
                 MAN

 7876 ADAMS      CLERK   $1,100.00
 7900 JAMES      CLERK     $950.00
 7902 FORD       ANALY   $3,000.00
                 ST

 7934 MILLER     CLERK   $1,300.00

14 rows retrieved.
```

11.1.2.6 CONNECT

Change the database connection to a different user and/or connect to a different database. There must be no white space between any of the parameters following the `CONNECT` command.

```
CON[NECT] username[/password][@{connectstring | variable } ]
```

Where:

username is a database username with which to connect to the database.

password is the password associated with the specified *username*. If a *password* is not provided, but a password is required for authentication, EDB*Plus will prompt for the password.

connectstring is the database connection string.

variable is a variable defined in the `login.sql` file that contains a database connection string. The `login.sql` file can be found in the `edbplus` subdirectory of the Advanced Server home directory.

In the following example, the database connection is changed to database `edb` on the localhost at port `5445` with username, `smith`.

```
SQL> CONNECT smith/mypassword@localhost:5445/edb
Disconnected from EnterpriseDB Database.
Connected to EnterpriseDB 9.5.0.0 (localhost:5445/edb) AS smith
```

From within the session shown above, the connection is changed to username `enterprisedb`. Also note that the host defaults to the localhost, the port defaults to 5444 (which is not the same as the port previously used), and the database defaults to `edb`.

```
SQL> CONNECT enterprisedb/password
Disconnected from EnterpriseDB Database.
Connected to EnterpriseDB 9.5.0.0 (localhost:5444/edb) AS enterprisedb
```

11.1.2.7 DEFINE

The `DEFINE` command creates or replaces the value of a *user variable* (also called a *substitution variable*).

```
DEF[INE ] [ variable [ = text ] ]
```

If the `DEFINE` command is given without any parameters, all current variables and their values are displayed.

If `DEFINE variable` is given, only `variable` is displayed with its value.

`DEFINE variable = text` assigns `text` to `variable`. `text` may be optionally enclosed within single or double quotation marks. Quotation marks must be used if `text` contains space characters.

The following example defines two variables, `dept` and `name`.

```
SQL> DEFINE dept = 20
SQL> DEFINE name = 'John Smith'
SQL> DEFINE
DEFINE EDB = "localhost:5445/edb"
DEFINE DEPT = "20"
DEFINE NAME = "John Smith"
```

Note: The variable `EDB` is read from the `login.sql` file located in the `edbplus` subdirectory of the Advanced Server home directory.

11.1.2.8 DEL

`DEL` is a line editor command that deletes one or more lines from the SQL buffer.

```
DEL [ n | n m | n * | n L[AST ] | * | * n | * L[AST ] |
  L[AST ] ]
```

The parameters specify which lines are to be deleted from the SQL buffer. Two parameters specify the start and end of a range of lines to be deleted. If the `DEL` command is given with no parameters, the current line is deleted.

n

n is an integer representing the *n*th line *n m*

n and *m* are integers where *m* is greater than *n* representing the *n*th through the *m*th lines

*

Current line

LAST

Last line

In the following example, the fifth and sixth lines containing columns `sal` and `comm`, respectively, are deleted from the `SELECT` command in the SQL buffer.

```
SQL> LIST
  1  SELECT
  2    empno
  3   ,ename
  4   ,job
  5   ,sal
  6   ,comm
  7   ,deptno
  8* FROM emp
SQL> DEL 5 6
SQL> LIST
  1  SELECT
  2    empno
  3   ,ename
  4   ,job
  5   ,deptno
  6* FROM emp
```

11.1.2.9 DESCRIBE

The `DESCRIBE` command displays:

- A list of columns, column data types, and column lengths for a table or view
- A list of parameters for a procedure or function
- A list of procedures and functions and their respective parameters for a package.

The `DESCRIBE` command will also display the structure of the database object referred to by a synonym. The syntax is:

```
DESC[RIBE] [ schema.]object
```

schema

Name of the schema containing the object to be described.

object

Name of the table, view, procedure, function, or package to be displayed, or the synonym of an object.

11.1.2.10 DISCONNECT

The `DISCONNECT` command closes the current database connection, but does not terminate EDB*Plus.

```
DISC[ONNECT ]
```

11.1.2.11 EDIT

The `EDIT` command invokes an external editor to edit the contents of an operating system file or the SQL buffer.

```
ED[IT ] [ filename[.ext ] ]
```

filename[.*ext*]

filename is the name of the file to open with an external editor. *ext* is the filename extension. If the filename extension is `sql`, then the `.sql` extension may be omitted when specifying *filename*. `EDIT` always assumes a `.sql` extension on filenames that are specified with no extension. If the filename parameter is omitted from the `EDIT` command, the contents of the SQL buffer are brought into the editor.

11.1.2.12 EXECUTE

The `EXECUTE` command executes an SPL procedure from EDB*Plus.

```
EXEC[UTE ] spl_procedure [ ([ parameters ]) ]
```

spl_procedure

The name of the SPL procedure to be executed.

parameters

Comma-delimited list of parameters. If there are no parameters, then a pair of empty parentheses may optionally be specified.

11.1.2.13 EXIT

The `EXIT` command terminates the EDB*Plus session and returns control to the operating system. `QUIT` is a synonym for `EXIT`. Specifying no parameters is equivalent to `EXIT SUCCESS COMMIT`.

```
{ EXIT | QUIT }
[ SUCCESS | FAILURE | WARNING | value |variable ]
[ COMMIT | ROLLBACK ]SUCCESS | FAILURE |WARNING
```

Returns an operating system dependent return code indicating successful operation, failure, or warning for `SUCCESS`, `FAILURE`, and `WARNING`, respectively. The default is `SUCCESS`.

value

An integer value that is returned as the return code.

variable

A variable created with the `DEFINE` command whose value is returned as the return code.

`COMMIT | ROLLBACK`

If `COMMIT` is specified, uncommitted updates are committed upon exit. If `ROLLBACK` is specified, uncommitted updates are rolled back upon exit. The default is `COMMIT`.

11.1.2.14 GET

The `GET` command loads the contents of the given file to the SQL buffer.

```
GET filename[.ext ] [ LIS[T ] | NOL[IST ] ]
```

filename[*.ext*]

> *filename* is the name of the file to load into the SQL buffer. *ext* is the filename extension. If the filename extension is `sql`, then the `.sql` extension may be omitted when specifying *filename*. `GET` always assumes a `.sql` extension on filenames that are specified with no extension.

LIST | NOLIST

> If `LIST` is specified, the content of the SQL buffer is displayed after the file is loaded. If `NOLIST` is specified, no listing is displayed. The default is `LIST`.

11.1.2.15 HELP

The `HELP` command obtains an index of topics or help on a specific topic. The question mark (`?`) is synonymous with specifying `HELP`.

```
{ HELP | ? } { INDEX | topic }
```

INDEX

> Displays an index of available topics.

topic

> The name of a specific topic – e.g., an EDB*Plus command, for which help is desired.

11.1.2.16 HOST

The `HOST` command executes an operating system command from EDB*Plus.

```
HO[ST ] [os_command]
```

os_command

> The operating system command to be executed. If you do not provide an operating system command, EDB*Plus pauses execution and opens a new shell prompt. When the shell exits, EDB*Plus resumes execution.

11.1.2.17 INPUT

The `INPUT` line editor command adds a line of text to the SQL buffer after the current line.

```
I[NPUT ] text
```

The following sequence of `INPUT` commands constructs a `SELECT` command.

```
SQL> INPUT SELECT empno, ename, job, sal, comm
SQL> INPUT FROM emp
SQL> INPUT WHERE deptno = 20
SQL> INPUT ORDER BY empno
SQL> LIST
  1  SELECT empno, ename, job, sal, comm
  2  FROM emp
  3  WHERE deptno = 20
  4* ORDER BY empno
```

11.1.2.18 LIST

`LIST` is a line editor command that displays the contents of the SQL buffer.

```
L[IST] [ n | n m | n * | n L[AST] | * | * n | * L[AST] | L[AST] ]
```

The buffer does not include a history of the EDB*Plus commands.

n

`n` represents the buffer line number.

n m

`n m` displays a list of lines between `n` and `m`.

*n **

`n *` displays a list of lines that range between line `n` and the current line.

n L[AST]

`n L[AST]` displays a list of lines that range from line `n` through the last line in the buffer.

`*`

`*` displays the current line.

`* n`

`* n` displays a list of lines that range from the current line through line `n`.

`* L[AST]`

`* L[AST]` displays a list of lines that range from the current line through the last line.

`L[AST]`

`L[AST]` displays the last line.

11.1.2.19 PASSWORD

Use the `PASSWORD` command to change your database password.

    PASSW[ORD] [user_name]

You must have sufficient privileges to use the `PASSWORD` command to change another user's password. The following example demonstrates using the `PASSWORD` command to change the password for a user named `acctg`:

```
SQL> PASSWORD acctg
Changing password for acctg
    New password:
    New password again:
Password successfully changed.
```

11.1.2.20 PAUSE

The `PAUSE` command displays a message, and waits for the user to press `ENTER`.

    PAU[SE] [optional_text]

`optional_text` specifies the text that will be displayed to the user. If the *`optional_text`* is omitted, Advanced Server will display two blank lines. If you double quote the *`optional_text`* string, the quotes will be included in the output.

11.1.2.21 PRINT

The `PRINT` command displays the value of a bind variable.

```
PRI[NT] [bind_variable_name]
```

`bind_variable_name` specifies the name of a bind variable. Omit *`bind_variable_name`* to generate a list that includes the values of all bind variables.

11.1.2.22 PROMPT

The `PROMPT` command displays a message to the user before continuing.

```
PRO[MPT] [message_text]
```

`message_text` specifies the text displayed to the user. Double quote the string to include quotes in the output.

11.1.2.23 QUIT

The `QUIT` command terminates the session and returns control to the operating system. `QUIT` is a synonym for `EXIT`.

```
QUIT

[SUCCESS | FAILURE | WARNING | value | sub_variable]

[COMMIT | ROLLBACK]
```

The default value is `QUIT SUCCESS COMMIT`.

11.1.2.24 REMARK

Use `REMARK` to include comments in a script.

```
REM[ARK] [optional_text]
```

You may also use the following convention to include a comment:

```
/*
```

```
 *  This is an example of a three line comment.
 */
```

11.1.2.25 SAVE

Use the `SAVE` command to write the SQL Buffer to an operating system file.

```
SAV[E] file_name
[CRE[ATE] | REP[LACE] | APP[END]]
```

`file_name`

> *`file_name`* specifies the name of the file (including the path) where the buffer contents are written. If you do not provide a file extension, `.sql` is appended to the end of the file name.

`CREATE`

> Include the `CREATE` keyword to create a new file. A new file is created *only* if a file with the specified name does not already exist. This is the default.

`REPLACE`

> Include the `REPLACE` keyword to specify that Advanced Server should overwrite an existing file.

`APPEND`

> Include the `APPEND` keyword to specify that Advanced Server should append the contents of the SQL buffer to the end of the specified file.

The following example saves the contents of the SQL buffer to a file named `example.sql`, located in the `temp` directory:

```
SQL> SAVE C:\example.sql CREATE
File "example.sql" written.
```

11.1.2.26 SET

Use the `SET` command to specify a value for a session level variable that controls EDB*Plus behavior. The following forms of the `SET` command are valid:

SET AUTOCOMMIT

Use the `SET AUTOCOMMIT` command to specify `COMMIT` behavior for Advanced Server transactions.

```
SET AUTO[COMMIT]

{ON | OFF | IMMEDIATE | statement_count}
```

Please note that EDB*Plus always automatically commits DDL statements.

ON

Specify `ON` to turn `AUTOCOMMIT` behavior on.

OFF

Specify `OFF` to turn `AUTOCOMMIT` behavior off.

IMMEDIATE

`IMMEDIATE` has the same effect as `ON`.

statement_count

Include a value for *statement_count* to instruct EDB*Plus to issue a commit after the specified count of successful SQL statements.

SET COLUMN SEPARATOR

Use the `SET COLUMN SEPARATOR` command to specify the text that Advanced Server displays between columns.

```
SET COLSEP column_separator
```

The default value of *column_separator* is a single space.

SET ECHO

Use the `SET ECHO` command to specify if SQL and EDB*Plus script statements should be displayed onscreen as they are executed.

```
SET ECHO {ON | OFF}
```

The default value is `OFF`.

SET FEEDBACK

The `SET FEEDBACK` command controls the display of interactive information after a SQL statement executes.

```
SET FEED[BACK] {ON | OFF | row_threshold}
```

`row_threshold`

> Specify an integer value for *`row_threshold`*. Setting *`row_threshold`* to `0` is same as setting `FEEDBACK` to `OFF`. Setting *`row_threshold`* equal `1` effectively sets `FEEDBACK` to `ON`.

SET FLUSH

Use the `SET FLUSH` command to control display buffering.

```
SET FLU[SH] {ON | OFF}
```

Set `FLUSH` to `OFF` to enable display buffering. If you enable buffering, messages bound for the screen may not appear until the script completes. Please note that setting `FLUSH` to `OFF` will offer better performance.

Set `FLUSH` to `ON` to disable display buffering. If you disable buffering, messages bound for the screen appear immediately.

SET HEADING

Use the `SET HEADING` variable to specify if Advanced Server should display column headings for `SELECT` statements.

```
SET HEA[DING] {ON | OFF}
```

SET HEAD SEPARATOR

The `SET HEADSEP` command sets the new heading separator character used by the `COLUMN HEADING` command. The default is '|'.

```
SET HEADS[EP]
```

SET LINESIZE

Use the `SET LINESIZE` command to specify the width of a line in characters.

```
SET LIN[ESIZE] width_of_line
```

width_of_line

The default value of *width_of_line* is `132`.

SET NEWPAGE

Use the `SET NEWPAGE` command to specify how many blank lines are printed after a page break.

```
SET NEWP[AGE] lines_per_page
```

lines_per_page

The default value of *lines_per_page* is `1`.

SET NULL

Use the `SET NULL` command to specify a string that is displayed to the user when a `NULL` column value is displayed in the output buffer.

```
SET NULL null_string
```

SET PAGESIZE

Use the `SET PAGESIZE` command to specify the number of printed lines that fit on a page.

```
SET PAGES[IZE] line_count
```

Use the `line_count` parameter to specify the number of lines per page.

SET SQLCASE

The `SET SQLCASE` command specifies if SQL statements transmitted to the server should be converted to upper or lower case.

```
SET SQLC[ASE] {MIX[ED] | UP[PER] | LO[WER]}
```

UPPER

Specify `UPPER` to convert the command text to uppercase.

`LOWER`

Specify `LOWER` to convert the command text to lowercase.

MIXED

Specify MIXED to leave the case of SQL commands unchanged. The default is MIXED.

SET PAUSE

The SET PAUSE command is most useful when included in a script; the command displays a prompt and waits for the user to press Return.

```
SET PAU[SE] {ON | OFF}
```

If SET PAUSE is ON, the message Hit ENTER to continue… will be displayed before each command is executed.

SET SPACE

Use the SET SPACE command to specify the number of spaces to display between columns:

```
SET SPACE number_of_spaces
```

SET SQLPROMPT

Use SET SQLPROMPT to set a value for a user-interactive prompt:

```
SET SQLP[ROMPT] "prompt"
```

By default, SQLPROMPT is set to "SQL> "

SQL TERMOUT

Use the SQL TERMOUT command to specify if command output should be displayed onscreen.

```
SET TERM[OUT] {ON | OFF}
```

SQL TIMING

The SQL TIMING command specifies if Advanced Server should display the execution time for each SQL statement after it is executed.

```
SET TIMI[NG] {ON | OFF}
```

SET VERIFY

Specifies if both the old and new values of a SQL statement are displayed when a substitution variable is encountered.

```
SET VER[IFY] { ON | OFF }
```

11.1.2.27 SHOW

Use the `SHOW` command to display current parameter values.

```
SHO[W] {ALL | parameter_name}
```

Display the current parameter settings by including the `ALL` keyword:

```
SQL> SHOW ALL
autocommit       OFF
colsep           " "
define           "&"
echo             OFF
FEEDBACK ON for 6 row(s).
flush            ON
heading          ON
headsep          "|"
linesize         78
newpage          1
null             " "
pagesize         14
pause            OFF
serveroutput     OFF
spool            OFF
sqlcase          MIXED
sqlprompt        "SQL> "
sqlterminator    ";"
suffix           ".sql"
termout          ON
timing           OFF
verify           ON
USER is          "enterprisedb"
HOST is          "localhost"
PORT is          "5444"
DATABASE is      "edb"
VERSION is       "9.5.0.0"
```

Or display a specific parameter setting by including the `parameter_name` in the `SHOW` command:

```
SQL> SHOW VERSION
VERSION is "9.5.0.0"
```

11.1.2.28 SPOOL

The `SPOOL` command sends output from the display to a file.

```
SP[OOL] output_file | OFF
```

Use the *`output_file`* parameter to specify a path name for the output file.

11.1.2.29 START

Use the `START` command to run an EDB*Plus script file; `START` is an alias for `@` command.

```
STA[RT] script_file
```

Specify the name of a script file in the *`script_file`* parameter.

11.1.2.30 UNDEFINE

The `UNDEFINE` command erases a user variable created by the `DEFINE` command.

```
UNDEF[INE] variable_name [ variable_name...]
```

Use the *`variable_name`* parameter to specify the name of a variable or variables.

11.1.2.31 WHENEVER SQLERROR

The `WHENEVER SQLERROR` command provides error handling for SQL errors or PL/SQL block errors. The syntax is:

```
WHENEVER SQLERROR
  {CONTINUE [COMMIT|ROLLBACK|NONE]
  |EXIT [SUCCESS|FAILURE|WARNING|n|sub_variable]
  [COMMIT|ROLLBACK]}
```

If Advanced Server encounters an error during the execution of a SQL command or PL/SQL block, EDB*Plus performs the action specified in the `WHENEVER SQLERROR` command:

Include the `CONTINUE` clause to instruct EDB*Plus to perform the specified action before continuing.

Include the `COMMIT` clause to instruct EDB*Plus to `COMMIT` the current transaction before exiting or continuing.

Include the `ROLLBACK` clause to instruct EDB*Plus to `ROLLBACK` the current transaction before exiting or continuing.

Include the `NONE` clause to instruct EDB*Plus to continue without committing or rolling back the transaction.

Include the `EXIT` clause to instruct EDB*Plus to perform the specified action and exit if it encounters an error.

Use the following options to specify a status code that EDB*Plus will return before exiting:

```
[SUCCESS|FAILURE|WARNING|n|sub_variable]
```

Please note that EDB*Plus supports substitution variables, but does not support bind variables.

*11.2 EDB*Loader*

EDB*Loader is a high-performance bulk data loader that provides an interface compatible with Oracle databases for Advanced Server. The EDB*Loader command line utility loads data from an input source, typically a file, into one or more tables using a subset of the parameters offered by Oracle SQL*Loader.

EDB*Loader features include:

- Support for the Oracle SQL*Loader data loading methods - conventional path load, direct path load, and parallel direct path load
- Syntax for control file directives compatible with Oracle SQL*Loader
- Input data with delimiter-separated or fixed-width fields
- Bad file for collecting rejected records
- Loading of multiple target tables
- Discard file for collecting records that do not meet the selection criteria of any target table
- Log file for recording the EDB*Loader session and any error messages
- Data loading from standard input and remote loading, particularly useful for large data sources on remote hosts

These features are explained in detail in the following sections.

Note: The following are important version compatibility restrictions between the EDB*Loader client and the database server.

- Invoking EDB*Loader is done using a client program called `edbldr`, which is used to pass parameters and directive information to the database server. **It is strongly recommended that the 9.5 EDB*Loader client (that is, the edbldr program supplied with Advanced Server 9.5) be used to load data only into version 9.5 of the database server. In general, the EDB*Loader client and database server should be the same version.**
- It is possible to use a 9.5 EDB*Loader client to load data into a 9.5 database server, but the new 9.5 EDB*Loader features may not be available under those circumstances.
- Use of a 9.5, 9.4 or 9.3 EDB*Loader client is not supported for database servers version 9.2 or earlier.

11.2.1 Data Loading Methods

As with Oracle SQL*Loader, EDB*Loader supports three data loading methods:

- Conventional path load
- Direct path load
- Parallel direct path load

Conventional path load is the default method used by EDB*Loader. Basic insert processing is used to add rows to the table.

The advantage of a conventional path load over the other methods is that table constraints and database objects defined on the table such as primary keys, not null constraints, check constraints, unique indexes, foreign key constraints, and triggers are enforced during a conventional path load.

One exception is that the Advanced Server *rules* defined on the table are not enforced. EDB*Loader can load tables on which rules are defined, but the rules are not executed. As a consequence, partitioned tables implemented using rules cannot be loaded using EDB*Loader.

Note: Advanced Server rules are created by the `CREATE RULE` command. Advanced Server rules are not the same database objects as rules and rule sets used in Oracle.

EDB*Loader also supports direct path loads. A direct path load is faster than a conventional path load, but requires the removal of most types of constraints and triggers from the table. See Section 11.2.5 for information on direct path loads.

Finally, EDB*Loader supports parallel direct path loads. A parallel direct path load provides even greater performance improvement by permitting multiple EDB*Loader sessions to run simultaneously to load a single table. See Section 11.2.6 for information on parallel direct path loads.

11.2.2 General Usage

EDB*Loader can load data files with either delimiter-separated or fixed-width fields, in single-byte or multi-byte character sets. The delimiter can be a string consisting of one or more single-byte or multi-byte characters. Data file encoding and the database encoding may be different. Character set conversion of the data file to the database encoding is supported.

Each EDB*Loader session runs as a single, independent transaction. If an error should occur during the EDB*Loader session that aborts the transaction, all changes made during the session are rolled back.

Generally, formatting errors in the data file do not result in an aborted transaction. Instead, the badly formatted records are written to a text file called the *bad file*. The reason for the error is recorded in the *log file*.

Records causing database integrity errors do result in an aborted transaction and rollback. As with formatting errors, the record causing the error is written to the bad file and the reason is recorded in the log file.

Note: EDB*Loader differs from Oracle SQL*Loader in that a database integrity error results in a rollback in EDB*Loader. In Oracle SQL*Loader, only the record causing the error is rejected. Records that were previously inserted into the table are retained and loading continues after the rejected record.

The following are examples of types of formatting errors that do not abort the transaction:

- Attempt to load non-numeric value into a numeric column
- Numeric value is too large for a numeric column
- Character value is too long for the maximum length of a character column
- Attempt to load improperly formatted date value into a date column

The following are examples of types of database errors that abort the transaction and result in the rollback of all changes made in the EDB*Loader session:

- Violation of a unique constraint such as a primary key or unique index
- Violation of a referential integrity constraint
- Violation of a check constraint
- Error thrown by a trigger fired as a result of inserting rows

11.2.3 Building the EDB*Loader Control File

When you invoke EDB*Loader, the list of arguments provided must include the name of a control file. The control file includes the instructions that EDB*Loader uses to load the table (or tables) from the input data file. The control file includes information such as:

- The name of the input data file containing the data to be loaded.
- The name of the table or tables to be loaded from the data file.
- Names of the columns within the table or tables and their corresponding field placement in the data file.
- Specification of whether the data file uses a delimiter string to separate the fields, or if the fields occupy fixed column positions.
- Optional selection criteria to choose which records from the data file to load into a given table.
- The name of the file that will collect illegally formatted records.
- The name of the discard file that will collect records that do not meet the selection criteria of any table.

The syntax for the EDB*Loader control file is as follows:

```
[ OPTIONS (param=value [, param=value ] ...) ]
LOAD DATA
  [ CHARACTERSET charset ]
  [ INFILE '{ data_file | stdin }' ]
  [ BADFILE 'bad_file' ]
  [ DISCARDFILE 'discard_file' ]
  [ { DISCARDMAX | DISCARDS } max_discard_recs ]
[ INSERT | APPEND | REPLACE | TRUNCATE ]
[ PRESERVE BLANKS ]
{ INTO TABLE target_table
  [ WHEN field_condition [ AND field_condition ] ...]
  [ FIELDS TERMINATED BY 'termstring'
    [ OPTIONALLY ENCLOSED BY 'enclstring' ] ]
  [ TRAILING NULLCOLS ]
   (field_def [, field_def ] ...)
} ...
```

where *field_def* defines a field in the specified *data_file* that describes the location, data format, or value of the data to be inserted into *column_name* of the *target_table*. The syntax of *field_def* is the following:

```
column_name {
  CONSTANT val |
  FILLER [ POSITION (start:end) ] [ fieldtype ] |
  [ POSITION (start:end) ] [ fieldtype ]
```

```
      [ PRESERVE BLANKS ] [ "expr" ]
    }
```

where *fieldtype* is one of:

```
    CHAR | DATE [ "datemask" ] | INTEGER EXTERNAL |
    FLOAT EXTERNAL | DECIMAL EXTERNAL | ZONED EXTERNAL |
    ZONED [(precision[,scale])]
```

Description

The specification of *data_file*, *bad_file*, and *discard_file* may include the full directory path or a relative directory path to the file name. If the file name is specified alone or with a relative directory path, the file is then assumed to exist (in the case of *data_file*), or is created (in the case of *bad_file* or *discard_file*), relative to the current working directory from which `edbldr` is invoked.

You can include references to environment variables within the EDB*Loader control file when referring to a directory path and/or file name. Environment variable references are formatted differently on Windows systems than on Linux systems:

- On Linux, the format is `$`*ENV_VARIABLE* or `${`*ENV_VARIABLE*`}`
- On Windows, the format is `%`*ENV_VARIABLE*`%`

Where *ENV_VARIABLE* is the environment variable that is set to the directory path and/or file name.

The `EDBLDR_ENV_STYLE` environment variable instructs Advanced Server to interpret environment variable references as Windows-styled references or Linux-styled references irregardless of the operating system on which EDB*Loader resides. You can use this environment variable to create portable control files for EDB*Loader.

- On a Windows system, set `EDBLDR_ENV_STYLE` to `linux` or `unix` to instruct Advanced Server to recognize Linux-style references within the control file.
- On a Linux system, set `EDBLDR_ENV_STYLE` to `windows` to instruct Advanced Server to recognize Windows-style references within the control file.

The operating system account `enterprisedb` must have read permission on the directory and file specified by *data_file*.

The operating system account `enterprisedb` must have write permission on the directories where *bad_file* and *discard_file* are to be written.

Note: It is suggested that the file names for *data_file*, *bad_file*, and *discard_file* include extensions of .dat, .bad, and .dsc, respectively. If the provided file name does not contain an extension, EDB*Loader assumes the actual file name includes the appropriate aforementioned extension.

If an EDB*Loader session results in data format errors and the BADFILE clause is not specified, nor is the BAD parameter given on the command line when edbldr is invoked, a bad file is created with the name *control_file_base*.bad in the current working directory from which edbldr is invoked. *control_file_base* is the base name of the control file (that is, the file name without any extension) used in the edbldr session.

If all of the following conditions are true, the discard file is not created even if the EDB*Loader session results in discarded records:

- The DISCARDFILE clause for specifying the discard file is not included in the control file.
- The DISCARD parameter for specifying the discard file is not included on the command line.
- The DISCARDMAX clause for specifying the maximum number of discarded records is not included in the control file.
- The DISCARDS clause for specifying the maximum number of discarded records is not included in the control file.
- The DISCARDMAX parameter for specifying the maximum number of discarded records is not included on the command line.

If neither the DISCARDFILE clause nor the DISCARD parameter for explicitly specifying the discard file name are specified, but DISCARDMAX or DISCARDS is specified, then the EDB*Loader session creates a discard file using the data file name with an extension of .dsc.

Note: There is a distinction between keywords DISCARD and DISCARDS. DISCARD is an EDB*Loader command line parameter used to specify the discard file name (see Section 11.2.4). DISCARDS is a clause of the LOAD DATA directive that may only appear in the control file. Keywords DISCARDS and DISCARDMAX provide the same functionality of specifying the maximum number of discarded records allowed before terminating the EDB*Loader session. Records loaded into the database before termination of the EDB*Loader session due to exceeding the DISCARDS or DISCARDMAX settings are kept in the database and are not rolled back.

If one of INSERT, APPEND, REPLACE, or TRUNCATE is specified, it establishes the default action of how rows are to be added to target tables. If omitted, the default action is as if INSERT had been specified.

If the FIELDS TERMINATED BY clause is specified, then the POSITION (*start*:*end*) clause may not be specified for any *field_def*. Alternatively if the FIELDS

`TERMINATED BY` clause is not specified, then every *`field_def`* must contain the `POSITION (`*`start`*`:`*`end`*`)` clause, excluding those with the `CONSTANT` clause.

Parameters

`OPTIONS` *`param=value`*

Use the `OPTIONS` clause to specify *`param=value`* pairs that represent an EDB*Loader directive. If a parameter is specified in both the `OPTIONS` clause and on the command line when `edbldr` is invoked, the command line setting is used.

Specify one or more of the following parameter/value pairs:

`DIRECT= { FALSE | TRUE }`

If `DIRECT` is set to `TRUE` EDB*Loader performs a direct path load instead of a conventional path load. The default value of `DIRECT` is `FALSE`.

See Section 11.2.5 for information on direct path loads.

`ERRORS=`*`error_count`*

`error_count` specifies the number of errors permitted before aborting the EDB*Loader session. The default is `50`.

`FREEZE= { FALSE | TRUE }`

Set `FREEZE` to `TRUE` to indicate that the data should be copied with the rows *frozen*. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wrap-around. For more information about frozen tuples, see the PostgreSQL core documentation at:

http://www.postgresql.org/docs/9.5/static/routine-vacuuming.html

You must specify a data-loading type of `TRUNCATE` in the control file when using the `FREEZE` option. `FREEZE` is not supported for direct loading.

By default, `FREEZE` is `FALSE`.

`PARALLEL= { FALSE | TRUE }`

Set `PARALLEL` to `TRUE` to indicate that this EDB*Loader session is one of a number of concurrent EDB*Loader sessions participating in a parallel direct path load. The default value of `PARALLEL` is `FALSE`.

When `PARALLEL` is `TRUE`, the `DIRECT` parameter must also be set to `TRUE` . See Section 11.2.6 for more information about parallel direct path loads.

`ROWS=`*`n`*

`n` specifies the number of rows that EDB*Loader will commit before loading the next set of *`n`* rows.

If EDB*Loader encounters an invalid row during a load (in which the `ROWS` parameter is specified), those rows committed prior to encountering the error will remain in the destination table.

`SKIP=`*`skip_count`*

`skip_count` specifies the number of records at the beginning of the input data file that should be skipped before loading begins. The default is `0`.

`SKIP_INDEX_MAINTENANCE={ FALSE | TRUE }`

If `SKIP_INDEX_MAINTENANCE` is `TRUE`, index maintenance is not performed as part of a direct path load, and indexes on the loaded table are marked as invalid. The default value of `SKIP_INDEX_MAINTENANCE` is `FALSE`.

Please note: During a parallel direct path load, target table indexes are not updated, and are marked as invalid after the load is complete.

You can use the `REINDEX` command to rebuild an index. For more information about the `REINDEX` command, see the PostgreSQL core documentation available at:

http://www.postgresql.org/docs/9.5/static/sql-reindex.html

`charset`

Use the `CHARACTERSET` clause to identify the character set encoding of *`data_file`* where *`charset`* is the character set name. This clause is required if the data file encoding differs from the control file encoding. (The control file encoding must always be in the encoding of the client where `edbldr` is invoked.)

Examples of *charset* settings are `UTF8`, `SQL_ASCII`, and `SJIS`.

For more information about client to database character set conversion, see the PostgreSQL core documentation available at:

http://www.postgresql.org/docs/9.5/static/multibyte.html

data_file

File containing the data to be loaded into *target_table*. Each record in the data file corresponds to a row to be inserted into *target_table*.

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of `.dat`, for example, `mydatafile.dat`.

Note: If the `DATA` parameter is specified on the command line when `edbldr` is invoked, the file given by the command line `DATA` parameter is used instead.

If the `INFILE` clause is omitted as well as the command line `DATA` parameter, then the data file name is assumed to be identical to the control file name, but with an extension of `.dat`.

`stdin`

Specify `stdin` (all lowercase letters) if you want to use standard input to pipe the data to be loaded directly to EDB*Loader. This is useful for data sources generating a large number of records to be loaded.

bad_file

File that receives *data_file* records that cannot be loaded due to errors.

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of `.bad`, for example, `mybadfile.bad`.

Note: If the `BAD` parameter is specified on the command line when `edbldr` is invoked, the file given by the command line `BAD` parameter is used instead.

discard_file

File that receives input data records that are not loaded into any table because none of the selection criteria are met for tables with the `WHEN` clause, and there are no tables without a `WHEN` clause. (All records meet the selection criteria of a table without a `WHEN` clause.)

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of `.dsc`, for example, `mydiscardfile.dsc`.

Note: If the `DISCARD` parameter is specified on the command line when `edbldr` is invoked, the file given by the command line `DISCARD` parameter is used instead.

`{ DISCARDMAX | DISCARDS }` *`max_discard_recs`*

Maximum number of discarded records that may be encountered from the input data records before terminating the EDB*Loader session. (A discarded record is described in the preceding description of the *`discard_file`* parameter.) Either keyword `DISCARDMAX` or `DISCARDS` may be used preceding the integer value specified by *`max_discard_recs`*.

For example, if *`max_discard_recs`* is `0`, then the EDB*Loader session is terminated if and when a first discarded record is encountered. If *`max_discard_recs`* is `1`, then the EDB*Loader session is terminated if and when a second discarded record is encountered.

When the EDB*Loader session is terminated due to exceeding *`max_discard_recs`*, prior input data records that have been loaded into the database are retained. They are not rolled back.

`INSERT | APPEND | REPLACE | TRUNCATE`

Specifies how data is to be loaded into the target tables. If one of `INSERT`, `APPEND`, `REPLACE`, or `TRUNCATE` is specified, it establishes the default action for all tables, overriding the default of `INSERT`.

`INSERT`

Data is to be loaded into an empty table. EDB*Loader throws an exception and does not load any data if the table is not initially empty.

Note: If the table contains rows, the `TRUNCATE` command must be used to empty the table prior to invoking EDB*Loader. EDB*Loader throws an exception if the `DELETE` command is used to empty the table instead of the `TRUNCATE` command. Oracle SQL*Loader allows the table to be emptied by using either the `DELETE` or `TRUNCATE` command.

`APPEND`

Data is to be added to any existing rows in the table. The table may be initially empty as well.

`REPLACE`

The `REPLACE` keyword and `TRUNCATE` keywords are functionally identical. The table is truncated by EDB*Loader prior to loading the new data.

Note: Delete triggers on the table are not fired as a result of the `REPLACE` operation.

`TRUNCATE`

The table is truncated by EDB*Loader prior to loading the new data. Delete triggers on the table are not fired as a result of the `TRUNCATE` operation.

`PRESERVE BLANKS`

For all target tables, retains leading white space when the optional enclosure delimiters are not present and leaves trailing white space intact when fields are specified with a predetermined size. When omitted, the default behavior is to trim leading and trailing white space.

`target_table`

Name of the table into which data is to be loaded. The table name may be schema-qualified (for example, `enterprisedb.emp`). The specified target must not be a view.

`field_condition`

Conditional clause taking the following form:

```
[ ( ] (start:end) { = | != | <> } 'val' [ ) ]
```

`start` and *`end`* are positive integers specifying the column positions in *`data_file`* that mark the beginning and end of a field that is to be compared with the constant *`val`*. The first character in each record begins with a *`start`* value of `1`.

In the `WHEN` *`field_condition`* `[ AND` *`field_condition`* `]` clause, if all such conditions evaluate to true for a given record, then EDB*Loader attempts to insert that record into *target_table*. If the insert operation fails, the record is written to *bad_file*.

All characters used in the *field_condition* text (particularly in the *val* string) must be valid in the database encoding. (For performing data conversion, EDB*Loader first converts the characters in *val* string to the database encoding and then to the data file encoding.)

If for a given record, none of the `WHEN` clauses evaluate to true for all `INTO TABLE` clauses, the record is written to *discard_file*, if a discard file was specified for the EDB*Loader session.

termstring

String of one or more characters that separates each field in *data_file*. The characters may be single-byte or multi-byte as long as they are valid in the database encoding. Two consecutive appearances of *termstring* with no intervening character results in the corresponding column set to null.

enclstring

String of one or more characters used to enclose a field value in *data_file*. The characters may be single-byte or multi-byte as long as they are valid in the database encoding. Use *enclstring* on fields where *termstring* appears as part of the data.

`TRAILING NULLCOLS`

If `TRAILING NULLCOLS` is specified, then the columns in the column list for which there is no data in *data_file* for a given record, are set to null when the row is inserted. This applies only to one or more consecutive columns at the end of the column list.

If fields are omitted at the end of a record and `TRAILING NULLCOLS` is not specified, EDB*Loader assumes the record contains formatting errors and writes it to the bad file.

column_name

Name of a column in *target_table* into which a field value defined by *field_def* is to be inserted.

`CONSTANT` *`val`*

Specifies a constant that is type-compatible with the column data type to which it is assigned in a field definition. Single or double quotes may enclose *`val`*. If *`val`* contains white space, then enclosing quotation marks must be used.

The use of the `CONSTANT` clause completely determines the value to be assigned to a column in each inserted row. No other clause may appear in the same field definition.

If the `TERMINATED BY` clause is used to delimit the fields in *`data_file`*, there must be no delimited field in *`data_file`* corresponding to any field definition with a `CONSTANT` clause. In other words, EDB*Loader assumes there is no field in *`data_file`* for any field definition with a `CONSTANT` clause.

`FILLER`

Specifies that the data in the field defined by the field definition is not to be loaded into the associated column. The column is set to null.

A column name defined with the `FILLER` clause must not be referenced in a SQL expression. See the discussion of the *`expr`* parameter.

`POSITION (`*`start`*`:`*`end`*`)`

Defines the location of the field in a record in a fixed-width field data file. *`start`* and *`end`* are positive integers. The first character in the record has a start value of `1`.

```
CHAR | DATE [ "datemask" ] | INTEGER EXTERNAL |
FLOAT EXTERNAL | DECIMAL EXTERNAL | ZONED EXTERNAL |
ZONED [(precision[,scale])]
```

Field type that describes the format of the data field in *`data_file`*.

Note: Specification of a field type is optional (for descriptive purposes only) and has no effect on whether or not EDB*Loader successfully inserts the data in the field into the table column. Successful loading depends upon the compatibility of the column data type and the field value. For example, a column with data type `NUMBER(7,2)` successfully accepts a field containing `2600`, but if the field contains a value such as `26XX`, the insertion fails and the record is written to *`bad_file`*.

Please note that `ZONED` data is not human-readable; `ZONED` data is stored in an internal format where each digit is encoded in a separate nibble/nybble/4-bit field.

In each `ZONED` value, the last byte contains a single digit (in the high-order 4 bits) and the sign (in the low-order 4 bits).

`precision`

Use *`precision`* to specify the length of the `ZONED` value.

If the *`precision`* value specified for `ZONED` conflicts with the length calculated by the server based on information provided with the `POSITION` clause, EDB*Loader will use the value specified for *`precision`*.

`scale`

`scale` specifies the number of digits to the right of the decimal point in a `ZONED` value.

`datemask`

Specifies the ordering and abbreviation of the day, month, and year components of a date field.

See Section 3.5.7 for date mask formatting information.

Note: If the `DATE` field type is specified along with a SQL expression for the column, then *`datemask`* must be specified after `DATE` and before the SQL expression. See the following discussion of the *`expr`* parameter.

`PRESERVE BLANKS`

For the column on which this option appears, retains leading white space when the optional enclosure delimiters are not present and leaves trailing white space intact when fields are specified with a predetermined size. When omitted, the default behavior is to trim leading and trailing white space.

`expr`

A SQL expression returning a scalar value that is type-compatible with the column data type to which it is assigned in a field definition. Double quotes must enclose *`expr`*. *`expr`* may contain a reference to any column in the field list (except for fields with the `FILLER` clause) by prefixing the column name by a colon character (`:`).

Examples

The following are some examples of control files and their corresponding data files.

The following control file uses a delimiter-separated data file that appends rows to the `emp` table:

```
LOAD DATA
  INFILE    'emp.dat'
    BADFILE 'emp.bad'
  APPEND
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
  )
```

In the preceding control file, the `APPEND` clause is used to allow the insertion of additional rows into the `emp` table.

The following is the corresponding delimiter-separated data file:

```
9101,ROGERS,CLERK,7902,17-DEC-10,1980.00,20
9102,PETERSON,SALESMAN,7698,20-DEC-10,2600.00,30,2300.00
9103,WARREN,SALESMAN,7698,22-DEC-10,5250.00,30,2500.00
9104,"JONES, JR.",MANAGER,7839,02-APR-09,7975.00,20
```

The use of the `TRAILING NULLCOLS` clause allows the last field supplying the `comm` column to be omitted from the first and last records. The `comm` column is set to null for the rows inserted from these records.

The double quotation mark enclosure character surrounds the value `JONES, JR.` in the last record since the comma delimiter character is part of the field value.

The following query displays the rows added to the table after the EDB*Loader session:

```
SELECT * FROM emp WHERE empno > 9100;

empno |   ename    |   job    | mgr  |      hiredate      |   sal    |  comm    | deptno
-------+------------+----------+------+--------------------+---------+---------+-------
-
  9101 | ROGERS     | CLERK    | 7902 | 17-DEC-10 00:00:00 | 1980.00 |         |     20
  9102 | PETERSON   | SALESMAN | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2300.00 |     30
  9103 | WARREN     | SALESMAN | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 2500.00 |     30
  9104 | JONES, JR. | MANAGER  | 7839 | 02-APR-09 00:00:00 | 7975.00 |         |     20
(4 rows)
```

The following example is a control file that loads the same rows into the `emp` table, but uses a data file containing fixed-width fields:

```
LOAD DATA
  INFILE        'emp_fixed.dat'
    BADFILE     'emp_fixed.bad'
  APPEND
  INTO TABLE emp
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46),
    deptno      POSITION (47:48),
    comm        POSITION (49:56)
  )
```

In the preceding control file, the `FIELDS TERMINATED BY` and `OPTIONALLY ENCLOSED BY` clauses are absent. Instead, each field now includes the `POSITION` clause.

The following is the corresponding data file containing fixed-width fields:

```
9101ROGERS    CLERK    790217-DEC-10   1980.0020
9102PETERSON  SALESMAN 769820-DEC-10   2600.0030 2300.00
9103WARREN    SALESMAN 769822-DEC-10   5250.0030 2500.00
9104JONES, JR.MANAGER  783902-APR-09   7975.0020
```

The following control file illustrates the use of the `FILLER` clause in the data fields for the `sal` and `comm` columns. EDB*Loader ignores the values in these fields and sets the corresponding columns to null.

```
LOAD DATA
  INFILE        'emp_fixed.dat'
    BADFILE     'emp_fixed.bad'
  APPEND
  INTO TABLE emp
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         FILLER POSITION (39:46),
    deptno      POSITION (47:48),
    comm        FILLER POSITION (49:56)
  )
```

Using the same fixed-width data file as in the prior example, the resulting rows in the table appear as follows:

```
SELECT * FROM emp WHERE empno > 9100;

empno |      ename       |  job     | mgr  |      hiredate       | sal | comm | deptno
```

```
-------+------------------+----------+------+--------------------+-----+------+--------
  9101 | ROGERS           | CLERK    | 7902 | 17-DEC-10 00:00:00 |     |      |     20
  9102 | PETERSON         | SALESMAN | 7698 | 20-DEC-10 00:00:00 |     |      |     30
  9103 | WARREN           | SALESMAN | 7698 | 22-DEC-10 00:00:00 |     |      |     30
  9104 | JONES, JR.       | MANAGER  | 7839 | 02-APR-09 00:00:00 |     |      |     20
(4 rows)
```

The following example illustrates the use of multiple `INTO TABLE` clauses. For this example, two empty tables are created with the same data definition as the `emp` table. The following `CREATE TABLE` commands create these two empty tables, while inserting no rows from the original `emp` table:

```
CREATE TABLE emp_research AS SELECT * FROM emp WHERE deptno = 99;
CREATE TABLE emp_sales AS SELECT * FROM emp WHERE deptno = 99;
```

The following control file contains two `INTO TABLE` clauses. Also note that there is no `APPEND` clause so the default operation of `INSERT` is used, which requires that tables `emp_research` and `emp_sales` be empty.

```
LOAD DATA
  INFILE        'emp_multitbl.dat'
    BADFILE     'emp_multitbl.bad'
    DISCARDFILE 'emp_multitbl.dsc'
  INTO TABLE emp_research
    WHEN (47:48) = '20'
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46),
    deptno      CONSTANT '20',
    comm        POSITION (49:56)
  )
  INTO TABLE emp_sales
    WHEN (47:48) = '30'
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46),
    deptno      CONSTANT '30',
    comm        POSITION (49:56) "ROUND(:comm + (:sal * .25), 0)"
  )
```

The `WHEN` clauses specify that when the field designated by columns 47 thru 48 contains `20`, the record is inserted into the `emp_research` table and when that same field contains `30`, the record is inserted into the `emp_sales` table. If neither condition is true, the record is written to the discard file named `emp_multitbl.dsc`.

The CONSTANT clause is given for column deptno so the specified constant value is inserted into deptno for each record. When the CONSTANT clause is used, it must be the only clause in the field definition other than the column name to which the constant value is assigned.

Finally, column comm of the emp_sales table is assigned a SQL expression. Column names may be referenced in the expression by prefixing the column name with a colon character (:).

The following is the corresponding data file:

```
9101ROGERS    CLERK    790217-DEC-10   1980.0020
9102PETERSON  SALESMAN 769820-DEC-10   2600.0030 2300.00
9103WARREN    SALESMAN 769822-DEC-10   5250.0030 2500.00
9104JONES, JR.MANAGER  783902-APR-09   7975.0020
9105ARNOLDS   CLERK    778213-SEP-10   3750.0010
9106JACKSON   ANALYST  756603-JAN-11   4500.0040
```

Since the records for employees ARNOLDS and JACKSON contain 10 and 40 in columns 47 thru 48, which do not satisfy any of the WHEN clauses, EDB*Loader writes these two records to the discard file, emp_multitbl.dsc, whose content is shown by the following:

```
9105ARNOLDS   CLERK    778213-SEP-10   3750.0010
9106JACKSON   ANALYST  756603-JAN-11   4500.0040
```

The following are the rows loaded into the emp_research and emp_sales tables:

```
SELECT * FROM emp_research;

empno |   ename    |   job   | mgr  |      hiredate      |   sal   | comm | deptno
-------+------------+---------+------+--------------------+---------+------+--------
  9101 | ROGERS     | CLERK   | 7902 | 17-DEC-10 00:00:00 | 1980.00 |      |  20.00
  9104 | JONES, JR. | MANAGER | 7839 | 02-APR-09 00:00:00 | 7975.00 |      |  20.00
(2 rows)

SELECT * FROM emp_sales;

empno |  ename   |   job    | mgr  |      hiredate      |   sal   |  comm   | deptno
-------+----------+----------+------+--------------------+---------+---------+--------
  9102 | PETERSON | SALESMAN | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2950.00 |  30.00
  9103 | WARREN   | SALESMAN | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 3813.00 |  30.00
(2 rows)
```

11.2.4 Invoking EDB*Loader

You must have superuser privileges to run EDB*Loader. Use the following command to invoke EDB*Loader from the command line:

```
edbldr [ -d dbname ] [ -p port ] [ -h host ]
[ USERID={ username/password | username/ | username | / } ]
  CONTROL=control_file
[ DATA=data_file ]
[ BAD=bad_file ]
[ DISCARD=discard_file ]
[ DISCARDMAX=max_discard_recs ]
[ LOG=log_file ]
[ PARFILE=param_file ]
[ DIRECT={ FALSE | TRUE } ]
[ FREEZE={ FALSE | TRUE } ]
[ ERRORS=error_count ]
[ PARALLEL={ FALSE | TRUE } ]
[ ROWS=n ]
[ SKIP=skip_count ]
[ SKIP_INDEX_MAINTENANCE={ FALSE | TRUE } ]
[ edb_resource_group=group_name ]
```

Description

If the `-d` option, the `-p` option, or the `-h` option are omitted, the defaults for the database, port, and host are determined according to the same rules as other Advanced Server utility programs such as `edb-psql`, for example.

Any parameter listed in the preceding syntax diagram except for the `-d` option, `-p` option, `-h` option, and the `PARFILE` parameter may be specified in a *parameter file*. The parameter file is specified on the command line when `edbldr` is invoked using `PARFILE=param_file`. Some parameters may be specified in the `OPTIONS` clause in the control file. See the description of the control file in Section 11.2.3.

The specification of *control_file*, *data_file*, *bad_file*, *discard_file*, *log_file*, and *param_file* may include the full directory path or a relative directory path to the file name. If the file name is specified alone or with a relative directory path, the file is assumed to exist (in the case of *control_file*, *data_file*, or *param_file*), or to be created (in the case of *bad_file*, *discard_file*, or *log_file*) relative to the current working directory from which `edbldr` is invoked.

Note: The control file must exist in the character set encoding of the client where `edbldr` is invoked. If the client is in a different encoding than the database encoding, then the `PGCLIENTENCODING` environment variable must be set on the client to the client's encoding prior to invoking `edbldr`. This must be done to ensure character set conversion is properly done between the client and the database server.

The operating system account used to invoke `edbldr` must have read permission on the directories and files specified by *`control_file`*, *`data_file`*, and *`param_file`*.

The operating system account `enterprisedb` must have write permission on the directories where *`bad_file`*, *`discard_file`*, and *`log_file`* are to be written.

Note: It is suggested that the file names for *`control_file`*, *`data_file`*, *`bad_file`*, *`discard_file`*, and *`log_file`* include extensions of `.ctl`, `.dat`, `.bad`, `.dsc`, and `.log`, respectively. If the provided file name does not contain an extension, EDB*Loader assumes the actual file name includes the appropriate aforementioned extension.

Parameters

`dbname`

Name of the database containing the tables to be loaded.

`port`

Port number on which the database server is accepting connections.

`host`

IP address of the host on which the database server is running.

`USERID={` *`username/password`* `|` *`username/`* `|` *`username`* `| / }`

EDB*Loader connects to the database with *`username`*. *`username`* must be a superuser. *`password`* is the password for *`username`*.

If the `USERID` parameter is omitted, EDB*Loader prompts for *`username`* and *`password`*. If `USERID=`*`username/`* is specified, then EDB*Loader 1) uses the password file specified by environment variable `PGPASSFILE` if `PGPASSFILE` is set, or 2) uses the `.pgpass` password file (`pgpass.conf` on Windows systems) if `PGPASSFILE` is not set. If `USERID=`*`username`* is specified, then EDB*Loader prompts for *`password`*. If `USERID=/` is specified, the connection is attempted using the operating system account as the user name.

Note: The Advanced Server connection environment variables `PGUSER` and `PGPASSWORD` are ignored by EDB*Loader. See the PostgreSQL core documentation for information on the `PGPASSFILE` environment variable and the password file.

`CONTROL=`*`control_file`*

> *`control_file`* specifies the name of the control file containing EDB*Loader directives. If a file extension is not specified, an extension of `.ctl` is assumed. See Section 11.2.3 for a description of the control file.

`DATA=`*`data_file`*

> *`data_file`* specifies the name of the file containing the data to be loaded into the target table. If a file extension is not specified, an extension of `.dat` is assumed. See Section 11.2.3 for a description of the *`data_file`*.
>
> **Note:** Specifying a *`data_file`* on the command line overrides the `INFILE` clause specified in the control file.

`BAD=`*`bad_file`*

> *`bad_file`* specifies the name of a file that receives input data records that cannot be loaded due to errors. See Section 11.2.3 for a description of the *`bad_file`*.
>
> **Note:** Specifying a *`bad_file`* on the command line overrides any `BADFILE` clause specified in the control file.

`DISCARD=`*`discard_file`*

> *`discard_file`* is the name of the file that receives input data records that do not meet any table's selection criteria. See the description of *`discard_file`* in Section 11.2.3.
>
> **Note:** Specifying a *`discard_file`* using the command line `DISCARD` parameter overrides the `DISCARDFILE` clause in the control file.

`DISCARDMAX=`*`max_discard_recs`*

> *`max_discard_recs`* is the maximum number of discarded records that may be encountered from the input data records before terminating the EDB*Loader session. See the description of *`max_discard_recs`* in Section11.2.3.
>
> **Note:** Specifying *`max_discard_recs`* using the command line `DISCARDMAX` parameter overrides the `DISCARDMAX` or `DISCARDS` clause in the control file.

`LOG=`*`log_file`*

> *`log_file`* specifies the name of the file in which EDB*Loader records the results of the EDB*Loader session.

If the `LOG` parameter is omitted, EDB*Loader creates a log file with the name *`control_file_base`*`.log` in the directory from which `edbldr` is invoked. *`control_file_base`* is the base name of the control file used in the EDB*Loader session. The operating system account `enterprisedb` must have write permission on the directory where the log file is to be written.

`PARFILE=`*`param_file`*

`param_file` specifies the name of the file that contains command line parameters for the EDB*Loader session. Any command line parameter listed in this section except for the `-d`, `-p`, and `-h` options, and the `PARFILE` parameter itself, can be specified in *`param_file`* instead of on the command line.

Any parameter given in *`param_file`* overrides the same parameter supplied on the command line before the `PARFILE` option. Any parameter given on the command line that appears after the `PARFILE` option overrides the same parameter given in *`param_file`*.

Note: Unlike other EDB*Loader files, there is no default file name or extension assumed for *`param_file`*, though by Oracle SQL*Loader convention, `.par` is typically used, but not required, as an extension.

`DIRECT= { FALSE | TRUE }`

If `DIRECT` is set to `TRUE` EDB*Loader performs a direct path load instead of a conventional path load. The default value of `DIRECT` is `FALSE`.

See Section 11.2.5 for information on direct path loads.

`FREEZE= { FALSE | TRUE }`

Set `FREEZE` to `TRUE` to indicate that the data should be copied with the rows *frozen*. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wrap-around. For more information about frozen tuples, see the PostgreSQL core documentation at:

http://www.postgresql.org/docs/9.5/static/routine-vacuuming.html

You must specify a data-loading type of `TRUNCATE` in the control file when using the `FREEZE` option. `FREEZE` is not supported for direct loading.

By default, `FREEZE` is `FALSE`.

`ERRORS=`*`error_count`*

> *`error_count`* specifies the number of errors permitted before aborting the EDB*Loader session. The default is `50`.

`PARALLEL= { FALSE | TRUE }`

> Set `PARALLEL` to `TRUE` to indicate that this EDB*Loader session is one of a number of concurrent EDB*Loader sessions participating in a parallel direct path load. The default value of `PARALLEL` is `FALSE`.
>
> When `PARALLEL` is `TRUE`, the `DIRECT` parameter must also be set to `TRUE` . See Section 11.2.6 for more information about parallel direct path loads.

`ROWS=`*`n`*

> *`n`* specifies the number of rows that EDB*Loader will commit before loading the next set of *`n`* rows.

`SKIP=`*`skip_count`*

> Number of records at the beginning of the input data file that should be skipped before loading begins. The default is `0`.

`SKIP_INDEX_MAINTENANCE= { FALSE | TRUE }`

> If set to `TRUE`, index maintenance is not performed as part of a direct path load, and indexes on the loaded table are marked as invalid. The default value of `SKIP_INDEX_MAINTENANCE` is `FALSE`.
>
> Please note: During a parallel direct path load, target table indexes are not updated, and are marked as invalid after the load is complete.
>
> You can use the `REINDEX` command to rebuild an index. For more information about the `REINDEX` command, see the PostgreSQL core documentation available at:
>
> http://www.postgresql.org/docs/9.5/static/sql-reindex.html

`edb_resource_group=`*`group_name`*

> *`group_name`* specifies the name of an EDB Resource Manager resource group to which the EDB*Loader session is to be assigned.
>
> Any default resource group that may have been assigned to the session (for example, a database user running the EDB*Loader session who had been assigned

a default resource group with the ALTER ROLE ... SET edb_resource_group command) is overridden by the resource group given by the edb_resource_group parameter specified on the edbldr command line.

For information about the EDB Resource Manager, see Chapter 5, "EDB Resource Manager" in the EDB Postgres Enterprise Edition Guide, available at:

http://www.enterprisedb.com/docs/en/9.5/eeguide/toc.html

Examples

In the following example EDB*Loader is invoked using a control file named emp.ctl located in the current working directory to load a table in database edb:

```
$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp.ctl
EDB*Loader: Copyright (c) 2007-2015, EnterpriseDB Corporation.

Successfully loaded (4) records
```

In the following example, EDB*Loader prompts for the user name and password since they are omitted from the command line. In addition, the files for the bad file and log file are specified with the BAD and LOG command line parameters.

```
$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb CONTROL=emp.ctl BAD=/tmp/emp.bad
LOG=/tmp/emp.log
Enter the user name : enterprisedb
Enter the password :
EDB*Loader: Copyright (c) 2007-2015, EnterpriseDB Corporation.

Successfully loaded (4) records
```

The following example runs EDB*Loader with the same parameters as shown in the preceding example, but using a parameter file located in the current working directory. The SKIP and ERRORS parameters are altered from their defaults in the parameter file as well. The parameter file, emp.par, contains the following:

```
CONTROL=emp.ctl
BAD=/tmp/emp.bad
LOG=/tmp/emp.log
SKIP=1
ERRORS=10
```

EDB*Loader is invoked with the parameter file as shown by the following:

```
$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb PARFILE=emp.par
Enter the user name : enterprisedb
Enter the password :
EDB*Loader: Copyright (c) 2007-2015, EnterpriseDB Corporation.

Successfully loaded (3) records
```

11.2.4.1 Exit Codes

When EDB*Loader exits, it will return one of the following codes:

Exit Code	Description
`0`	Indicates that all rows loaded successfully.
`1`	Indicates that EDB*Loader encountered command line or syntax errors, or aborted the load operation due to an unrecoverable error.
`2`	Indicates that the load completed, but some (or all) rows were rejected or discarded.
`3`	Indicates that EDB*Loader encountered fatal errors (such as OS errors). This class of errors is equivalent to the `FATAL` or `PANIC` severity levels of PostgreSQL errors.

11.2.5 Direct Path Load

During a direct path load, EDB*Loader writes the data directly to the database pages, which is then synchronized to disk. The insert processing associated with a conventional path load is bypassed, thereby resulting in a performance improvement.

Bypassing insert processing reduces the types of constraints that may exist on the target table. The following types of constraints are permitted on the target table of a direct path load:

- Primary key
- Not null constraints
- Indexes (unique or non-unique)

The restrictions on the target table of a direct path load are the following:

- Triggers are not permitted
- Check constraints are not permitted
- Foreign key constraints on the target table referencing another table are not permitted
- Foreign key constraints on other tables referencing the target table are not permitted
- The table must not be partitioned
- Rules may exist on the target table, but they are not executed

Note: Currently, a direct path load in EDB*Loader is more restrictive than in Oracle SQL*Loader. The preceding restrictions do not apply to Oracle SQL*Loader in most cases. The following restrictions apply to a control file used in a direct path load:

- Multiple table loads are not supported. That is, only one `INTO TABLE` clause may be specified in the control file.
- SQL expressions may not be used in the data field definitions of the `INTO TABLE` clause.
- The `FREEZE` option is not supported for direct path loading.

To run a direct path load, add the `DIRECT=TRUE` option as shown by the following example:

```
$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp.ctl DIRECT=TRUE
EDB*Loader: Copyright (c) 2007-2015, EnterpriseDB Corporation.

Successfully loaded (4) records
```

11.2.6 Parallel Direct Path Load

The performance of a direct path load can be further improved by distributing the loading process over two or more sessions running concurrently. Each session runs a direct path load into the same table.

Since the same table is loaded from multiple sessions, the input records to be loaded into the table must be divided amongst several data files so that each EDB*Loader session uses its own data file and the same record is not loaded more than once into the table.

The target table of a parallel direct path load is under the same restrictions as a direct path load run in a single session.

The restrictions on the target table of a direct path load are the following:

- Triggers are not permitted
- Check constraints are not permitted
- Foreign key constraints on the target table referencing another table are not permitted
- Foreign key constraints on other tables referencing the target table are not permitted
- The table must not be partitioned
- Rules may exist on the target table, but they are not executed

In addition, the `APPEND` clause must be specified in the control file used by each EDB*Loader session.

To run a parallel direct path load, run EDB*Loader in a separate session for each participant of the parallel direct path load. Invocation of each such EDB*Loader session must include the `DIRECT=TRUE` and `PARALLEL=TRUE` parameters.

Each EDB*Loader session runs as an independent transaction so if one of the parallel sessions aborts and rolls back its changes, the loading done by the other parallel sessions are not affected.

Note: In a parallel direct path load, each EDB*Loader session reserves a fixed number of blocks in the target table in a round-robin fashion. Some of the blocks in the last allocated chunk may not be used, and those blocks remain uninitialized. A subsequent use of the `VACUUM` command on the target table may show warnings regarding these uninitialized blocks such as the following:

```
WARNING:  relation "emp" page 98264 is uninitialized --- fixing

WARNING:  relation "emp" page 98265 is uninitialized --- fixing
```

```
WARNING:  relation "emp" page 98266 is uninitialized --- fixing
```

This is an expected behavior and does not indicate data corruption.

Indexes on the target table are not updated during a parallel direct path load and are therefore marked as invalid after the load is complete. You must use the `REINDEX` command to rebuild the indexes.

The following example shows the use of a parallel direct path load on the `emp` table.

Note: If you attempt a parallel direct path load on the sample `emp` table provided with Advanced Server, you must first remove the triggers and constraints referencing the `emp` table. In addition the primary key column, `empno`, was expanded from `NUMBER(4)` to `NUMBER` in this example to allow for the insertion of a larger number of rows.

The following is the control file used in the first session:

```
LOAD DATA
  INFILE    '/home/user/loader/emp_parallel_1.dat'
  APPEND
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
  )
```

The `APPEND` clause must be specified in the control file for a parallel direct path load.

The following shows the invocation of EDB*Loader in the first session. The `DIRECT=TRUE` and `PARALLEL=TRUE` parameters must be specified.

```
$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp_parallel_1.ctl DIRECT=TRUE PARALLEL=TRUE
WARNING:  index maintenance will be skipped with PARALLEL load
EDB*Loader: Copyright (c) 2007-2015, EnterpriseDB Corporation.
```

The control file used for the second session appears as follows. Note that it is the same as the one used in the first session, but uses a different data file.

```
LOAD DATA
  INFILE    '/home/user/loader/emp_parallel_2.dat'
  APPEND
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
  )
```

The preceding control file is used in a second session as shown by the following:

```
$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp_parallel_2.ctl DIRECT=TRUE PARALLEL=TRUE
WARNING:  index maintenance will be skipped with PARALLEL load
EDB*Loader: Copyright (c) 2007-2015, EnterpriseDB Corporation.
```

EDB*Loader displays the following message in each session when its respective load operation completes:

```
Successfully loaded (10000) records
```

The following query shows that the index on the `emp` table has been marked as `INVALID`:

```
SELECT index_name, status FROM user_indexes WHERE table_name = 'EMP';

 index_name | status
------------+---------
 EMP_PK     | INVALID
(1 row)
```

Note: `user_indexes` is the view of indexes compatible with Oracle databases owned by the current user.

Queries on the `emp` table will not utilize the index unless it is rebuilt using the `REINDEX` command as shown by the following:

```
REINDEX INDEX emp_pk;
```

A subsequent query on `user_indexes` shows that the index is now marked as `VALID`:

```
SELECT index_name, status FROM user_indexes WHERE table_name = 'EMP';

 index_name | status
------------+--------
 EMP_PK     | VALID
(1 row)
```

11.2.7 Remote Loading

EDB*Loader supports a feature called *remote loading*. In remote loading, the database containing the table to be loaded is running on a database server on a different host than from where EDB*Loader is invoked with the input data source.

This feature is useful if you have a large amount of data to be loaded, and you do not want to create a large data file on the host running the database server.

In addition, you can use the standard input feature to pipe the data from the data source such as another program or script, directly to EDB*Loader, which then loads the table in the remote database. This bypasses the process of having to create a data file on disk for EDB*Loader.

Performing remote loading along with using standard input requires the following:

- The `edbldr` program must be installed on the client host on which it is to be invoked with the data source for the EDB*Loader session.
- The control file must contain the clause `INFILE 'stdin'` so you can pipe the data directly into EDB*Loader's standard input. See Section 11.2.3 for information on the `INFILE` clause and the EDB*Loader control file.
- All files used by EDB*Loader such as the control file, bad file, discard file, and log file must reside on, or are created on, the client host on which `edbldr` is invoked.
- When invoking EDB*Loader, use the `-h` option to specify the IP address of the remote database server. See Section 11.2.4 for information on invoking EDB*Loader.
- Use the operating system pipe operator (`|`) or input redirection operator (`<`) to supply the input data to EDB*Loader.

The following example loads a database running on a database server at `192.168.1.14` using data piped from a source named `datasource`.

```
datasource | ./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password
CONTROL=remote.ctl
```

The following is another example of how standard input can be used:

```
./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password
CONTROL=remote.ctl < datasource
```

11.2.8 Updating a Table with a Conventional Path Load

You can use EDB*Loader with a conventional path load to update the rows within a table, merging new data with the existing data. When you invoke EDB*Loader to perform an update, the server searches the table for an existing row with a matching primary key:

- If the server locates a row with a matching key, it replaces the existing row with the new row.
- If the server does not locate a row with a matching key, it adds the new row to the table.

To use EDB*Loader to update a table, the table must have a primary key. Please note that you cannot use EDB*Loader to `UPDATE` a partitioned table.

To perform an `UPDATE`, use the same steps as when performing a conventional path load:

1. Create a data file that contains the rows you wish to `UPDATE` or `INSERT`.
2. Define a control file that uses the `INFILE` keyword to specify the name of the data file. For information about building the EDB*Loader control file, see Section 11.2.3
3. Invoke EDB*Loader, specifying the database name, connection information, and the name of the control file. For information about invoking EDB*Loader, see Section 11.2.4.

The following example uses the `emp` table that is distributed with the Advanced Server sample data. By default, the table contains:

```
edb=# select * from emp;
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)
```

The following control file (`emp_update.ctl`) specifies the fields in the table in a comma-delimited list. The control file performs an `UPDATE` on the `emp` table:

```
LOAD DATA
  INFILE 'emp_update.dat'
  BADFILE 'emp_update.bad'
  DISCARDFILE 'emp_update.dsc'
UPDATE INTO TABLE emp
FIELDS TERMINATED BY ","
(empno, ename, job, mgr, hiredate, sal, comm, deptno)
```

The data that is being updated or inserted is saved in the `emp_update.dat` file. `emp_update.dat` contains:

```
7521,WARD,MANAGER,7839,22-FEB-81 00:00:00,3000.00,0.00,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,3500.00,0.00,20
7903,BAKER,SALESMAN,7521,10-JUN-13 00:00:00,1800.00,500.00,20
7904,MILLS,SALESMAN,7839,13-JUN-13 00:00:00,1800.00,500.00,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1500.00,400.00,30
```

Invoke EDB*Loader, specifying the name of the database (`edb`), the name of a database superuser (and their associated password) and the name of the control file (`emp_update.ctl`):

```
edbldr -d edb userid=user_name/password control=emp_update.ctl
```

After performing the update, the `emp` table contains:

```
edb=# select * from emp;
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  |MANAGER   | 7839 | 22-FEB-81 00:00:00 | 3000.00 |0.00    |      30
7566 |JONES |MANAGER   | 7839 | 02-APR-81 00:00:00 | 3500.00 |0.00    |      20
7654 |MARTIN|SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1500.00 |400.00  |      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
7903 |BAKER |SALESMAN |7521   | 10-JUN-13 00:00:00 | 1800.00 |500.00  |      20
7904 |MILLS |SALESMAN |7839   |13-JUN-13 00:00:00  |1800.00  |500.00  |      20
7934 |MILLER|CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |        |      10
(16 rows)
```

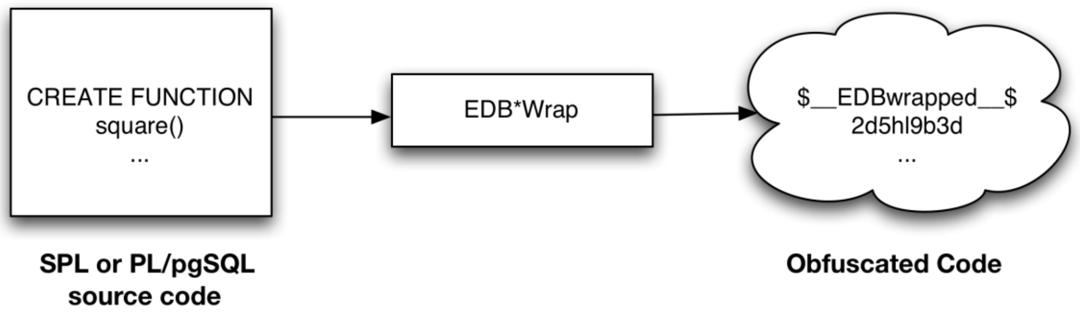
The rows containing information for the three employees that are currently in the `emp` table are updated, while rows are added for the new employees (`BAKER` and `MILLS`)

11.3 EDB*Wrap

The EDB*Wrap utility protects proprietary source code and programs (functions, stored procedures, triggers, and packages) from unauthorized scrutiny. The EDB*Wrap program translates a file that contains SPL or PL/pgSQL source code (the plaintext) into a file that contains the same code in a form that is nearly impossible to read. Once you have the obfuscated form of the code, you can send that code to the PostgreSQL server and the server will store those programs in obfuscated form. While EDB*Wrap does obscure code, table definitions are still exposed.

Everything you wrap is stored in obfuscated form. If you wrap an entire package, the package body source, as well as the prototypes contained in the package header and the functions and procedures contained in the package body are stored in obfuscated form.

If you wrap a `CREATE PACKAGE` statement, you hide the package API from other developers. You may want to wrap the package body, but not the package header so users can see the package prototypes and other public variables that are defined in the package body. To allow users to see what prototypes the package contains, use EDBWrap to obfuscate only the `'CREATE PACKAGE BODY'` statement in the edbwrap input file, omitting the `'CREATE PACKAGE'` statement. The package header source will be stored plaintext, while the package body source and package functions and procedures will be stored obfuscated.



Once wrapped, source code and programs cannot be unwrapped or debugged. Reverse engineering is possible, but would be very difficult.

The entire source file is wrapped into one unit. Any `psql` meta-commands included in the wrapped file will not be recognized when the file is executed; executing an obfuscated file that contains a psql meta-command will cause a syntax error. `edbwrap` does not validate SQL source code - if the plaintext form contains a syntax error, `edbwrap` will not complain. Instead, the server will report an error and abort the entire file when you try to execute the obfuscated form.

11.3.1 Using EDB*Wrap to Obfuscate Source Code

EDB*Wrap is a command line utility; it accepts a single input source file, obfuscates the contents and returns a single output file. When you invoke the `edbwrap` utility, you must provide the name of the file that contains the source code to obfuscate. You may also specify the name of the file where `edbwrap` will write the obfuscated form of the code. `edbwrap` offers three different command-line styles. The first style is compatible with Oracle's `wrap` utility:

```
edbwrap iname=input_file [oname=output_file]
```

The `iname=input_file` argument specifies the name of the input file; if `input_file` does not contain an extension, `edbwrap` will search for a file named *`input_file`*`.sql`

The `oname=`*`output_file`* argument (which is optional) specifies the name of the output file; if *`output_file`* does not contain an extension, `edbwrap` will append `.plb` to the name.

If you do not specify an output file name, `edbwrap` writes to a file whose name is derived from the input file name: `edbwrap` strips the suffix (typically `.sql`) from the input file name and adds `.plb`.

`edbwrap` offers two other command-line styles that may feel more familiar:

```
edbwrap --iname input_file [--oname output_file]
edbwrap -i input_file [-o output_file]
```

You may mix command-line styles; the rules for deriving input and output file names are identical regardless of which style you use.

Once `edbwrap` has produced a file that contains obfuscated code, you typically feed that file into the PostgreSQL server using a client application such as `edb-psql`. The server executes the obfuscated code line by line and stores the source code for SPL and PL/pgSQL programs in wrapped form.

In summary, to obfuscate code with EDB*Wrap, you:

1. Create the source code file.
2. Invoke EDB*Wrap to obfuscate the code.
3. Import the file as if it were in plaintext form.

The following sequence demonstrates `edbwrap` functionality.

First, create the source code for the `list_emp` procedure (in plaintext form):

```
[bash] cat listemp.sql
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;
/
```

You can import the `list_emp` procedure with a client application such as `edb-psql`:

```
[bash] edb-psql edb
Welcome to edb-psql 8.4.3.2, the EnterpriseDB interactive terminal.

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

edb=# \i listemp.sql
CREATE PROCEDURE
```

You can view the plaintext source code (stored in the server) by examining the `pg_proc` system table:

```
edb=# SELECT prosrc FROM pg_proc WHERE proname = 'list_emp';
                            prosrc
--------------------------------------------------------------

     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
(1 row)

edb=# quit
```

Next, obfuscate the plaintext file with EDB*Wrap:

```
[bash] edbwrap -i listemp.sql
EDB*Wrap Utility: Release 8.4.3.2

Copyright (c) 2004-2013 EnterpriseDB Corporation.  All Rights Reserved.

Using encoding UTF8 for input
Processing listemp.sql to listemp.plb

Examining the contents of the output file (listemp.plb) file reveals that the
code is obfuscated:

[bash] cat listemp.plb
$__EDBwrapped__$
UTF8
d+6DL30RVaGjYMIzkuoSzAQgtBw7MhYFuAFkBsfYfhdJ0rjwBv+bHr1FCyH6j9SgH
movU+bYI+jR+hR2jbzq3sovHKEyZIp9y3/GckbQgualRhIlGpyWfE0dltDUpkYRLN
/OUXmk0/P4H6EI98sAHevGDhOWI+58DjJ44qhZ+l5NNEVxbWDztpb/s5sdx4660qQ
Ozx3/gh8VkqS2JbcxYMpjmrwVr6fAXfb68Ml9mW2Hl7fNtxcb5kjSzXvfWR2XYzJf
KFNrEhbL1DTVlSEC5wE6lGlwhYvXOf22m1R2IFns0MtF9fwcnBWAs1YqjR00j6+fc
er/f/efAFh4=
$__EDBwrapped__$
```

You may notice that the second line of the wrapped file contains an encoding name (in this case, the encoding is UTF8). When you obfuscate a file, edbwrap infers the encoding of the input file by examining the locale. For example, if you are running edbwrap while your locale is set to en_US.utf8, edbwrap assumes that the input file is encoded in UTF8. Be sure to examine the output file after running edbwrap; if the locale contained in the wrapped file does not match the encoding of the input file, you should change your locale and rewrap the input file.

You can import the obfuscated code into the PostgreSQL server using the same tools that work with plaintext code:

```
[bash] edb-psql edb
Welcome to edb-psql 8.4.3.2, the EnterpriseDB interactive terminal.

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

edb=# \i listemp.plb
CREATE PROCEDURE

Now, the pg_proc system table contains the obfuscated code:

edb=# SELECT prosrc FROM pg_proc WHERE proname = 'list_emp';
                                    prosrc
----------------------------------------------------------------
 $__EDBwrapped__$
 UTF8
 dw4B9Tz69J3WOsy0GgYJQa+G2sLZ3IOyxS8pDyuOTFuiYe/EXiEatwwG3h3tdJk
 ea+AIp35dS/4idbN8wpegM3s994dQ3R97NgNHfvTQnO2vtd4wQtsQ/Zc4v4Lhfj
 nlV+A4UpHI5oQEnXeAch2LcRD87hkU0uo1ESeQV8IrXaj9BsZr+ueROnwhGs/Ec
 pva/tRV4m9RusFn0wyr38u4Z8w4dfnPW184Y3o6It4b3aH07WxTkWrMLmOZW1jJ
 Nu6u4o+ezO64G9QKPazgehslv4JB9NQnuocActfDSPMY7R7anmgw
 $__EDBwrapped__$
(1 row)
```

Invoke the obfuscated code in the same way that you would invoke the plaintext form:

edb=# 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

EDB-SPL Procedure successfully completed
edb=# quit
```

When you use `pg_dump` to back up a database, wrapped programs remain obfuscated in the archive file.

Be aware that audit logs produced by the Postgres server will show wrapped programs in plaintext form. Source code is also displayed in plaintext in SQL error messages generated during the execution of a program.

Note: At this time, the bodies of the objects created by the following statements will not be stored in obfuscated form:

```
CREATE [OR REPLACE] TYPE type_name AS OBJECT
CREATE [OR REPLACE] TYPE type_name UNDER type_name
CREATE [OR REPLACE] TYPE BODY type_name
```

12 Dynamic Runtime Instrumentation Tools Architecture (DRITA)

The Dynamic Runtime Instrumentation Tools Architecture (DRITA) allows a DBA to query catalog views to determine the *wait events* that affect the performance of individual sessions or the system as a whole. DRITA records the number of times each event occurs as well as the time spent waiting; you can use this information to diagnose performance problems. DRITA offers this functionality, while consuming minimal system resources.

DRITA compares *snapshots* to evaluate the performance of a system. A snapshot is a saved set of system performance data at a given point in time. Each snapshot is identified by a unique ID number; you can use snapshot ID numbers with DRITA reporting functions to return system performance statistics.

12.1 Configuring and Using DRITA

Advanced Server's `postgresql.conf` file includes a configuration parameter named `timed_statistics` that controls the collection of timing data. The valid parameter values are `TRUE` or `FALSE`; the default value is `FALSE`.

This is a dynamic parameter which can be modified in the `postgresql.conf` file, or while a session is in progress. To enable DRITA, you must either:

> Modify the `postgresql.conf` file, setting the `timed_statistics` parameter to `TRUE`.
>
> or
>
> Connect to the server with the EDB-PSQL client, and invoke the command:

```
SET timed_statistics = TRUE
```

After modifying the `timed_statistics` parameter, take a starting snapshot. A snapshot captures the current state of each timer and event counter. The server will compare the starting snapshot to a later snapshot to gauge system performance.

Use the `edbsnap()` function to take the beginning snapshot:

```
edb=# SELECT * FROM edbsnap();
       edbsnap
----------------------
 Statement processed.
(1 row)
```

Then, run the workload that you would like to evaluate; when the workload has completed (or at a strategic point during the workload), take another snapshot:

```
edb=# SELECT * FROM edbsnap();
       edbsnap
----------------------
 Statement processed.
(1 row)
```

You can capture multiple snapshots during a session. Then, use the DRITA functions and reports to manage and compare the snapshots to evaluate performance information.

12.2 DRITA Functions

You can use DRITA functions to gather wait information and manage snapshots. DRITA functions are fully supported by Advanced Server 9.5 whether your installation is made compatible with Oracle databases or is made in PostgreSQL-compatible mode.

12.2.1 get_snaps()

The `get_snaps()` function returns a list of the current snapshots. The signature is:

```
get_snaps()
```

The following example demonstrates using the `get_snaps()` function to display a list of snapshots:

```
edb=# SELECT * FROM get_snaps();
          get_snaps
------------------------------
 1  11-FEB-10 10:41:05.668852
 2  11-FEB-10 10:42:27.26154
 3  11-FEB-10 10:45:48.999992
 4  11-FEB-10 11:01:58.345163
 5  11-FEB-10 11:05:14.092683
 6  11-FEB-10 11:06:33.151002
 7  11-FEB-10 11:11:16.405664
 8  11-FEB-10 11:13:29.458405
 9  11-FEB-10 11:23:57.595916
 10 11-FEB-10 11:29:02.214014
 11 11-FEB-10 11:31:44.244038
(11 rows)
```

The first column in the result list displays the snapshot identifier; the second column displays the date and time that the snapshot was captured.

12.2.2 sys_rpt()

The `sys_rpt()` function returns system wait information. The signature is:

sys_rpt(*beginning_id*, *ending_id*, *top_n*)

Parameters

beginning_id

beginning_id is an integer value that represents the beginning session identifier.

ending_id

ending_id is an integer value that represents the ending session identifier.

top_n

top_n represents the number of rows to return

This example demonstrates a call to the `sys_rpt()` function:

```
edb=# SELECT * FROM sys_rpt(9, 10, 10);
                                   sys_rpt
-----------------------------------------------------------------------------
 WAIT NAME                                 COUNT      WAIT TIME        % WAIT
 ---------------------------------------------------------------------------
 wal write                                 21250      104.723772       36.31
 db file read                              121407     72.143274        25.01
 wal flush                                 84185      51.652495        17.91
 wal file sync                             712        29.482206        10.22
 infinitecache write                       84178      15.814444        5.48
 db file write                             84177      14.447718        5.01
 infinitecache read                        672        0.098691         0.03
 db file extend                            190        0.040386         0.01
 query plan                                52         0.024400         0.01
 wal insert lock acquire                   4          0.000837         0.00
(12 rows)
```

The information displayed in the result set includes:

Column Name	Description
`WAIT NAME`	The name of the wait.
`COUNT`	The number of times that the wait event occurred.
`WAIT TIME`	The time of the wait event in milliseconds.
`% WAIT`	The percentage of the total wait time used by this wait for this session.

12.2.3 sess_rpt()

The `sess_rpt()` function returns session wait information. The signature is:

```
sess_rpt(beginning_id, ending_id, top_n)
```

Parameters

beginning_id

beginning_id is an integer value that represents the beginning session identifier.

ending_id

ending_id is an integer value that represents the ending session identifier.

top_n

top_n represents the number of rows to return

The following example demonstrates a call to the `sess_rpt()` function:

```
SELECT * FROM sess_rpt(18, 19, 10);

                              sess_rpt
------------------------------------------------------------------------------
ID    USER        WAIT NAME              COUNT TIME(ms)   %WAIT SES   %WAIT ALL
 -----------------------------------------------------------------------------

 17373 enterprise db file read             30   0.175713   85.24       85.24
 17373 enterprise query plan               18   0.014930   7.24        7.24
 17373 enterprise wal flush                6    0.004067   1.97        1.97
 17373 enterprise wal write                1    0.004063   1.97        1.97
 17373 enterprise wal file sync            1    0.003664   1.78        1.78
 17373 enterprise infinitecache read       38   0.003076   1.49        1.49
 17373 enterprise infinitecache write      5    0.000548   0.27        0.27
 17373 enterprise db file extend           190  0.04.386   0.03        0.03
 17373 enterprise db file write            5    0.000082   0.04        0.04
 (11 rows)
```

The information displayed in the result set includes:

Column Name	Description
`ID`	The processID of the session.
`USER`	The name of the user incurring the wait.
`WAIT NAME`	The name of the wait event.
`COUNT`	The number of times that the wait event occurred.
`TIME (ms)`	The length of the wait event in milliseconds.
`% WAIT SES`	The percentage of the total wait time used by this wait for this session.
`% WAIT ALL`	The percentage of the total wait time used by this wait (for all sessions).

12.2.4 sessid_rpt()

The `sessid_rpt()` function returns session ID information for a specified backend. The signature is:

`sessid_rpt(`*beginning_id*`, `*ending_id*`, `*backend_id*`)`

Parameters

beginning_id

beginning_id is an integer value that represents the beginning session identifier.

ending_id

ending_id is an integer value that represents the ending session identifier.

backend_id

backend_id is an integer value that represents the backend identifier.

The following code sample demonstrates a call to `sessid_rpt()`:

```
SELECT * FROM sessid_rpt(18, 19, 17373);

                                sessid_rpt
---------------------------------------------------------------------------------
 ID    USER       WAIT NAME             COUNT TIME(ms)  %WAIT SES   %WAIT ALL
 --------------------------------------------------------------------------------
 17373 enterprise db file read            30   0.175713  85.24       85.24
 17373 enterprise query plan              18   0.014930  7.24        7.24
 17373 enterprise wal flush               6    0.004067  1.97        1.97
 17373 enterprise wal write               1    0.004063  1.97        1.97
 17373 enterprise wal file sync           1    0.003664  1.78        1.78
 17373 enterprise infinitecache read      38   0.003076  1.49        1.49
 17373 enterprise infinitecache write     5    0.000548  0.27        0.27
 17373 enterprise db file extend          190  0.040386  0.03        0.03
 17373 enterprise db file write           5    0.000082  0.04        0.04
(11 rows)
```

The information displayed in the result set includes:

Column Name	Description
`ID`	The process ID of the wait.
`USER`	The name of the user that owns the session.
`WAIT NAME`	The name of the wait event.
`COUNT`	The number of times that the wait event occurred.
`TIME (ms)`	The length of the wait in milliseconds.
`% WAIT SES`	The percentage of the total wait time used by this wait for this session.
`% WAIT ALL`	The percentage of the total wait time used by this wait (for all sessions).

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

12.2.5 sesshist_rpt()

The `sesshist_rpt()` function returns session wait information for a specified backend. The signature is:

```
sesshist_rpt(snapshot_id, session_id)
```

Parameters

snapshot_id

snapshot_id is an integer value that identifies the snapshot.

session_id

session_id is an integer value that represents the session.

The following example demonstrates a call to the `sesshist_rpt()` function:

```
edb=# SELECT * FROM sesshist_rpt (9, 5531);
                              sesshist_rpt
------------------------------------------------------------------------------
 ID    USER       SEQ  WAIT NAME
   ELAPSED(ms)   File  Name                    # of Blk   Sum of Blks
 ------------------------------------------------------------------------------
 5531 enterprise 1     db file read
   18546        14309  session_waits_pk        1          1
 5531 enterprise 2     infinitecache read
   125          14309  session_waits_pk        1          1
 5531 enterprise 3     db file read
   376          14304  edb$session_waits       0          1
 5531 enterprise 4     infinitecache read
   166          14304  edb$session_waits       0          1
 5531 enterprise 5     db file read
   7978         1260   pg_authid               0          1
 5531 enterprise 6     infinitecache read
   154          1260   pg_authid               0          1
 5531 enterprise 7     db file read
   628          14302  system_waits_pk         1          1
 5531 enterprise 8     infinitecache read
   463          14302  system_waits_pk         1          1
 5531 enterprise 9     db file read
   3446         14297  edb$system_waits        0          1
 5531 enterprise 10    infinitecache read
   187          14297  edb$system_waits        0          1
 5531 enterprise 11    db file read
   14750        14295  snap_pk                 1          1
 5531 enterprise 12    infinitecache read
   416          14295  snap_pk                 1          1
 5531 enterprise 13    db file read
   7139         14290  edb$snap                0          1
 5531 enterprise 14    infinitecache read
   158          14290  edb$snap                0          1
 5531 enterprise 15    db file read
```

```
    27287        14288  snapshot_num_seq     0           1
  5531 enterprise 16    infinitecache read
 (17 rows)
```

The information displayed in the result set includes:

Column Name	Description
`ID`	The system-assigned identifier of the wait.
`USER`	The name of the user that incurred the wait.
`SEQ`	The sequence number of the wait event.
`WAIT NAME`	The name of the wait event.
`ELAPSED (ms)`	The length of the wait event in milliseconds.
`File`	The relfilenode number of the file.
`Name`	If available, the name of the file name related to the wait event.
`# of Blk`	The block number read or written for a specific instance of the event .
`Sum of Blks`	The number of blocks read.

12.2.6 purgesnap()

The `purgesnap()` function purges a range of snapshots from the snapshot tables. The signature is:

`purgesnap(`*`beginning_id`*`, `*`ending_id`*`)`

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier .

`purgesnap()` removes all snapshots between *`beginning_id`* and *`ending_id`* (inclusive):

```
SELECT * FROM purgesnap(6, 9);

             purgesnap
------------------------------------
  Snapshots in range 6 to 9 deleted.
 (1 row)
```

A call to the `get_snaps()` function after executing the example shows that snapshots `6` through `9` have been purged from the snapshot tables:

```
edb=# SELECT * FROM get_snaps();
          get_snaps
------------------------------
 1  11-FEB-10 10:41:05.668852
 2  11-FEB-10 10:42:27.26154
 3  11-FEB-10 10:45:48.999992
 4  11-FEB-10 11:01:58.345163
 5  11-FEB-10 11:05:14.092683
 10 11-FEB-10 11:29:02.214014
 11 11-FEB-10 11:31:44.244038
(7 rows)
```

12.2.7 truncsnap()

Use the `truncsnap()` function to delete all records from the snapshot table. The signature is:

```
truncsnap()
```

For example:

```
SELECT * FROM truncsnap();

      truncsnap
----------------------
 Snapshots truncated.
(1 row)
```

A call to the `get_snaps()` function after calling the `truncsnap()` function shows that all records have been removed from the snapshot tables:

```
SELECT * FROM get_snaps();
 get_snaps
-----------
(0 rows)
```

12.3 Simulating Statspack AWR Reports

The functions described in this section return information comparable to the information contained in an Oracle Statspack/AWR (Automatic Workload Repository) report. When taking a snapshot, performance data from system catalog tables is saved into history tables. The reporting functions listed below report on the differences between two given snapshots.

- `stat_db_rpt()`
- `stat_tables_rpt()`
- `statio_tables_rpt()`
- `stat_indexes_rpt()`
- `statio_indexes_rpt()`

The reporting functions can be executed individually or you can execute all five functions by calling the `edbreport()` function.

12.3.1 edbreport()

The `edbreport()` function includes data from the other reporting functions, plus additional system information. The signature is:

```
edbreport(beginning_id, ending_id)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

The call to the `edbreport()` function returns a composite report that contains system information and the reports returned by the other statspack functions. :

```
edb=# SELECT * FROM edbreport(9, 10);

edbreport
-----------------------------------------------------------------------------
    EnterpriseDB Report for database edb        23-AUG-15
 Version: EnterpriseDB 9.5.0.0 on i686-pc-linux-gnu
      Begin snapshot: 9 at 23-AUG-15 13:45:07.165123
      End snapshot:   10 at 23-AUG-15 13:45:35.653036
```

```
Size of database edb is 155 MB
     Tablespace: pg_default Size: 179 MB Owner: enterprisedb
     Tablespace: pg_global  Size: 435 kB Owner: enterprisedb

Schema: pg_toast_temp_1          Size: 0 bytes    Owner: enterprisedb
Schema: public                   Size: 0 bytes    Owner: enterprisedb
Schema: enterprisedb             Size: 143 MB     Owner: enterprisedb
Schema: pgagent                  Size: 192 kB     Owner: enterprisedb
Schema: dbms_job_procedure       Size: 0 bytes    Owner: enterprisedb
```

The information displayed in the report introduction includes the database name and version, the current date, the beginning and ending snapshot date and times, database and tablespace details and schema information.

```
               Top 10 Relations by pages

TABLE                                            RELPAGES
----------------------------------------------------------------------------
pgbench_accounts                                  15874
pg_proc                                           102
edb$statio_all_indexes                            73
edb$stat_all_indexes                              73
pg_attribute                                      67
pg_depend                                         58
edb$statio_all_tables                             49
edb$stat_all_tables                               47
pgbench_tellers                                   37
pg_description                                    32
```

The information displayed in the `Top 10 Relations by pages` section includes:

Column Name	Description
`TABLE`	The name of the table.
`RELPAGES`	The number of pages in the table.

```
               Top 10 Indexes by pages

INDEX                                            RELPAGES
----------------------------------------------------------------------------
pgbench_accounts_pkey                             2198
pg_depend_depender_index                          32
pg_depend_reference_index                         31
pg_proc_proname_args_nsp_index                    30
pg_attribute_relid_attnam_index                   23
pg_attribute_relid_attnum_index                   17
pg_description_o_c_o_index                        15
edb$statio_idx_pk                                 11
edb$stat_idx_pk                                   11
pg_proc_oid_index                                 9
```

The information displayed in the `Top 10 Indexes by pages` section includes:

Column Name	Description
INDEX	The name of the index.
RELPAGES	The number of pages in the index.

```
                Top 10 Relations by DML

 SCHEMA          RELATION                       UPDATES    DELETES    INSERTS
 ---------------------------------------------------------------------------
 enterprisedb    pgbench_accounts               10400      0          1000000
 enterprisedb    pgbench_tellers                10400      0          100
 enterprisedb    pgbench_branches               10400      0          10
 enterprisedb    pgbench_history                0          0          10400
 pgagent         pga_jobclass                   0          0          6
 pgagent         pga_exception                  0          0          0
 pgagent         pga_job                        0          0          0
 pgagent         pga_jobagent                   0          0          0
 pgagent         pga_joblog                     0          0          0
 pgagent         pga_jobstep                    0          0          0
```

The information displayed in the `Top 10 Relations by DML` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
UPDATES	The number of UPDATES performed on the table.
DELETES	The number of DELETES performed on the table.
INSERTS	The number of INSERTS performed on the table.

```
   DATA from pg_stat_database

 DATABASE   NUMBACKENDS  XACT COMMIT  XACT ROLLBACK   BLKS READ  BLKS HIT
BLKS ICACHE HIT       HIT RATIO   ICACHE HIT RATIO
 ---------------------------------------------------------------------------
 edb         0            142          0               78         10446
    0                  99.26      0.00

   DATA from pg_buffercache not included because pg_buffercache is not
installed
```

The information displayed in the `DATA from pg_stat_database` section of the report includes:

Column Name	Description
DATABASE	The name of the database.
NUMBACKENDS	Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset.
XACT COMMIT	Number of transactions in this database that have been committed.
XACT ROLLBACK	Number of transactions in this database that have been rolled back.
BLKS READ	Number of disk blocks read in this database.
BLKS HIT	Number of times disk blocks were found already in the buffer cache

Column Name	Description
	(when a read was not necessary).
BLKS ICACHE HIT	The number of blocks found in Infinite Cache.
HIT RATIO	The percentage of times that a block was found in the shared buffer cache.
ICACHE HIT RATIO	The percentage of times that a block was found in Infinite Cache.

```
   DATA from pg_stat_all_tables ordered by seq scan

 SCHEMA                 RELATION                        SEQ SCAN   REL TUP READ
IDX SCAN   IDX TUP READ INS    UPD    DEL
 -----------------------------------------------------------------------------
 pg_catalog             pg_class                        16         7162
546        319            0      1      0
 pg_catalog             pg_am                           13         13
0          0              0      0      0
 pg_catalog             pg_database                     4          16
42         42             0      0      0
 pg_catalog             pg_index                        4          660
145        149            0      0      0
 pg_catalog             pg_namespace                    4          100
49         49             0      0      0
 sys                    edb$snap                        1          9
0          0              1      0      0
 pg_catalog             pg_authid                       1          1
25         25             0      0      0
 sys                    edb$session_wait_history        0          0
0          0              50     0      0
 sys                    edb$session_waits               0          0
0          0              2      0      0
 sys                    edb$stat_all_indexes            0          0
0          0              165    0      0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by seq scan` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
SEQ SCAN	The number of sequential scans initiated on this table..
REL TUP READ	The number of tuples read in the table.
IDX SCAN	The number of index scans initiated on the table.
IDX TUP READ	The number of index tuples read.
INS	The number of rows inserted.
UPD	The number of rows updated.
DEL	The number of rows deleted.

```
   DATA from pg_stat_all_tables ordered by rel tup read

 SCHEMA                 RELATION                        SEQ SCAN   REL TUP READ
IDX SCAN   IDX TUP READ INS    UPD    DEL
 -----------------------------------------------------------------------------
 pg_catalog             pg_class                        16         7162
546        319            0      1      0
```

```
 pg_catalog           pg_index                     4          660
145         149         0      0      0
 pg_catalog           pg_namespace                 4          100
49          49          0      0      0
 pg_catalog           pg_database                  4          16
42          42          0      0      0
 pg_catalog           pg_am                        13         13
0           0           0      0      0
 sys                  edb$snap                     1          9
0           0           1      0      0
 pg_catalog           pg_authid                    1          1
25          25          0      0      0
 sys                  edb$session_wait_history     0          0
0           0           50     0      0
 sys                  edb$session_waits            0          0
0           0           2      0      0
 sys                  edb$stat_all_indexes         0          0
0           0           165    0      0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by rel tup read` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`SEQ SCAN`	The number of sequential scans performed on the table.
`REL TUP READ`	The number of tuples read from the table.
`IDX SCAN`	The number of index scans performed on the table.
`IDX TUP READ`	The number of index tuples read.
`INS`	The number of rows inserted.
`UPD`	The number of rows updated.
`DEL`	The number of rows deleted.

```
   DATA from pg_statio_all_tables

 SCHEMA      RELATION             HEAP     HEAP     HEAP     IDX      IDX
                                  READ     HIT      ICACHE   READ     HIT
                                                    HIT

             IDX      TOAST    TOAST    TOAST    TIDX     TIDX     TIDX
             ICACHE   READ     HIT      ICACHE   READ     HIT      ICACHE
             HIT                        HIT                        HIT
-----------------------------------------------------------------------------
 public      pgbench_accounts     92766    67215    288      59       32126
             9        0        0        0        0        0        0
 pg_catalog  pg_class             0        296      0        3        16
             0        0        0        0        0        0        0
 sys         edb$stat_all_indexes 8        125      0        4        233
             0        0        0        0        0        0        0
 sys         edb$statio_all_index 8        125      0        4        233
             0        0        0        0        0        0        0
 sys         edb$stat_all_tables  6        91       0        2        174
             0        0        0        0        0        0        0
 sys         edb$statio_all_table 6        91       0        2        174
             0        0        0        0        0        0        0
 pg_catalog  pg_namespace         3        72       0        0        0
             0        0        0        0        0        0        0
 sys         edb$session_wait_his 1        24       0        4        47
```

```
             0          0          0          0          0          0          0
 pg_catalog  pg_opclass               3          13          0          2          0
             0          0          0          0          0          0          0
 pg_catalog  pg_trigger               0          12          0          1          15
             0          0          0          0          0          0          0
```

The information displayed in the `Data from pg_statio_all_tables` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`HEAP READ`	The number of heap blocks read.
`HEAP HIT`	The number of heap blocks hit.
`HEAP ICACHE HIT`	The number of heap blocks in Infinite Cache.
`IDX READ`	The number of index blocks read.
`IDX HIT`	The number of index blocks hit.
`IDX ICACHE HIT`	The number of index blocks in Infinite Cache.
`TOAST READ`	The number of toast blocks read.
`TOAST HIT`	The number of toast blocks hit.
`TOAST ICACHE HIT`	The number of toast blocks in Infinite Cache.
`TIDX READ`	The number of toast index blocks read.
`TIDX HIT`	The number of toast index blocks hit.
`TIDX ICACHE HIT`	The number of toast index blocks in Infinite Cache.

```
   DATA from pg_stat_all_indexes

 SCHEMA                 RELATION                   INDEX
IDX SCAN   IDX TUP READ IDX TUP FETCH
 -----------------------------------------------------------------------------
 pg_catalog             pg_attribute
pg_attribute_relid_attnum_index     427        907          907
 pg_catalog             pg_class                   pg_class_relname_nsp_index
289        62             62
 pg_catalog             pg_class                   pg_class_oid_index
257        257            257
 pg_catalog             pg_statistic
pg_statistic_relid_att_inh_index    207        196          196
 enterprisedb           pgbench_accounts           pgbench_accounts_pkey
200        255            200
 pg_catalog             pg_cast                    pg_cast_source_target_index
199        50             50
 pg_catalog             pg_proc                    pg_proc_oid_index
116        116            116
 pg_catalog             edb_partition              edb_partition_partrelid_index
112        0              0
 pg_catalog             edb_policy                 edb_policy_object_name_index
112        0              0
 enterprisedb           pgbench_branches           pgbench_branches_pkey
101        110            0
```

The information displayed in the `DATA from pg_stat_all_indexes` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the index resides.
RELATION	The name of the table on which the index is defined.
INDEX	The name of the index.
IDX SCAN	The number of indexes scans initiated on this index.
IDX TUP READ	Number of index entries returned by scans on this index
IDX TUP FETCH	Number of live table rows fetched by simple index scans using this index.

```
   DATA from pg_statio_all_indexes

 SCHEMA                    RELATION                  INDEX
IDX BLKS READ   IDX BLKS HIT    IDX BLKS ICACHE HIT
 -----------------------------------------------------------------------------
 pg_catalog                pg_attribute
pg_attribute_relid_attnum_index     0               867             0
 enterprisedb              pgbench_accounts          pgbench_accounts_pkey
1               778             0
 pg_catalog                pg_class                  pg_class_relname_nsp_index
0               590             0
 pg_catalog                pg_class                  pg_class_oid_index
0               527             0
 pg_catalog                pg_statistic
pg_statistic_relid_att_inh_index    0               441             0
 sys                       edb$stat_all_indexes      edb$stat_idx_pk
1               332             0
 sys                       edb$statio_all_indexes    edb$statio_idx_pk
1               332             0
 pg_catalog                pg_proc                   pg_proc_oid_index
0               244             0
 sys                       edb$stat_all_tables       edb$stat_tab_pk
0               241             0
 sys                       edb$statio_all_tables     edb$statio_tab_pk
0               241             0
```

The information displayed in the `DATA from pg_statio_all_indexes` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the index resides.
RELATION	The name of the table on which the index is defined.
INDEX	The name of the index.
IDX BLKS READ	The number of index blocks read.
IDX BLKS HIT	The number of index blocks hit.
IDX BLKS ICACHE HIT	The number of index blocks in Infinite Cache that were hit.

```
   System Wait Information

 WAIT NAME                                 COUNT      WAIT TIME       % WAIT
 ---------------------------------------------------------------------------
 query plan                                0          0.000407        100.00
 db file read                              0          0.000000        0.00
```

The information displayed in the `System Wait Information` section includes:

Column Name	Description
WAIT NAME	The name of the wait.
COUNT	The number of times that the wait event occurred.
WAIT TIME	The length of the wait time in milliseconds.
% WAIT	The percentage of the total wait time used by this wait for this session.

```
    Database Parameters from postgresql.conf

 PARAMETER                         SETTING
CONTEXT     MINVAL       MAXVAL
 ----------------------------------------------------------------------------
 allow_system_table_mods            off
postmaster
 application_name                   psql
user
 archive_command                    (disabled)
sighup
 archive_mode                       off
postmaster
 archive_timeout                    0
sighup      0            2147483647
 array_nulls                        on
user
 authentication_timeout             60
sighup      1            600
 autovacuum                         on
sighup
 autovacuum_analyze_scale_factor    0.1
sighup      0            100
 autovacuum_analyze_threshold       50
sighup      0            2147483647
 autovacuum_freeze_max_age          200000000
postmaster  100000000    2000000000
 autovacuum_max_workers             3
postmaster  1            8388607
 autovacuum_naptime                 60
sighup      1            2147483
 autovacuum_vacuum_cost_delay       20
...
```

The information displayed in the `Database Parameters from postgresql.conf` section includes:

Column Name	Description
PARAMETER	The name of the parameter.
SETTING	The current value assigned to the parameter.
CONTEXT	The context required to set the parameter value.
MINVAL	The minimum value allowed for the parameter.
MAXVAL	The maximum value allowed for the parameter.

12.3.2 stat_db_rpt()

The signature is:

```
stat_db_rpt(beginning_id, ending_id)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

The following example demonstrates the `stat_db_rpt()` function:

```
SELECT * FROM stat_db_rpt(9, 10);
                               stat_db_rpt
-----------------------------------------------------------------------------
   DATA from pg_stat_database

 DATABASE   NUMBACKENDS  XACT COMMIT  XACT ROLLBACK   BLKS READ  BLKS HIT
        BLKS ICACHE HIT      HIT RATIO      ICACHE HIT RATIO
-----------------------------------------------------------------------------
 edb          1            21           0               92928      101217
        301                    52.05          0.15
```

The information displayed in the `DATA from pg_stat_database` section of the report includes:

Column Name	Description
`DATABASE`	The name of the database.
`NUMBACKENDS`	Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset.
`XACT COMMIT`	The number of transactions in this database that have been committed.
`XACT ROLLBACK`	The number of transactions in this database that have been rolled back.
`BLKS READ`	The number of blocks read.
`BLKS HIT`	The number of blocks hit.
`BLKS ICACHE HIT`	The number of blocks in Infinite Cache that were hit.
`HIT RATIO`	The percentage of times that a block was found in the shared buffer cache.
`ICACHE HIT RATIO`	The percentage of times that a block was found in Infinite Cache.

12.3.3 stat_tables_rpt()

The signature is:

```
function_name(beginning_id, ending_id, top_n, scope)
```

Parameters

`beginning_id`

> `beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

> `ending_id` is an integer value that represents the ending session identifier.

`top_n`

> `top_n` represents the number of rows to return

`scope`

> `scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:
>
> - `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.
> - `USER` indicates that the function should return information about user-defined tables.
> - `ALL` specifies that the function should return information about all tables.

The `stat_tables_rpt()` function returns a two-part report. The first portion of the report contains:

```
SELECT * FROM stat_tables_rpt(18, 19, 10, 'ALL');

stat_tables_rpt
------------------------------------------------------------------------------
DATA from pg_stat_all_tables ordered by seq scan

SCHEMA         RELATION
    SEQ SCAN    REL TUP READ IDX SCAN   IDX TUP READ   INS    UPD    DEL
------------------------------------------------------------------------------
pg_catalog     pg_class
    8           2952         78         65             0      0      0
pg_catalog     pg_index
    4           448          23         28             0      0      0
pg_catalog     pg_namespace
    4           76           1          1              0      0      0
pg_catalog     pg_database
    3           6            0          0              0      0      0
pg_catalog     pg_authid
    2           1            0          0              0      0      0
sys            edb$snap
    1           15           0          0              1      0      0
public         accounts
    0           0            0          0              0      0      0
public         branches
    0           0            0          0              0      0      0
sys            edb$session_wait_history
    0           0            0          0              25     0      0
sys            edb$session_waits
    0           0            0          0              10     0      0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by seq scan` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`SEQ SCAN`	The number of sequential scans on the table.
`REL TUP READ`	The number of tuples read from the table.
`IDX SCAN`	The number of index scans performed on the table.
`IDX TUP READ`	The number of index tuples read from the table.
`INS`	The number of rows inserted.
`UPD`	The number of rows updated.
`DEL`	The number of rows deleted.

The second portion of the report contains:

```
DATA from pg_stat_all_tables ordered by rel tup read

SCHEMA        RELATION
    SEQ SCAN   REL TUP READ IDX SCAN   IDX TUP READ INS    UPD    DEL
-----------------------------------------------------------------------------
pg_catalog    pg_class
    8          2952         78         65           0      0      0
pg_catalog    pg_index
    4          448          23         28           0      0      0
pg_catalog    pg_namespace
    4          76           1          1            0      0      0
sys           edb$snap
    1          15           0          0            1      0      0
pg_catalog    pg_database
    3          6            0          0            0      0      0
pg_catalog    pg_authid
    2          1            0          0            0      0      0
public        accounts
    0          0            0          0            0      0      0
public        branches
    0          0            0          0            0      0      0
sys           edb$session_wait_history
    0          0            0          0            25     0      0
sys           edb$session_waits
    0          0            0          0            10     0      0
(29 rows)
```

The information displayed in the `DATA from pg_stat_all_tables ordered by rel tup read` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`SEQ SCAN`	The number of sequential scans performed on the table.
`REL TUP READ`	The number of tuples read from the table.
`IDX SCAN`	The number of index scans performed on the table.
`IDX TUP READ`	The number of live rows fetched by index scans.
`INS`	The number of rows inserted.
`UPD`	The number of rows updated.
`DEL`	The number of rows deleted.

12.3.4 statio_tables_rpt()

The signature is:

```
statio_tables_rpt(beginning_id, ending_id, top_n, scope)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.
- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should return information about all tables.

The `statio_tables_rpt()` function returns a report that contains:

```
edb=# SELECT * FROM statio_tables_rpt(9, 10, 10, 'SYS');

                               statio_tables_rpt
-----------------------------------------------------------------------------
   DATA from pg_statio_all_tables

 SCHEMA      RELATION                 HEAP     HEAP     HEAP     IDX      IDX
                                      READ     HIT      ICACHE   READ     HIT
                                                        HIT

             IDX      TOAST    TOAST    TOAST    TIDX     TIDX     TIDX
             ICACHE   READ     HIT      ICACHE   READ     HIT      ICACHE
             HIT                      HIT                        HIT
-----------------------------------------------------------------------------
 public      pgbench_accounts    92766    67215    288      59       32126
             9        0        0        0        0        0        0
 pg_catalog  pg_class            0        296      0        3        16
             0        0        0        0        0        0        0
 sys         edb$stat_all_indexes 8       125      0        4        233
             0        0        0        0        0        0        0
 sys         edb$statio_all_index 8       125      0        4        233
             0        0        0        0        0        0        0
 sys         edb$stat_all_tables  6       91       0        2        174
             0        0        0        0        0        0        0
 sys         edb$statio_all_table 6       91       0        2        174
             0        0        0        0        0        0        0
 pg_catalog  pg_namespace        3        72       0        0        0
             0        0        0        0        0        0        0
 sys         edb$session_wait_his 1       24       0        4        47
             0        0        0        0        0        0        0
 pg_catalog  pg_opclass          3        13       0        2        0
             0        0        0        0        0        0        0
 pg_catalog  pg_trigger          0        12       0        1        15
             0        0        0        0        0        0        0
(16 rows)
```

The information displayed in the `Data from pg_statio_all_tables` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`RELATION`	The name of the relation.
`HEAP READ`	The number of heap blocks read.
`HEAP HIT`	The number of heap blocks hit.
`HEAP ICACHE HIT`	The number of heap blocks in Infinite Cache.
`IDX READ`	The number of index blocks read.
`IDX HIT`	The number of index blocks hit.
`IDX ICACHE HIT`	The number of index blocks in Infinite Cache.
`TOAST READ`	The number of toast blocks read.
`TOAST HIT`	The number of toast blocks hit.
`TOAST ICACHE HIT`	The number of toast blocks in Infinite Cache.
`TIDX READ`	The number of toast index blocks read.
`TIDX HIT`	The number of toast index blocks hit.
`TIDX ICACHE HIT`	The number of toast index blocks in Infinite Cache.

12.3.5 stat_indexes_rpt()

The signature is:

```
stat_indexes_rpt(beginning_id, ending_id, top_n, scope)
```

Parameters

`beginning_id`

> `beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

> `ending_id` is an integer value that represents the ending session identifier.

`top_n`

> `top_n` represents the number of rows to return

`scope`

> `scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:
>
> - `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.
> - `USER` indicates that the function should return information about user-defined tables.
> - `ALL` specifies that the function should return information about all tables.

The `stat_indexes_rpt()` function returns a report that contains:

```
edb=# SELECT * FROM stat_indexes_rpt(9, 10, 10, 'ALL');

                            stat_indexes_rpt
-----------------------------------------------------------------------------
   DATA from pg_stat_all_indexes

 SCHEMA        RELATION        INDEX
                          IDX SCAN    IDX TUP READ    IDX TUP FETCH
-----------------------------------------------------------------------------
 pg_catalog    pg_cast         pg_cast_source_target_index
                          30          7               7
 pg_catalog    pg_class        pg_class_oid_index
                          15          15              15
 pg_catalog    pg_trigger      pg_trigger_tgrelid_tgname_index
                          12          12              12
 pg_catalog    pg_attribute    pg_attribute_relid_attnum_index
                          7           31              31
 pg_catalog    pg_statistic    pg_statistic_relid_att_index
                          7           0               0
 pg_catalog    pg_database     pg_database_oid_index
                          5           5               5
 pg_catalog    pg_proc         pg_proc_oid_index
                          5           5               5
 pg_catalog    pg_operator     pg_operator_oprname_l_r_n_index
                          3           1               1
 pg_catalog    pg_type         pg_type_typname_nsp_index
                          3           1               1
 pg_catalog    pg_amop         pg_amop_opr_fam_index
                          2           3               3
(14 rows)
```

The information displayed in the `DATA from pg_stat_all_indexes` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`RELATION`	The name of the relation.
`INDEX`	The name of the index.
`IDX SCAN`	The number of indexes scanned.
`IDX TUP READ`	The number of index tuples read.
`IDX TUP FETCH`	The number of index tuples fetched.

12.3.6 statio_indexes_rpt()

The signature is:

```
statio_indexes_rpt(beginning_id, ending_id, top_n, scope)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.
- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should return information about all tables.

The `statio_indexes_rpt()` function returns a report that contains:

```
edb=# SELECT * FROM statio_indexes_rpt(9, 10, 10, 'SYS');

                            statio_indexes_rpt
-----------------------------------------------------------------------------
   DATA from pg_statio_all_indexes

 SCHEMA      RELATION         INDEX
                        IDX BLKS READ   IDX BLKS HIT    IDX BLKS ICACHE HIT
-----------------------------------------------------------------------------
public                pgbench_accounts           pgbench_accounts_pkey
                        59              32126           9
 sys                  edb$stat_all_indexes      edb$stat_idx_pk
                        4              233             0
 sys                  edb$statio_all_indexes    edb$statio_idx_pk
                        4              233             0
 sys                  edb$stat_all_tables       edb$stat_tab_pk
                        2              174             0
 sys                  edb$statio_all_tables     edb$statio_tab_pk
                        2              174             0
 sys                  edb$session_wait_history  session_waits_hist_pk
                        4              47              0
 pg_catalog           pg_cast                   pg_cast_source_target_index
                        1              29              0
 pg_catalog           pg_trigger                pg_trig_tgrelid_tgname_index
                        1              15              0
 pg_catalog           pg_class                  pg_class_oid_index
                        1              14              0
 pg_catalog           pg_statistic              pg_statistic_relid_att_index
                        2              12              0
(14 rows)
```

The information displayed in the `DATA from pg_statio_all_indexes` report includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`RELATION`	The name of the table on which the index is defined.
`INDEX`	The name of the index.
`IDX BLKS READ`	The number of index blocks read.
`IDX BLKS HIT`	The number of index blocks hit.
`IDX BLKS ICACHE HIT`	The number of index blocks in Infinite Cache that were hit.

12.4 Performance Tuning Recommendations

To use DRITA reports for performance tuning, review the top five events in a given report, looking for any event that takes a disproportionately large percentage of resources. In a streamlined system, user I/O will probably make up the largest number of waits. Waits should be evaluated in the context of CPU usage and total time; an event may not be significant if it takes 2 minutes out of a total measurement interval of 2 hours, if the rest of the time is consumed by CPU time. The component of response time (CPU "work" time or other "wait" time) that consumes the highest percentage of overall time should be evaluated.

When evaluating events, watch for:

Event type	**Description**
Checkpoint waits	Checkpoint waits may indicate that checkpoint parameters need to be adjusted, (`checkpoint_segments` and `checkpoint_timeout`).
WAL-related waits	WAL-related waits may indicate `wal_buffers` are under-sized.
SQL Parse waits	If the number of waits is high, try to use prepared statements.
db file random reads	If high, check that appropriate indexes and statistics exist.
db file random writes	If high, may need to decrease `bgwriter_delay`.
btree random lock acquires	May indicate indexes are being rebuilt. Schedule index builds during less active time.

Performance reviews should also include careful scrutiny of the hardware, the operating system, the network and the application SQL statements.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

12.5 Event Descriptions

Event Name	Description
`add in shmem lock acquire`	Obsolete/unused
`bgwriter communication lock acquire`	The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process.
`btree vacuum lock acquire`	The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID.
`buffer free list lock acquire`	The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory).
`checkpoint lock acquire:`	A server process has waited for the short-term lock that prevents simultaneous checkpoints.
`checkpoint start lock acquire`	The server has waited for the short-term lock that synchronizes access to the bgwriter checkpoint schedule.
`clog control lock acquire`	The server has waited for the short-term lock that synchronizes access to the commit log.
`control file lock acquire`	The server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number).
`db file extend`	A server process has waited for the operating system while adding a new page to the end of a file.
`db file read`	A server process has waited for the completion of a read (from disk).
`db file write`	A server process has waited for the completion of a write (to disk).
`db file sync`	A server process has waited for the operating system to flush all changes to disk.
`first buf mapping lock acquire`	The server has waited for a short-term lock that synchronizes access to the shared-buffer mapping table.
`freespace lock acquire`	The server has waited for the short-term lock that synchronizes access to the freespace map.
`Infinite Cache read`	The server has waited for an Infinite Cache read request.
`Infinite Cache write`	The server has waited for an Infinite Cache write request.
`lwlock acquire`	The server has waited for a short-term lock that has not been described elsewhere in this section.
`multi xact gen lock acquire`	The server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes).
`multi xact member lock acquire`	The server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes).
`multi xact offset lock acquire`	The server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes).
`oid gen lock acquire`	The server has waited for the short-term lock that synchronizes access to the next available OID (object ID).
`query plan`	The server has computed the execution plan for a SQL statement.
`rel cache init lock acquire`	The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads.
`shmem index lock acquire`	The server has waited for the short-term lock that synchronizes access to the shared-memory map.
`sinval lock acquire`	The server has waited for the short-term lock that synchronizes access to the cache invalidation state.
`sql parse`	The server has parsed a SQL statement.

`subtrans control lock acquire`	The server has waited for the short-term lock that synchronizes access to the subtransaction log.
`tablespace create lock acquire`	The server has waited for the short-term lock that prevents simultaneous `CREATE TABLESPACE` or `DROP TABLESPACE` commands.
`two phase state lock acquire`	The server has waited for the short-term lock that synchronizes access to the list of prepared transactions.
`wal insert lock acquire`	The server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small.
`wal write lock acquire`	The server has waited for the short-term lock that synchronizes write-ahead log flushes.
`wal file sync`	The server has waited for the write-ahead log to sync to disk (related to the wal_sync_method parameter which, by default, is 'fsync' - better performance can be gained by changing this parameter to open_sync).
`wal flush`	The server has waited for the write-ahead log to flush to disk.
`wal write`	The server has waited for a write to the write-ahead log buffer (expect this value to be high).
`xid gen lock acquire`	The server has waited for the short-term lock that synchronizes access to the next available transaction ID.

12.6 Catalog Views

The following DRITA catalog views provide access to performance information relating to system waits.

12.6.1 edb$system_waits

The edb$system_waits view summarizes the number of waits and the total wait time per session for each wait named. It also displays the average and max wait times. The following example shows the result of a SELECT statement on the edb$system_waits view:

```
select * from sys.edb$system_waits;

 edb_id | dbname |wait_name  | wait_count |avg_wait | max_wait | totalwait
--------+--------+-----------+------------+---------+----------+----------
      1 | edb    |db fileread|        301 |0.011516 | 0.629986 | 2.742500
      1 | edb    |wal flush  |         26 |0.010364 | 0.085380 | 0.269452
      1 | edb    |wal write  |         26 |0.010355 | 0.085371 | 0.269232
      1 | edb    |query plan |        277 |0.001367 | 0.049425 | 0.192442
      2 | edb    |wal flush  |         28 |0.040443 | 0.095150 | 0.431984
      2 | edb    |wal write  |         28 |0.040434 | 0.095093 | 0.431698
      2 | edb    |query plan |        299 |0.001479 | 0.049425 | 0.262596
```

edb$system_waits summarizes the following information:

Column Name	Type	Description
edb_id	BIGINT	Wait identifier.
dbname	NAME	Name of the database in which the wait occurs.
wait_name	TEXT	Name of the wait event.
wait_count	BIGINT	Number of times the wait event has occurred.
avg_wait	NUMERIC	Average wait time in milliseconds.
max_wait	NUMERIC(50,6)	Maximum wait time in milliseconds.
totalwait	NUMERIC(50,6)	Total wait time in milliseconds.

12.6.2 edb$session_waits

The edb$session_waits view summarizes the number of waits and the total wait time per session for each wait named and identified by backend ID. It also displays the average and max wait times. The following code sample shows the result of a SELECT statement on the edb$session_waits view:

```
SELECT * FROM sys.edb$session_waits;

 edb_id | dbname | backend_id |  wait_name     | wait_count | avg_wait_time |
max_wait_time| total_wait_time |  usename      |  current_query
--------+--------+------------+----------------+------------+---------------+-
-------------+-----------------+--------------+---------------------------
      1 | edb    |      22935 | db file read   |        175 |      0.008399 |
    0.629986 |        1.469887 | enterprisedb | <IDLE>
      1 | edb    |      22988 | db file read   |        116 |      0.009556 |
    0.040627 |        1.108438 | enterprisedb | select * from edbsnap();
      1 | edb    |      22988 | wal flush      |        26  |      0.010364 |
    0.085380 |        0.269452 | enterprisedb | select * from edbsnap();
(3 rows)
```

edb$session_waits summarizes the following information:

Column Name	Type	Description
edb_id	BIGINT	Wait identifier.
dbname	NAME	Name of the database in which the wait occurs.
backend_id	BIGINT	The backend ID of the process.
wait_name	TEXT	Name of the wait event.
wait_count	BIGINT	Number of times the wait event has occurred.
avg_wait_time	NUMERIC(50,6)	Average wait time in milliseconds.
max_wait_time	NUMERIC	Maximum wait time in milliseconds.
total_wait_time	NUMERIC	Total wait time in milliseconds.
use_name	NAME	The name of the user invoking the query.
current_query	TEXT	The query that is currently executing.

12.6.3 edb$session_wait_history

The `edb$session_wait_history` view contains the last 25 wait events for each backend ID active during the session. The following code sample shows the result of a `SELECT` statement on the `edb$session_wait_history` view:

```
SELECT * FROM sys.edb$session_wait_history;

 edb_id | dbname | backend_id | seq |   wait_name    | elapsed | p1 | p2 | p3
--------+--------+------------+-----+----------------+---------+----+----+----
      1 | edb    |      22935 |   1 | query plan     |      54 |  0 |  0 |  0
      1 | edb    |      22935 |   2 | db file read   |    1116 |2689|  8 |  1
      1 | edb    |      22935 |   3 | db file read   |     983 |1255| 32 |  1
      1 | edb    |      22935 |   4 | db file read   |   13717 |2691| 19 |  1
      1 | edb    |      22935 |   5 | query plan     |      75 |   0|  0 |  0
      1 | edb    |      22935 |   6 | db file read   |   11053 |1255|  7 |  1
      1 | edb    |      22935 |   7 | db file read   |     404 |2689|  4 |  1
 (7 rows)
```

The `edb$session_wait_history` view includes the following information:

Column Name	Type	Description
`edb_id`	`BIGINT`	Wait identifier.
`dbname`	`TEXT`	Name of the database in which the wait occurs.
`backend_id`	`BIGINT`	The session identifier of the process in which the wait occurs.
`seq`	`BIGINT`	The sequence number of the event (value 1 through 25).
`wait_name`	`TEXT`	Name of the wait event.
`elapsed`	`BIGINT`	Elapsed time in milliseconds.
`p1`	`BIGINT`	Wait specific – see table below.
`p2`	`BIGINT`	Wait specific – see table below.
`p3`	`BIGINT`	Wait specific – see table below.

The values contained in the `p1`, `p2`, and `p3` columns are wait-specific. The following waits include information in those columns:

Wait Name	p1	p2	p3
`wal file sync`	0 means Fsync 1 means Fdatasync 2 means open 3 means Fsync writethrough 4 means open dsync For more information, please see the documentation for `WAL_SYNC_METHOD`	unused	unused
`Infinite Cache write`	The Infinite Cache node ID that was written	The file ID from `pg_class.relfilenode`	The block number that was written
`Infinite Cache read`	The file ID from `pg_class.relfilenode`	The block number that was written	unused
`db file extend`	The file ID from `pg_class.relfilenode`	The block number that was extended	Skip Fsync; `1` if True, `0` if False
`db file read`	The file ID from	The block number that was	unused

	pg_class.relfilenode	read	
db file write	The file ID from pg_class.relfilenode	The block number that was written	unused

For all other event types, the p1, p2, and p3 columns are unused.

13 Table Partitioning

In a partitioned table, one logically large table is broken into smaller physical pieces. Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning allows you to omit the partition column from the front of an index, reducing index size and making it more likely that the heavily used parts of the index fits in memory.

- When a query or update accesses a large percentage of a single partition, performance may improve because the server will perform a sequential scan of the partition instead of using an index and random access reads scattered across the whole table.

- A bulk load (or unload) can be implemented by adding or removing partitions, if you plan that requirement into the partitioning design. `ALTER TABLE` is far faster than a bulk operation. It also entirely avoids the `VACUUM` overhead caused by a bulk `DELETE`.

- Seldom-used data can be migrated to less-expensive (or slower) storage media.

Table partitioning is worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application; a good rule of thumb is that the size of the table should exceed the physical memory of the database server.

This document discusses the aspects of table partitioning compatible with Oracle databases that are supported by Advanced Server.

The PostgreSQL 9.5 `INSERT… ON CONFLICT DO NOTHING/UPDATE` clause (commonly known as UPSERT) is not supported on Oracle-styled partitioned tables. If you include the `ON CONFLICT DO NOTHING/UPDATE` clause when invoking the `INSERT` command to add data to a partitioned table, the server will return an error.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

13.1 Selecting a Partition Type

When you create a partitioned table, you specify `LIST`, `RANGE`, or `HASH` partitioning rules. The partitioning rules provide a set of constraints that define the data that is stored in each partition. As new rows are added to the partitioned table, the server uses the partitioning rules to decide which partition should contain each row.

Advanced Server can also use the partitioning rules to enforce partition pruning, improving performance when responding to user queries. When selecting a partitioning type and partitioning keys for a table, you should take into consideration how the data that is stored within a table will be queried, and include often-queried columns in the partitioning rules.

List Partitioning

When you create a list-partitioned table, you specify a single partitioning key column. When adding a row to the table, the server compares the key values specified in the partitioning rule to the corresponding column within the row. If the column value matches a value in the partitioning rule, the row is stored in the partition named in the rule.

Range Partitioning

When you create a range-partitioned table, you specify one or more partitioning key columns. When you add a new row to the table, the server compares the value of the partitioning key (or keys) to the corresponding column (or columns) in a table entry. If the column values satisfy the conditions specified in the partitioning rule, the row is stored in the partition named in the rule.

Hash Partitioning

When you create a hash-partitioned table, you specify one or more partitioning key columns. Data is divided into (approx.) equal-sized partitions amongst the specified partitions. When you add a row to a hash-partitioned table, the server computes a hash value for the data in the specified column (or columns), and stores the row in a partition according to the hash value.

Subpartitioning

Subpartitioning breaks a partitioned table into smaller subsets that may or may not be stored on the same server. A table is typically subpartitioned by a different set of columns, and may be a different subpartitioning type than the parent partition. If one partition is subpartitioned, then each partition will have at least one subpartition.

If a table is subpartitioned, no data will be stored in any of the partition tables; the data will be stored instead in the corresponding subpartitions.

13.2 Using Partition Pruning

Advanced Server's query planner uses *partition pruning* to compute an efficient plan to locate a row (or rows) that matches the conditions specified in the `WHERE` clause of a `SELECT` statement. To successfully prune partitions from an execution plan, the `WHERE` clause must constrain the information that is compared to the partitioning key column specified when creating the partitioned table. When querying a:

- list-partitioned table, partition pruning is effective when the `WHERE` clause compares a literal value to the partitioning key using operators like equal (=) or `AND`.

- range-partitioned table, partition pruning is effective when the `WHERE` clause compares a literal value to a partitioning key using operators such as equal (=), less than (<), or greater than (>).

- hash-partitioned table, partition pruning is effective when the `WHERE` clause compares a literal value to the partitioning key using an operator such as equal (=).

The partition pruning mechanism uses two optimization techniques:

- Fast Pruning

- Constraint exclusion

Partition pruning techniques limit the search for data to only those partitions in which the values for which you are searching might reside. Both pruning techniques remove partitions from a query's execution plan, increasing performance.

The difference between the fast pruning and constraint exclusion is that fast pruning understands the relationship between the partitions in an Oracle-partitioned table, while constraint exclusion does not. For example, when a query searches for a specific value within a list-partitioned table, fast pruning can reason that only a specific partition may hold that value, while constraint exclusion must examine the constraints defined for *each* partition. Fast pruning occurs early in the planning process to reduce the number of partitions that the planner must consider, while constraint exclusion occurs late in the planning process.

Using Constraint Exclusion

The `constraint_exclusion` parameter controls constraint exclusion. The `constraint_exclusion` parameter may have a value of `on`, `off`, or `partition`. To enable constraint exclusion, the parameter must be set to *either* `partition` or `on`. By default, the parameter is set to `partition`.

For more information about constraint exclusion, see:

http://www.postgresql.org/docs/9.5/static/ddl-partitioning.html

When constraint exclusion is enabled, the server examines the constraints defined for each partition to determine if that partition can satisfy a query.

When you execute a `SELECT` statement that *does not* contain a `WHERE` clause, the query planner must recommend an execution plan that searches the entire table. When you execute a `SELECT` statement that *does* contain a `WHERE` clause, the query planner determines in which partition that row would be stored, and sends query fragments to that partition, pruning the partitions that could not contain that row from the execution plan. If you are are not using partitioned tables, disabling constraint exclusion may improve performance.

Fast Pruning

Like constraint exclusion, fast pruning can only optimize queries that include a `WHERE` (or join) clause, and only when the qualifiers in the `WHERE` clause match a certain form. In both cases, the query planner will avoid searching for data within partitions that cannot possibly hold the data required by the query.

Fast pruning is controlled by a boolean configuration parameter named `edb_enable_pruning`. If `edb_enable_pruning` is `ON`, Advanced Server will fast prune certain queries. If `edb_enable_pruning` is `OFF`, the server will disable fast pruning.

Please note: Fast pruning cannot optimize queries against subpartitioned tables or optimize queries against range-partitioned tables that are partitioned on more than one column.

For LIST-partitioned tables, Advanced Server can fast prune queries that contain a `WHERE` clause that constrains a partitioning column to a literal value. For example, given a LIST-partitioned table such as:

```
CREATE TABLE sales_hist(..., country text, ...)
PARTITION BY LIST(country)

(
    PARTITION americas VALUES('US', 'CA', 'MX'),
    PARTITION europe VALUES('BE', 'NL', 'FR'),
    PARTITION asia VALUES('JP', 'PK', 'CN'),
    PARTITION others VALUES(DEFAULT)
)
```

Fast pruning can reason about `WHERE` clauses such as:

```
      WHERE country = 'US'

      WHERE country IS NULL;
```

Given the first `WHERE` clause, fast pruning would eliminate partitions `europe`, `asia`, and `others` because those partitions cannot hold rows that satisfy the qualifier: `WHERE country = 'US'`.

Given the second `WHERE` clause, fast pruning would eliminate partitions `americas`, `europe`, and `asia` because because those partitions cannot hold rows where `country IS NULL`.

The operator specified in the `WHERE` clause must be an equal sign (=) or the equality operator appropriate for the data type of the partitioning column.

For range-partitioned tables, Advanced Server can fast prune queries that contain a `WHERE` clause that constrains a partitioning column to a literal value, but the operator may be any of the following:

```
      >
      >=
      =
      <=
      <
```

Fast pruning will also reason about more complex expressions involving `AND` and `BETWEEN` operators, such as:

```
WHERE size > 100 AND size <= 200
WHERE size BETWEEN 100 AND 200
```

But cannot prune based on expressions involving `OR` or `IN`.

For example, when querying a RANGE-partitioned table, such as:

```
CREATE TABLE boxes(id int, size int, color text)
  PARTITION BY RANGE(size)
(
    PARTITION small VALUES LESS THAN(100),
    PARTITION medium VALUES LESS THAN(200),
    PARTITION large VALUES LESS THAN(300)
)
```

Fast pruning can reason about `WHERE` clauses such as:

```
WHERE size > 100     -- scan partitions 'medium' and 'large'

WHERE size >= 100    -- scan partitions 'medium' and 'large'
WHERE size = 100     -- scan partition 'medium'

WHERE size <= 100    -- scan partitions 'small' and 'medium'

WHERE size < 100     -- scan partition 'small'

WHERE size > 100 AND size < 199     -- scan partition 'medium'

WHERE size BETWEEN 100 AND 199      -- scan partition 'medium'

WHERE color = 'red' AND size = 100  -- scan 'medium'

WHERE color = 'red' AND (size > 100 AND size < 199) -- scan 'medium'
```

In each case, fast pruning requires that the qualifier must refer to a partitioning column and literal value (or `IS NULL`/`IS NOT NULL`).

Note that fast pruning can also optimize `DELETE` and `UPDATE` statements containing `WHERE` clauses of the forms described above.

13.2.1 Example - Partition Pruning

The `EXPLAIN` statement displays the execution plan of a statement. You can use the `EXPLAIN` statement to confirm that Advanced Server is pruning partitions from the execution plan of a query.

To demonstrate the efficiency of partition pruning, first create a simple table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Then, perform a constrained query that includes the `EXPLAIN` statement:

```
EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'INDIA';
```

The resulting query plan shows that the server will scan only the `sales_asia` table - the table in which a row with a `country` value of `INDIA` would be stored:

```
edb=# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'INDIA';
                    QUERY PLAN
---------------------------------------------------
 Append
   ->  Seq Scan on sales
         Filter: ((country)::text = 'INDIA'::text)
   ->  Seq Scan on sales_asia
         Filter: ((country)::text = 'INDIA'::text)
(5 rows)
```

If you perform a query that searches for a row that matches a value not included in the partitioning key:

```
EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE dept_no = '30';
```

The resulting query plan shows that the server must look in all of the partitions to locate the rows that satisfy the query:

```
edb=# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE dept_no = '30';
               QUERY PLAN
-----------------------------------------
 Append
   ->  Seq Scan on sales
         Filter: (dept_no = 30::numeric)
   ->  Seq Scan on sales_europe
         Filter: (dept_no = 30::numeric)
   ->  Seq Scan on sales_asia
         Filter: (dept_no = 30::numeric)
   ->  Seq Scan on sales_americas
         Filter: (dept_no = 30::numeric)
(9 rows)
```

Constraint exclusion also applies when querying subpartitioned tables:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date) SUBPARTITION BY LIST(country)
(
  PARTITION "2011" VALUES LESS THAN('01-JAN-2012')
  (
    SUBPARTITION europe_2011 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2011 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2011 VALUES ('US', 'CANADA')
  ),
  PARTITION "2012" VALUES LESS THAN('01-JAN-2013')
  (
    SUBPARTITION europe_2012 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2012 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2012 VALUES ('US', 'CANADA')
  ),
  PARTITION "2013" VALUES LESS THAN('01-JAN-2015')
  (
    SUBPARTITION europe_2013 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2013 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2013 VALUES ('US', 'CANADA')
  )
);
```

When you query the table, the query planner prunes any partitions or subpartitions from the search path that cannot possibly contain the desired result set:

```
edb=# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'US' AND date =
'Dec 12, 2012';
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Append
```

```
   ->  Seq Scan on sales
         Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12
00:00:00'::timestamp without time zone))
   ->  Seq Scan on sales_2012
         Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12
00:00:00'::timestamp without time zone))
   ->  Seq Scan on sales_americas_2012
         Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12
00:00:00'::timestamp without time zone))
(7 rows)
```

13.3 Partitioning Commands Compatible with Oracle Databases

The following sections provide information about using the table partitioning syntax compatible with Oracle databases supported by Advanced Server.

13.3.1 CREATE TABLE…PARTITION BY

Use the `PARTITION BY` clause of the `CREATE TABLE` command to create a partitioned table with data distributed amongst one or more partitions (and subpartitions). The command syntax comes in the following forms:

List Partitioning Syntax

Use the first form to create a list-partitioned table:

```
CREATE TABLE [ schema. ]table_name
   table_definition
   PARTITION BY LIST(column)
   [SUBPARTITION BY {RANGE|LIST|HASH} (column[, column ]...)]
   (list_partition_definition[, list_partition_definition]...);
```

Where *list_partition_definition* is:

```
      PARTITION [partition_name]
        VALUES (value[, value]...)
        [TABLESPACE tablespace_name]
        [(subpartition, ...)]
```

Range Partitioning Syntax

Use the second form to create a range-partitioned table:

```
CREATE TABLE [ schema. ]table_name
   table_definition
   PARTITION BY RANGE(column[, column ]...)
   [SUBPARTITION BY {RANGE|LIST|HASH} (column[, column ]...)]
   (range_partition_definition[, range_partition_definition]...);
```

Where *range_partition_definition* is:

```
      PARTITION [partition_name]
        VALUES LESS THAN (value[, value]...)
        [TABLESPACE tablespace_name]
        [(subpartition, ...)]
```

Hash Partitioning Syntax

Use the third form to create a hash-partitioned table:

```
CREATE TABLE [ schema. ]table_name
   table_definition
   PARTITION BY HASH(column[, column ]...)
   [SUBPARTITION BY {RANGE|LIST|HASH} (column[, column ]...)]
   (hash_partition_definition[, hash_partition_definition]...);
```

Where `hash_partition_definition` is:

```
      [PARTITION partition_name]
        [TABLESPACE tablespace_name]
        [(subpartition, ...)]
```

Subpartitioning Syntax

`subpartition` may be one of the following:

```
      {list_subpartition | range_subpartition | hash_subpartition}
```

where `list_subpartition` is:

```
           SUBPARTITION [subpartition_name]
             VALUES (value[, value]...)
             [TABLESPACE tablespace_name]
```

where `range_subpartition` is:

```
           SUBPARTITION [subpartition_name]
             VALUES LESS THAN (value[, value]...)
             [TABLESPACE tablespace_name]
```

where `hash_subpartition` is:

```
           [SUBPARTITION subpartition_name]
             [TABLESPACE tablespace_name]
```

Description

The `CREATE TABLE… PARTITION BY` command creates a table with one or more partitions; each partition may have one or more subpartitions. There is no upper limit to the number of defined partitions, but if you include the `PARTITION BY` clause, you must specify at least one partitioning rule. The resulting table will be owned by the user that creates it.

Use the `PARTITION BY LIST` clause to divide a table into partitions based on the values entered in a specified column. Each partitioning rule must specify at least one literal value, but there is no upper limit placed on the number of values you may specify. Include a rule that specifies a matching value of `DEFAULT` to direct any un-qualified rows to the given partition; for more information about using the DEFAULT keyword, see Section 13.4.

Use the PARTITION BY RANGE clause to specify boundary rules by which to create partitions. Each partitioning rule must contain at least one column of a data type that has two operators (i.e., a greater-than or equal to operator, and a less-than operator). Range boundaries are evaluated against a LESS THAN clause and are non-inclusive; a date boundary of January 1, 2013 will include only those date values that fall on or before December 31, 2012.

Range partition rules must be specified in ascending order. INSERT commands that store rows with values that exceed the top boundary of a range-partitioned table will fail unless the partitioning rules include a boundary rule that specifies a value of MAXVALUE. If you do not include a MAXVALUE partitioning rule, any row that exceeds the maximum limit specified by the boundary rules will result in an error.

For more information about using the MAXVALUE keyword, see Section 13.4.

Use the PARTITION BY HASH clause to create a hash-partitioned table. In a HASH partitioned table, data is divided amongst equal-sized partitions based on the hash value of the column specified in the partitioning syntax. When specifying a HASH partition, choose a column (or combination of columns) that is as close to unique as possible to help ensure that data is evenly distributed amongst the partitions. When selecting a partitioning column (or combination of columns), select a column (or columns) that you frequently search for exact matches for best performance.

Use the TABLESPACE keyword to specify the name of a tablespace on which a partition or subpartition will reside; if you do not specify a tablespace, the partition or subpartition will reside in the default tablespace.

If a table definition includes the SUBPARTITION BY clause, each partition within that table will have at least one subpartition. Each subpartition may be explicitly defined or system-defined.

If the subpartition is system-defined, the server-generated subpartition will reside in the default tablespace, and the name of the subpartition will be assigned by the server. The server will create:

- A DEFAULT subpartition if the SUBPARTITION BY clause specifies LIST.
- A MAXVALUE subpartition if the SUBPARTITION BY clause specifies RANGE.

The server will generate a subpartition name that is a combination of the partition table name and a unique identifier. You can query the ALL_TAB_SUBPARTITIONS table to review a complete list of subpartition names.

Parameters

`table_name`

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

`table_definition`

The column names, data types, and constraint information as described in the PostgreSQL core documentation for the `CREATE TABLE` statement, available at:

http://www.postgresql.org/docs/9.5/static/sql-createtable.html

`partition_name`

The name of the partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`subpartition_name`

The name of the subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`column`

The name of a column on which the partitioning rules are based. Each row will be stored in a partition that corresponds to the *`value`* of the specified column(s).

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

Use `value` to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. `value` may be `NULL`, `DEFAULT` (if specifying a `LIST` partition), or `MAXVALUE` (if specifying a `RANGE` partition).

When specifying rules for a list-partitioned table, include the `DEFAULT` keyword in the last partition rule to direct any un-matched rows to the given partition. If you do not include a rule that includes a value of `DEFAULT`, any `INSERT` statement that attempts to add a row that does not match the specified rules of at least one partition will fail, and return an error.

When specifying rules for a list-partitioned table, include the `MAXVALUE` keyword in the last partition rule to direct any un-categorized rows to the given partition. If you do not include a `MAXVALUE` partition, any `INSERT` statement that attempts to add a row where the partitioning key is greater than the highest value specified will fail, and return an error.

`tablespace_name`

The name of the tablespace in which the partition or subpartition resides.

13.3.1.1 Example - PARTITION BY LIST

The following example creates a partitioned table (`sales`) using the `PARTITION BY LIST` clause. The `sales` table stores information in three partitions (`europe`, `asia`, and `americas`):

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

The resulting table is partitioned by the value specified in the `country` column:

```
acctg=# SELECT partition_name, high_value from ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 americas       | 'US', 'CANADA'
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(3 rows)
```

- Rows with a value of `US` or `CANADA` in the `country` column are stored in the `americas` partition.
- Rows with a value of `INDIA` or `PAKISTAN` in the `country` column are stored in the `asia` partition.
- Rows with a value of `FRANCE` or `ITALY` in the `country` column are stored in the `europe` partition.

The server would evaluate the following statement against the partitioning rules, and store the row in the `europe` partition:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012',
'650000');
```

13.3.1.2 Example - PARTITION BY RANGE

The following example creates a partitioned table (`sales`) using the `PARTITION BY RANGE` clause. The `sales` table stores information in four partitions (`q1_2012`, `q2_2012`, `q3_2012` and `q4_2012`) :

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012
    VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
);
```

The resulting table is partitioned by the value specified in the `date` column:

```
acctg=# SELECT partition_name, high_value from ALL_TAB_PARTITIONS;
 partition_name |  high_value
----------------+---------------
 q4_2012        | '2013-Jan-01'
 q3_2012        | '2012-Oct-01'
 q2_2012        | '2012-Jul-01'
 q1_2012        | '2012-Apr-01'
(4 rows)
```

- Any row with a value in the `date` column before April 1, 2012 is stored in a partition named `q1_2012`.

- Any row with a value in the `date` column before July 1, 2012 is stored in a partition named `q2_2012`.

- Any row with a value in the `date` column before October 1, 2012 is stored in a partition named `q3_2012`.
- Any row with a value in the `date` column before January 1, 2013 is stored in a partition named `q4_2012`.

The server would evaluate the following statement against the partitioning rules and store the row in the `q3_2012` partition:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012',
'650000');
```

13.3.1.3 Example - PARTITION BY HASH

The following example creates a partitioned table (`sales`) using the `PARTITION BY HASH` clause. The `sales` table stores information in three partitions (`p1`, `p2`, and `p3`:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY HASH (part_no)
(
  PARTITION p1,
  PARTITION p2,
  PARTITION p3
);
```

The table is partitioned by the hash value of the value specified in the `part_no` column:

```
acctg=# SELECT partition_name, partition_position from ALL_TAB_PARTITIONS;
 partition_name | partition_position
----------------+--------------------
 p3             |                  3
 p2             |                  2
 p1             |                  1
(3 rows)
```

The server will evaluate the hash value of the `part_no` column, and distribute the rows into approximately equal partitions.

13.3.1.4 Example - PARTITION BY RANGE, SUBPARTITION BY LIST

The following example creates a partitioned table (`sales`) that is first partitioned by the transaction date; the range partitions (`q1_2012`, `q2_2012`, `q3_2012` and `q4_2012`) are then list-subpartitioned using the value of the `country` column.

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST(country)
  (
    PARTITION q1_2012
      VALUES LESS THAN('2012-Apr-01')
      (
        SUBPARTITION q1_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q1_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q1_americas VALUES ('US', 'CANADA')
       ),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01')
      (
        SUBPARTITION q2_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q2_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q2_americas VALUES ('US', 'CANADA')
       ),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01')
      (
        SUBPARTITION q3_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q3_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q3_americas VALUES ('US', 'CANADA')
       ),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
      (
        SUBPARTITION q4_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q4_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q4_americas VALUES ('US', 'CANADA')
       )
);
```

This statement creates a table with four partitions; each partition has three subpartitions:

```
acctg=# SELECT subpartition_name, high_value, partition_name FROM
ALL_TAB_SUBPARTITIONS;
 subpartition_name |     high_value       | partition_name
-------------------+----------------------+----------------
 q4_asia           | 'INDIA', 'PAKISTAN' | q4_2012
 q4_europe         | 'FRANCE', 'ITALY'   | q4_2012
 q4_americas       | 'US', 'CANADA'      | q4_2012
 q3_americas       | 'US', 'CANADA'      | q3_2012
 q3_asia           | 'INDIA', 'PAKISTAN' | q3_2012
 q3_europe         | 'FRANCE', 'ITALY'   | q3_2012
 q2_americas       | 'US', 'CANADA'      | q2_2012
 q2_asia           | 'INDIA', 'PAKISTAN' | q2_2012
 q2_europe         | 'FRANCE', 'ITALY'   | q2_2012
 q1_americas       | 'US', 'CANADA'      | q1_2012
 q1_asia           | 'INDIA', 'PAKISTAN' | q1_2012
 q1_europe         | 'FRANCE', 'ITALY'   | q1_2012
(12 rows)
```

When a row is added to this table, the value in the `date` column is compared to the values specified in the range partitioning rules, and the server selects the partition in which the row should reside. The value in the `country` column is then compared to the values specified in the list subpartitioning rules; when the server locates a match for the value, the row is stored in the corresponding subpartition.

Any row added to the table will be stored in a subpartition, so the partitions will contain no data.

The server would evaluate the following statement against the partitioning and subpartitioning rulesand store the row in the `q3_europe` partition:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012',
'650000');
```

13.3.2 ALTER TABLE...ADD PARTITION

Use the `ALTER TABLE… ADD PARTITION` command to add a partition to an existing partitioned table. The syntax is:

```
ALTER TABLE table_name ADD PARTITION partition_definition;
```

Where *partition_definition* is:

```
      {list_partition | range_partition }
```

and *list_partition* is:

```
      PARTITION [partition_name]
        VALUES (value[, value]...)
        [TABLESPACE tablespace_name]
        [(subpartition, ...)]
```

and *range_partition* is:

```
      PARTITION [partition_name]
        VALUES LESS THAN (value[, value]...)
        [TABLESPACE tablespace_name]
        [(subpartition, ...)]
```

Where *subpartition* is:

```
      {list_subpartition | range_subpartition | hash_subpartition}
```

and *list_subpartition* is:

```
      SUBPARTITION [subpartition_name]
        VALUES (value[, value]...)
        [TABLESPACE tablespace_name]
```

and *range_subpartition* is:

```
      SUBPARTITION [subpartition_name ]
        VALUES LESS THAN (value[, value]...)
        [TABLESPACE tablespace_name]
```

Description

The `ALTER TABLE… ADD PARTITION` command adds a partition to an existing partitioned table. There is no upper limit to the number of defined partitions in a partitioned table.

New partitions must be of the same type (`LIST`, `RANGE` or `HASH`) as existing partitions. The new partition rules must reference the same column specified in the partitioning rules that define the existing partition(s).

You cannot use the `ALTER TABLE… ADD PARTITION` statement to add a partition to a table with a `MAXVALUE` or `DEFAULT` rule. Note that you can alternatively use the `ALTER TABLE… SPLIT PARTITION` statement to split an existing partition, effectively increasing the number of partitions in a table.

`RANGE` partitions must be specified in ascending order. You cannot add a new partition that precedes existing partitions in a `RANGE` partitioned table.

Include the `TABLESPACE` clause to specify the tablespace in which the new partition will reside. If you do not specify a tablespace, the partition will reside in the default tablespace.

If the table is indexed, the index will be created on the new partition.

To use the `ALTER TABLE... ADD PARTITION` command you must be the table owner, or have superuser (or administrative) privileges.

Parameters

`table_name`

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

`partition_name`

The name of the partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`subpartition_name`

The name of the subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

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

Use *`value`* to specify a quoted literal value (or comma-delimited list of literal values) by which rows will be distributed into partitions. Each partitioning rule must specify at least one *`value`*, but there is no limit placed on the number of values specified within a rule. *value* may also be NULL, DEFAULT (if specifying a LIST partition), or MAXVALUE (if specifying a RANGE partition).

For information about creating a DEFAULT or MAXVALUE partition, see Section 13.4.

tablespace_name

The name of the tablespace in which a partition or subpartition resides.

13.3.2.1 Example - Adding a Partition to a LIST Partitioned Table

The example that follows adds a partition to the list-partitioned sales table. The table was created using the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

The table contains three partitions (americas, asia, and europe) :

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 americas       | 'US', 'CANADA'
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(3 rows)
```

The following command adds a partition named east_asia to the sales table:

```
ALTER TABLE sales ADD PARTITION east_asia
  VALUES ('CHINA', 'KOREA');
```

After invoking the command, the table includes the east_asia partition:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
```

```
 east_asia      | 'CHINA', 'KOREA'
 americas       | 'US', 'CANADA'
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(4 rows)
```

13.3.2.2 Example - Adding a Partition to a RANGE Partitioned Table

The example that follows adds a partition to a range-partitioned table named `sales`:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012
    VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
);
```

The table contains four partitions (`q1_2012`, `q2_2012, q3_2012`, and `q4_2012`):

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |  high_value
----------------+---------------
 q4_2012        | '2013-Jan-01'
 q3_2012        | '2012-Oct-01'
 q2_2012        | '2012-Jul-01'
 q1_2012        | '2012-Apr-01'
(4 rows)
```

The following command adds a partition named `q1_2013` to the `sales` table:

```
ALTER TABLE sales ADD PARTITION q1_2013
  VALUES LESS THAN('01-APR-2013');
```

After invoking the command, the table includes the `q1_2013` partition:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
```

```
 partition_name |  high_value
----------------+---------------
 q1_2012        | '2012-Apr-01'
 q2_2012        | '2012-Jul-01'
 q3_2012        | '2012-Oct-01'
 q4_2012        | '2013-Jan-01'
 q1_2013        | '01-APR-2013'
(5 rows)
```

13.3.3 ALTER TABLE… ADD SUBPARTITION

The `ALTER TABLE… ADD SUBPARTITION` command adds a subpartition to an existing subpartitioned partition. The syntax is:

```
ALTER TABLE table_name MODIFY PARTITION partition_name
      ADD SUBPARTITION subpartition_definition;
```

Where *`subpartition_definition`* is:

```
{list_subpartition | range_subpartition}
```

and *`list_subpartition`* is:

```
SUBPARTITION [subpartition_name]
  VALUES (value[, value]...)
  [TABLESPACE tablespace_name]
```

and *`range_subpartition`* is:

```
SUBPARTITION [subpartition_name]
  VALUES LESS THAN (value[, value]...)
  [TABLESPACE tablespace_name]
```

Description

The `ALTER TABLE… ADD SUBPARTITION` command adds a subpartition to an existing partition; the partition must already be subpartitioned. There is no upper limit to the number of defined subpartitions.

New subpartitions must be of the same type (`LIST`, `RANGE` or `HASH`) as existing subpartitions. The new subpartition rules must reference the same column specified in the subpartitioning rules that define the existing subpartition(s).

You cannot use the `ALTER TABLE… ADD SUBPARTITION` statement to add a subpartition to a table with a `MAXVALUE` or `DEFAULT` rule , but you can split an existing subpartition with the `ALTER TABLE… SPLIT SUBPARTITION` statement, effectively adding a subpartition to a table.

You cannot add a new subpartition that precedes existing subpartitions in a range subpartitioned table; range subpartitions must be specified in ascending order.

Include the `TABLESPACE` clause to specify the tablespace in which the subpartition will reside. If you do not specify a tablespace, the subpartition will be created in the default tablespace.

If the table is indexed, the index will be created on the new subpartition.

To use the `ALTER TABLE... ADD SUBPARTITION` command you must be the table owner, or have superuser (or administrative) privileges.

Parameters

`table_name`

The name (optionally schema-qualified) of the partitioned table in which the subpartition will reside.

`partition_name`

The name of the partition in which the new subpartition will reside.

`subpartition_name`

The name of the subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

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

Use `value` to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. `value` may also be `NULL`, `DEFAULT` (if specifying a `LIST` partition), or `MAXVALUE` (if specifying a `RANGE` partition).

For information about creating a `DEFAULT` or `MAXVALUE` partition, see Section 13.4.

`tablespace_name`

The name of the tablespace in which the subpartition resides.

13.3.3.1 Example - Adding a Subpartition to a LIST-RANGE Partitioned Table

The following example adds a `RANGE` subpartition to the list-partitioned `sales` table. The `sales` table was created with the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
  SUBPARTITION BY RANGE(date)
(
  PARTITION europe VALUES('FRANCE', 'ITALY')
    (
      SUBPARTITION europe_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION europe_2012
        VALUES LESS THAN('2013-Jan-01')
    ),
  PARTITION asia VALUES('INDIA', 'PAKISTAN')
    (
      SUBPARTITION asia_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION asia_2012
        VALUES LESS THAN('2013-Jan-01')
    ),
  PARTITION americas VALUES('US', 'CANADA')
    (
      SUBPARTITION americas_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION americas_2012
        VALUES LESS THAN('2013-Jan-01')
    )
);
```

The `sales` table has three partitions, named `europe`, `asia`, and `americas`. Each partition has two range-defined subpartitions:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2011       | '2012-Jan-01'
 europe         | europe_2012       | '2013-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
 asia           | asia_2012         | '2013-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_2012     | '2013-Jan-01'
(6 rows)
```

The following command adds a subpartition named `europe_2013`:

```
ALTER TABLE sales MODIFY PARTITION europe
  ADD SUBPARTITION europe_2013
  VALUES LESS THAN('2015-Jan-01');
```

After invoking the command, the table includes a subpartition named `europe_2013`:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2011       | '2012-Jan-01'
 europe         | europe_2012       | '2013-Jan-01'
 europe         | europe_2013       | '2015-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
 asia           | asia_2012         | '2013-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_2012     | '2013-Jan-01'
(7 rows)
```

Note that when adding a new range subpartition, the subpartitioning rules must specify a range that falls *after* any existing subpartitions.

13.3.3.2 Example - Adding a Subpartition to a RANGE-LIST Partitioned Table

The following example adds a `LIST` subpartition to the `RANGE` partitioned `sales` table. The `sales` table was created with the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST (country)
  (
    PARTITION first_half_2012 VALUES LESS THAN('01-JUL-2012')
    (
      SUBPARTITION europe VALUES ('ITALY', 'FRANCE'),
      SUBPARTITION americas VALUES ('US', 'CANADA')
    ),
    PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013')
    (
      SUBPARTITION asia VALUES ('INDIA', 'PAKISTAN')
    )
  );
```

After executing the above command, the `sales` table will have two partitions, named `first_half_2012` and `second_half_2012`. The `first_half_2012` partition has two subpartitions, named `europe` and `americas`, and the `second_half_2012` partition has one partition, named `asia`:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |     high_value
------------------+-------------------+---------------------
 first_half_2012  | europe            | 'ITALY', 'FRANCE'
 first_half_2012  | americas          | 'US', 'CANADA'
 second_half_2012 | asia              | 'INDIA', 'PAKISTAN'
(3 rows)
```

The following command adds a subpartition to the `second_half_2012` partition, named `east_asia`:

```
ALTER TABLE sales MODIFY PARTITION second_half_2012
  ADD SUBPARTITION east_asia VALUES ('CHINA');
```

After invoking the command, the table includes a subpartition named `east_asia`:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |     high_value
------------------+-------------------+---------------------
 first_half_2012  | europe            | 'ITALY', 'FRANCE'
 first_half_2012  | americas          | 'US', 'CANADA'
 second_half_2012 | asia              | 'INDIA', 'PAKISTAN'
 second_half_2012 | east_asia         | 'CHINA'
(4 rows)
```

13.3.4 ALTER TABLE...SPLIT PARTITION

Use the `ALTER TABLE… SPLIT PARTITION` command to divide a single partition into two partitions, and redistribute the partition's contents between the new partitions. The command syntax comes in two forms.

The first form splits a `RANGE` partition into two partitions:

```
ALTER TABLE table_name SPLIT PARTITION partition_name
  AT (range_part_value)
  INTO
  (
    PARTITION new_part1
      [TABLESPACE tablespace_name],
    PARTITION new_part2
      [TABLESPACE tablespace_name]
  );
```

The second form splits a `LIST` partition into two partitions:

```
ALTER TABLE table_name SPLIT PARTITION partition_name
  VALUES (value[, value]...)
  INTO
  (
    PARTITION new_part1
      [TABLESPACE tablespace_name],
    PARTITION new_part2
      [TABLESPACE tablespace_name]
  );
```

Description

The `ALTER TABLE...SPLIT PARTITION` command adds a partition to an existing `LIST` or `RANGE` partitioned table. Please note that the `ALTER TABLE… SPLIT PARTITION` command cannot add a partition to a `HASH` partitioned table. There is no upper limit to the number of partitions that a table may have.

When you execute an `ALTER TABLE...SPLIT PARTITION` command, Advanced Server creates two new partitions, and redistributes the content of the old partition between them (as constrained by the partitioning rules).

Include the `TABLESPACE` clause to specify the tablespace in which a partition will reside. If you do not specify a tablespace, the partition will reside in the default tablespace.

If the table is indexed, the index will be created on the new partition.

To use the `ALTER TABLE... SPLIT PARTITION` command you must be the table owner, or have superuser (or administrative) privileges.

Parameters

`table_name`

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

`partition_name`

The name of the partition that is being split.

`new_part1`

The name of the first new partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_part1` will receive the rows that meet the subpartitioning constraints specified in the `ALTER TABLE… SPLIT SUBPARTITION` command.

`new_part2`

The name of the second new partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_part2` will receive the rows are not directed to *`new_part1`* by the partitioning constraints specified in the `ALTER TABLE… SPLIT PARTITION` command.

`range_part_value`

Use *`range_part_value`* to specify the boundary rules by which to create the new partition. The partitioning rule must contain at least one column of a data type that has two operators (i.e., a greater-than-or-equal to operator, and a less-than operator). Range boundaries are evaluated against a `LESS THAN` clause and are non-inclusive; a date boundary of January 1, 2010 will include only those date values that fall on or before December 31, 2009.

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

Use *`value`* to specify a quoted literal value (or comma-delimited list of literal values) by which rows will be distributed into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule.

For information about creating a `DEFAULT` or `MAXVALUE` partition, see Section 13.4.

`tablespace_name`

The name of the tablespace in which the partition or subpartition resides.

13.3.4.1 Example - Splitting a LIST Partition

Our example will divide one of the partitions in the list-partitioned `sales` table into two new partitions, and redistribute the contents of the partition between them. The `sales` table is created with the statement:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

The table definition creates three partitions (`europe`, `asia`, and `americas`). The following command adds rows to each partition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

The rows are distributed amongst the partitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date        | amount
----------------+---------+---------+----------+--------------------+-------
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
(17 rows)
```

The following command splits the `americas` partition into two partitions named `us` and `canada`:

```
ALTER TABLE sales SPLIT PARTITION americas
  VALUES ('US')
  INTO (PARTITION us, PARTITION canada);
```

A `SELECT` statement confirms that the rows have been redistributed across the correct partitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid   | dept_no | part_no | country  |        date        | amount
--------------+---------+---------+----------+--------------------+--------
 sales_europe |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia   |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia   |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia   |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia   |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia   |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_us     |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_us     |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_us     |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_us     |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_us     |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
 sales_canada |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_canada |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_canada |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
```

```
(17 rows)
```

13.3.4.2 Example - Splitting a RANGE Partition

This example divides the `q4_2012` partition (of the range-partitioned `sales` table) into two partitions, and redistribute the partition's contents. Use the following command to create the `sales` table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012
    VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
);
```

The table definition creates four partitions (`q1_2012`, `q2_2012`, `q3_2012`, and `q4_2012`). The following command adds rows to each partition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

A `SELECT` statement confirms that the rows are distributed amongst the partitions as expected:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | dept_no | part_no | country  |        date         | amount
---------------+---------+---------+----------+---------------------+--------
 sales_q1_2012 |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_q1_2012 |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_q1_2012 |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_q2_2012 |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_q2_2012 |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_q2_2012 |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_q2_2012 |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_q3_2012 |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_q3_2012 |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_q3_2012 |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_q3_2012 |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_q3_2012 |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_q4_2012 |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_q4_2012 |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_q4_2012 |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |   15000
 sales_q4_2012 |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_q4_2012 |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(17 rows)
```

The following command splits the `q4_2012` partition into two partitions named `q4_2012_p1` and `q4_2012_p2`:

```
    ALTER TABLE sales SPLIT PARTITION q4_2012
      AT ('15-Nov-2012')
      INTO
      (
        PARTITION q4_2012_p1,
        PARTITION q4_2012_p2
      );
```

A `SELECT` statement confirms that the rows have been redistributed across the new partitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid     | dept_no | part_no | country  |        date        |amount
------------------+---------+---------+----------+--------------------+------
 sales_q1_2012    |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 | 45000
 sales_q1_2012    |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 | 75000
 sales_q1_2012    |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 | 75000
 sales_q2_2012    |      40 | 9519b   | US       | 12-APR-12 00:00:00 |145000
 sales_q2_2012    |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 | 37500
 sales_q2_2012    |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |120000
 sales_q2_2012    |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |  4950
 sales_q3_2012    |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 | 15000
 sales_q3_2012    |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |650000
 sales_q3_2012    |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |650000
 sales_q3_2012    |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |  5090
 sales_q3_2012    |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |  4950
 sales_q4_2012_p1 |      40 | 4577b   | US       | 11-NOV-12 00:00:00 | 25000
 sales_q4_2012_p1 |      40 | 4788b   | US       | 09-OCT-12 00:00:00 | 15000
 sales_q4_2012_p1 |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |650000
 sales_q4_2012_p2 |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 | 50000
 sales_q4_2012_p2 |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |  5090
(17 rows)
```

13.3.5 ALTER TABLE...SPLIT SUBPARTITION

Use the `ALTER TABLE… SPLIT SUBPARTITION` command to divide a single subpartition into two subpartitions, and redistribute the subpartition's contents. The command comes in two variations.

The first variation splits a range subpartition into two subpartitions:

```
ALTER TABLE table_name SPLIT SUBPARTITION subpartition_name
  AT (range_part_value)
  INTO
  (
    SUBPARTITION new_subpart1
      [TABLESPACE tablespace_name],
    SUBPARTITION new_subpart2
      [TABLESPACE tablespace_name]
  );
```

The second variation splits a list subpartition into two subpartitions:

```
ALTER TABLE table_name SPLIT SUBPARTITION subpartition_name
  VALUES (value[, value]...)
  INTO
  (
    SUBPARTITION new_subpart1
      [TABLESPACE tablespace_name],
    SUBPARTITION new_subpart2
      [TABLESPACE tablespace_name]
  );
```

Description

The `ALTER TABLE...SPLIT SUBPARTITION` command adds a subpartition to an existing subpartitioned table. There is no upper limit to the number of defined subpartitions. When you execute an `ALTER TABLE...SPLIT SUBPARTITION` command, Advanced Server creates two new subpartitions, moving any rows that contain values that are constrained by the specified subpartition rules into *`new_subpart1`*, and any remaining rows into *`new_subpart2`*.

The new subpartition rules must reference the column specified in the rules that define the existing subpartition(s).

Include the `TABLESPACE` clause to specify a tablespace in which a new subpartition will reside. If you do not specify a tablespace, the subpartition will be created in the default tablespace.

If the table is indexed, the index will be created on the new subpartition.

To use the `ALTER TABLE... SPLIT SUBPARTITION` command you must be the table owner, or have superuser (or administrative) privileges.

Parameters

`table_name`

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

`subpartition_name`

The name of the subpartition that is being split.

`new_subpart1`

The name of the first new subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_subpart1` will receive the rows that meet the subpartitioning constraints specified in the `ALTER TABLE… SPLIT SUBPARTITION` command.

`new_subpart2`

The name of the second new subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_subpart2` will receive the rows are not directed to *`new_subpart1`* by the subpartitioning constraints specified in the `ALTER TABLE… SPLIT SUBPARTITION` command.

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

Use *`value`* to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. *`value`* may also be `NULL`, `DEFAULT` (if specifying a `LIST` subpartition), or `MAXVALUE` (if specifying a `RANGE` subpartition).

For information about creating a `DEFAULT` or `MAXVALUE` partition, see Section 13.4.

`tablespace_name`

The name of the tablespace in which the partition or subpartition resides.

13.3.5.1 Example - Splitting a LIST Subpartition

The following example splits a list subpartition, redistributing the subpartition's contents between two new subpartitions. The sample table (`sales`) was created with the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST(country)
  (
    PARTITION first_half_2012 VALUES LESS THAN('01-JUL-2012')
    (
      SUBPARTITION p1_europe VALUES ('ITALY', 'FRANCE'),
      SUBPARTITION p1_americas VALUES ('US', 'CANADA')
    ),
    PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013')
    (
      SUBPARTITION p2_europe VALUES ('ITALY', 'FRANCE'),
      SUBPARTITION p2_americas VALUES ('US', 'CANADA')
    )
  );
```

The `sales` table has two partitions, named `first_half_2012`, and `second_half_2012`. Each partition has two range-defined subpartitions that distribute the partition's contents into subpartitions based on the value of the `country` column:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |    high_value
------------------+-------------------+-------------------
 second_half_2012 | p2_europe         | 'ITALY', 'FRANCE'
 first_half_2012  | p1_europe         | 'ITALY', 'FRANCE'
 second_half_2012 | p2_americas       | 'US', 'CANADA'
 first_half_2012  | p1_americas       | 'US', 'CANADA'
(4 rows)
```

The following command adds rows to each subpartition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000');
```

A `SELECT` statement confirms that the rows are correctly distributed amongst the subpartitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid      | dept_no | part_no | country|        date        |amount
-------------------+---------+---------+--------+--------------------+------
 sales_p1_europe   |      10 | 4519b   | FRANCE | 17-JAN-12 00:00:00 |  45000
 sales_p1_europe   |      10 | 4519b   | FRANCE | 17-JAN-12 00:00:00 |  45000
 sales_p1_americas |      40 | 9519b   | US     | 12-APR-12 00:00:00 | 145000
 sales_p1_americas |      30 | 9519b   | CANADA | 01-FEB-12 00:00:00 |  75000
 sales_p1_americas |      30 | 4519b   | CANADA | 08-APR-12 00:00:00 | 120000
 sales_p1_americas |      40 | 3788a   | US     | 12-MAY-12 00:00:00 |   4950
 sales_p2_europe   |      10 | 9519b   | ITALY  | 07-JUL-12 00:00:00 |  15000
 sales_p2_europe   |      10 | 9519a   | FRANCE | 18-AUG-12 00:00:00 | 650000
 sales_p2_europe   |      10 | 9519b   | FRANCE | 18-AUG-12 00:00:00 | 650000
 sales_p2_americas |      40 | 4577b   | US     | 11-NOV-12 00:00:00 |  25000
 sales_p2_americas |      30 | 7588b   | CANADA | 14-DEC-12 00:00:00 |  50000
 sales_p2_americas |      40 | 4788a   | US     | 23-SEP-12 00:00:00 |   4950
 sales_p2_americas |      40 | 4788b   | US     | 09-OCT-12 00:00:00 |  15000
(13 rows)
```

The following command splits the `p2_americas` subpartition into two new subpartitions, and redistributes the contents:

```
ALTER TABLE sales SPLIT SUBPARTITION p2_americas
  VALUES ('US')
  INTO
  (
    SUBPARTITION p2_us,
    SUBPARTITION p2_canada
  );
```

After invoking the command, the `p2_americas` subpartition has been deleted; in it's place, the server has created two new subpartitions (`p2_us` and `p2_canada`):

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
```

```
  partition_name  | subpartition_name |    high_value
------------------+-------------------+-------------------
 first_half_2012  | p1_europe         | 'ITALY', 'FRANCE'
 first_half_2012  | p1_americas       | 'US', 'CANADA'
 second_half_2012 | p2_europe         | 'ITALY', 'FRANCE'
 second_half_2012 | p2_canada         | 'CANADA'
 second_half_2012 | p2_us             | 'US'
(5 rows)
```

Querying the `sales` table demonstrates that the content of the `p2_americas` subpartition has been redistributed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid      | dept_no | part_no | country |        date        |amount
-------------------+---------+---------+---------+--------------------+------
 sales_p1_europe   |      10 | 4519b   | FRANCE  | 17-JAN-12 00:00:00 | 45000
 sales_p1_europe   |      10 | 4519b   | FRANCE  | 17-JAN-12 00:00:00 | 45000
 sales_p1_americas |      40 | 9519b   | US      | 12-APR-12 00:00:00 |145000
 sales_p1_americas |      30 | 9519b   | CANADA  | 01-FEB-12 00:00:00 | 75000
 sales_p1_americas |      30 | 4519b   | CANADA  | 08-APR-12 00:00:00 |120000
 sales_p1_americas |      40 | 3788a   | US      | 12-MAY-12 00:00:00 |  4950
 sales_p2_europe   |      10 | 9519b   | ITALY   | 07-JUL-12 00:00:00 | 15000
 sales_p2_europe   |      10 | 9519a   | FRANCE  | 18-AUG-12 00:00:00 |650000
 sales_p2_europe   |      10 | 9519b   | FRANCE  | 18-AUG-12 00:00:00 |650000
 sales_p2_us       |      40 | 4577b   | US      | 11-NOV-12 00:00:00 | 25000
 sales_p2_us       |      40 | 4788a   | US      | 23-SEP-12 00:00:00 |  4950
 sales_p2_us       |      40 | 4788b   | US      | 09-OCT-12 00:00:00 | 15000
 sales_p2_canada   |      30 | 7588b   | CANADA  | 14-DEC-12 00:00:00 | 50000
(13 rows)
```

13.3.5.2 Example - Splitting a RANGE Subpartition

The following example splits a range subpartition, redistributing the subpartition's contents between two new subpartitions. The sample table (`sales`) was created with the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
  SUBPARTITION BY RANGE(date)
(
  PARTITION europe VALUES('FRANCE', 'ITALY')
    (
      SUBPARTITION europe_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION europe_2012
        VALUES LESS THAN('2013-Jan-01')
    ),
  PARTITION asia VALUES('INDIA', 'PAKISTAN')
    (
      SUBPARTITION asia_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION asia_2012
        VALUES LESS THAN('2013-Jan-01')
    ),
  PARTITION americas VALUES('US', 'CANADA')
    (
      SUBPARTITION americas_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION americas_2012
        VALUES LESS THAN('2013-Jan-01')
    )
);
```

The `sales` table has three partitions (`europe, asia`, and `americas`). Each partition has two range-defined subpartitions that sort the partitions contents into subpartitions by the value of the `date` column:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2011       | '2012-Jan-01'
 europe         | europe_2012       | '2013-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
 asia           | asia_2012         | '2013-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_2012     | '2013-Jan-01'
(6 rows)
```

The following command adds rows to each subpartition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

A `SELECT` statement confirms that the rows are distributed amongst the subpartions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
      tableoid       | dept_no|part_no| country |        date        |amount
---------------------+--------+-------+---------+--------------------+---
 sales_europe_2012   |      10| 4519b | FRANCE  | 17-JAN-12 00:00:00 | 45000
 sales_europe_2012   |      10| 9519b | ITALY   | 07-JUL-12 00:00:00 | 15000
 sales_europe_2012   |      10| 9519a | FRANCE  | 18-AUG-12 00:00:00 | 650000
 sales_europe_2012   |      10| 9519b | FRANCE  | 18-AUG-12 00:00:00 | 650000
 sales_asia_2012     |      20| 3788a | INDIA   | 01-MAR-12 00:00:00 | 75000
 sales_asia_2012     |      20| 3788a | PAKISTAN| 04-JUN-12 00:00:00 | 37500
 sales_asia_2012     |      20| 3788b | INDIA   | 21-SEP-12 00:00:00 | 5090
 sales_asia_2012     |      20| 4519a | INDIA   | 18-OCT-12 00:00:00 | 650000
 sales_asia_2012     |      20| 4519b | INDIA   | 02-DEC-12 00:00:00 | 5090
 sales_americas_2012 |      40| 9519b | US      | 12-APR-12 00:00:00 | 145000
 sales_americas_2012 |      40| 4577b | US      | 11-NOV-12 00:00:00 | 25000
 sales_americas_2012 |      30| 7588b | CANADA  | 14-DEC-12 00:00:00 | 50000
 sales_americas_2012 |      30| 9519b | CANADA  | 01-FEB-12 00:00:00 | 75000
 sales_americas_2012 |      30| 4519b | CANADA  | 08-APR-12 00:00:00 | 120000
 sales_americas_2012 |      40| 3788a | US      | 12-MAY-12 00:00:00 | 4950
 sales_americas_2012 |      40| 4788a | US      | 23-SEP-12 00:00:00 | 4950
 sales_americas_2012 |      40| 4788b | US      | 09-OCT-12 00:00:00 | 15000
(17 rows)
```

The following command splits the `americas_2012` subpartition into two new subpartitions, and redistributes the contents:

```
    ALTER TABLE sales
      SPLIT SUBPARTITION americas_2012
      AT('2012-Jun-01')
      INTO
       (
         SUBPARTITION americas_p1_2012,
         SUBPARTITION americas_p2_2012
       );
```

After invoking the command, the `americas_2012` subpartition has been deleted; in it's place, the server has created two new subpartitions (`americas_p1_2012` and `americas_p2_2012`):

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2012       | '2013-Jan-01'
 europe         | europe_2011       | '2012-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_p2_2012  | '2013-Jan-01'
 americas       | americas_p1_2012  | '2012-Jun-01'
 asia           | asia_2012         | '2013-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
(7 rows)
```

Querying the `sales` table demonstrates that the subpartition's contents are redistributed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
      tableoid         | dept_no|part_no|country |      date         |amount
-----------------------+--------+-------+--------+-------------------+-------
sales_europe_2012      |      10| 4519b |FRANCE  | 17-JAN-12 00:00:00|  45000
 sales_europe_2012     |      10| 9519b |ITALY   | 07-JUL-12 00:00:00|  15000
 sales_europe_2012     |      10| 9519a |FRANCE  | 18-AUG-12 00:00:00| 650000
 sales_europe_2012     |      10| 9519b |FRANCE  | 18-AUG-12 00:00:00| 650000
 sales_asia_2012       |      20| 3788a |INDIA   | 01-MAR-12 00:00:00|  75000
 sales_asia_2012       |      20| 3788a |PAKISTAN| 04-JUN-12 00:00:00|  37500
 sales_asia_2012       |      20| 3788b |INDIA   | 21-SEP-12 00:00:00|   5090
 sales_asia_2012       |      20| 4519a |INDIA   | 18-OCT-12 00:00:00| 650000
 sales_asia_2012       |      20| 4519b |INDIA   | 02-DEC-12 00:00:00|   5090
 sales_americas_p1_2012|      40| 9519b |US      | 12-APR-12 00:00:00| 145000
 sales_americas_p1_2012|      30| 9519b |CANADA  | 01-FEB-12 00:00:00|  75000
 sales_americas_p1_2012|      30| 4519b |CANADA  | 08-APR-12 00:00:00| 120000
 sales_americas_p1_2012|      40| 3788a |US      | 12-MAY-12 00:00:00|   4950
 sales_americas_p2_2012|      40| 4577b |US      | 11-NOV-12 00:00:00|  25000
 sales_americas_p2_2012|      30| 7588b |CANADA  | 14-DEC-12 00:00:00|  50000
 sales_americas_p2_2012|      40| 4788a |US      | 23-SEP-12 00:00:00|   4950
 sales_americas_p2_2012|      40| 4788b |US      | 09-OCT-12 00:00:00|  15000
(17 rows)
```

13.3.6 ALTER TABLE… EXCHANGE PARTITION

The `ALTER TABLE…EXCHANGE PARTITION` command swaps an existing table with a partition or subpartition. If you plan to add a large quantity of data to a partitioned table, you can use the `ALTER TABLE… EXCHANGE PARTITION` command to implement a bulk load. You can also use the `ALTER TABLE… EXCHANGE PARTITION` command to remove old or unneeded data for storage.

The command syntax is available in two forms. The first form swaps a table for a partition:

```
ALTER TABLE target_table
  EXCHANGE PARTITION target_partition
  WITH TABLE source_table
  [(WITH | WITHOUT) VALIDATION];
```

The second form swaps a table for a subpartition:

```
ALTER TABLE target_table
  EXCHANGE SUBPARTITION target_subpartition
  WITH TABLE source_table
  [(WITH | WITHOUT) VALIDATION];
```

This command makes no distinction between a partition and a subpartition:

- You can exchange a partition with the `EXCHANGE PARTITION` or `EXCHANGE SUBPARTITION` clause.
- You can exchange a subpartition with `EXCHANGE PARTITION` or `EXCHANGE SUBPARTITION` clause.

Description

When the `ALTER TABLE… EXCHANGE PARTITION` command completes, the data originally located in the *`target_partition`* will be located in the *`source_table`*, and the data originally located in the *`source_table`* will be located in the *`target_partition`*.

The `ALTER TABLE… EXCHANGE PARTITION` command can exchange partitions in a `LIST`, `RANGE` or `HASH` partitioned table. The structure of the *`source_table`* must match the structure of the *`target_table`* (both tables must have matching columns and data types), and the data contained within the table must adhere to the partitioning constraints.

Advanced Server accepts the `WITHOUT VALIDATION` clause, but ignores it; the new table is always validated.

You must own a table to invoke `ALTER TABLE… EXCHANGE PARTITION` or `ALTER TABLE… EXCHANGE SUBPARTITION` against that table.

Parameters:

`target_table`

The name (optionally schema-qualified) of the table in which the partition resides.

`target_partition`

The name of the partition or subpartition to be replaced.

`source_table`

The name of the table that will replace the *`target_partition`*.

13.3.6.1 Example - Exchanging a Table for a Partition

The example that follows demonstrates swapping a table for a partition (`americas`) of the `sales` table. You can create the `sales` table with the following command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Use the following command to add sample data to the `sales` table:

```
INSERT INTO sales VALUES
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that only one row resides in the `americas` partition:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no| part_no | country |      date     | amount
---------------+--------+---------+---------+-------------------+-----------
 sales_europe  |      10| 4519b   | FRANCE  | 17-JAN-12 00:00:00|      45000
 sales_europe  |      10| 9519b   | ITALY   | 07-JUL-12 00:00:00|      15000
 sales_europe  |      10| 9519a   | FRANCE  | 18-AUG-12 00:00:00|     650000
 sales_europe  |      10| 9519b   | FRANCE  | 18-AUG-12 00:00:00|     650000
 sales_asia    |      20| 3788a   | INDIA   | 01-MAR-12 00:00:00|      75000
 sales_asia    |      20| 3788a   | PAKISTAN| 04-JUN-12 00:00:00|      37500
 sales_asia    |      20| 3788b   | INDIA   | 21-SEP-12 00:00:00|       5090
 sales_asia    |      20| 4519a   | INDIA   | 18-OCT-12 00:00:00|     650000
 sales_asia    |      20| 4519b   | INDIA   | 02-DEC-12 00:00:00|       5090
 sales_americas|      40| 9519b   | US      | 12-APR-12 00:00:00|     145000
(10 rows)
```

The following command creates a table (`n_america`) that matches the definition of the `sales` table:

```
CREATE TABLE n_america
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
);
```

The following command adds data to the `n_america` table. The data conforms to the partitioning rules of the `americas` partition:

```
INSERT INTO n_america VALUES
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000');
```

The following command swaps the table into the partitioned table:

```
ALTER TABLE sales
  EXCHANGE PARTITION americas
  WITH TABLE n_america;
```

Querying the `sales` table shows that the contents of the `n_america` table has been exchanged for the content of the `americas` partition:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no| part_no | country |        date        | amount
---------------+--------+--------+----------+--------------------+-----------
 sales_europe  |      10| 4519b   | FRANCE   | 17-JAN-12 00:00:00 |      45000
 sales_europe  |      10| 9519b   | ITALY    | 07-JUL-12 00:00:00 |      15000
 sales_europe  |      10| 9519a   | FRANCE   | 18-AUG-12 00:00:00 |     650000
 sales_europe  |      10| 9519b   | FRANCE   | 18-AUG-12 00:00:00 |     650000
 sales_asia    |      20| 3788a   | INDIA    | 01-MAR-12 00:00:00 |      75000
 sales_asia    |      20| 3788a   | PAKISTAN| 04-JUN-12 00:00:00 |      37500
 sales_asia    |      20| 3788b   | INDIA    | 21-SEP-12 00:00:00 |       5090
 sales_asia    |      20| 4519a   | INDIA    | 18-OCT-12 00:00:00 |     650000
 sales_asia    |      20| 4519b   | INDIA    | 02-DEC-12 00:00:00 |       5090
 sales_americas|      40| 9519b   | US       | 12-APR-12 00:00:00 |     145000
 sales_americas|      40| 4577b   | US       | 11-NOV-12 00:00:00 |      25000
 sales_americas|      30| 7588b   | CANADA   | 14-DEC-12 00:00:00 |      50000
 sales_americas|      30| 9519b   | CANADA   | 01-FEB-12 00:00:00 |      75000
 sales_americas|      30| 4519b   | CANADA   | 08-APR-12 00:00:00 |     120000
 sales_americas|      40| 3788a   | US       | 12-MAY-12 00:00:00 |       4950
 sales_americas|      40| 4788a   | US       | 23-SEP-12 00:00:00 |       4950
 sales_americas|      40| 4788b   | US       | 09-OCT-12 00:00:00 |      15000
(17 rows)
```

Querying the `n_america` table shows that the row that was previously stored in the `americas` partition has been moved to the `n_america` table:

```
acctg=# SELECT tableoid::regclass, * FROM n_america;
 tableoid  | dept_no | part_no | country |        date        | amount
-----------+---------+---------+---------+--------------------+------------
 n_america |      40 | 9519b   | US      | 12-APR-12 00:00:00 |     145000
(1 row)
```

13.3.7 ALTER TABLE… MOVE PARTITION

Use the `ALTER TABLE… MOVE PARTITION` command to move a partition or subpartition to a different tablespace. The command takes two forms.

The first form moves a partition to a new tablespace:

```
ALTER TABLE table_name
  MOVE PARTITION partition_name
   TABLESPACE tablespace_name;
```

The second form moves a subpartition to a new tablespace:

```
ALTER TABLE table_name
  MOVE SUBPARTITION subpartition_name
   TABLESPACE tablespace_name;
```

The command syntax makes no distinctions between a partition and a subpartition:

- You can move a partition with the `MOVE PARTITION` or `MOVE SUBPARTITION` clause.

- You can move a subpartition with `MOVE PARTITION` or `MOVE SUBPARTITION` clause.

Description

The `ALTER TABLE…MOVE PARTITION` command moves a partition or subpartition from its current tablespace to a different tablespace. The `ALTER TABLE… MOVE PARTITION` command can move partitions (or subpartitions) of a `LIST`, `RANGE` or `HASH` partitioned (or subpartitioned) table. You must own a table to invoke `ALTER TABLE… MOVE PARTITION` or `ALTER TABLE… MOVE SUBPARTITION`.

Parameters

`table_name`

The name (optionally schema-qualified) of the table in which the partition resides.

`partition_name`

The name of the partition or subpartition to be moved.

tablespace_name

The name of the tablespace to which the partition or subpartition will be moved.

13.3.7.1 Example - Moving a Partition to a Different Tablespace

The following example moves a partition of the `sales` table from one tablespace to another. First, create the `sales` table with the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012 VALUES LESS THAN ('2012-Apr-01'),
  PARTITION q2_2012 VALUES LESS THAN ('2012-Jul-01'),
  PARTITION q3_2012 VALUES LESS THAN ('2012-Oct-01'),
  PARTITION q4_2012 VALUES LESS THAN ('2013-Jan-01') TABLESPACE ts_1,
  PARTITION q1_2013 VALUES LESS THAN ('2013-Mar-01') TABLESPACE ts_2
);
```

Querying the `ALL_TAB_PARTITIONS` view confirms that the partitions reside on the expected servers and tablespaces:

```
acctg=# SELECT partition_name, tablespace_name FROM ALL_TAB_PARTITIONS;
 partition_name | tablespace_name
----------------+-----------------+-----------------
 q1_2013        | ts_2
 q4_2012        | ts_1
 q3_2012        |
 q2_2012        |
 q1_2012        |
(5 rows)
```

After preparing the target tablespace, invoke the `ALTER TABLE… MOVE PARTITION` command to move the `q1_2013` partition from a tablespace named `ts_2` to a tablespace named `ts_3`:

```
ALTER TABLE sales MOVE PARTITION q1_2013 TABLESPACE ts_3;
```

Querying the `ALL_TAB_PARTITIONS` view shows that the move was successful:

```
acctg=# SELECT partition_name, tablespace_name FROM ALL_TAB_PARTITIONS;
```

```
 partition_name | tablespace_name
----------------+-----------------
 q1_2013        | ts_3
 q4_2012        | ts_1
 q3_2012        |
 q2_2012        |
 q1_2012        |
(5 rows)
```

13.3.8 ALTER TABLE… RENAME PARTITION

Use the `ALTER TABLE… RENAME PARTITION` command to rename a table partition. The syntax takes two forms:

```
ALTER TABLE table_name
 RENAME PARTITION partition_name
 TO new_name;
and
ALTER TABLE table_name
 RENAME SUBPARTITION subpartition_name
 TO new_name;
```

This command makes no distinctions between a partition and a subpartition:

- You can rename a partition with the `RENAME PARTITION` or `RENAME SUBPARTITION` clause.

- You can rename a subpartition with `RENAME PARTITION` or `RENAME SUBPARTITION` clause.

Description

The `ALTER TABLE… RENAME PARTITION` and `ALTER TABLE… RENAME SUBPARTITION` commands rename a partition or subpartition. You must own the specified table to invoke `ALTER TABLE… RENAME PARTITION` or `ALTER TABLE… RENAME SUBPARTITION`.

Parameters

`table_name`

The name (optionally schema-qualified) of the table in which the partition resides.

`partition_name`

The name of the partition or subpartition to be renamed.

`new_name`

The new name of the partition or subpartition.

13.3.8.1 Example - Renaming a Partition

The following command creates a list-partitioned table named `sales`:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Query the `ALL_TAB_PARTITIONS` view to display the partition names:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 europe         | 'FRANCE', 'ITALY'
 asia           | 'INDIA', 'PAKISTAN'
 americas       | 'US', 'CANADA'
(3 rows)
```

The following command renames the `americas` partition to `n_america`:

```
ALTER TABLE sales
 RENAME PARTITION americas TO n_america;
```

Querying the `ALL_TAB_PARTITIONS` view demonstrates that the partition has been successfully renamed:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 europe         | 'FRANCE', 'ITALY'
 asia           | 'INDIA', 'PAKISTAN'
 n_america      | 'US', 'CANADA'
(3 rows)
```

13.3.9 DROP TABLE

Use the PostgreSQL `DROP TABLE` command to remove a partitioned table definition, it's partitions and subpartitions, and delete the table contents. The syntax is:

```
    DROP TABLE table_name
```

Parameters

`table_name`

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

Description

The `DROP TABLE` command removes an entire table, and the data that resides in that table. When you delete a table, any partitions or subpartitions (of that table) are deleted as well.

To use the `DROP TABLE` command, you must be the owner of the partitioning root, a member of a group that owns the table, the schema owner, or a database superuser.

Example

To delete a table, connect to the controller node (the host of the partitioning root), and invoke the `DROP TABLE` command. For example, to delete the `sales` table, invoke the following command:

```
    DROP TABLE sales;
```

The server will confirm that the table has been dropped:

```
acctg=# drop table sales;
DROP TABLE
acctg=#
```

For more information about the `DROP TABLE` command, please see the PostgreSQL core documentation at:

http://www.postgresql.org/docs/9.5/static/sql-droptable.html

13.3.10 ALTER TABLE… DROP PARTITION

Use the `ALTER TABLE… DROP PARTITION` command to delete a partition definition, and the data stored in that partition. The syntax is:

```
      ALTER TABLE table_name DROP PARTITION partition_name;
```

Parameters

`table_name`

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

`partition_name`

The name of the partition to be deleted.

Description

The `ALTER TABLE… DROP PARTITION` command deletes a partition and any data stored on that partition. The `ALTER TABLE… DROP PARTITION` command can drop partitions of a `LIST` or `RANGE` partitioned table; please note that this command does not work on a `HASH` partitioned table. When you delete a partition, any subpartitions (of that partition) are deleted as well.

To use the `DROP PARTITION` clause, you must be the owner of the partitioning root, a member of a group that owns the table, or have database superuser or administrative privileges.

13.3.10.1 Example - Deleting a Partition

The example that follows deletes a partition of the `sales` table. Use the following command to create the `sales` table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Querying the `ALL_TAB_PARTITIONS` view displays the partition names:

```
acctg=# SELECT partition_name, server_name, high_value FROM
ALL_TAB_PARTITIONS;
 partition_name | server_name |     high_value
----------------+-------------+---------------------
 europe         | seattle     | 'FRANCE', 'ITALY'
 asia           | chicago     | 'INDIA', 'PAKISTAN'
 americas       | boston      | 'US', 'CANADA'
(3 rows)
```

To delete the americas partition from the `sales` table, invoke the following command:

```
    ALTER TABLE sales DROP PARTITION americas;
```

Querying the `ALL_TAB_PARTITIONS` view demonstrates that the partition has been successfully deleted:

```
acctg=# SELECT partition_name, server_name, high_value FROM
ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(2 rows)
```

13.3.11 ALTER TABLE… DROP SUBPARTITION

Use the `ALTER TABLE… DROP SUBPARTITION` command to drop a subpartition definition, and the data stored in that subpartition. The syntax is:

```
ALTER TABLE table_name DROP SUBPARTITION subpartition_name;
```

Parameters

table_name

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

subpartition_name

The name of the subpartition to be deleted.

Description

The `ALTER TABLE… DROP SUBPARTITION` command deletes a subpartition, and the data stored in that subpartition. To use the `DROP SUBPARTITION` clause, you must be the owner of the partitioning root, a member of a group that owns the table, or have superuser or administrative privileges.

13.3.11.1 Example - Deleting a Subpartition

The example that follows deletes a subpartition of the `sales` table. Use the following command to create the `sales` table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST(country)
  (
    PARTITION first_half_2012 VALUES LESS THAN('01-JUL-2012')
    (
      SUBPARTITION europe VALUES ('ITALY', 'FRANCE'),
      SUBPARTITION americas VALUES ('CANADA', 'US'),
      SUBPARTITION asia VALUES ('PAKISTAN', 'INDIA')
    ),
    PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013')
  );
```

Querying the `ALL_TAB_SUBPARTITIONS` view displays the subpartition names:

```
acctg=# SELECT subpartition_name, high_value, server_name FROM
ALL_TAB_SUBPARTITIONS; subpartition_name |     high_value      | server_name
-------------------+---------------------+-------------
 europe            | 'ITALY', 'FRANCE'   | chicago
 americas          | 'CANADA', 'US'      | seattle
 asia              | 'PAKISTAN', 'INDIA' | boston
(3 rows)
```

To delete the `americas` subpartition from the `sales` table, invoke the following command:

```
    ALTER TABLE sales DROP SUBPARTITION americas;
```

Querying the `ALL_TAB_SUBPARTITIONS` view demonstrates that the subpartition has been successfully deleted:

```
acctg=# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name |     high_value
-------------------+--------------------
 europe            | 'ITALY', 'FRANCE'
 asia              | 'PAKISTAN', 'INDIA'
(2 rows)
```

13.3.12 TRUNCATE TABLE

Use the `TRUNCATE TABLE` command to remove the contents of a table, while preserving the table definition. When you truncate a table, any partitions or subpartitions of that table are also truncated. The syntax is:

```
TRUNCATE TABLE table_name
```

Description

The `TRUNCATE TABLE` command removes an entire table, and the data that resides in that table. When you delete a table, any partitions or subpartitions (of that table) are deleted as well.

To use the `TRUNCATE TABLE` command, you must be the owner of the partitioning root, a member of a group that owns the table, the schema owner, or a database superuser.

Parameters

`table_name`

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

13.3.12.1 Example - Emptying a Table

The example that follows removes the data from the `sales` table. Use the following command to create the `sales` table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Populate the `sales` table with the command:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that the partitions are populated with data:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid   |dept_no | part_no | country  |        date         | amount
--------------+--------+---------+----------+---------------------+-----------
sales_europe  |     10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |     45000
sales_europe  |     10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |     15000
sales_europe  |     10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |    650000
 sales_europe |     10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |    650000
sales_asia    |     20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |     75000
sales_asia    |     20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |     37500
sales_asia    |     20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |      5090
sales_asia    |     20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |    650000
sales_asia    |     20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |      5090
sales_americas|     40 | 9519b   | US       | 12-APR-12 00:00:00 |    145000
sales_americas|     40 | 4577b   | US       | 11-NOV-12 00:00:00 |     25000
sales_americas|     30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |     50000
sales_americas|     30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |     75000
sales_americas|     30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |    120000
sales_americas|     40 | 3788a   | US       | 12-MAY-12 00:00:00 |      4950
sales_americas|     40 | 4788a   | US       | 23-SEP-12 00:00:00 |      4950
sales_americas|     40 | 4788b   | US       | 09-OCT-12 00:00:00 |     15000
(17 rows)
```

To delete the contents of the `sales` table, invoke the following command:

```
    TRUNCATE TABLE sales;
```

Now, querying the `sales` table shows that the data has been removed but the structure is intact:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
 tableoid | dept_no | part_no | country |   date   | amount
```

```
----------+---------+---------+---------+----------+------------
(0 rows)
```

For more information about the `TRUNCATE TABLE` command, please see the PostgreSQL documentation at:

http://www.postgresql.org/docs/9.5/static/sql-truncate.html

13.3.13 ALTER TABLE… TRUNCATE PARTITION

Use the `ALTER TABLE… TRUNCATE PARTITION` command to remove the data from the specified partition, leaving the partition structure intact. The syntax is:

```
ALTER TABLE table_name TRUNCATE PARTITION partition_name
  [{DROP|REUSE} STORAGE]
```

Description

Use the `ALTER TABLE… TRUNCATE PARTITION` command to remove the data from the specified partition, leaving the partition structure intact. When you truncate a partition, any subpartitions of that partition are also truncated.

`ALTER TABLE… TRUNCATE PARTITION` will not cause `ON DELETE` triggers that might exist for the table to fire, but it will fire `ON TRUNCATE` triggers. If an `ON TRUNCATE` trigger is defined for the partition, all `BEFORE TRUNCATE` triggers are fired before any truncation happens, and all `AFTER TRUNCATE` triggers are fired after the last truncation occurs.

You must have the `TRUNCATE` privilege on a table to invoke `ALTER TABLE… TRUNCATE PARTITION`.

Parameters

table_name

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

partition_name

The name of the partition to be deleted.

`DROP STORAGE` and `REUSE STORAGE` are included for compatibility only; the clauses are parsed and ignored.

13.3.13.1 Example - Emptying a Partition

The example that follows removes the data from a partition of the `sales` table. Use the following command to create the `sales` table:

```
CREATE TABLE sales
```

```
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Populate the `sales` table with the command:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that the partitions are populated with data:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date        | amount
----------------+---------+---------+----------+--------------------+--------
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
(17 rows)
```

To delete the contents of the `americas` partition, invoke the following command:

```
ALTER TABLE sales TRUNCATE PARTITION americas;
```

Now, querying the `sales` table shows that the content of the `americas` partition has been removed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | dept_no | part_no | country  |        date        | amount
--------------+---------+---------+----------+--------------------+--------
 sales_europe |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia   |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia   |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia   |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia   |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia   |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(9 rows)
```

While the rows have been removed, the structure of the `americas` partition is still intact:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 europe         | 'FRANCE', 'ITALY'
 asia           | 'INDIA', 'PAKISTAN'
 americas       | 'US', 'CANADA'
(3 rows)
```

13.3.14 ALTER TABLE… TRUNCATE SUBPARTITION

Use the `ALTER TABLE… TRUNCATE SUBPARTITION` command to remove all of the data from the specified subpartition, leaving the subpartition structure intact. The syntax is:

```
ALTER TABLE table_name
  TRUNCATE SUBPARTITION subpartition_name
  [{DROP|REUSE} STORAGE]
```

Description

The `ALTER TABLE… TRUNCATE SUBPARTITION` command removes all data from a specified subpartition, leaving the subpartition structure intact.

`ALTER TABLE… TRUNCATE SUBPARTITION` will not cause `ON DELETE` triggers that might exist for the table to fire, but it will fire `ON TRUNCATE` triggers. If an `ON TRUNCATE` trigger is defined for the subpartition, all `BEFORE TRUNCATE` triggers are fired before any truncation happens, and all `AFTER TRUNCATE` triggers are fired after the last truncation occurs.

You must have the `TRUNCATE` privilege on a table to invoke `ALTER TABLE… TRUNCATE SUBPARTITION`.

Parameters

`table_name`

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

`subpartition_name`

The name of the subpartition to be truncated.

The `DROP STORAGE` and `REUSE STORAGE` clauses are included for compatibility only; the clauses are parsed and ignored.

13.3.14.1 Example - Emptying a Subpartition

The example that follows removes the data from a subpartition of the `sales` table. Use the following command to create the `sales` table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date) SUBPARTITION BY LIST (country)
(
  PARTITION "2011" VALUES LESS THAN('01-JAN-2012')
  (
    SUBPARTITION europe_2011 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2011 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2011 VALUES ('US', 'CANADA')
  ),
  PARTITION "2012" VALUES LESS THAN('01-JAN-2013')
  (
    SUBPARTITION europe_2012 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2012 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2012 VALUES ('US', 'CANADA')
  ),
  PARTITION "2013" VALUES LESS THAN('01-JAN-2015')
  (
    SUBPARTITION europe_2013 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2013 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2013 VALUES ('US', 'CANADA')
  )
);
```

Populate the `sales` table with the command:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2011', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2011', '50000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2011', '4950'),
  (20, '3788a', 'US', '04-Apr-2012', '37500'),
  (40, '4577b', 'INDIA', '11-Jun-2011', '25000'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that the rows have been distributed amongst the subpartitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
      tableoid      | dept_no| part_no| country  |        date       |amount
--------------------+--------+--------+----------+-------------------+-------
```

```
 sales_2011_europe  |      10| 4519b  | FRANCE   | 17-JAN-11 00:00:00|  45000
 sales_2011_asia    |      40| 4577b  | INDIA    | 11-JUN-11 00:00:00|  25000
 sales_2011_americas|      30| 7588b  | CANADA   | 14-DEC-11 00:00:00|  50000
 sales_2011_americas|      40| 3788a  | US       | 12-MAY-11 00:00:00|   4950
 sales_2012_europe  |      10| 9519b  | ITALY    | 07-JUL-12 00:00:00|  15000
 sales_2012_asia    |      20| 3788a  | INDIA    | 01-MAR-12 00:00:00|  75000
 sales_2012_asia    |      20| 3788a  | PAKISTAN | 04-JUN-12 00:00:00|  37500
 sales_2012_asia    |      20| 4519b  | INDIA    | 02-DEC-12 00:00:00|   5090
 sales_2012_americas|      40| 9519b  | US       | 12-APR-12 00:00:00| 145000
 sales_2012_americas|      40| 4577b  | US       | 11-NOV-12 00:00:00| 25000
 sales_2012_americas|      30| 4519b  | CANADA   | 08-APR-12 00:00:00| 120000
 sales_2012_americas|      20| 3788a  | US       | 04-APR-12 00:00:00|  37500
(12 rows)
```

To delete the contents of the `2012_americas` partition, invoke the following command:

```
ALTER TABLE sales TRUNCATE SUBPARTITION "americas_2012";
```

Now, querying the `sales` table shows that the content of the `americas_2012` partition has been removed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
      tableoid      | dept_no|part_no| country  |        date        | amount
--------------------+--------+-------+----------+--------------------+-------
 sales_2011_europe  |      10| 4519b | FRANCE   | 17-JAN-11 00:00:00 |  45000
 sales_2011_asia    |      40| 4577b | INDIA    | 11-JUN-11 00:00:00 |  25000
 sales_2011_americas|      30| 7588b | CANADA   | 14-DEC-11 00:00:00 |  50000
 sales_2011_americas|      40| 3788a | US       | 12-MAY-11 00:00:00 |   4950
 sales_2012_europe  |      10| 9519b | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_2012_asia    |      20| 3788a | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_2012_asia    |      20| 3788a | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_2012_asia    |      20| 4519b | INDIA    | 02-DEC-12 00:00:00 |   5090
(8 rows)
```

While the rows have been removed, the structure of the `2012_americas` partition is still intact:

```
acctg=# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name |     high_value
-------------------+---------------------
 2013_europe       | 'ITALY', 'FRANCE'
 2012_europe       | 'ITALY', 'FRANCE'
 2011_europe       | 'ITALY', 'FRANCE'
 2013_asia         | 'PAKISTAN', 'INDIA'
 2012_asia         | 'PAKISTAN', 'INDIA'
 2011_asia         | 'PAKISTAN', 'INDIA'
 2013_americas     | 'US', 'CANADA'
 2012_americas     | 'US', 'CANADA'
 2011_americas     | 'US', 'CANADA'
(9
rows)
```

13.4 Handling Stray Values in a LIST or RANGE Partitioned Table

A `DEFAULT` or `MAXVALUE` partition or subpartition will capture any rows that do not meet the other partitioning rules defined for a table.

Defining a DEFAULT Partition

A `DEFAULT` partition will capture any rows that do not fit into any other partition in a `LIST` partitioned (or subpartitioned) table. If you do not include a `DEFAULT` rule, any row that does not match one of the values in the partitioning constraints will result in an error. Each `LIST` partition or subpartition may have its own `DEFAULT` rule.

The syntax of a `DEFAULT` rule is:

```
     PARTITION [partition_name] VALUES (DEFAULT)
```

Where *`partition_name`* specifies the name of the partition or subpartition that will store any rows that do not match the rules specified for other partititions.

The last example created a list partitioned table in which the server decided which partition to store the data based upon the value of the `country` column. If you attempt to add a row in which the value of the `country` column contains a value not listed in the rules, Advanced Server reports an error:

```
acctg=# INSERT INTO sales VALUES
acctg-#   (40, '3000x', 'IRELAND', '01-Mar-2012', '45000');
ERROR:  inserted partition key does not map to any partition
```

The following example creates the same table, but adds a `DEFAULT` partition. The server will store any rows that do not match a value specified in the partitioning rules for `europe`, `asia`, or `americas` partitions in the `others` partition:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA'),
  PARTITION others VALUES (DEFAULT)
);
```

To test the `DEFAULT` partition, add row with a value in the `country` column that does not match one of the countries specified in the partitioning constraints:

```
INSERT INTO sales VALUES
  (40, '3000x', 'IRELAND', '01-Mar-2012', '45000');
```

Querying the contents of the `sales` table confirms that the previously rejected row is now stored in the `sales_others` partition:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date        | amount
----------------+---------+---------+----------+--------------------+--------
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |   45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |   15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |  650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |  650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |   75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |   37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |    5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |  650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |    5090
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 |  145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |   25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |   50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |   75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |  120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |    4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |    4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |   15000
 sales_others   |      40 | 3000x   | IRELAND  | 01-MAR-12 00:00:00 |   45000
(18 rows)
```

Please note that Advanced Server does not have a way to re-assign the contents of a `DEFAULT` partition or subpartition:

- You cannot use the `ALTER TABLE… ADD PARTITION` command to add a partition to a table with a `DEFAULT` rule, but you can use the `ALTER TABLE… SPLIT PARTITION` command to split an existing partition.

- You cannot use the `ALTER TABLE… ADD SUBPARTITION` command to add a subpartition to a table with a `DEFAULT` rule, but you can use the `ALTER TABLE… SPLIT SUBPARTITION` command to split an existing subpartition.

Defining a MAXVALUE Partition

A `MAXVALUE` partition (or subpartition) will capture any rows that do not fit into any other partition in a range-partitioned (or subpartitioned) table. If you do not include a `MAXVALUE` rule, any row that exceeds the maximum limit specified by the partitioning rules will result in an error. Each partition or subpartition may have its own `MAXVALUE` partition.

The syntax of a `MAXVALUE` rule is:

```
PARTITION [partition_name] VALUES LESS THAN (MAXVALUE)
```

Where *`partition_name`* specifies the name of the partition that will store any rows that do not match the rules specified for other partititions.

The last example created a range-partitioned table in which the data was partitioned based upon the value of the `date` column. If you attempt to add a row with a `date` that exceeds a date listed in the partitioning constraints, Advanced Server reports an error:

```
acctg=# INSERT INTO sales VALUES
acctg-#   (40, '3000x', 'IRELAND', '01-Mar-2013', '45000');
ERROR:  inserted partition key does not map to any partition
```

The following `CREATE TABLE` command creates the same table, but with a `MAXVALUE` partition. Instead of throwing an error, the server will store any rows that do not match the previous partitioning constraints in the `others` partition:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012 VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012 VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012 VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012 VALUES LESS THAN('2013-Jan-01'),
  PARTITION others VALUES LESS THAN (MAXVALUE)
);
```

To test the `MAXVALUE` partition, add a row with a value in the `date` column that exceeds the last date value listed in a partitioning rule. The server will store the row in the `others` partition:

```
     INSERT INTO sales VALUES
       (40, '3000x', 'IRELAND', '01-Mar-2013', '45000');
```

Querying the contents of the `sales` table confirms that the previously rejected row is now stored in the `sales_others` partition :

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | dept_no | part_no | country  |        date        | amount
---------------+---------+---------+----------+--------------------+---------
```

```
 sales_q1_2012 |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |    45000
 sales_q1_2012 |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |    75000
 sales_q1_2012 |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |    75000
 sales_q2_2012 |      40 | 9519b   | US       | 12-APR-12 00:00:00 |   145000
 sales_q2_2012 |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |    37500
 sales_q2_2012 |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |   120000
 sales_q2_2012 |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |     4950
 sales_q3_2012 |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |    15000
 sales_q3_2012 |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |   650000
 sales_q3_2012 |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |   650000
 sales_q3_2012 |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |     5090
 sales_q3_2012 |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |     4950
 sales_q4_2012 |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |    25000
 sales_q4_2012 |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |    50000
 sales_q4_2012 |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |    15000
 sales_q4_2012 |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |   650000
 sales_q4_2012 |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |     5090
 sales_others  |      40 | 3000x   | IRELAND  | 01-MAR-13 00:00:00 |    45000
(18 rows)
```

Please note that Advanced Server does not have a way to re-assign the contents of a `MAXVALUE` partition or subpartition:

- You cannot use the `ALTER TABLE… ADD PARTITION` statement to add a partition to a table with a `MAXVALUE` rule, but you can use the `ALTER TABLE… SPLIT PARTITION` statement to split an existing partition.

- You cannot use the `ALTER TABLE… ADD SUBPARTITION` statement to add a subpartition to a table with a `MAXVALUE` rule , but you can split an existing subpartition with the `ALTER TABLE… SPLIT SUBPARTITION` statement.

13.5 Specifying Multiple Partitioning Keys in a RANGE Partitioned Table

You can often improve performance by specifying multiple key columns for a `RANGE` partitioned table. If you often select rows using comparison operators (based on a greater-than or less-than value) on a small set of columns, consider using those columns in `RANGE` partitioning rules.

Specifying Multiple Keys in a Range-Partitioned Table

Range-partitioned table definitions may include multiple columns in the partitioning key. To specify multiple partitioning keys for a range-partitioned table, include the column names in a comma-separated list after the `PARTITION BY RANGE` clause:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  sale_year    number,
  sale_month   number,
  sale_day     number,
  amount      number
)
PARTITION BY RANGE(sale_year, sale_month)
(
  PARTITION q1_2012
    VALUES LESS THAN(2012, 4),
  PARTITION q2_2012
    VALUES LESS THAN(2012, 7),
  PARTITION q3_2012
    VALUES LESS THAN(2012, 10),
  PARTITION q4_2012
    VALUES LESS THAN(2013, 1)
);
```

If a table is created with multiple partitioning keys, you must specify multiple key values when querying the table to take full advantage of partition pruning:

```
acctg=# EXPLAIN SELECT * FROM sales WHERE sale_year = 2012 AND sale_month =
8;
                                   QUERY PLAN
-----------------------------------------------------------------------------
Result  (cost=0.00..14.35 rows=2 width=250)
   ->  Append  (cost=0.00..14.35 rows=2 width=250)
         ->  Seq Scan on sales  (cost=0.00..0.00 rows=1 width=250)
               Filter: ((sale_year = 2012::numeric) AND (sale_month =
8::numeric))
         ->  Seq Scan on sales_q3_2012 sales  (cost=0.00..14.35 rows=1
width=250)
```

```
                Filter: ((sale_year = 2012::numeric) AND (sale_month =
8::numeric))
(6 rows)
```

Since all rows with a value of `8` in the `sale_month` column and a value of `2012` in the `sale_year` column will be stored in the `q3_2012` partition, Advanced Server searches only that partition.

13.6 Retrieving Information about a Partitioned Table

Advanced Server provides five system catalog views that you can use to view information about the structure of partitioned tables.

Querying the Partitioning Views

You can query the following views to retrieve information about partitioned and subpartitioned tables:

- `ALL_PART_TABLES`
- `ALL_TAB_PARTITIONS`
- `ALL_TAB_SUBPARTITIONS`
- `ALL_PART_KEY_COLUMNS`
- `ALL_SUBPART_KEY_COLUMNS`

The structure of each view is explained in Section 13.5.1, *Table Partitioning Views*. If you are using the EDB-PSQL client, you can also discover the structure of a view by entering:

```
\d view_name
```

Where *`view_name`* specifies the name of the table partitioning view.

Querying a view can provide information about the structure of a partitioned or subpartitioned table. For example, the following code snippit displays the system-assigned names of a subpartitioned table:

```
acctg=# SELECT subpartition_name, partition_name FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name | partition_name
-------------------+----------------
 SYS_SUBP107       | americas
 SYS_SUBP104       | asia
 SYS_SUBP101       | europe
 SYS_SUBP108       | americas
 SYS_SUBP105       | asia
 SYS_SUBP102       | europe
 SYS_SUBP109       | americas
 SYS_SUBP106       | asia
 SYS_SUBP103       | europe
(9 rows)
```

13.6.1 Table Partitioning Views - Reference

Query the following catalog views, compatible with Oracle databases, to review detailed information about your partitioned tables.

13.6.1.1 ALL_PART_TABLES

The following table lists the information available in the `ALL_PART_TABLES` view:

Column	Type	Description
`owner`	`name`	The owner of the table.
`table_name`	`name`	The name of the table.
`schema_name`	`name`	The schema in which the table resides.
`partitioning_type`	`text`	RANGE, LIST or HASH
`subpartitioning_type`	`text`	RANGE, LIST, HASH, or NONE
`partition_count`	`bigint`	The number of partitions.
`def_subpartition_count`	`integer`	The default subpartition count - this will always be 0.
`partitioning_key_count`	`integer`	The number of columns listed in the partition by clause.
`subpartitioning_key_count`	`integer`	The number of columns in the subpartition by clause.
`status`	`character varying(8)`	This column will always be VALID.
`def_tablespace_name`	`character varying(30)`	This column will always be NULL.
`def_pct_free`	`numeric`	This column will always be NULL.
`def_pct_used`	`numeric`	This column will always be NULL.
`def_ini_trans`	`numeric`	This column will always be NULL.
`def_max_trans`	`numeric`	This column will always be NULL.
`def_initial_extent`	`character varying(40)`	This column will always be NULL.
`def_next_extent`	`character varying(40)`	This column will always be NULL.
`def_min_extents`	`character varying(40)`	This column will always be NULL.
`def_max_extents`	`character varying(40)`	This column will always be NULL.
`def_pct_increase`	`character varying(40)`	This column will always be NULL.
`def_freelists`	`numeric`	This column will always be NULL.
`def_freelist_groups`	`numeric`	This column will always be NULL.
`def_logging`	`character varying(7)`	This column will always be YES
`def_compression`	`character varying(8)`	This column will always be NONE
`def_buffer_pool`	`character varying(7)`	This column will always be DEFAULT
`ref_ptn_constraint_name`	`character varying(30)`	This column will always be NULL
`interval`	`character varying(1000)`	This column will always be NULL

13.6.1.2 ALL_TAB_PARTITIONS

The following table lists the information available in the `ALL_TAB_PARTITIONS` view:

Column	Type	Description
`table_owner`	`name`	The owner of the table.
`table_name`	`name`	The name of the table.
`schema_name`	`name`	The schema in which the table resides.
`composite`	`text`	YES if the table is subpartioned; NO if it is not subpartitioned.
`partition_name`	`name`	The name of the partition.
`subpartition_count`	`bigint`	The number of subpartitions for this partition.
`high_value`	`text`	The partition limit for RANGE partitions, or the partition value for LIST partitions.
`high_value_length`	`integer`	The length of high_value.
`partition_position`	`integer`	The ordinal position of this partition.
`tablespace_name`	`name`	The tablespace in which this partition resides.
`pct_free`	`numeric`	This column will always be 0.
`pct_used`	`numeric`	This column will always be 0.
`ini_trans`	`numeric`	This column will always be 0.
`max_trans`	`numeric`	This column will always be 0.
`initial_extent`	`numeric`	This column will always be NULL.
`next_extent`	`numeric`	This column will always be NULL.
`min_extent`	`numeric`	This column will always be 0.
`max_extent`	`numeric`	This column will always be 0.
`pct_increase`	`numeric`	This column will always be 0.
`freelists`	`numeric`	This column will always be NULL
`freelist_groups`	`numeric`	This column will always be NULL
`logging`	`character varying(7)`	This column will always be YES.
`compression`	`character varying(8)`	This column will always be NONE.
`num_rows`	`numeric`	The approx. number of rows in this partition.
`blocks`	`integer`	The approx. number of blocks in this partition.
`empty_blocks`	`numeric`	This column will always be NULL
`avg_space`	`numeric`	This column will always be NULL
`chain_cnt`	`numeric`	This column will always be NULL
`avg_row_len`	`numeric`	This column will always be NULL
`sample_size`	`numeric`	This column will always be NULL
`last_analyzed`	`timestamp without time zone`	This column will always be NULL
`buffer_pool`	`character varying(7)`	This column will always be NULL
`global_stats`	`character varying(3)`	This column will always be YES.
`user_stats`	`character varying(3)`	This column will always be NO.
`backing_table`	`regclass`	OID of the backing table for this partition.
`server_name`	`name`	The name of the server on which the partition resides.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

13.6.1.3 ALL_TAB_SUBPARTITIONS

The following table lists the information available in the `ALL_TAB_SUBPARTITIONS` view:

Column	Type	Description
`table_owner`	`name`	The name of the owner of the table.
`table_name`	`name`	The name of the table.
`schema_name`	`name`	The name of the schema in which the table resides.
`partition_name`	`name`	The name of the partition.
`high_value`	`text`	The subpartition limit for RANGE subpartitions, or the subpartition value for LIST subpartitions.
`high_value_length`	`integer`	The length of high_value.
`subpartition_name`	`name`	The name of the subpartition.
`subpartition_position`	`integer`	The ordinal position of this subpartition.
`tablespace_name`	`name`	The tablespace in which this subpartition resides.
`pct_free`	`numeric`	This column will always be 0.
`pct_used`	`numeric`	This column will always be 0.
`ini_trans`	`numeric`	This column will always be 0.
`max_trans`	`numeric`	This column will always be 0.
`initial_extent`	`numeric`	This column will always be NULL.
`next_extent`	`numeric`	This column will always be NULL.
`min_extent`	`numeric`	This column will always be 0.
`max_extent`	`numeric`	This column will always be 0.
`pct_increase`	`numeric`	This column will always be 0.
`freelists`	`numeric`	This column will always be NULL.
`freelist_groups`	`numeric`	This column will always be NULL.
`logging`	`character varying(7)`	This column will always be YES.
`compression`	`character varying(8)`	This column will always be NONE.
`num_rows`	`numeric`	The approx. number of rows in this subpartition.
`blocks`	`integer`	The approx. number of blocks in this subpartition.
`empty_blocks`	`numeric`	This column will always be NULL.
`avg_space`	`numeric`	This column will always be NULL.
`chain_cnt`	`numeric`	This column will always be NULL.
`avg_row_len`	`numeric`	This column will always be NULL.
`sample_size`	`numeric`	This column will always be NULL.
`last_analyzed`	`timestamp without time zone`	This column will always be NULL.
`buffer_pool`	`character varying(7)`	This column will always be NULL.
`global_stats`	`character varying(3)`	This column will always be YES.
`user_stats`	`character varying(3)`	This column will always be NO.
`backing_table`	`regclass`	OID of the backing table for this subpartition.
`server_name`	`name`	The name of the server on which the subpartition resides.

13.6.1.4 ALL_PART_KEY_COLUMNS

The following table lists the information available in the `ALL_PART_KEY_COLUMNS` view:

Column	Type	Description
`owner`	`name`	The name of the table owner.
`name`	`name`	The name of the table.
`schema`	`name`	The name of the schema on which the table resides.
`object_type`	`character(5)`	This column will always be TABLE.
`column_name`	`name`	The name of the partitioning key column.
`column_position`	`integer`	The position of this column within the partitioning key (the first column has a column position of 1, the second column has a column position of 2...)

13.6.1.5 ALL_SUBPART_KEY_COLUMNS

The following table lists the information available in the `ALL_SUBPART_KEY_COLUMNS` view:

Column	Type	Description
`owner`	`name`	The name of the table owner.
`name`	`name`	The name of the table.
`schema`	`name`	The name of the schema on which the table resides.
`object_type`	`character(5)`	This column will always be TABLE.
`column_name`	`name`	The name of the partitioning key column.
`column_position`	`integer`	The position of this column within the subpartitioning key (the first column has a column position of 1, the second column has a column position of 2...)

14 ECPGPlus

EnterpriseDB has enhanced ECPG (the PostgreSQL pre-compiler) to create ECPGPlus. ECPGPlus allows you to include embedded SQL commands in C applications; when you use ECPGPlus to compile an application that contains embedded SQL commands, the SQL code is syntax-checked and translated into C.

ECPGPlus supports Pro*C compatible syntax in C programs when connected to an Advanced Server database. ECPGPlus supports:

- Oracle Dynamic SQL – Method 4 (ODS-M4).
- Pro*C compatible anonymous blocks.
- A `CALL` statement compatible with Oracle databases.

As part of ECPGPlus's Pro*C compatibility, you do not need to include the `BEGIN DECLARE SECTION` and `END DECLARE SECTION` directives.

For more information about using ECPGPlus, please see the EDB Postgres Advanced Server ECPG Connector Guide available from the EnterpriseDB website at:

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

14.1 C-preprocessor Directives

The ECPGPlus C-preprocessor enforces two behaviors that are dependent on the mode in which you invoke ECPGPlus:

- `PROC` mode
- non-`PROC` mode

Compiling in PROC mode

In `PROC` mode, ECPGPlus allows you to:

- Declare host variables outside of an `EXEC SQL BEGIN/END DECLARE SECTION.`
- Use any C variable as a host variable as long as it is of a data type compatible with ECPG.

When you invoke ECPGPlus in `PROC` mode (by including the `–C PROC` keywords), the ECPG compiler honors the following C-preprocessor directives:

```
#include
#if expression
#ifdef symbolName
#ifndef symbolName
#else
#elif expression
#endif
#define symbolName expansion
#define symbolName([macro arguments]) expansion
#undef symbolName
#defined(symbolName)
```

Pre-processor directives are used to effect or direct the code that is received by the compiler. For example, using the following code sample:

```
#if HAVE_LONG_LONG == 1
#define BALANCE_TYPE long long
#else
#define BALANCE_TYPE double
#endif
...
BALANCE_TYPE customerBalance;
```

If you invoke ECPGPlus with the following command-line arguments:

```
ecpg –C PROC –DHAVE_LONG_LONG=1
```

ECPGPlus will copy the entire fragment (without change) to the output file, but will only send the following tokens to the ECPG parser:

```
long long customerBalance;
```

On the other hand, if you invoke ECPGPlus with the following command-line arguments:

```
ecpg -C PROC -DHAVE_LONG_LONG=0
```

The ECPG parser will receive the following tokens:

```
double customerBalance;
```

If your code uses preprocessor directives to filter the code that is sent to the compiler, the complete code is retained in the original code, while the ECPG parser sees only the processed token stream.

Compiling in non-PROC mode

If you do not include the `-C PROC` command-line option:

- C preprocessor directives are copied to the output file without change.
- You must declare the type and name of each C variable that you intend to use as a host variable within an `EXEC SQL BEGIN/END DECLARE` section.

When invoked in non-`PROC` mode, ECPG implements the behavior described in the PostgreSQL Core documentation, available at:

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

14.2 Supported C Data Types

An ECPGPlus application must deal with two sets of data types: SQL data types (such as `SMALLINT`, `DOUBLE PRECISION` and `CHARACTER VARYING`) and C data types (like `short`, `double` and `varchar[n]`). When an application fetches data from the server, ECPGPlus will map each SQL data type to the type of the C variable into which the data is returned.

In general, ECPGPlus can convert most SQL server types into similar C types, but not all combinations are valid. For example, ECPGPlus will try to convert a SQL character value into a C integer value, but the conversion may fail (at execution time) if the SQL character value contains non-numeric characters.

The reverse is also true; when an application sends a value to the server, ECPGPlus will try to convert the C data type into the required SQL type. Again, the conversion may fail (at execution time) if the C value cannot be converted into the required SQL type.

ECPGPlus can convert any SQL type into C character values (`char[n]` or `varchar[n]`). Although it is safe to convert any SQL type to/from `char[n]` or `varchar[n]`, it is often convenient to use more natural C types such as `int`, `double`, or `float`. The supported C data types are:

- `short`
- `int`
- `unsigned int`
- `long long int`
- `float`
- `double`
- `char[n+1]`
- `varchar[n+1]`
- `bool`
- and any equivalent created by a `typedef`

In addition to the numeric and character types supported by C, the `pgtypeslib` run-time library offers custom data types (and functions to operate on those types) for dealing with date/time and exact numeric values:

- `timestamp`
- `interval`
- `date`
- `decimal`
- `numeric`

To use a data type supplied by `pgtypeslib`, you must `#include` the proper header file.

14.3 Type Codes

The following table contains the type codes for *external* data types. An external data type is used to indicate the type of a C host variable. When an application binds a value to a parameter or binds a buffer to a `SELECT`-list item, the type code in the corresponding SQLDA descriptor (`*descriptor*->T[*column*]`) should be set to one of the following values:

Type Code	Host Variable Type (C Data Type)
`1, 2, 8, 11, 12, 15, 23, 24, 91, 94, 95, 96, 97`	`char[]`
`3`	`int`
`4, 7, 21`	`float`
`5, 6`	`null-terminated string (char[length+1])`
`9`	`varchar`
`22`	`double`
`68`	`unsigned int`

The following table contains the type codes for *internal* data types. An internal type code is used to indicate the type of a value as it resides in the database. The `DESCRIBE SELECT LIST` statement populates the data type array (`*descriptor*->T[*column*]`) using the following values.

Internal Type Code	Server Type
1	VARCHAR2
2	NUMBER
8	LONG
11	ROWID
12	DATE
23	RAW
24	LONG RAW
96	CHAR
100	BINARY FLOAT
101	BINARY DOUBLE
104	UROWID
187	TIMESTAMP
188	TIMESTAMP W/TIMEZONE
189	INTERVAL YEAR TO MONTH
190	INTERVAL DAY TO SECOND
232	TIMESTAMP LOCAL_TZ

14.4 The SQLDA Structure

Oracle Dynamic SQL method 4 uses the SQLDA data structure to hold the data and metadata for a dynamic SQL statement. A SQLDA structure can describe a set of input parameters corresponding to the parameter markers found in the text of a dynamic statement or the result set of a dynamic statement. The layout of the SQLDA structure is:

```
struct SQLDA
{
  int      N;     /* Number of entries              */
  char  **V;     /* Variables                      */
  int     *L;     /* Variable lengths               */
  short  *T;     /* Variable types                 */
  short **I;     /* Indicators                     */
  int      F;     /* Count of variables discovered by DESCRIBE */
  char  **S;     /* Variable names                 */
  short  *M;     /* Variable name maximum lengths  */
  short  *C;     /* Variable name actual lengths   */
  char  **X;     /* Indicator names                */
  short  *Y;     /* Indicator name maximum lengths */
  short  *Z;     /* Indicator name actual lengths  */
};
```

Parameters

`N` – *maximum number of entries*

The `N` structure member contains the maximum number of entries that the SQLDA may describe. This member is populated by the `sqlald()` function when you allocate the SQLDA structure. Before using a descriptor in an `OPEN` or `FETCH` statement, you must set `N` to the *actual* number of values described.

`V` – *data values*

The `V` structure member is a pointer to an array of data values.

> For a `SELECT`-list descriptor, `V` points to an array of values returned by a `FETCH` statement (each member in the array corresponds to a column in the result set).
>
> For a bind descriptor, `V` points to an array of parameter values (you must populate the values in this array before opening a cursor that uses the descriptor).

Your application must allocate the space required to hold each value.

`L` – *length of each data value*

The `L` structure member is a pointer to an array of lengths. Each member of this array must indicate the amount of memory available in the corresponding member of the `V` array. For example, if `V[5]` points to a buffer large enough to hold a 20-byte NULL-terminated string, `L[5]` should contain the value 21 (20 bytes for the characters in the string plus 1 byte for the NULL-terminator). Your application must set each member of the `L` array.

`T` – *data types*

The `T` structure member points to an array of data types, one for each column (or parameter) described by the descriptor.

> For a bind descriptor, you must set each member of the `T` array to tell ECPGPlus the data type of each parameter.
>
> For a `SELECT`-list descriptor, the `DESCRIBE SELECT LIST` statement sets each member of the `T` array to reflect the type of data found in the corresponding column.

You may change any member of the `T` array before executing a `FETCH` statement to force ECPGPlus to convert the corresponding value to a specific data type. For example, if the `DESCRIBE SELECT LIST` statement indicates that a given column is of type `DATE`, you may change the corresponding `T` member to request that the next `FETCH` statement return that value in the form of a NULL-terminated string. Each member of the T array is a numeric type code. The type codes returned by a `DESCRIBE SELECT LIST` statement differ from those expected by a `FETCH` statement. After executing a `DESCRIBE SELECT LIST` statement, each member of `T` encodes a data type *and* a flag indicating whether the corresponding column is nullable. You can use the `sqlnul()` function to extract the type code and nullable flag from a member of the T array. The signature of the `sqlnul()` function is as follows:

```
void sqlnul(unsigned short *valType,
            unsigned short *typeCode,
            int            *isNull)
```

For example, to find the type code and nullable flag for the third column of a descriptor named results, you would invoke `sqlnul()` as follows:

```
sqlnul(&results->T[2], &typeCode, &isNull);
```

`I` – *indicator variables*

The `I` structure member points to an array of indicator variables. This array is allocated for you when your application calls the `sqlald()` function to allocate the descriptor.

For a SELECT-list descriptor, each member of the I array indicates whether the corresponding column contains a NULL (non-zero) or non-NULL (zero) value.

For a bind parameter, your application must set each member of the I array to indicate whether the corresponding parameter value is NULL.

F – *number of entries*

The F structure member indicates how many values are described by the descriptor (the N structure member indicates the *maximum* number of values which may be described by the descriptor; F indicates the actual number of values). The value of the F member is set by ECPGPlus when you execute a DESCRIBE statement. F may be positive, negative, or zero.

For a SELECT-list descriptor, F will contain a positive value if the number of columns in the result set is equal to or less than the maximum number of values permitted by the descriptor (as determined by the N structure member); 0 if the statement is *not* a SELECT statement, or a negative value if the query returns more columns than allowed by the N structure member.

For a bind descriptor, F will contain a positive number if the number of parameters found in the statement is less than or equal to the maximum number of values permitted by the descriptor (as determined by the N structure member); 0 if the statement contains no parameters markers, or a negative value if the statement contains more parameter markers than allowed by the N structure member.

If F contains a positive number (after executing a DESCRIBE statement), that number reflects the count of columns in the result set (for a SELECT-list descriptor) or the number of parameter markers found in the statement (for a bind descriptor). If F contains a negative value, you may compute the absolute value of F to discover how many values (or parameter markers) are required. For example, if F contains -24 after describing a SELECT list, you know that the query returns 24 columns.

S – *column/parameter names*

The S structure member points to an array of NULL-terminated strings.

For a SELECT-list descriptor, the DESCRIBE SELECT LIST statement sets each member of this array to the name of the corresponding column in the result set.

For a bind descriptor, the DESCRIBE BIND VARIABLES statement sets each member of this array to the name of the corresponding bind variable.

In this release, the name of each bind variable is determined by the left-to-right order of the parameter marker within the query - for example, the name of the first parameter is always `?0`, the name of the second parameter is always `?1`, and so on.

`M` – *maximum column/parameter name length*

The `M` structure member points to an array of lengths. Each member in this array specifies the *maximum* length of the corresponding member of the `S` array (that is, `M[0]` specifies the maximum length of the column/parameter name found at `S[0]`). This array is populated by the `sqlald()` function.

`C` – *actual column/parameter name length*

The `C` structure member points to an array of lengths. Each member in this array specifies the *actual* length of the corresponding member of the `S` array (that is, `C[0]` specifies the actual length of the column/parameter name found at `S[0]`).

This array is populated by the `DESCRIBE` statement.

`X` – *indicator variable names*

The `X` structure member points to an array of NULL-terminated strings - each string represents the name of a NULL indicator for the corresponding value.

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

`Y` – *maximum indicator name length*

The `Y` structure member points to an array of lengths. Each member in this array specifies the *maximum* length of the corresponding member of the `X` array (that is, `Y[0]` specifies the maximum length of the indicator name found at `X[0]`).

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

`Z` – *actual indicator name length*

The `Z` structure member points to an array of lengths. Each member in this array specifies the *actual* length of the corresponding member of the `X` array (that is, `Z[0]` specifies the actual length of the indicator name found at `X[0]`).

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

14.5 ECPGPlus Statements

An embedded SQL statement allows your client application to interact with the server, while an embedded directive is an instruction to the ECPGPlus compiler.

You can embed any Advanced Server SQL statement in a C program. Each statement should begin with the keywords `EXEC SQL`, and must be terminated with a semi-colon (`;`). Within the C program, a SQL statement takes the form:

```
EXEC SQL sql_command_body;
```

Where *`sql_command_body`* represents a standard SQL statement. You can use a host variable anywhere that the SQL statement expects a value expression.

ECPGPlus extends the PostgreSQL server-side syntax for some statements; for those statements, syntax differences are outlined in the following reference sections. For a complete reference to the supported syntax of other SQL commands, please see the *PostgreSQL Core Documentation* at:

http://www.postgresql.org/docs/9.5/static/sql-commands.html

14.5.1 ALLOCATE DESCRIPTOR

Use the `ALLOCATE DESCRIPTOR` statement to allocate an SQL descriptor area:

```
EXEC SQL [FOR array_size] ALLOCATE DESCRIPTOR descriptor_name
    [WITH MAX variable_count];
```

Where:

`array_size` is a variable that specifies the number of array elements to allocate for the descriptor. *`array_size`* may be an `INTEGER` value or a host variable.

`descriptor_name` is the host variable that contains the name of the descriptor, or the name of the descriptor. This value may take the form of an identifier, a quoted string literal, or of a host variable.

`variable_count` specifies the maximum number of host variables in the descriptor. The default value of *`variable_count`* is `100`.

The following code fragment allocates a descriptor named `emp_query` that may be processed as an array (`emp_array`):

```
EXEC SQL FOR :emp_array ALLOCATE DESCRIPTOR emp_query;
```

14.5.2 CALL

Use the `CALL` statement to invoke a procedure or function on the server. The `CALL` statement works only on Advanced Server. The `CALL` statement comes in two forms; the first form is used to call a *function*:

```
EXEC SQL CALL program_name '('[actual_arguments]')'
  INTO [[:ret_variable][:ret_indicator]];
```

The second form is used to call a *procedure*:

```
EXEC SQL CALL program_name '('[actual_arguments]')';
```

Where:

`program_name` is the name of the stored procedure or function that the `CALL` statement invokes. The program name may be schema-qualified or package-qualified (or both); if you do not specify the schema or package in which the program resides, ECPGPlus will use the value of `search_path` to locate the program.

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

`:`*`ret_variable`* specifies a host variable that will receive the value returned if the program is a function.

:*`ret_indicator`* specifies a host variable that will receive the indicator value returned, if the program is a function.

For example, the following statement invokes the `get_job_desc` function with the value contained in the :`ename` host variable, and captures the value returned by that function in the :`job` host variable:

```
EXEC SQL CALL get_job_desc(:ename)
  INTO :job;
```

14.5.3 CLOSE

Use the `CLOSE` statement to close a cursor, and free any resources currently in use by the cursor. A client application cannot fetch rows from a closed cursor. The syntax of the `CLOSE` statement is:

```
      EXEC SQL CLOSE [cursor_name];
```

Where:

`cursor_name` is the name of the cursor closed by the statement. The cursor name may take the form of an identifier or of a host variable.

The `OPEN` statement initializes a cursor. Once initialized, a cursor result set will remain unchanged unless the cursor is re-opened. You do not need to `CLOSE` a cursor before re-opening it.

To manually close a cursor named `emp_cursor`, use the command:

```
      EXEC SQL CLOSE emp_cursor;
```

A cursor is automatically closed when an application terminates.

14.5.4 COMMIT

Use the `COMMIT` statement to complete the current transaction, making all changes permanent and visible to other users. The syntax is:

```
      EXEC SQL [AT database_name] COMMIT [WORK]
               [COMMENT 'text'] [COMMENT 'text' RELEASE];
```

Where:

`database_name` is the name of the database (or host variable that contains the name of the database) in which the work resides. This value may take the form of an unquoted string literal, or of a host variable.

For compatibility, ECPGPlus accepts the `COMMENT` clause without error but does *not* store any text included with the `COMMENT` clause.

Include the `RELEASE` clause to close the current connection after performing the commit.

For example, the following command commits all work performed on the `dept` database and closes the current connection:

```
EXEC SQL AT dept COMMIT RELEASE;
```

By default, statements are committed only when a client application performs a `COMMIT` statement. Include the `-t` option when invoking ECPGPlus to specify that a client application should invoke `AUTOCOMMIT` functionality. You can also control `AUTOCOMMIT` functionality in a client application with the following statements:

```
EXEC SQL SET AUTOCOMMIT TO ON
```

and

```
EXEC SQL SET AUTOCOMMIT TO OFF
```

14.5.5 CONNECT

Use the `CONNECT` statement to establish a connection to a database. The `CONNECT` statement is available in two forms: one form is compatible with Oracle databases, the other is not.

The first form is compatible with Oracle databases:

```
EXEC SQL CONNECT
  {{:user_name IDENTIFIED BY :password} | :connection_id}
  [AT database_name]
  [USING :database_string]
  [ALTER AUTHORIZATION :new_password];
```

Where:

`user_name` is a host variable that contains the role that the client application will use to connect to the server.

`password` is a host variable that contains the password associated with that role.

`connection_id` is a host variable that contains a slash-delimited user name and password used to connect to the database.

Include the `AT` clause to specify the database to which the connection is established. *`database_name`* is the name of the database to which the client is connecting; specify the value in the form of a variable, or as a string literal.

Include the `USING` clause to specify a host variable that contains a null-terminated string identifying the database to which the connection will be established.

The `ALTER AUTHORIZATION` clause is supported for syntax compatibility only; ECPGPlus parses the `ALTER AUTHORIZATION` clause, and reports a warning.

Using the first form of the `CONNECT` statement, a client application might establish a connection with a host variable named `user` that contains the identity of the connecting role, and a host variable named `password` that contains the associated password using the following command:

```
EXEC SQL CONNECT :user IDENTIFIED BY :password;
```

A client application could also use the first form of the `CONNECT` statement to establish a connection using a single host variable named `:connection_id`. In the following example, `connection_id` contains the slash-delimited role name and associated password for the user:

```
EXEC SQL CONNECT :connection_id;
```

The syntax of the second form of the `CONNECT` statement is:

```
EXEC SQL CONNECT TO database_name
[AS connection_name] [credentials];
```

Where *`credentials`* is one of the following:

```
USER user_name password
USER user_name IDENTIFIED BY password
USER user_name USING password
```

In the second form:

`database_name` is the name or identity of the database to which the client is connecting. Specify *`database_name`* as a variable, or as a string literal, in one of the following forms:

```
database_name[@hostname][:port]

tcp:postgresql://hostname[:port][/database_name][options]

unix:postgresql://hostname[:port][/database_name][options]
```

Where:

`hostname` is the name or IP address of the server on which the database resides.

`port` is the port on which the server listens.

> You can also specify a value of `DEFAULT` to establish a connection with the default database, using the default role name. If you specify `DEFAULT` as the target database, do not include a *`connection_name`* or *`credentials`*.

`connection_name` is the name of the connection to the database. *`connection_name`* should take the form of an identifier (that is, not a string literal or a variable). You can open multiple connections, by providing a unique *`connection_name`* for each connection.

> If you do not specify a name for a connection, `ecpglib` assigns a name of `DEFAULT` to the connection. You can refer to the connection by name (`DEFAULT`) in any `EXEC SQL` statement.
>
> `CURRENT` is the most recently opened or the connection mentioned in the most-recent `SET CONNECTION TO` statement. If you do not refer to a connection by name in an `EXEC SQL` statement, ECPG assumes the name of the connection to be `CURRENT`.

`user_name` is the role used to establish the connection with the Advanced Server database. The privileges of the specified role will be applied to all commands performed through the connection.

`password` is the password associated with the specified *`user_name`*.

The following code fragment uses the second form of the `CONNECT` statement to establish a connection to a database named `edb`, using the role `alice` and the password associated with that role, `1safepwd`:

```
EXEC SQL CONNECT TO edb AS acctg_conn
  USER 'alice' IDENTIFIED BY '1safepwd';
```

The name of the connection is `acctg_conn`; you can use the connection name when changing the connection name using the `SET CONNECTION` statement.

14.5.6 DEALLOCATE DESCRIPTOR

Use the `DEALLOCATE DESCRIPTOR` statement to free memory in use by an allocated descriptor. The syntax of the statement is:

```
EXEC SQL DEALLOCATE DESCRIPTOR descriptor_name
```

Where:

descriptor_name is the name of the descriptor. This value may take the form of a quoted string literal, or of a host variable.

The following example deallocates a descriptor named `emp_query`:

```
EXEC SQL DEALLOCATE DESCRIPTOR emp_query;
```

14.5.7 DECLARE CURSOR

Use the `DECLARE CURSOR` statement to define a cursor. The syntax of the statement is:

```
EXEC SQL [AT database_name] DECLARE cursor_name CURSOR FOR
(select_statement | statement_name);
```

Where:

database_name is the name of the database on which the cursor operates. This value may take the form of an identifier or of a host variable. If you do not specify a database name, the default value of *database_name* is the default database.

cursor_name is the name of the cursor.

select_statement is the text of the `SELECT` statement that defines the cursor result set; the `SELECT` statement cannot contain an `INTO` clause.

statement_name is the name of a SQL statement or block that defines the cursor result set.

The following example declares a cursor named `employees`:

```
EXEC SQL DECLARE employees CURSOR FOR
  SELECT
    empno, ename, sal, comm
  FROM
    emp;
```

The cursor generates a result set that contains the employee number, employee name, salary and commission for each employee record that is stored in the `emp` table.

14.5.8 DECLARE DATABASE

Use the `DECLARE DATABASE` statement to declare a database identifier for use in subsequent SQL statements (for example, in a `CONNECT` statement). The syntax is:

```
EXEC SQL DECLARE database_name DATABASE;
```

Where:

`database_name` specifies the name of the database.

The following example demonstrates declaring an identifier for the `acctg` database:

```
EXEC SQL DECLARE acctg DATABASE;
```

After invoking the command declaring `acctg` as a database identifier, the `acctg` database can be referenced by name when establishing a connection or in `AT` clauses.

This statement has no effect and is provided for Pro*C compatibility only.

14.5.9 DECLARE STATEMENT

Use the `DECLARE STATEMENT` directive to declare an identifier for an SQL statement. Advanced Server supports two versions of the `DECLARE STATEMENT` directive:

```
EXEC SQL [database_name] DECLARE statement_name STATEMENT;
```

and

```
EXEC SQL DECLARE STATEMENT statement_name;
```

Where:

`statement_name` specifies the identifier associated with the statement.

`database_name` specifies the name of the database. This value may take the form of an identifier or of a host variable that contains the identifier.

A typical usage sequence that includes the `DECLARE STATEMENT` directive might be:

```
EXEC SQL DECLARE give_raise STATEMENT;      // give_raise
is now a statement handle (not prepared)
```

```
EXEC SQL PREPARE give_raise FROM :stmtText; // give_raise
is now associated with a statement
EXEC SQL EXECUTE give_raise;
```

This statement has no effect and is provided for Pro*C compatibility only.

14.5.10 DELETE

Use the `DELETE` statement to delete one or more rows from a table. The syntax for the ECPGPlus `DELETE` statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that an expression is allowed. The syntax is:

```
[FOR exec_count] DELETE FROM [ONLY] table [[AS] alias]
  [USING using_list]
  [WHERE condition | WHERE CURRENT OF cursor_name]
  [{RETURNING|RETURN} * | output_expression [[ AS] output_name]
[, ...] INTO host_variable_list ]
```

Where:

Include the `FOR` *`exec_count`* clause to specify the number of times the statement will execute; this clause is valid only if the `VALUES` clause references an array or a pointer to an array.

`table` is the name (optionally schema-qualified) of an existing table. Include the `ONLY` clause to limit processing to the specified table; if you do not include the `ONLY` clause, any tables inheriting from the named table are also processed.

`alias` is a substitute name for the target table.

`using_list` is a list of table expressions, allowing columns from other tables to appear in the `WHERE` condition.

Include the `WHERE` clause to specify which rows should be deleted. If you do not include a `WHERE` clause in the statement, `DELETE` will delete all rows from the table, leaving the table definition intact.

`condition` is an expression, host variable or parameter marker that returns a value of type `BOOLEAN`. Those rows for which *`condition`* returns true will be deleted.

`cursor_name` is the name of the cursor to use in the `WHERE CURRENT OF` clause; the row to be deleted will be the one most recently fetched from this cursor. The cursor must be a non-grouping query on the `DELETE` statements target table. You cannot specify `WHERE CURRENT OF` in a `DELETE` statement that includes a Boolean condition.

The `RETURN`/`RETURNING` clause specifies an *`output_expression`* or *`host_variable_list`* that is returned by the `DELETE` command after each row is deleted:

> *`output_expression`* is an expression to be computed and returned by the `DELETE` command after each row is deleted. *`output_name`* is the name of the returned column; include `*` to return all columns.
>
> *`host_variable_list`* is a comma-separated list of host variables and optional indicator variables. Each host variable receives a corresponding value from the `RETURNING` clause.

For example, the following statement deletes all rows from the `emp` table where the `sal` column contains a value greater than the value specified in the host variable, `:max_sal`:

```
DELETE FROM emp WHERE sal > :max_sal;
```

For more information about using the `DELETE` statement, please see the PostgreSQL Core documentation available at:

http://www.postgresql.org/docs/9.5/static/sql-delete.html

14.5.11 DESCRIBE

Use the `DESCRIBE` statement to find the number of input values required by a prepared statement or the number of output values returned by a prepared statement. The `DESCRIBE` statement is used to analyze a SQL statement whose shape is unknown at the time you write your application.

The `DESCRIBE` statement populates an `SQLDA` descriptor; to populate a SQL descriptor, use the `ALLOCATE DESCRIPTOR` and `DESCRIBE…DESCRIPTOR` statements.

```
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name INTO
descriptor;
```

or

```
EXEC SQL DESCRIBE SELECT LIST FOR statement_name INTO
descriptor;
```

Where:

`statement_name` is the identifier associated with a prepared SQL statement or PL/SQL block.

descriptor is the name of C variable of type SQLDA*. You must allocate the space for the descriptor by calling sqlald() (and initialize the descriptor) before executing the DESCRIBE statement.

When you execute the first form of the DESCRIBE statement, ECPG populates the given descriptor with a description of each input variable *required* by the statement. For example, given two descriptors:

```
SQLDA *query_values_in;
SQLDA *query_values_out;
```

You might prepare a query that returns information from the emp table:

```
EXEC SQL PREPARE get_emp FROM
  "SELECT ename, empno, sal FROM emp WHERE empno = ?";
```

The command requires one input variable (for the parameter marker (?)).

```
EXEC SQL DESCRIBE BIND VARIABLES
  FOR get_emp INTO query_values_in;
```

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.

When you execute the second form, ECPG populates the given descriptor with a description of each value *returned* by the statement. For example, the following statement returns three values:

```
EXEC SQL DESCRIBE SELECT LIST
  FOR get_emp  INTO query_values_out;
```

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

Before *executing* the statement, you must bind a variable for each input value and a variable for each output value. The variables that you bind for the input values specify the actual values used by the statement. The variables that you bind for the output values tell ECPGPlus where to put the values when you execute the statement.

This is alternate Pro*C compatible syntax for the DESCRIBE DESCRIPTOR statement.

14.5.12 DESCRIBE DESCRIPTOR

Use the DESCRIBE DESCRIPTOR statement to retrieve information about a SQL statement, and store that information in a SQL descriptor. Before using DESCRIBE

DESCRIPTOR, you must allocate the descriptor with the ALLOCATE DESCRIPTOR statement. The syntax is:

```
EXEC SQL DESCRIBE [INPUT | OUTPUT] statement_identifier
  USING [SQL] DESCRIPTOR descriptor_name;
```

Where:

statement_name is the name of a prepared SQL statement.

descriptor_name is the name of the descriptor. *descriptor_name* can be a quoted string value or a host variable that contains the name of the descriptor.

If you include the INPUT clause, ECPGPlus populates the given descriptor with a description of each input variable *required* by the statement.

For example, given two descriptors:

```
EXEC SQL ALLOCATE DESCRIPTOR query_values_in;
EXEC SQL ALLOCATE DESCRIPTOR query_values_out;
```

You might prepare a query that returns information from the emp table:

```
EXEC SQL PREPARE get_emp FROM
  "SELECT ename, empno, sal FROM emp WHERE empno = ?";
```

The command requires one input variable (for the parameter marker (?)).

```
EXEC SQL DESCRIBE INPUT get_emp USING
'query_values_in';
```

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.

If you do not specify the INPUT clause, DESCRIBE DESCRIPTOR populates the specified descriptor with the values returned by the statement.

If you include the OUTPUT clause, ECPGPlus populates the given descriptor with a description of each value *returned* by the statement.

For example, the following statement returns three values:

```
EXEC SQL DESCRIBE OUTPUT FOR get_emp USING
'query_values_out';
```

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

14.5.13 DISCONNECT

Use the DISCONNECT statement to close the connection to the server. The syntax is:

```
EXEC SQL DISCONNECT [connection_name][CURRENT][DEFAULT][ALL];
```

Where:

connection_name is the connection name specified in the CONNECT statement used to establish the connection. If you do not specify a connection name, the current connection is closed.

Include the CURRENT keyword to specify that ECPGPlus should close the most-recently used connection.

Include the DEFAULT keyword to specify that ECPGPlus should close the connection named DEFAULT. If you do not specify a name when opening a connection, ECPGPlus assigns the name, DEFAULT, to the connection.

Include the ALL keyword to instruct ECPGPlus to close all active connections.

The following example creates a connection (named hr_connection) that connects to the hr database, and then disconnects from the connection:

```
/* client.pgc*/
int main()
{
    EXEC SQL CONNECT TO hr AS connection_name;
    EXEC SQL DISCONNECT connection_name;
    return(0);
}
```

14.5.14 EXECUTE

Use the EXECUTE statement to execute a statement previously prepared using an EXEC SQL PREPARE statement. The syntax is:

```
EXEC SQL [FOR array_size] EXECUTE statement_name
  [USING {DESCRIPTOR SQLDA_descriptor
  |:host_variable [[INDICATOR] :indicator_variable]}];
```

Where:

array_size is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you omit the FOR clause, the statement is executed once for each member of the array.

statement_name specifies the name assigned to the statement when the statement was created (using the EXEC SQL PREPARE statement).

Include the USING clause to supply values for parameters within the prepared statement:

> Include the DESCRIPTOR *SQLDA_descriptor* clause to provide an SQLDA descriptor value for a parameter.
>
> Use a *host_variable* (and an optional *indicator_variable*) to provide a user-specified value for a parameter.

The following example creates a prepared statement that inserts a record into the emp table:

```
EXEC SQL PREPARE add_emp (numeric, text, text, numeric) AS
    INSERT INTO emp VALUES($1, $2, $3, $4);
```

Each time you invoke the prepared statement, provide fresh parameter values for the statement:

```
EXEC SQL EXECUTE add_emp USING 8000, 'DAWSON', 'CLERK',
7788;
EXEC SQL EXECUTE add_emp USING 8001, 'EDWARDS', 'ANALYST',
7698;
```

14.5.15 EXECUTE DESCRIPTOR

Use the EXECUTE statement to execute a statement previously prepared by an EXEC SQL PREPARE statement, using an SQL descriptor. The syntax is:

```
EXEC SQL [FOR array_size] EXECUTE statement_identifier
  [USING [SQL] DESCRIPTOR descriptor_name]
  [INTO [SQL] DESCRIPTOR descriptor_name];
```

Where:

array_size is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you omit the FOR clause, the statement is executed once for each member of the array.

statement_identifier specifies the identifier assigned to the statement with the EXEC SQL PREPARE statement.

Include the USING clause to specify values for any input parameters required by the prepared statement.

Include the INTO clause to specify a descriptor into which the EXECUTE statement will write the results returned by the prepared statement.

descriptor_name specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

The following example executes the prepared statement, give_raise, using the values contained in the descriptor stmtText:

```
EXEC SQL PREPARE give_raise FROM :stmtText;
EXEC SQL EXECUTE give_raise USING DESCRIPTOR :stmtText;
```

14.5.16 EXECUTE...END EXEC

Use the EXECUTE…END-EXEC statement to embed an anonymous block into a client application. The syntax is:

```
EXEC SQL [AT database_name] EXECUTE anonymous_block END-EXEC;
```

Where:

database_name is the database identifier or a host variable that contains the database identifier. If you omit the AT clause, the statement will be executed on the current default database.

anonymous_block is an inline sequence of PL/pgSQL or SPL statements and declarations. You may include host variables and optional indicator variables within the block; each such variable is treated as an IN/OUT value.

The following example executes an anonymous block:

```
EXEC SQL EXECUTE
  BEGIN
    IF (current_user = :admin_user_name) THEN
      DBMS_OUTPUT.PUT_LINE('You are an administrator');
    END IF;
END-EXEC;
```

Please Note: the `EXECUTE…END EXEC` statement is supported only by Advanced Server.

14.5.17 EXECUTE IMMEDIATE

Use the `EXECUTE IMMEDIATE` statement to execute a string that contains a SQL command. The syntax is:

```
EXEC SQL [AT database_name] EXECUTE IMMEDIATE command_text;
```

Where:

database_name is the database identifier or a host variable that contains the database identifier. If you omit the `AT` clause, the statement will be executed on the current default database.

command_text is the command executed by the `EXECUTE IMMEDIATE` statement.

This dynamic SQL statement is useful when you don't know the text of an SQL statement (ie., when writing a client application). For example, a client application may prompt a (trusted) user for a statement to execute. After the user provides the text of the statement as a string value, the statement is then executed with an `EXECUTE IMMEDIATE` command.

The statement text may not contain references to host variables. If the statement may contain parameter markers or returns one or more values, you must use the `PREPARE` and `DESCRIBE` statements.

The following example executes the command contained in the `:command_text` host variable:

```
EXEC SQL EXECUTE IMMEDIATE :command_text;
```

14.5.18 FETCH

Use the `FETCH` statement to return rows from a cursor into an SQLDA descriptor or a target list of host variables. Before using a `FETCH` statement to retrieve information from a cursor, you must prepare the cursor using `DECLARE` and `OPEN` statements. The statement syntax is:

```
EXEC SQL [FOR array_size] FETCH cursor
  { USING DESCRIPTOR SQLDA_descriptor }|{ INTO target_list };
```

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor from which rows are being fetched, or a host variable that contains the name of the cursor.

If you include a USING clause, the FETCH statement will populate the specified SQLDA descriptor with the values returned by the server.

If you include an INTO clause, the FETCH statement will populate the host variables (and optional indicator variables) specified in the target_list.

The following code fragment declares a cursor named employees that retrieves the employee number, name and salary from the emp table:

```
EXEC SQL DECLARE employees CURSOR FOR
    SELECT empno, ename, esal FROM emp;
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO :emp_no, :emp_name, :emp_sal;
```

14.5.19 FETCH DESCRIPTOR

Use the FETCH DESCRIPTOR statement to retrieve rows from a cursor into an SQL descriptor. The syntax is:

```
EXEC SQL [FOR array_size] FETCH cursor
  INTO [SQL] DESCRIPTOR descriptor_name;
```

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor from which rows are fetched, or a host variable that contains the name of the cursor. The client must DECLARE and OPEN the cursor before calling the FETCH DESCRIPTOR statement.

Include the INTO clause to specify an SQL descriptor into which the EXECUTE statement will write the results returned by the prepared statement. *descriptor_name* specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor. Prior to use, the descriptor must be allocated using an `ALLOCATE DESCRIPTOR` statement.

The following example allocates a descriptor named `row_desc` that will hold the description and the values of a specific row in the result set. It then declares and opens a cursor for a prepared statement (`my_cursor`), before looping through the rows in result set, using a `FETCH` to retrieve the next row from the cursor into the descriptor:

```
EXEC SQL ALLOCATE DESCRIPTOR 'row_desc';
EXEC SQL DECLARE my_cursor CURSOR FOR query;
EXEC SQL OPEN my_cursor;

for( row = 0; ; row++ )
{
  EXEC SQL BEGIN DECLARE SECTION;
    int     col;
  EXEC SQL END DECLARE SECTION;
  EXEC SQL FETCH my_cursor INTO SQL DESCRIPTOR 'row_desc';
```

14.5.20 GET DESCRIPTOR

Use the `GET DESCRIPTOR` statement to retrieve information from a descriptor. The `GET DESCRIPTOR` statement comes in two forms. The first form returns the number of values (or columns) in the descriptor.

```
EXEC SQL GET DESCRIPTOR descriptor_name
  :host_variable = COUNT;
```

The second form returns information about a specific value (specified by the `VALUE` *`column_number`* clause).

```
EXEC SQL [FOR array_size] GET DESCRIPTOR descriptor_name
  VALUE column_number {:host_variable = descriptor_item {,…}};
```

Where:

`array_size` is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you specify an `array_size`, the `host_variable` must be an array of that size; for example, if `array_size` is `10`, `:host_variable` must be a 10-member array of `host_variables`. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`descriptor_name` specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

Include the `VALUE` clause to specify the information retrieved from the descriptor.

> *`column_number`* identifies the position of the variable within the descriptor.
>
> *`host_variable`* specifies the name of the host variable that will receive the value of the item.
>
> *`descriptor_item`* specifies the type of the retrieved descriptor item.

ECPGPlus implements the following *`descriptor_item`* types:

- `TYPE`
- `LENGTH`
- `OCTET_LENGTH`
- `RETURNED_LENGTH`
- `RETURNED_OCTET_LENGTH`
- `PRECISION`
- `SCALE`
- `NULLABLE`
- `INDICATOR`
- `DATA`
- `NAME`

The following code fragment demonstrates using a `GET DESCRIPTOR` statement to obtain the number of columns entered in a user-provided string:

```
EXEC SQL ALLOCATE DESCRIPTOR parse_desc;
EXEC SQL PREPARE query FROM :stmt;
EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse_desc;
EXEC SQL GET DESCRIPTOR parse_desc :col_count = COUNT;
```

The example allocates an SQL descriptor (named `parse_desc`), before using a `PREPARE` statement to syntax check the string provided by the user (`:stmt`). A `DESCRIBE` statement moves the user-provided string into the descriptor, `parse_desc`. The call to `EXEC SQL GET DESCRIPTOR` interrogates the descriptor to discover the number of columns (`:col_count`) in the result set.

14.5.21 INSERT

Use the `INSERT` statement to add one or more rows to a table. The syntax for the ECPGPlus `INSERT` statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that a value is allowed. The syntax is:

```
[FOR exec_count] INSERT INTO table [(column [, ...])]
  {DEFAULT VALUES |
   VALUES ({expression | DEFAULT} [, ...])[, ...] | query}
  [RETURNING * | output_expression [[ AS ] output_name] [, ...]]
```

Where:

Include the `FOR` *`exec_count`* clause to specify the number of times the statement will execute; this clause is valid only if the `VALUES` clause references an array or a pointer to an array.

`table` specifies the (optionally schema-qualified) name of an existing table.

`column` is the name of a column in the table. The column name may be qualified with a subfield name or array subscript. Specify the `DEFAULT VALUES` clause to use default values for all columns.

`expression` is the expression, value, host variable or parameter marker that will be assigned to the corresponding column. Specify `DEFAULT` to fill the corresponding column with its default value.

`query` specifies a `SELECT` statement that supplies the row(s) to be inserted.

`output_expression` is an expression that will be computed and returned by the `INSERT` command after each row is inserted. The expression can refer to any column within the table. Specify `*` to return all columns of the inserted row(s).

`output_name` specifies a name to use for a returned column.

The following example adds a row to the `employees` table:

```
INSERT INTO emp (empno, ename, job, hiredate)
    VALUES ('8400', :ename, 'CLERK', '2011-10-31');
```

Note that the `INSERT` statement uses a host variable (:`ename`) to specify the value of the `ename` column.

For more information about using the `INSERT` statement, please see the PostgreSQL Core documentation:

http://www.postgresql.org/docs/9.5/static/sql-insert.html

14.5.22 OPEN

Use the `OPEN` statement to open a cursor. The syntax is:

```
EXEC SQL [FOR array_size] OPEN cursor [USING parameters];
```

Where *parameters* is one of the following:

```
  DESCRIPTOR SQLDA_descriptor
or
  host_variable [ [ INDICATOR ] indicator_variable, … ]
```

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.

cursor is the name of the cursor being opened.

parameters is either `DESCRIPTOR` *SQLDA_descriptor* or a comma-separated list of host variables (and optional indicator variables) that initialize the cursor. If specifying an *SQLDA_descriptor*, the descriptor must be initialized with a `DESCRIBE` statement.

The `OPEN` statement initializes a cursor using the values provided in *parameters*. Once initialized, the cursor result set will remain unchanged unless the cursor is closed and re-opened. A cursor is automatically closed when an application terminates.

The following example declares a cursor named `employees`, that queries the `emp` table, returning the employee number, name, salary and commission of an employee whose name matches a user-supplied value (stored in the host variable, `:emp_name`).

```
  EXEC SQL DECLARE employees CURSOR FOR
    SELECT
      empno, ename, sal, comm
    FROM
      emp
    WHERE ename = :emp_name;
  EXEC SQL OPEN employees;
...
```

After declaring the cursor, the example uses an `OPEN` statement to make the contents of the cursor available to a client application.

14.5.23 OPEN DESCRIPTOR

Use the `OPEN DESCRIPTOR` statement to open a cursor with a SQL descriptor. The syntax is:

```
EXEC SQL [FOR array_size] OPEN cursor
  [USING [SQL] DESCRIPTOR descriptor_name]
  [INTO [SQL] DESCRIPTOR descriptor_name];
```

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor being opened.

descriptor_name specifies the name of an SQL descriptor (in the form of a single-quoted string literal) or a host variable that contains the name of an SQL descriptor that contains the query that initializes the cursor.

For example, the following statement opens a cursor (named emp_cursor), using the host variable, :employees:

```
EXEC SQL OPEN emp_cursor USING DESCRIPTOR :employees;
```

14.5.24 PREPARE

Prepared statements are useful when a client application must perform a task multiple times; the statement is parsed, written and planned only once, rather than each time the statement is executed, saving repetitive processing time.

Use the PREPARE statement to prepare an SQL statement or PL/pgSQL block for execution. The statement is available in two forms; the first form is:

```
EXEC SQL [AT database_name] PREPARE statement_name
  FROM sql_statement;
```

The second form is:

```
EXEC SQL [AT database_name] PREPARE statement_name
  AS sql_statement;
```

Where:

database_name is the database identifier or a host variable that contains the database identifier against which the statement will execute. If you omit the AT clause, the statement will execute against the current default database.

statement_name is the identifier associated with a prepared SQL statement or PL/SQL block.

sql_statement may take the form of a SELECT statement, a single-quoted string literal or host variable that contains the text of an SQL statement.

To include variables within a prepared statement, substitute placeholders ($1, $2, $3, etc.) for statement values that might change when you PREPARE the statement. When you EXECUTE the statement, provide a value for each parameter. The values must be provided in the order in which they will replace placeholders.

The following example creates a prepared statement (named add_emp) that inserts a record into the emp table:

```
EXEC SQL PREPARE add_emp (int, text, text, numeric) AS
    INSERT INTO emp VALUES($1, $2, $3, $4);
```

Each time you invoke the statement, provide fresh parameter values for the statement:

```
EXEC SQL EXECUTE add_emp(8003, 'Davis', 'CLERK', 2000.00);
EXEC SQL EXECUTE add_emp(8004, 'Myer', 'CLERK', 2000.00);
```

Please note: A client application must issue a PREPARE statement within each session in which a statement will be executed; prepared statements persist only for the duration of the current session.

14.5.25 ROLLBACK

Use the ROLLBACK statement to abort the current transaction, and discard any updates made by the transaction. The syntax is:

```
EXEC SQL [AT database_name] ROLLBACK [WORK]
  [ { TO [SAVEPOINT] savepoint } | RELEASE ]
```

Where:

database_name is the database identifier or a host variable that contains the database identifier against which the statement will execute. If you omit the AT clause, the statement will execute against the current default database.

Include the TO clause to abort any commands that were executed after the specified *savepoint*; use the SAVEPOINT statement to define the *savepoint*. If you omit the TO clause, the ROLLBACK statement will abort the transaction, discarding all updates.

Include the RELEASE clause to cause the application to execute an EXEC SQL COMMIT RELEASE and close the connection.

Use the following statement to rollback a complete transaction:

```
EXEC SQL ROLLBACK;
```

Invoking this statement will abort the transaction, undoing all changes, erasing any savepoints, and releasing all transaction locks. If you include a savepoint (`my_savepoint` in the following example):

```
EXEC SQL ROLLBACK TO SAVEPOINT my_savepoint;
```

Only the portion of the transaction that occurred after the `my_savepoint` is rolled back; `my_savepoint` is retained, but any savepoints created after `my_savepoint` will be erased.

Rolling back to a specified savepoint releases all locks acquired after the savepoint.

14.5.26 SAVEPOINT

Use the `SAVEPOINT` statement to define a *savepoint*; a savepoint is a marker within a transaction. You can use a `ROLLBACK` statement to abort the current transaction, returning the state of the server to its condition prior to the specified savepoint. The syntax of a `SAVEPOINT` statement is:

```
EXEC SQL [AT database_name] SAVEPOINT savepoint_name
```

Where:

`database_name` is the database identifier or a host variable that contains the database identifier against which the savepoint resides. If you omit the `AT` clause, the statement will execute against the current default database.

`savepoint_name` is the name of the savepoint. If you re-use a *`savepoint_name`*, the original savepoint is discarded.

Savepoints can only be established within a transaction block. A transaction block may contain multiple savepoints.

To create a savepoint named `my_savepoint`, include the statement:

```
EXEC SQL SAVEPOINT my_savepoint;
```

14.5.27 SELECT

ECPGPlus extends support of the SQL `SELECT` statement by providing the `INTO` *`host_variables`* clause. The clause allows you to select specified information from an Advanced Server database into a host variable. The syntax for the `SELECT` statement is:

```
EXEC SQL [AT database_name]
SELECT
  [ hint ]
  [ ALL | DISTINCT [ ON(expression, ...) ]]
  select_list INTO host_variables

  [ FROM from_item [, from_item ]...]
  [ WHERE condition ]
  [ hierarchical_query_clause ]
  [ GROUP BY expression [, ...]]
  [ HAVING condition ]
  [ { UNION [ ALL ] | INTERSECT | MINUS } (subquery) ]
  [ ORDER BY expression [order_by_options]]
  [ LIMIT { count | ALL }]
  [ OFFSET start [ ROW | ROWS ] ]
  [ FETCH { FIRST | NEXT }[ count ] { ROW | ROWS } ONLY ]
  [ FOR { UPDATE | SHARE } [OF table_name [, ...]][NOWAIT ][...]]
```

Where:

`database_name` is the name of the database (or host variable that contains the name of the database) in which the table resides. This value may take the form of an unquoted string literal, or of a host variable.

`host_variables` is a list of host variables that will be populated by the `SELECT` statement. If the `SELECT` statement returns more than a single row, *`host_variables`* must be an array.

ECPGPlus provides support for the additional clauses of the SQL `SELECT` statement as documented in the PostgreSQL Core documentation available at:

http://www.postgresql.org/docs/9.5/static/sql-select.html

To use the `INTO` *`host_variables`* clause, include the names of defined host variables when specifying the `SELECT` statement. For example, the following `SELECT` statement populates the `:emp_name` and `:emp_sal` host variables with a list of employee names and salaries:

```
EXEC SQL SELECT ename, sal
  INTO :emp_name, :emp_sal
  FROM emp
  WHERE empno = 7988;
```

The enhanced `SELECT` statement also allows you to include parameter markers (question marks) in any clause where a value would be permitted. For example, the following query contains a parameter marker in the `WHERE` clause:

```
SELECT * FROM emp WHERE dept_no = ?;
```

This `SELECT` statement allows you to provide a value at run-time for the `dept_no` parameter marker.

14.5.28 SET CONNECTION

There are (at least) three reasons you may need more than one connection in a given client application:

- You may want different privileges for different statements;
- You may need to interact with multiple databases within the same client.
- Multiple threads of execution (within a client application) cannot share a connection concurrently.

The syntax for the `SET CONNECTION` statement is:

```
EXEC SQL SET CONNECTION connection_name;
```

Where:

`connection_name` is the name of the connection to the database.

To use the `SET CONNECTION` statement, you should open the connection to the database using the second form of the `CONNECT` statement; include the `AS` clause to specify a *`connection_name`*.

By default, the current thread uses the current connection; use the `SET CONNECTION` statement to specify a default connection for the current thread to use. The default connection is only used when you execute an `EXEC SQL` statement that does not explicitly specify a connection name. For example, the following statement will use the default connection because it does not include an `AT` *`connection_name`* clause. :

```
EXEC SQL DELETE FROM emp;
```

This statement will not use the default connection because it specifies a connection name using the `AT` *`connection_name`* clause:

```
EXEC SQL AT acctg_conn DELETE FROM emp;
```

For example, a client application that creates and maintains multiple connections (such as):

```
EXEC SQL CONNECT TO edb AS acctg_conn
  USER 'alice' IDENTIFIED BY 'acctpwd';
```

and

```
EXEC SQL CONNECT TO edb AS hr_conn
  USER 'bob' IDENTIFIED BY 'hrpwd';
```

Can change between the connections with the `SET CONNECTION` statement:

```
SET CONNECTION acctg_conn;
```

or

```
SET CONNECTION hr_conn;
```

The server will use the privileges associated with the connection when determining the privileges available to the connecting client. When using the `acctg_conn` connection, the client will have the privileges associated with the role, `alice`; when connected using `hr_conn`, the client will have the privileges associated with `bob`.

14.5.29 SET DESCRIPTOR

Use the `SET DESCRIPTOR` statement to assign a value to a descriptor area using information provided by the client application in the form of a host variable or an integer value. The statement comes in two forms; the first form is:

```
EXEC SQL [FOR array_size] SET DESCRIPTOR descriptor_name
  VALUE column_number descriptor_item = host_variable;
```

The second form is:

```
EXEC SQL [FOR array_size] SET DESCRIPTOR descriptor_name
  COUNT = integer;
```

Where:

`array_size` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`descriptor_name` specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

Include the `VALUE` clause to describe the information stored in the descriptor.

> `column_number` identifies the position of the variable within the descriptor.
>
> `descriptor_item` specifies the type of the descriptor item.
>
> `host_variable` specifies the name of the host variable that contains the value of the item.

ECPGPlus implements the following `descriptor_item` types:

- `TYPE`
- `LENGTH`
- `[REF] INDICATOR`
- `[REF] DATA`
- `[REF] RETURNED LENGTH`

For example, a client application might prompt a user for a dynamically created query:

```
query_text = promptUser("Enter a query");
```

To execute a dynamically created query, you must first *prepare* the query (parsing and validating the syntax of the query), and then *describe* the *input* parameters found in the query using the `EXEC SQL DESCRIBE INPUT` statement.

```
EXEC SQL ALLOCATE DESCRIPTOR query_params;
EXEC SQL PREPARE emp_query FROM :query_text;

EXEC SQL DESCRIBE INPUT emp_query
  USING SQL DESCRIPTOR 'query_params';
```

After describing the query, the `query_params` descriptor contains information about each parameter required by the query.

For this example, we'll assume that the user has entered:

```
SELECT ename FROM emp WHERE sal > ? AND job = ?;,
```

In this case, the descriptor describes two parameters:

- one for `sal > ?`

- one for `job = ?`

To discover the number of parameter markers (question marks) in the query (and therefore, the number of values you must provide before executing the query), use:

```
EXEC SQL GET DESCRIPTOR … :host_variable = COUNT;
```

Then, you can use `EXEC SQL GET DESCRIPTOR` to retrieve the name of each parameter. You can also use `EXEC SQL GET DESCRIPTOR` to retrieve the type of each parameter (along with the number of parameters) from the descriptor, or you can supply each *value* in the form of a character string and ECPG will convert that string into the required data type.

The data type of the first parameter is `numeric`; the type of the second parameter is `varchar`. The name of the first parameter is `sal`; the name of the second parameter is `job`.

Next, loop through each parameter, prompting the user for a value, and store those values in host variables. You can use `GET DESCRIPTOR … COUNT` to find the number of parameters in the query.

```
EXEC SQL GET DESCRIPTOR 'query_params'
  :param_count = COUNT;

for(param_number = 1;
    param_number <= param_count;
    param_number++)
{
```

Use `GET DESCRIPTOR` to copy the name of the parameter into the `param_name` host variable:

```
   EXEC SQL GET DESCRIPTOR 'query_params'
      VALUE :param_number :param_name = NAME;

   reply = promptUser(param_name);
   if (reply == NULL)
    reply_ind = 1;  /* NULL */
else
    reply_ind = 0;  /* NOT NULL */
```

To associate a *value* with each parameter, you use the `EXEC SQL SET DESCRIPTOR` statement. For example:

```
  EXEC SQL SET DESCRIPTOR 'query_params'
    VALUE :param_number DATA = :reply;
  EXEC SQL SET DESCRIPTOR 'query_params'
    VALUE :param_number INDICATOR = :reply_ind;
}
```

Now, you can use the `EXEC SQL EXECUTE DESCRIPTOR` statement to execute the prepared statement on the server.

14.5.30 UPDATE

Use an `UPDATE` statement to modify the data stored in a table. The syntax is:

```
EXEC SQL [AT database_name][FOR exec_count]
    UPDATE [ ONLY ] table [ [ AS ] alias ]
    SET {column = { expression | DEFAULT } |
        (column [, ...]) = ({ expression|DEFAULT } [, ...])} [, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [[ AS ] output_name] [, ...] ]
```

Where:

database_name is the name of the database (or host variable that contains the name of the database) in which the table resides. This value may take the form of an unquoted string literal, or of a host variable.

Include the `FOR` *exec_count* clause to specify the number of times the statement will execute; this clause is valid only if the `SET` or `WHERE` clause contains an array.

ECPGPlus provides support for the additional clauses of the SQL `UPDATE` statement as documented in the PostgreSQL Core documentation available at:

http://www.postgresql.org/docs/9.5/static/sql-update.html

A host variable can be used in any clause that specifies a value. To use a host variable, simply substitute a defined variable for any value associated with any of the documented `UPDATE` clauses.

The following `UPDATE` statement changes the job description of an employee (identified by the `:ename` host variable) to the value contained in the `:new_job` host variable, and increases the employees salary, by multiplying the current salary by the value in the `:increase` host variable:

```
EXEC SQL UPDATE emp
  SET job = :new_job, sal = sal * :increase
  WHERE ename = :ename;
```

The enhanced `UPDATE` statement also allows you to include parameter markers (question marks) in any clause where an input value would be permitted. For example, we can write the same update statement with a parameter marker in the `WHERE` clause:

```
EXEC SQL UPDATE emp
  SET job = ?, sal = sal * ?
  WHERE ename = :ename;
```

This `UPDATE` statement could allow you to prompt the user for a new value for the `job` column and provide the amount by which the `sal` column is incremented for the employee specified by `:ename`.

14.5.31 WHENEVER

Use the `WHENEVER` statement to specify the action taken by a client application when it encounters an SQL error or warning. The syntax is:

```
EXEC SQL WHENEVER condition action;
```

The following table describes the different conditions that might trigger an *`action`*:

Condition	Description
`NOT FOUND`	The server returns a `NOT FOUND` condition when it encounters a `SELECT` that returns no rows, or when a `FETCH` reaches the end of a result set.
`SQLERROR`	The server returns an `SQLERROR` condition when it encounters a serious error returned by an SQL statement.
`SQLWARNING`	The server returns an `SQLWARNING` condition when it encounters a non-fatal warning returned by an SQL statement.

The following table describes the actions that result from a client encountering a *`condition`*:

Action	Description
`CALL` *`function`*`([`*`args`*`])`	Instructs the client application to call the named *`function`*.
`CONTINUE`	Instructs the client application to proceed to the next statement.
`DO BREAK`	Instructs the client application to a C break statement. A break statement may appear in a `loop` or a `switch` statement. If executed, the break statement terminate the `loop` or the `switch` statement..
`DO CONTINUE`	Instructs the client application to emit a C `continue` statement. A `continue` statement may only exist within a loop, and if executed, will cause the flow of control to return to the top of the loop.
`DO` *`function`*`([`*`args`*`])`	Instructs the client application to call the named *`function`*.
`GOTO` *`label`* or `GO TO` *`label`*	Instructs the client application to proceed to the statement that contains the *`label`*.
`SQLPRINT`	Instructs the client application to print a message to standard error.
`STOP`	Instructs the client application to stop execution.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

The following code fragment prints a message if the client application encounters a warning, and aborts the application if it encounters an error:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

Include the following code to specify that a client should continue processing after warning a user of a problem:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
```

Include the following code to call a function if a query returns no rows, or when a cursor reaches the end of a result set:

```
EXEC SQL WHENEVER NOT FOUND CALL error_handler(__LINE__);
```

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

15 dblink_ora

dblink_ora provides an OCI-based database link that allows you `SELECT`, `INSERT`, `UPDATE` or `DELETE` data stored on an Oracle system from within Advanced Server.

Connecting to an Oracle Database

To enable Oracle connectivity, download Oracle's freely available OCI drivers from their website, presently at:

http://www.oracle.com/technetwork/database/features/instant-client/index-100365.html

Before creating a link to an Oracle server, you must tell Advanced Server where to find the OCI driver. You can either set the `LD_LIBRARY_PATH` environment variable (or `PATH` on Windows) to the `lib` directory of the Oracle client installation or 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 `LD_LIBRARY_PATH` (or `PATH`) environment variable.

If you use the `LD_LIBRARY_PATH` (or `PATH`) environment variable, you must ensure that the variable is properly set each time you start Advanced Server.

If the Oracle instant client that you've downloaded does not include the `libclntsh.so` library, you must create a symbolic link named `libclntsh.so` that points to the downloaded version. Navigate to the Instant Client directory, and execute the following command:

```
ln -s libclntsh.so<version_number> libclntsh.so
```

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

```
ln -s libclntsh.so.11.1 libclntsh.so
```

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

```
oracle_home = 'lib_directory'
```

Where *`lib_directory`* is the name of the directory that contains `libclntsh.so` (on Linux) or `oci.dll` (on Windows).

After setting the `oracle_home` configuration parameter, you must restart the server for the changes to take effect.

15.1 dblink_ora Functions and Procedures

dblink_ora supports the following functions and procedures.

15.1.1 dblink_ora_connect()

The `dblink_ora_connect()` function establishes a connection to an Oracle database with user-specified connection information. The function comes in two forms; the signature of the first form is:

```
dblink_ora_connect(conn_name, server_name, service_name,
user_name, password, port, asDBA)
```

Where:

`conn_name` specifies the name of the link.

`server_name` specifies the name of the host.

`service_name` specifies the name of the service.

`user_name` specifies the name used to connect to the server.

`password` specifies the password associated with the user name.

`port` specifies the port number.

`asDBA` is `True` if you wish to request `SYSDBA` privileges on the Oracle server. This parameter is optional; if omitted, the default value is `FALSE`.

The first form of `dblink_ora_connect()` returns a `TEXT` value.

The signature of the second form of the `dblink_ora_connect()` function is:

```
dblink_ora_connect(foreign_server_name, asDBA)
```

Where:

`foreign_server_name` specifies the name of a foreign server.

> *asDBA* is True if you wish to request SYSDBA privileges on the Oracle server. This parameter is optional; if omitted, the default value is FALSE.

The second form of the dblink_ora_connect() function allows you to use the connection properties of a pre-defined foreign server when establishing a connection to the server.

Before invoking the second form of the dblink_ora_connect() function, use the CREATE SERVER command to store the connection properties for the link to a system table. When you call the dblink_ora_connect() function, substitute the server name specified in the CREATE SERVER command for the name of the link.

The second form of dblink_ora_connect() returns a TEXT value.

15.1.2 dblink_ora_status()

The dblink_ora_status() function returns the database connection status. The signature is:

```
dblink_ora_status(conn_name)
```

Where:

> *conn_name* specifies the name of the link.

If the specified connection is active, the function returns a TEXT value of OK.

15.1.3 dblink_ora_disconnect()

The dblink_ora_disconnect() function closes a database connection. The signature is:

```
dblink_ora_disconnect(conn_name)
```

Where:

> *conn_name* specifies the name of the link.

The function returns a TEXT value.

15.1.4 dblink_ora_record()

The `dblink_ora_record()` function retrieves information from a database. The signature is:

```
dblink_ora_record(conn_name, query_text)
```

Where:

`conn_name` specifies the name of the link.

`query_text` specifies the text of the SQL `SELECT` statement that will be invoked on the Oracle server.

The function returns a `SETOF` record.

15.1.5 dblink_ora_call()

The `dblink_ora_call()` function executes a non-`SELECT` statement on an Oracle database and returns a result set. The signature is:

```
dblink_ora_call(conn_name, command, iterations)
```

Where:

`conn_name` specifies the name of the link.

`command` specifies the text of the SQL statement that will be invoked on the Oracle server.

`iterations` specifies the number of times the statement is executed.

The function returns a `SETOF` record.

15.1.6 dblink_ora_exec()

The `dblink_ora_exec()` procedure executes a DML or DDL statement in the remote database. The signature is:

```
dblink_ora_exec(conn_name, command)
```

Where:

> *conn_name* specifies the name of the link.
>
> *command* specifies the text of the `INSERT`, `UPDATE`, or `DELETE` SQL statement that will be invoked on the Oracle server.

The function returns a `VOID`.

15.1.7 dblink_ora_copy()

The dblink_ora_copy() function copies an Oracle table to an EnterpriseDB table. The dblink_ora_copy() function returns a BIGINT value that represents the number of rows copied. The signature is:

```
dblink_ora_copy(conn_name, command, schema_name,
table_name, truncate, count)
```

Where:

> *conn_name* specifies the name of the link.
>
> *command* specifies the text of the SQL `SELECT` statement that will be invoked on the Oracle server.
>
> *schema_name* specifies the name of the target schema.
>
> *table_name* specifies the name of the target table.
>
> *truncate* specifies if the server should `TRUNCATE` the table prior to copying; specify `TRUE` to indicate that the server should `TRUNCATE` the table. *truncate* is optional; if omitted, the value is `FALSE`.
>
> *count* instructs the server to report status information every *n* record, where *n* is the number specified. During the execution of the function, Advanced Server raises a notice of severity `INFO` with each iteration of the count. For example, if FeedbackCount is `10`, `dblink_ora_copy()` raises a notice every `10` records. *count* is optional; if omitted, the value is `0`.

15.2 Calling dblink_ora Functions

The following command establishes a connection using the `dblink_ora_connect()` function:

```
SELECT dblink_ora_connect('acctg', 'localhost', 'xe', 'hr',
'pwd', 1521);
```

The example connects to a service named `xe` running on port `1521` (on the `localhost`) with a user name of `hr` and a password of `pwd`. You can use the connection name `acctg` to refer to this connection when calling other dblink_ora functions.

The following command uses the `dblink_ora_copy()` function over a connection named `edb_conn` to copy the `empid` and `deptno` columns from a table (on an Oracle server) named `ora_acctg` to a table located in the `public` schema on an instance of Advanced Server named `as_acctg`. The `TRUNCATE` option is enforced, and a feedback count of `3` is specified:

```
edb=# SELECT dblink_ora_copy('edb_conn','select empid,
deptno FROM ora_acctg', 'public', 'as_acctg', true, 3);

INFO:  Row: 0
INFO:  Row: 3
INFO:  Row: 6
INFO:  Row: 9
INFO:  Row: 12

 dblink_ora_copy
-----------------
 12

(1 row)
```

The following `SELECT` statement uses `dblink_ora_record()` function and the `acctg` connection to retrieve information from the Oracle server:

```
SELECT * FROM dblink_ora_record( 'acctg', 'SELECT
first_name from employees') AS t1(id VARCHAR);
```

The command retrieves a list that includes all of the entries in the `first_name` column of the `employees` table.

16 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.

16.1.1 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.

16.1.2 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.

16.1.3 edb_partdef

The `edb_partdef` table contains one row for each

Column	Type	Modifiers	Description
`pdefrel`	`oid`	`not null`	The OID of the partitioning root (comes from pg_class).
`pdeftype`	`char`	`not null`	The partitioning type: 'r' for range 'l' for list 'h' for hash.
`pdefsubtype`	`char`	`not null`	The subpartitioning type: 'r' for range 'l' for list 'h' for hash.
`pdefcols`	`int2vector`	`not null`	The partitioning key columns (a vector of pg_attribute OIDs).
`pdefsubcols`	`int2vector`	`not null`	The subpartitioning key columns (a vector of pg_attribute OIDs).
`pdefkeyexpr`	`pg_node_tree`		Currently unused.
`pdefinsertexpr`	`pg_node_tree`		Currently unused.

16.1.4 edb_partition

The `edb_partition` table contains one row for each partition or subpartition.

Column	Type	Modifiers	Description
`partname`	`name`	`not_null`	The partition or subpartition name.
`partpos`	`integer`	`not_null`	The partition or subpartition position.
`partpdefid`	`oid`	`not_null`	The OID of the edb_partdef tuple (points to edb_partdef).
`partrelid`	`oid`	`not_null`	The OID of the partition backing table (points to pg_class).
`partparent`	`oid`	`not_null`	The OID of the parent edb_partition tuple (for subpartitions).
`partcons`	`oid`	`not_null`	The OID of the CHECK constraint for the partition (points to pg_constraint).
`parttablespace`	`oid`	`not_null`	The OID of the TABLESPACE (points to pg_tablespace).
`partistemplate`	`boolean`	`not_null`	Identifies this partition as a template partition (currently unused).
`partvals`	`pg_node_tree`		A list of partition key values in pg_getexpr() form.

16.1.5 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.

16.1.6 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.

16.1.7 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 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.

Copyright © 2007 - 2021 EnterpriseDB Corporation. All rights reserved.

16.1.8 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. 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.

16.1.9 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.

16.1.10 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.

17 Acknowledgements

The PostgreSQL 8.3, 8.4, 9.0, 9.1, 9.2, 9.3 and 9.4 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 © 1996-2016 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.