EnterpriseDB®

Oracle Compatibility Developer's Guide

Postgres Plus Advanced Server 8.3 R2

July 2, 2009

Oracle® Compatibility Developer's Guide, version 2.14
by EnterpriseDB Corporation
Copyright © 2009 EnterpriseDB Corporation. All rights reserved.

EnterpriseDB Corporation, 235 Littleton Road, Westford, MA 01866, USA
T +1 978 589 5700 **F** +1 978 589 5701 **E** info@enterprisedb.com **www**.enterprisedb.com

Table of Contents

1 Introduction

This guide describes the Oracle compatibility features of Postgres Plus Advanced Server. Oracle compatibility means that an application runs in an Oracle environment as well as in a Postgres Plus Advanced Server environment with minimal or no changes to the application code.

Postgres Plus Advanced Server contains a rich set of features that enables development of database applications for PostgreSQL or Oracle. This guide focuses solely on the features that are compatible with Oracle. To learn about all of the features of Postgres Plus Advanced Server, consult the Postgres Plus documentation set.

Developing an Oracle compatible application in Postgres Plus Advanced Server requires special attention to which features are used in the construction of the application. For example, developing an Oracle compatible application means choosing:

- Oracle compatible data types to define the application's database tables
- SQL statements that are compatible with Oracle SQL
- Oracle compatible system and built-in functions for use in SQL statements and procedural logic
- 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

Postgres Plus 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

This section lists the Oracle compatibility features that have been added to Postgres Plus Advanced Server 8.3 (R1) to form Postgres Plus Advanced Server 8.3 (R2).

- The COMMIT and ROLLBACK commands are now supported in SPL programs. See Section 4.6 for information on explicit transaction control in SPL.
- The CREATE_TYPE and DROP_TYPE commands are now supported to create and delete user-defined types.
- Nested tables are now supported as a collection type in addition to associative arrays. See Section 4.10.2 for information on nested tables.
- The NEW_TIME function is now supported. See Section 3.5.7.5 for more information.
- Support for the object type specification and object type attributes have been added to SPL. See Chapter 8.
- Support for DBMS_SQL built-in package (Oracle compatible dynamic SQL). See Section 7.5.
- Support for DBMS_JOB built-in package (Oracle compatible job management). See Section 7.6.
- Support for DBMS_LOB built-in package (Oracle compatible large object management). See Section 7.7.
- Support for DBMS_UTILITY built-in package (Oracle compatible utilities). See Section 7.8.
- Support for UTL_MAIL built-in package (Oracle compatible email management). See Section 7.9.
- Support for UTL_SMTP built-in package (Oracle compatible SMTP implementation). See Section 7.10.
- Support for five new Oracle views: ALL_TYPES, DBA_TYPES, USER_TYPES, DBA_JOBS and USER_JOBS.
- Source code obfuscation is now supported with the EDB*Wrap utility. See section 11.3.

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 Oracle Compatible Configuration Parameters

Postgres Plus Advanced Server supports the development and execution of PostgreSQL and Oracle applications. There are a number of system behaviors that can be altered to act in a more PostgreSQL or in a more Oracle compliant manner. These are controlled by configuration parameters that can be found in the `postgresql.conf` file in the database cluster data directory. Changing the parameters in the `postgresql.conf` file changes the behavior over all databases in the cluster. More fine-grained adjustment of these parameters can be done by database, by user or group, or by session. These parameters are the following:

- `edb_redwood_date` – Controls whether or not a time component is stored in `DATE` columns. For Oracle compatible behavior, set `edb_redwood_date` to "true".
- `edb_redwood_strings` – Equates null to an empty string for purposes of string concatenation operations. For Oracle compatible behavior, set `edb_redwood_strings` to "true".
- `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 Oracle compatible behavior, set `edb_stmt_level_tx` to "true"; however, use only when absolutely necessary. See Section 1.3.3.
- `oracle_home` – Point Postgres Plus Advanced Server to the correct Oracle installation directory. See Section 1.3.4.

1.3.1 edb_redwood_date

When `DATE` appears as the data type of a column in the commands,

CREATE TABLE or

ALTER TABLE, 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_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 chapter 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)
```

1.3.3 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.

1.3.4 oracle_home

The configuration parameter, `oracle_home` directs Postgres Plus Advanced Server to the correct Oracle Home directory in the file system. This parameter can only be set by a superuser and can be set at any level both on and off-line.; thereby allowing a single EDB server to use multiple Oracle client installs and access both older and newer versions of Oracle at the same time. By implementation, if set, the oracle_home configuration parameter will override the default ORACLE_HOME environment variable.

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 Postgres Plus Advanced Server the selection for Oracle compatible configuration and defaults must be chosen in order to reproduce the same results as the examples shown in this guide. A default Oracle 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 Postgres Plus 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 on to Postgres Plus 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 chapter 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

Postgres Plus 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 Postgres Plus Advanced Server 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 Postgres Plus 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 Postgres Plus 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.

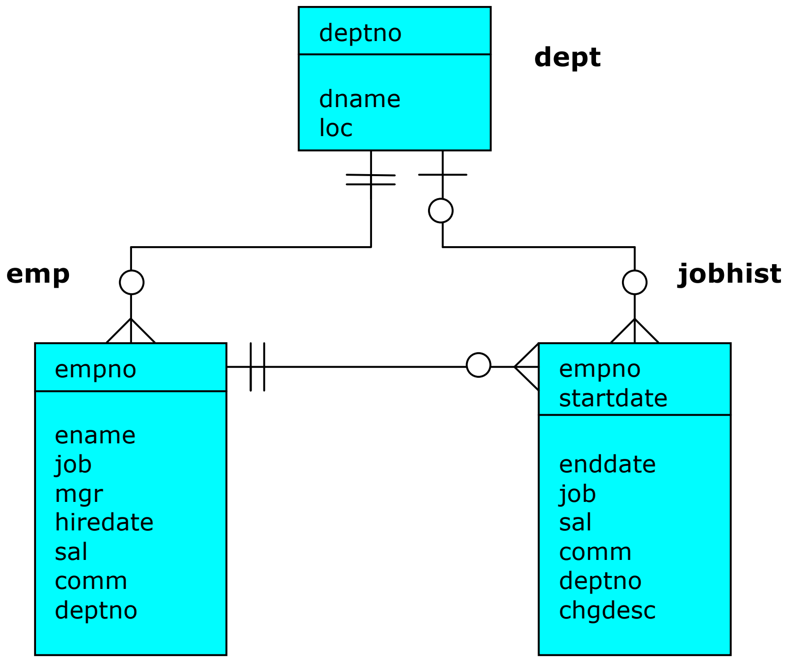
Each employee has an identification number, name, hire date, salary, and manager. Some employees earn a commission in addition to their salary. All employee-related information is stored in the `emp` table.

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;
```

The following sections begin the discussion of basic SQL commands.

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.

Postgres Plus 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 relabel 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, Postgres Plus 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, Postgres Plus 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

In the previous chapter the basics of using SQL to store and access your data in Postgres Plus Advanced Server was covered. This section discusses more advanced SQL features that 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.

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. Postgres Plus 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. The types of database objects for which a synonym may be created are a table, view, sequence, or another synonym.

There are two types of synonyms - public synonyms and private synonyms. A *public synonym* is a synonym that is globally available in a database and can be referenced by any user in the database cluster. A public synonym does not belong to any schema. A private synonym is one that does belong to a specific schema. Postgres Plus Advanced Server currently supports only public synonyms.

2.2.4.1 Creating a Public Synonym

The command, `CREATE PUBLIC SYNONYM`, is used to create a public synonym. The public synonym must be assigned an identifier that is not already used for an existing public synonym. For example:

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

Now, the `emp` table in the `enterprisedb` schema can be referenced in any SQL statement, both DDL and 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)
```

See the

CREATE PUBLIC SYNONYM command additional information.

2.2.4.2 Deleting a Public Synonym

To delete a public synonym, use the command, `DROP PUBLIC SYNONYM`. In the following example, the synonym, `personnel`, created in the previous example is dropped.

```
DROP PUBLIC SYNONYM personnel;
```

See the DROP PUBLIC SYNONYM command for additional information.

2.2.4.3 Public Synonym Namespace

The name given to a public synonym can be any valid identifier as long as there is no other public synonym in the same database with the same name. This means, that a public synonym can have the same name as an existing schema, table, view, or any other database object.

Thus, it is important to choose public synonym names carefully as unexpected results may occur if the same name is used by other objects in the search path as explained in the next section.

2.2.4.4 Public Synonym Name Resolution and the Search Path

Name resolution is the process of determining exactly which particular object is to be acted upon in a SQL command. If an object is fully-qualified by its schema name, there is no ambiguity. The desired object is the one belonging to the named schema. However, if an object is not qualified by its schema name, then there is a series of steps to determine where the desired object resides.

If an unqualified name appears in a SQL command, and only if that name does not appear in any schema to which the user has access in the current search path, the public synonyms in the database are examined to see if this name is a public synonym. If so, then the name resolves to the object underlying the public synonym.

As a consequence, if there is a public synonym defined which is intended for use in a SQL command, but the current search path happens to contain another identically named object in a schema accessible by the user, the name will resolve to the object in the search path and not to the public synonym.

2.2.4.5 Public Synonyms and Privileges

Any user can create a public synonym. There are no special privileges for public synonym creation. Any user can reference a public synonym in a SQL command. However, when the SQL command is executed, the privileges of the current user are checked against the synonym's underlying database object and if the user does not have the proper permissions for that object, the SQL command will fail.

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.

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.

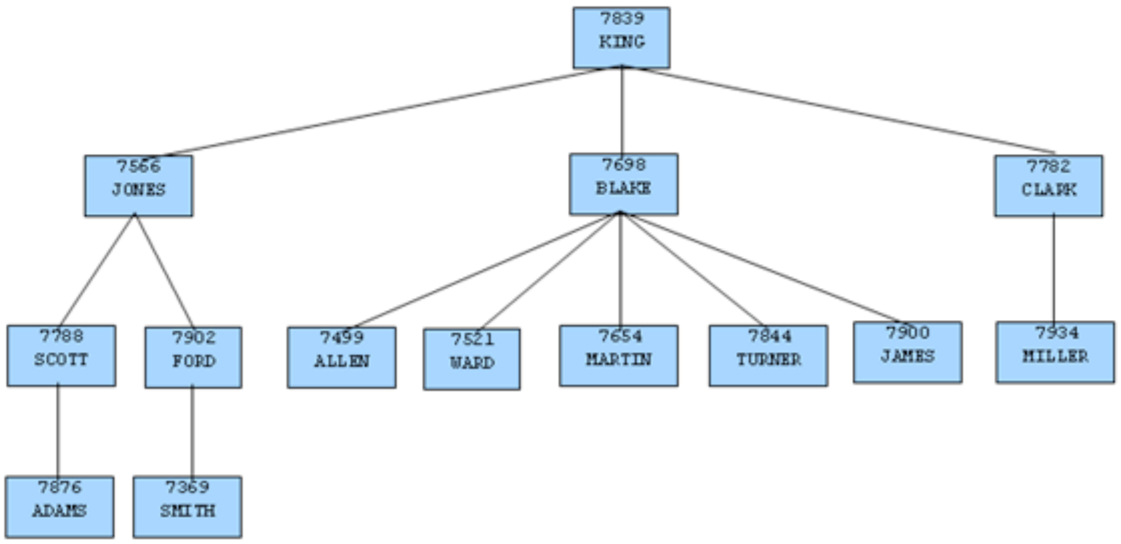


Figure 2 Employee Organization Hierarchy

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 is a direct report of 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)
```

2.2.5.5 Ordering the Siblings

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.

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)
```

3 The SQL Language

This chapter describes the subset of the Postgres Plus Advanced Server SQL language that is Oracle compatible. The SQL syntax, commands, data types, functions, etc. described in this chapter work in both Postgres Plus Advanced Server and in Oracle.

Other aspects of the Postgres Plus Advanced Server SQL language that are not Oracle compatible can be found in the Postgres Plus documentation set. The Postgres Plus documentation set includes syntax and commands for extended functionality not included in this guide.

This chapter is organized into the following sections:

- General discussion of Postgres Plus 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 code zero. (To include a double quote, write two double quotes.) This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers `FOO`, `foo`, and `"foo"` are considered the same by Postgres Plus Advanced Server, but `"Foo"` and `"FOO"` are different from these three and each other. (The folding of unquoted names to lower case in Postgres Plus Advanced Server is an area of Oracle-incompatibility. In Oracle unquoted names are folded to upper case. Thus, `foo` is equivalent to `"FOO"` not `"foo"` in Oracle. 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 Postgres Plus 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 `*/`.

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

3.2 Data Types

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

Table 3-1 Data Types

Name	Alias	Description
`BLOB`	`LONG RAW, RAW(n)`	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`	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*
`VARCHAR2(n)`	`CHAR VARYING(n), CHARACTER VARYING(n), VARCHAR(n)`	Variable-length character string with a maximum length of *n* characters

The following sections describe each data type in more detail.

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 3-2 Numeric Types

Name	Storage Size	Description	Range
`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
`REAL`	4 bytes	Variable-precision, inexact	6 decimal digits precision
`DOUBLE PRECISION`	8 bytes	Variable-precision, inexact	15 decimal digits

Name	Storage Size	Description	Range
			precision

The following sections describe the types in detail.

3.2.1.1 Integer Type

The type, `INTEGER`, stores whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error.

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.

Postgres Plus 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. Postgres Plus 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 shows the general-purpose character types available in Postgres Plus Advanced Server.

Table 3-3 Character Types

Name	Description
`CHAR [ (n) ]`	Fixed-length, blank-padded
`CLOB`	Large variable-length up to 1 GB
`VARCHAR2(n)`	Variable-length with limit

The two primary character types are `CHAR(n)` and `VARCHAR2(n)`, where *n* is a positive integer. Both of these types can store strings up to *n* characters in length. In the case of type `CHAR`, *n* defaults to 1 if omitted. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. If the string to be stored is shorter than the declared length, values of type `CHAR` will be space-padded; values of type `VARCHAR2` will simply store the shorter string.

If one explicitly casts a value to `VARCHAR2(n)` or `CHAR(n)`, then an over-length value will be truncated to *n* characters without raising an error. (This too is required by the SQL standard.)

Values of type `CHAR` are physically padded with spaces to the specified width *n*, and are stored and displayed that way. However, the padding spaces are treated as semantically insignificant. 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. Note that trailing spaces *are* semantically significant in `VARCHAR2` values.

A third character type used for storing large character strings is the `CLOB` data type. `CLOB` is semantically equivalent to `VARCHAR2` except no length limit is specified. Generally, use `CLOB` over `VARCHAR2` 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 three 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 also 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 `BLOB` data type allows storage of binary strings.

Table 3-4 Binary Large Object

Name	Storage Size	Description

Name	Storage Size	Description
`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

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.

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

Table 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
`TIMESTAMP [ (`*p*`) ]`	8 bytes	Date and time	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.

3.2.4.1 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. See Section 3.5.6.

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.

3.2.4.1.1 Dates

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

Table 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.

3.2.4.1.2 Times

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

Table 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

Example	Description
`04:05 PM`	Same as 16:05; input hour must be <= 12

3.2.4.1.3 *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 3-6 Date Input. The time portion of the time stamp can be formatted according to any of examples shown in Table 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
```

3.2.4.2 Date/Time Output

The default output format of the date/time types will be either the Oracle compatible style (dd-MON-yy) referred to as the *Redwood date style*, or the ISO 8601 format (yyyy-mm-dd) 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 3-8 Date/Time Output Styles

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

3.2.4.3 Internals

Postgres Plus 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

Postgres Plus 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 3-9 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`.

Note: The `BOOLEAN` data type can only be used for a variable declaration in an SPL program - it cannot be used to define a column data type in a table.

3.3 SQL Commands

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

Note the following points:

- Postgres Plus 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 in the command. Syntax, options, and functionality that are not Oracle compatible have been omitted from the command description and syntax.
- The Postgres Plus documentation set contains aspects of the command that may not be Oracle compatible.

3.3.1 ALTER INDEX

Name

`ALTER INDEX` -- change the definition of an index

Synopsis

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

Description

`ALTER INDEX` changes the definition of an existing index. `RENAME` changes the name of the index. There is no effect on the stored data.

Parameters

name

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

new_name

New name for the index.

Examples

To rename an existing index:

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

See Also

CREATE INDEX,

DROP INDEX

3.3.2 ALTER ROLE

Name

`ALTER ROLE` -- change a database role

Synopsis

```
ALTER ROLE name IDENTIFIED BY password
```

Description

`ALTER ROLE` changes the password of a role. Only superusers or users with the `CREATEROLE` attribute can use this command. If the role to be altered has the `SUPERUSER` attribute, then only a superuser can give this command. Note that unless the role has the `LOGIN` attribute, the password serves no real purpose.

Parameters

`name`

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

`password`

The role's new password.

Notes

Use

GRANT and

REVOKE to change a role's memberships.

Examples

Change a role's password:

```
ALTER ROLE admins IDENTIFIED BY xyRP35z;
```

See Also

CREATE ROLE, DROP ROLE,

GRANT,

REVOKE, SET ROLE

3.3.3 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 Oracle compatible.

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 Oracle compatible.

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 Oracle compatible.

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

3.3.4 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 Oracle syntax compatibility and have no effect whatsoever on the runtime behavior of Postgres Plus Advanced Server. Others will alter a corresponding Postgres Plus 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 Postgres Plus Advanced Server `datestyle` runtime configuration parameter.

`NLS_LANGUAGE (string)`

Sets the language in which messages are displayed. Has the same effect as setting the Postgres Plus 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 Postgres Plus 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 Postgres Plus Advanced Server. See Section 3.4 for more information.

`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 Postgres Plus 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 Postgres Plus 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 Postgres Plus Advanced Server. This form is not Oracle compatible.

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

Set the date display format.

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

3.3.5 ALTER TABLE

Name

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

Synopsis

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

where *`action`* is one of:

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

Description

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

`ADD` *`column type`*

This form adds a new column to the table using the same syntax as

CREATE TABLE.

`DROP COLUMN`

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

`ADD` *`table_constraint`*

This form adds a new constraint to a table using the same syntax as

CREATE TABLE.

`DROP CONSTRAINT`

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

`RENAME`

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

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

Parameters

`name`

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

`column`

Name of a new or existing column.

`new_column`

New name for an existing column.

`new_name`

New name for the table.

`type`

Data type of the new column.

`table_constraint`

New table constraint for the table.

`constraint_name`

Name of an existing constraint to drop.

`CASCADE`

Automatically drop objects that depend on the dropped constraint.

Notes

When a column is added with `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

3.3.6 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

3.3.7 ALTER USER

Name

`ALTER USER` -- change a database user account

Synopsis

```
ALTER USER name IDENTIFIED BY password
```

Description

`ALTER USER` is used to change the password of a Postgres Plus Advanced Server user account. A database superuser or a user with the `CREATEROLE` privilege can use this command. Ordinary users can also use this command to change their own password.

Parameters

`name`

The name of the user whose attributes are to be altered.

`password`

The new password to be used for this account.

Examples

Change a user password:

```
ALTER USER john IDENTIFIED BY xyz;
```

See Also

CREATE USER,

DROP USER

3.3.8 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). Therefore, don't 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 theses comments:

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

3.3.9 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,

SAVEPOINT

3.3.10 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;
```

3.3.11 CREATE DATABASE LINK

Name

`CREATE DATABASE LINK` -- create a new database link

Synopsis

```
CREATE [ PUBLIC ] DATABASE LINK name
  CONNECT TO username IDENTIFIED BY 'password'
  USING { libpq 'host=hostname port=portnum dbname=database' |
    [ oci ] '//hostname[:portnum]/database' }
```

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 Postgres Plus 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.

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 to be created.

username

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

password

The password for *username*.

`libpq`

Specify connection to a remote Postgres Plus Advanced Server database.

oci

Specify connection to a remote Oracle database. This is the default if omitted.

hostname

Name or IP address of the server hosting the remote database.

portnum

Port number accepting connections to the remote database server.

database

The remote database name.

Notes

If a SQL command is to be executed that references a database link to a remote Oracle database, the server needs some way to know where the correct Oracle installation resides on disk. There are two ways to point Postgres Plus Advanced Server to the correct Oracle installation home directory upon start up:

- The environment variable, `ORACLE_HOME`, may be set to the correct directory. This is the default Oracle configuration.
- The postgresql.conf configuration parameter `oracle_home` will also direct Postgres Plus Advanced Server to the correct Oracle Home directory in the file system. See Section 1.3.4 for information on `oracle_home`.

Examples

The following examples assume that a copy of the Postgres Plus Advanced Server sample application's `emp` table has been created in an Oracle database and a second Postgres Plus 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 a Postgres Plus Advanced Server database named, `edb`, located on localhost, running on port 5443. Connect to the Postgres Plus 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 Postgres Plus 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

3.3.12 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. 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 0 for information on 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.

A directory alias must be deleted directly from the `pg_catalog.edb_dir` system catalog table if it is desired to remove the directory alias. This operation must be performed by a superuser. Note that `edb_dir` is not an Oracle compatible table.

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)
```

Remove directory, `empdir`:

```
DELETE FROM pg_catalog.edb_dir WHERE dirname = 'empdir';
```

3.3.13 CREATE FUNCTION

Name

`CREATE FUNCTION` -- define a new function

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION name
  [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
    [, ...]) ]
  RETURN rettype
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
    [ declaration; ] [, ...]
  BEGIN
    statement; [...]
[ EXCEPTION
  { WHEN exception [ OR exception ] [...] THEN
      statement; [, ...] } [, ...]
]
  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 argument types may share a name (this is called overloading). (Overloading of functions is a Postgres Plus Advanced Server feature - overloading of stored functions is not Oracle compatible.)

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`

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

`argname`

The name of an argument. The argument is referenced by this name within the function 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 the function's arguments. The argument types 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 function independent from changes to the definition of a table.

`DEFAULT value`

Supplies a default value for an input argument if one is not supplied in the function call. `DEFAULT` may not be specified for arguments with modes `IN OUT` or `OUT`.

`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`.

`DEFINER | CURRENT_USER`

Specifies whether the privileges of the function owner (`DEFINER`) or the privileges of the current user executing the function (`CURRENT_USER`) are to be used to determine whether or not access is allowed to database objects referenced in the function. Also, under `DEFINER`, the search path of the function owner is used to resolve references to unqualified database objects while under

CURRENT_USER, the search path of the current user executing the function is used to resolve references to unqualified database objects. DEFINER is the default.

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.

Notes

Postgres Plus Advanced Server allows function overloading; that is, the same name can be used for several different functions so long as they have distinct argument 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
```

See Also

DROP FUNCTION

3.3.14 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.

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

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

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

Parameters

`UNIQUE`

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

name

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

table

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

column

The name of a column in the table.

expression

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

tablespace

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

Notes

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

Examples

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

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

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

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

See Also

ALTER INDEX,

DROP INDEX

3.3.15 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 Chapter 6 for more information on 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. Also, under `DEFINER`, the search path of the package owner is used to resolve references to unqualified database objects while under `CURRENT_USER`, the search path of the current user executing a program in the package is used to resolve references to unqualified database objects. `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

CREATE PACKAGE BODY,

DROP PACKAGE

3.3.16 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

3.3.17 CREATE PROCEDURE

Name

`CREATE PROCEDURE` -- define a new stored procedure

Synopsis

```
CREATE [ OR REPLACE ] PROCEDURE name
  [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
    [, ...]) ]
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
    [ declaration; ] [, ...]
  BEGIN
    statement; [...]
[ EXCEPTION
  { WHEN exception [ OR exception ] [...] THEN
      statement; [, ...] } [, ...]
]
  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 on procedures.

Parameters

name

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

argname

The name of an 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 the procedure's arguments. The argument types 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`*

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`.

`DEFINER | CURRENT_USER`

Specifies whether the privileges of the procedure owner (`DEFINER`) or the privileges of the current user executing the procedure (`CURRENT_USER`) are to be used to determine whether or not access is allowed to database objects referenced in the procedure. Also, under `DEFINER`, the search path of the procedure owner is used to resolve references to unqualified database objects while under `CURRENT_USER`, the search path of the current user executing the procedure is used to resolve references to unqualified database objects. `DEFINER` is the default.

`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.

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
```

See Also

DROP PROCEDURE

3.3.18 CREATE PUBLIC SYNONYM

Name

`CREATE PUBLIC SYNONYM` -- define a new public synonym

Synopsis

```
CREATE [ OR REPLACE ] PUBLIC SYNONYM name FOR object
```

Description

`CREATE PUBLIC SYNONYM` defines a public synonym for certain types of database objects. A synonym is an alternate name that can be used to refer to the object. A public synonym is a synonym globally available in the database that can be referenced by any user in the database cluster.

`CREATE OR REPLACE PUBLIC SYNONYM` is similar, but if a public synonym of the same name already exists, it is replaced.

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

See Section 2.2.4 for additional information on public synonyms.

Parameters

`name`

The name of a public synonym to be created.

`object`

The name (optionally schema-qualified) of a database object for which a public synonym is created. The database object may be a table, view, sequence, or another synonym.

Notes

Any user can create a public synonym - no special permission is required.

A public synonym can be referenced by any user in any SQL statement, however, the statement will only be successfully executed if the user has the proper permissions on the database object referenced by the synonym.

A public synonym is not a member of any schema, but is a database-wide name.

Public synonyms can be created for non-existent objects.

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

Examples

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

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

See Also

DROP PUBLIC SYNONYM

3.3.19 CREATE ROLE

Name

`CREATE ROLE` -- define a new database role

Synopsis

```
CREATE ROLE name [ IDENTIFIED BY password ]
```

Description

`CREATE ROLE` adds a new role to a Postgres Plus 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

3.3.20 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. (Oracle compatibility note: In Oracle, 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 Oracle compatible.)

`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. (Oracle compatibility note: In Postgres Plus 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 Postgres Plus Advanced Server, there are other forms of the CREATE SCHEMA command that are not Oracle compatible.

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;
```

3.3.21 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 3.5.8.

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 Oracle compatible.

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

3.3.22 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 Oracle compatible.) 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 Postgres Plus 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:

`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 Oracle compatible. The Oracle default is `DELETE ROWS`.)

`DELETE ROWS`

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

`TABLESPACE` *`tablespace`*

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

`USING INDEX TABLESPACE` *`tablespace`*

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

Notes

Postgres Plus Advanced Server automatically creates an index for each unique constraint and primary key constraint to enforce the uniqueness. Thus, it is not necessary to create an explicit index for primary key columns. (See

CREATE INDEX for more information.)

Examples

Create table `dept` and table `emp`:

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14),
    loc             VARCHAR2(13)
);
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
```

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

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

Define a check column constraint:

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

Define a check table constraint:

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno),
    CONSTRAINT new_emp_ck CHECK (ename IS NOT NULL AND empno > 7000)
);
```

Define a primary key table constraint for the table `jobhist`. Primary key table constraints can be defined on one or more columns of the table.

```
CREATE TABLE jobhist (
    empno           NUMBER(4) NOT NULL,
    startdate       DATE NOT NULL,
    enddate         DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2),
    chgdesc         VARCHAR2(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate)
);
```

This assigns a literal constant default value for the column, `job` and makes the default value of `hiredate` be the date at which the row is inserted.

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

Create table `dept` in tablespace `diskvol1`:

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14),
    loc             VARCHAR2(13)
) TABLESPACE diskvol1;
```

See Also

ALTER TABLE, DROP TABLE

3.3.23 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 (that is, a `SELECT` command). Refer to

SELECT for a description of the allowed syntax.

See Also

CREATE TABLE,

SELECT

3.3.24 CREATE TRIGGER

Name

`CREATE TRIGGER` -- define a new trigger

Synopsis

```
CREATE [ OR REPLACE ] TRIGGER name
  { BEFORE | AFTER }
  { INSERT | UPDATE | DELETE }
      [ OR { INSERT | UPDATE | DELETE } ] [, ...]
    ON table
  [ FOR EACH ROW ]
  [ 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.

The name of the new trigger must not match any existing trigger defined on the same table unless the intent is to update the definition of an existing trigger, in which case use `CREATE OR REPLACE TRIGGER`.

The trigger is created in the same schema as the table on which the triggering event is defined.

See Chapter 5 for more information on 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.

`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

3.3.25 CREATE TYPE

Name

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

Synopsis

```
CREATE [ OR REPLACE ] TYPE name { IS | AS } OBJECT
  ({ attribute { datatype | objtype } } [, ...])

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

Description

`CREATE TYPE` defines a new user-defined data type. The types that can be created are an object type or a nested table type. `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.

To update the definition of an existing type, use `CREATE OR REPLACE 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 Chapter 8 for more information on object types.

Parameters

`name`

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

`attribute`

The name of an attribute in the object type.

`datatype`

The data type of an attribute in an object type, or of the table of a nested table type.

objtype

The name of a previously defined object type.

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 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

DROP TYPE

3.3.26 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 a Postgres Plus 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`

Sets 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

ALTER USER,

DROP USER

3.3.27 CREATE VIEW

Name

`CREATE VIEW` -- define a new view

Synopsis

```
CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ]
  AS query
```

Description

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

`CREATE OR REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced.

If a schema name is given (for example, `CREATE VIEW myschema.myview` ...) then the view is created in the specified schema. Otherwise it is created in the current schema. The view name must be distinct from the name of any other view, table, sequence, or index in the same schema.

Parameters

`name`

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

`column_name`

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

`query`

A query (that is, a `SELECT` statement) which will provide the columns and rows of the view.

Refer to

SELECT for more information about valid queries.

Notes

Currently, views are read only - the system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts, etc. on the view into appropriate actions on other tables. See the `CREATE RULE` command in the Postgres Plus 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

3.3.28 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

3.3.29 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

3.3.30 DROP FUNCTION

Name

`DROP FUNCTION` -- remove a function

Synopsis

```
DROP FUNCTION name [ ([ type [, ...] ]) ]
```

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. The argument types to the function must be specified if there is at least one argument. (This requirement is not Oracle compatible. Postgres Plus Advanced Server allows overloading of function names, so the function signature is required in the Postgres Plus Advanced Server `DROP FUNCTION` command.)

Parameters

name

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

type

The data type of an argument of the function.

Examples

The following command removes the `emp_comp` function.

```
DROP FUNCTION emp_comp(NUMBER, NUMBER);
```

See Also

CREATE FUNCTION

3.3.31 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

3.3.32 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

3.3.33 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

3.3.34 DROP PUBLIC SYNONYM

Name

`DROP PUBLIC SYNONYM` -- remove a public synonym

Synopsis

`DROP PUBLIC SYNONYM` *`name`*

Description

`DROP PUBLIC SYNONYM` drops existing public synonyms. To execute this command you must be a superuser or the owner of the public synonym.

See Section 2.2.4 for additional information on public synonyms.

Parameters

`name`

The name of a public synonym to be removed

Examples

This command will remove the public synonym named, `personnel`:

```
DROP PUBLIC SYNONYM personnel;
```

See Also

CREATE PUBLIC SYNONYM

3.3.35 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.

However, 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

ALTER ROLE,

CREATE ROLE, SET ROLE

3.3.36 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

3.3.37 DROP TABLE

Name

`DROP TABLE` -- remove a table

Synopsis

`DROP TABLE` *`name`*

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.

Examples

To destroy table, `emp`:

```
DROP TABLE emp;
```

See Also

ALTER TABLE,

CREATE TABLE

3.3.38 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

3.3.39 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

3.3.40 DROP TYPE

Name

`DROP TYPE` -- remove a type definition

Synopsis

```
DROP TYPE name
```

Description

`DROP TYPE` removes the type definition. To execute this command you must be a superuser or the owner of the type. 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

3.3.41 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

ALTER USER, CREATE USER

3.3.42 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 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).

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

3.3.43 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 { 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 ]
```

Description

The `GRANT` command has two basic variants: one that grants privileges on a database object (table, view, sequence, or program), and one that grants membership in a role. These variants are similar in many ways, but they are different enough to be described separately.

In Postgres Plus 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.

3.3.44 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 Postgres Plus Advanced Server syntax for granting the `EXECUTE` privilege is not fully Oracle compatible. Postgres Plus 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. In addition for functions, Postgres Plus Advanced Server requires the full function signature after the function name (including an empty parenthesis if there are no function arguments). For procedures, the full signature is required if the procedure has one or more 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 Postgres Plus 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.

3.3.45 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 some 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, Postgres Plus 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

3.3.46 INSERT

Name

`INSERT` -- create new rows in a table

Synopsis

```
INSERT INTO table[@dblink ] [ ( column [, ...] ) ]
  { VALUES ( { expression | DEFAULT } [, ...] )
    [ RETURNING return_expression [, ...]
        { INTO { record | variable [, ...] }
        | BULK COLLECT INTO collection [, ...] } ]
  | query }
```

Description

`INSERT` allows you to insert new rows into a table. You can insert a single row at a time or several rows as a result of a query.

The columns in the target list may be listed in any order. Each column not present in the target list will be inserted using a default value, either its declared default value or null.

If the expression for each column is not of the correct data type, automatic type conversion will be attempted.

The `RETURNING INTO {` *`record`* `|` *`variable`* `[, ...] }` clause may only be specified when the `INSERT` command is used within an SPL program and only when the `VALUES` clause is used.

The `RETURNING BULK COLLECT INTO` *`collection`* `[, ...]` clause may only be specified if the `INSERT` command is used within an SPL program. If more than one *`collection`* is specified as the target of the `BULK COLLECT INTO` clause, then each *`collection`* must consist of a single, scalar field – i.e., *`collection`* must not be a record. *`return_expression`* evaluated for each inserted row, becomes an element in *`collection`* starting with the first element. Any existing rows in *`collection`* are deleted. If the result set is empty, then *`collection`* will be empty.

You must have `INSERT` privilege to a table in order to insert into it. If you use the *`query`* clause to insert rows from a query, you also need to have `SELECT` privilege on any table used in the query.

Parameters

table

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

dblink

Database link name identifying a remote database. See the

CREATE DATABASE LINK command for information on database links.

column

The name of a column in *table*.

expression

An expression or value to assign to *column*.

DEFAULT

This column will be filled with its default value.

query

A query (SELECT statement) that supplies the rows to be inserted. Refer to the

SELECT command for a description of the syntax.

return_expression

An expression that may include one or more columns from *table*. If a column name from *table* is specified in *return_expression*, the value substituted for the column when *return_expression* is evaluated is determined as follows:

If the column specified in *return_expression* is assigned a value in the INSERT command, then the assigned value is used in the evaluation of *return_expression*.

If the column specified in *return_expression* is not assigned a value in the INSERT command and there is no default value for the column in the table's column definition, then null is used in the evaluation of *return_expression*.

If the column specified in *return_expression* is not assigned a value in the INSERT command and there is a default value for the column in the table's column definition, then the default value is used in the evaluation of *return_expression*.

record

A record whose field the evaluated *return_expression* is to be assigned. The first *return_expression* is assigned to the first field in *record*, the second *return_expression* is assigned to the second field in *record*, etc. The number of fields in *record* must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

variable

A variable to which the evaluated *return_expression* is to be assigned. If more than one *return_expression* and *variable* are specified, the first *return_expression* is assigned to the first *variable*, the second *return_expression* is assigned to the second *variable*, etc. The number of variables specified following the `INTO` keyword must exactly match the number of expressions following the `RETURNING` keyword and the variables must be type-compatible with their assigned expressions.

collection

A collection in which an element is created from the evaluated *return_expression*. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding *return_expression* and *collection* field must be type-compatible.

Examples

Insert a single row into table `emp`:

```
INSERT INTO emp VALUES (8021,'JOHN','SALESMAN',7698,'22-FEB-07',1250,500,30);
```

In this second example, the column, `comm`, is omitted and therefore it will have the default value of null:

```
INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, deptno)
    VALUES (8022,'PETERS','CLERK',7698,'03-DEC-06',950,30);
```

The third example uses the `DEFAULT` clause for the `hiredate` and `comm` columns rather than specifying a value:

```
INSERT INTO emp VALUES (8023,'FORD','ANALYST',7566,NULL,3000,NULL,20);
```

This example creates a table for the department names and then inserts into the table by selecting from the `dname` column of the `dept` table:

```
CREATE TABLE deptnames (
    deptname        VARCHAR2(14)
);
INSERT INTO deptnames SELECT dname FROM dept;
```

3.3.47 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, Postgres Plus 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 `ACCESS EXCLUSIVE`, the most restrictive mode, is used. (`ACCESS EXCLUSIVE` is not an Oracle compatible term. In Postgres Plus Advanced Server, this 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.

3.3.48 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 } [, ...]
```

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 Oracle compatible option. By default Oracle always cascades dependent privileges, but Postgres Plus 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.

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;
```

See Also

GRANT, SET ROLE

3.3.49 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

3.3.50 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

3.3.51 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 rollback 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

3.3.52 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 [, ...] [ LEVEL ] ]
  [ HAVING condition [, ...] ]
  [ { UNION [ ALL ] | INTERSECT | MINUS } select ]
  [ ORDER BY expression [ ASC | DESC ] [, ...] ]
  [ FOR UPDATE ]
```

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 on optimizer hints.

The remaining parameters are discussed within the following sections.

3.3.52.1 FROM Clause

The `FROM` clause specifies one or more source tables for the `SELECT`. 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.

The `FROM` clause can contain the 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

```
[ 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.

3.3.52.2 WHERE Clause

The optional `WHERE` clause has the general 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.

3.3.52.3 GROUP BY Clause

The optional `GROUP BY` clause has the general form

```
GROUP BY expression [, ...]
```

`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.

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.

3.3.52.4 HAVING Clause

The optional `HAVING` clause has the general form

```
HAVING condition
```

where *`condition`* is the same as specified for the `WHERE` clause.

`HAVING` eliminates group rows that do not satisfy the 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.

3.3.52.5 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.

3.3.52.6 UNION Clause

The `UNION` clause has this general 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`.

3.3.52.7 INTERSECT Clause

The `INTERSECT` clause has this general 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).

3.3.52.8 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.

3.3.52.9 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.

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

3.3.52.10 ORDER BY Clause

The optional `ORDER BY` clause has this general 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.

3.3.52.11 DISTINCT Clause

If DISTINCT is specified, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). ALL specifies the opposite: all rows are kept; that is the default.

3.3.52.12 FOR UPDATE Clause

The FOR UPDATE clause has this form:

```
FOR UPDATE
```

FOR UPDATE causes the rows retrieved by the SELECT statement to be locked as though for update. This prevents them from being modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt UPDATE, DELETE, or SELECT FOR UPDATE of these rows will be blocked until the current transaction ends. Also, 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 other 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 can't be clearly identified with individual table rows; for example it can't be used with aggregation.

Examples

To join table, dept with table, emp:

```
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)
```

To sum the column, `sal` of all employees and group 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)
```

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)
```

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)
```

3.3.53 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.

3.3.54 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 identify of role `admins`:

```
SET ROLE admins;
```

User `mary` reverts back to her own identity:

```
SET ROLE NONE;
```

See Also

ALTER ROLE,

CREATE ROLE, DROP ROLE,

GRANT,

REVOKE

3.3.55 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.

3.3.56 TRUNCATE

Name

`TRUNCATE` -- empty a table

Synopsis

```
TRUNCATE TABLE name
```

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.

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

DELETE

3.3.57 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 command for information on database links.

column

The name of a column in table.

expression

An expression to assign to the column. The expression may use the old values of this and other columns in the table.

`DEFAULT`

Set the column to its default value (which will be null if no specific default expression has been assigned to it).

condition

An expression that returns a value of type `BOOLEAN`. Only rows for which this expression returns true will be updated.

return_expression

An expression that may include one or more columns from table. If a column name from table is specified in *return_expression*, the value substituted for the column when *return_expression* is evaluated is determined as follows:

If the column specified in *return_expression* is assigned a value in the `UPDATE` command, then the assigned value is used in the evaluation of *return_expression*.

If the column specified in *return_expression* is not assigned a value in the `UPDATE` command, then the column's current value in the affected row is used in the evaluation of *return_expression*.

record

A record whose field the evaluated *return_expression* is to be assigned. The first *return_expression* is assigned to the first field in *record*, the second *return_expression* is assigned to the second field in *record*, etc. The number of fields in *record* must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

variable

A variable to which the evaluated *return_expression* is to be assigned. If more than one *return_expression* and *variable* are specified, the first *return_expression* is assigned to the first *variable*, the second *return_expression* is assigned to the second *variable*, etc. The number of variables specified following the `INTO` keyword must exactly match the number of expressions following the `RETURNING` keyword and the variables must be type-compatible with their assigned expressions.

collection

A collection in which an element is created from the evaluated *return_expression*. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding *return_expression* and *collection* field must be type-compatible.

Examples

Change the location to AUSTIN for department 20 in the `dept` table:

```
UPDATE dept SET loc = 'AUSTIN' WHERE deptno = 20;
```

For all employees with job SALESMAN in the `emp` table, update the salary by 10% and increase the commission by 500.

```
UPDATE emp SET sal = sal * 1.1, comm = comm + 500 WHERE job = 'SALESMAN';
```

3.4 Optimizer Hints

When a `DELETE`, `SELECT`, or `UPDATE` command is issued, the Postgres Plus Advanced Server database server goes through a process to produce the result set of the command which is the final set of rows returned by the database server. How this result set is produced is the job of the *query planner*, also known as the *query optimizer*. Depending upon the specific command, there may be one or more alternatives, called *query plans*, the planner may consider as possible ways to create the result set. The selection of the plan to be used to actually execute the command is dependent upon various factors including:

- Costs assigned to various operations to retrieve the data (see the Planner Cost Constants in the `postgresql.conf` file).
- Settings of various planner method parameters (see the Planner Method Configuration section in the `postgresql.conf` file).
- Column statistics that have been gathered on the table data by the `ANALYZE` command (see the *Postgres Plus* documentation set for information on the `ANALYZE` command and column statistics).

Generally speaking, of the various feasible plans, the query planner chooses the one of least estimated cost for actual execution.

However, it is possible in any given `DELETE`, `SELECT`, or `UPDATE` command to directly influence selection of all or part of the final plan by using optimizer hints. *Optimizer hints* are directives embedded in comment-like syntax immediately following the `DELETE`, `SELECT`, or `UPDATE` key words that tell the planner to utilize or not utilize a certain approach for producing the result set.

Synopsis

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

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

Optimizer hints may be given in two different formats as shown above. Note that in both formats, a plus sign (+) must immediately follow the `/*` or `--` opening comment symbols with no intervening space in order for the following tokens to be interpreted as hints.

In the first format, the hint and optional comment may span multiple lines. In the second format, all hints and comments must be on a single line. The remainder of the statement must start on a new line.

Description

The following points regarding the usage of optimizer hints should be noted:

- 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 *Postgres Plus* 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`, `SELECT`, or `UPDATE` command.

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

3.4.1 Default Optimization Modes

There are a number of optimization modes that can be chosen as the default setting for a Postgres Plus 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 3-10 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 Oracle compatible `SHOW` command 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 3-11 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 3-10 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 Postgres Plus Advanced Server `dbserver\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/EnterpriseDB/8.3/dbserver/bin:$PATH
```

The following example was run in Windows.

```
SET PATH=C:\EnterpriseDB\8.3\dbserver\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 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 3-12 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)
```

3.4.4 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 3-11 or Table 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)
```

3.4.5 Conflicting Hints

This final section on hints deals with cases where two or more conflicting hints are given in a SQL command. In such cases, the hints that contradict each other are ignored. The following table lists hints that are contradictory to each other.

Table 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

Postgres Plus 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 3-14 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 3-15 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 3-16 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.

3.5.3 Mathematical Functions and Operators

Mathematical operators are provided for many Postgres Plus 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 3-17 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`

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 3-18 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`
`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`	`NUMBER`	Truncate to s decimal	`TRUNC(42.4382, 2)`	`42.43`

Function	Return Type	Description	Example	Result
`INTEGER)`		places		
`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 3-19 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 all 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 the 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 3-20 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`
`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.	`INSTR('PETER PIPER PICKED UP A PACK OF PICKED PEPPERS','PI',1,3)`	`33`
`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	`SUBSTR('This is a test',6,2)`	`is`

Function	Return Type	Description	Example	Result
		from the start till the end.		
`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 3-20.

Table 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 and `default` not specified, returns null.	`DECODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found')`	`Three`
`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`
`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`	`LPAD('hi', 5, 'xy')`	`xyxhi`

Function	Return Type	Description	Example	Result
		then it is truncated (on the right).		
`NVL(`*`expr1, expr2`*`)`	Same as argument types; both arguments have the same data type	If *`expr1`* is not null, return *`expr1`*, otherwise return *`expr2`*	`NVL(null, 'abc')`	`abc`
`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 Using the LIKE Operator

Postgres Plus 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.

3.5.6 Data Type Formatting Functions

The Postgres Plus Advanced Server formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. Table 3-22 lists them. 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 3-22 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(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`*`)`	`CLOB`	Convert a floating-point number to a string with output,	`TO_CHAR(CAST(123.5282 AS REAL), '999.99')`	`123.53`

Function	Return Type	Description	Example	Result
		format		
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 3-23 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)
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	Era indicator (lowercase)

Pattern	Description
or a.d.	
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)
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 3-24 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

Modifier	Description	Example
`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 3-25 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 3-26 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'`
`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 Date/Time Functions and Operators

Table 3-28 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 3-27 illustrates the behaviors of the basic arithmetic operators (+, -). For formatting functions, refer to Section 3.5.6. You should be familiar with the background information on date/time data types from Section 3.2.4.

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

Table 3-28 Date/Time Functions

Function	Return Type	Description	Example	Result
`ADD_MONTHS(DATE, NUMBER)`	`DATE`	Add months to a date; see Section REF _Ref171326180 \n \h 3.5.7.1	`ADD_MONTHS('28-FEB-97', 3.8)`	`31-MAY-97 00:00:00`
`CURRENT_DATE`	`DATE`	Current date; see Section REF _Ref171321439 \n \h 3.5.7.8	`CURRENT_DATE`	`04-JUL-07`
`EXTRACT(`*`field`*` FROM TIMESTAMP)`	`DOUBLE PRECISION`	Get subfield; see Section REF _Ref171325337 \n \h 3.5.7.2	`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 REF _Ref171321439 \n \h 3.5.7.8	`LOCALTIMESTAMP`	`04-JUL-07 15:33:23.484`
`MONTHS_BETWEEN(DATE, DATE)`	`NUMBER`	Number of months between two dates; see Section REF _Ref171393068 \n \h 3.5.7.3	`MONTHS_BETWEEN('28-FEB-07', '30-NOV-06')`	`3`
`NEXT_DAY(DATE, `*`dayofweek`*`)`	`DATE`	Date falling on *`dayofweek`* following specified date; see Section REF _Ref171394200 \n \h 3.5.7.4	`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`
`ROUND(DATE [, `*`format`*` ])`	`DATE`	Date rounded according to *`format`*; see Section REF	`ROUND(TO_DATE('29-MAY-05'),'MON')`	`01-JUN-05 00:00:00`

Function	Return Type	Description	Example	Result
		_Ref171394533 \n \h 3.5.7.5		
`SYSDATE`	`DATE`	Current date and time	`SYSDATE`	`06-AUG-07 10:06:27`
`TRUNC(DATE [, format ])`	`DATE`	Truncate according to *format*; see Section REF _Ref174165473 \n \h 3.5.7.7	`TRUNC(TO_DATE('29-MAY-05'), 'MON')`	`01-MAY-05 00:00:00`

3.5.7.1 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)
```

3.5.7.2 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)
```

3.5.7.3 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)
```

3.5.7.4 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)
```

3.5.7.5 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)
```

3.5.7.6 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 3-29 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 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)
```

3.5.7.7 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 3-30 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

Pattern	Description
`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)
```

3.5.7.8 CURRENT DATE/TIME

Postgres Plus 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`
- `LOCALTIMESTAMP`
- `LOCALTIMESTAMP(`*`precision`*`)`
- `SYSDATE`

`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 CURRENT_DATE FROM DUAL;

   date
-----------
 06-AUG-07
(1 row)

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)

SELECT SYSDATE FROM DUAL;

     timestamp
--------------------
 06-AUG-07 16:11:48
```

```
(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 Sequence Manipulation Functions

This section describes Postgres Plus 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.

3.5.9 Conditional Expressions

This section describes the SQL-compliant conditional expressions available in Postgres Plus Advanced Server.

3.5.9.1 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;
```

3.5.9.2 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, 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.

3.5.9.3 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`*.

3.5.9.4 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.

3.5.10 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 3-31 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, or date/time type	Same as argument type	Maximum value of expression across all input values

Function	Argument Type	Return Type	Description
`MIN(`*`expression`*`)`	Any numeric, string, or date/time 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 3-32 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*)
`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)

Function	Argument Type	Return Type	Description
`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)

3.5.11 Subquery Expressions

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

3.5.11.1 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)
```

3.5.11.2 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.

3.5.11.3 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.

3.5.11.4 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.

3.5.11.5 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 Postgres Plus Advanced Server.

SPL provides the following benefits:

- Adds full procedural programming functionality to complement the SQL language
- Provides a single, common language to create stored procedures, functions, triggers, and packages for the Postgres Plus Advanced Server database
- Is integrated with pgAdmin III to provide a seamless development and testing environment
- Promotes the use of reusable code
- Is easy to use

This chapter first describes the basic elements of an SPL program. The chapter then provides 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 Chapter 5 for information on triggers. Packages are discussed in Chapter 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 Postgres Plus 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.

It is suggested that qualified names be avoided if at all possible. In this chapter, the following conventions are adopted to avoid such 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.

4.2 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 Chapter 5 and packages are addressed in Chapter 6.

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`.

There may be an optional declaration section that 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`.

Finally, there may be an optional exception section which appears 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 this block is executed the output would be as follows.

```
75 divided by 14 is 5.36
```

The following block consists of all three sections - the declaration, executable, and exception sections.

```
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

The preceding section demonstrated the basic structure of a block. A block can simply be executed in Postgres Plus Advanced Server.

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.

Anonymous blocks are useful for quick, one-time programs such as for testing.

Typically, however, the same block of code would 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.

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) ]
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
  [ declarations ]
  BEGIN
    statements
  END [ name ];
```

`name` is the identifier of the procedure. If [`OR REPLACE`] is specified and a procedure with the same name already exists in the schema, the new procedure replaces the existing one. If [`OR REPLACE`] is not specified, the new procedure will not be allowed to replace an existing one with the same name in the same schema. *`parameters`* is a list of formal parameters. If the `AUTHID` clause is omitted or `DEFINER` is specified, the rights and search path of the procedure owner are used to determine access privileges to database objects and resolve unqualified database object references, respectively. If `CURRENT_USER` is specified, the rights and search path of the current user executing the procedure are used to determine access privileges and resolve unqualified object references. *`declarations`* are variable, cursor, or type declarations. *`statements`* are SPL program statements. The `BEGIN` - `END` block may contain an `EXCEPTION` section.

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 Postgres Plus Advanced Server.

See the

CREATE PROCEDURE command for more information on creating a procedure.

4.2.3.2 Calling a Procedure

The 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) ];
```

`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 must be called with no opening and closing parenthesis.

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. 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;
```

`name` is the name of the procedure to be dropped.

The previously created procedure is dropped in this example:

```
DROP PROCEDURE simple_procedure;
```

See the

DROP PROCEDURE command for more details.

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
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
```

```
[ declarations ]
BEGIN
  statements
END [ name ];
```

`name` is the identifier of the function. If [`OR REPLACE`] is specified and a function with the same name already exists in the schema, the new function replaces the existing one. If [`OR REPLACE`] is not specified, the new function will not be allowed to replace an existing one with the same name in the same schema. `parameters` is a list of formal parameters. `data_type` is the data type of the value that is returned by the function. If the `AUTHID` clause is omitted or `DEFINER` is specified, the rights and search path of the function owner are used to determine access privileges to database objects and resolve unqualified database object references, respectively. If `CURRENT_USER` is specified, the rights and search path of the current user executing the function are used to determine access privileges and resolve unqualified object references. `declarations` are variable, cursor, or type declarations. `statements` are SPL program statements. The `BEGIN` - `END` block may contain an `EXCEPTION` section.

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 is another function that takes two input parameters. Parameters will be 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;
```

See the

CREATE FUNCTION command for more information.

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 can 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;
```

name is the name of the function to be dropped.

The previously created function is dropped in this example:

```
DROP FUNCTION simple_function;
```

See the

DROP FUNCTION command for more details.

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 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 4-33 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.

4.2.6 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.

4.2.6.1 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 Postgres Plus 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 the

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.

4.2.6.2 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 Oracle compatible. 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 Postgres Plus Advanced Server, a user and a schema are two distinct objects.

4.2.6.3 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 4.5.5 for information on exception handling.

The final topic discusses exactly who is the current user.

4.2.6.4 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*. It is the current user's search path that will be used to resolve any unqualified object references. 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 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 search path is used to resolve unqualified object references and 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.

4.2.6.5 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.

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:/EnterpriseDB/8.3/samples/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 Postgres Plus Advanced Server and is not Oracle compatible 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.

```
\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 Postgres Plus 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 force the `hire_clerk` function to call the `new_empno` function in the `hr_mgr` schema, the call must be changed to a fully qualified name. Since `hire_clerk` is now an invoker's rights program, an unqualified call to `new_empno` would result in a search for `new_empno` in a schema in the search path of `hire_clerk`'s caller rather than specifically in schema, `hr_mgr`, where this program actually resides.

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.

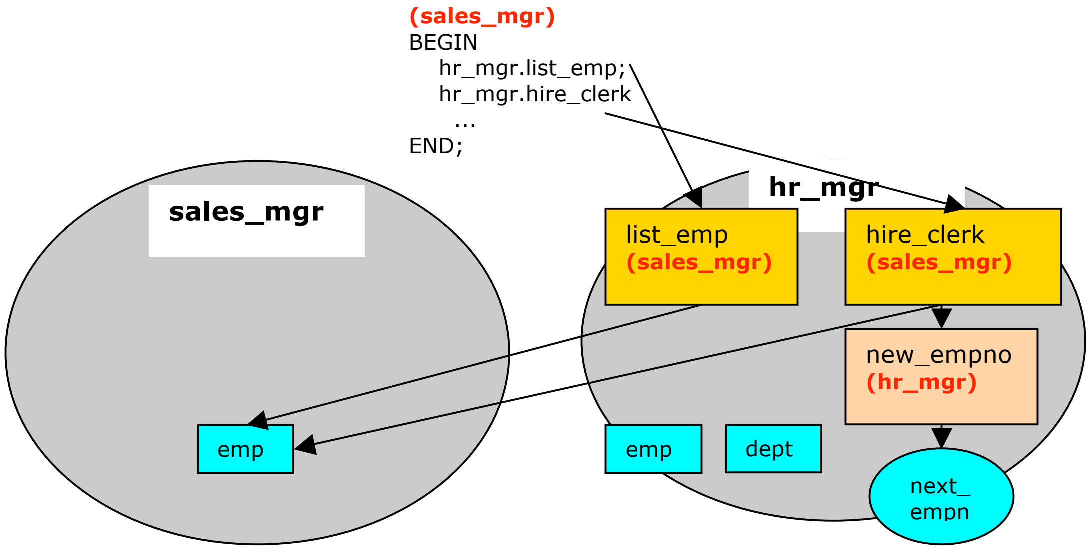


Figure 3 - Invoker's Rights Programs

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`.

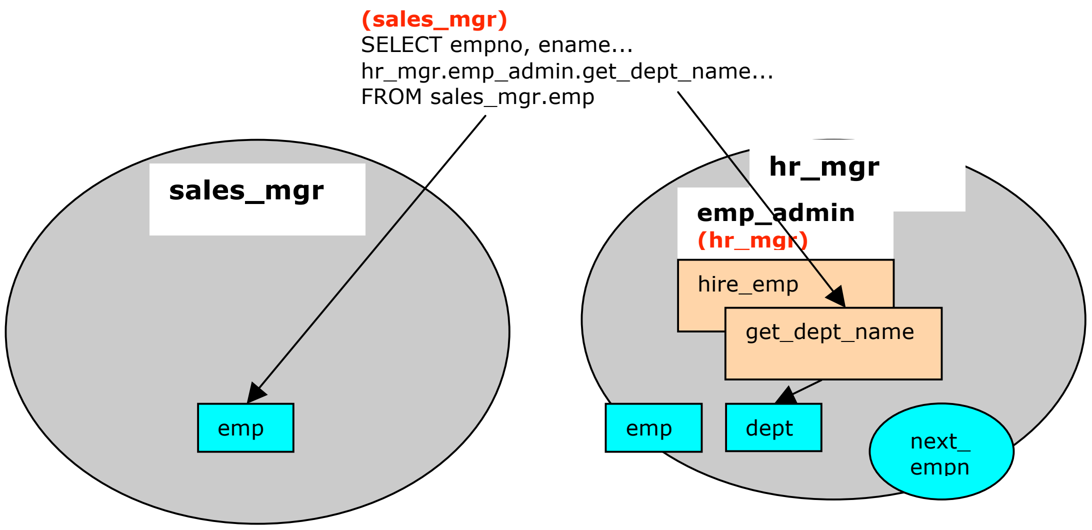


Figure 4 Definer's Rights Package

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`.

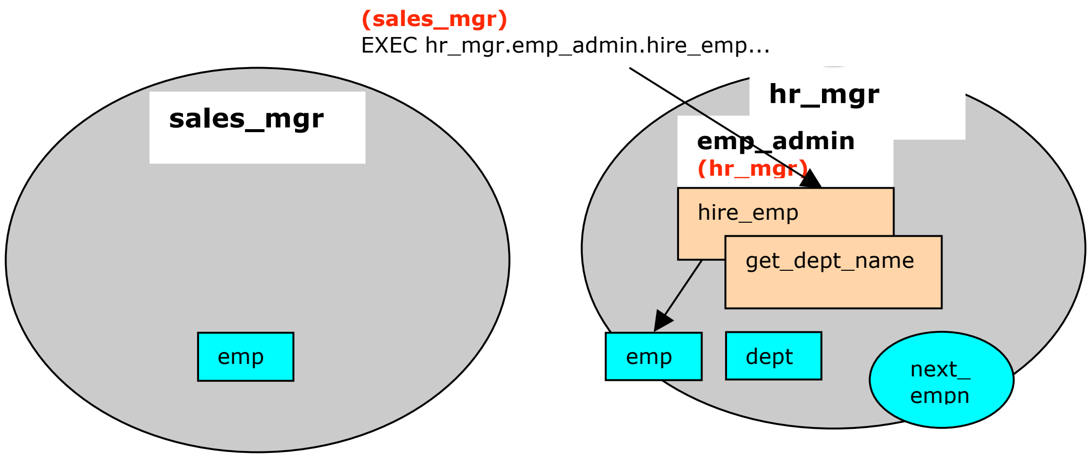


Figure 5 Definer's Rights Package

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

As discussed in Section 4.2.1 SPL is a block-structured language. The first section that can appear in a block is the declaration section. The declaration section contains the definition of variables, cursors, and other types that can be used in SPL statements contained in the block. In this section, variable declarations are examined in more detail.

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

Using the `%TYPE` attribute provides an easy way to create a variable dependent upon a column's data type. Using the `%ROWTYPE` attribute, a record can be defined that contains fields corresponding to all columns of a given table. Each field takes on the data type of its corresponding column.

Note: 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.

A record can be declared using the `%ROWTYPE` attribute. 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 whose columns are to define the fields in the record. A view may be used as well to define a 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 following is the syntax for defining a record type.

```
TYPE rectype IS RECORD (field_1 datatype_1
  [, field_2 datatype_2 ] ...);
```

`rectype` is an identifier assigned to the record type. *`field_1, field_2,`*... are identifiers assigned to the fields of the record type. *`datatype_1, datatype_2,`*... are the data types of *`field_1, field_2,`*... respectively.

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*.

While the `dept_salary_rpt` example in Section 4.3 showed assignment statements used in variable declarations, a variation of this 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 4.10.5.1 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 4.5.5 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 4.5.5 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 4.10.4 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 4.10.4 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 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 4.10.4 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 a Postgres Plus Advanced Server extension and is not Oracle compatible.) *`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 4.10.5 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 of Section 4.4.4 still exists in 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 of Section 4.4.6 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
```

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 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*.

4.5.2.1 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
```

4.5.2.2 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
```

4.5.3 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*.

4.5.3.1 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
```

4.5.3.2 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
```

4.5.4 Loops

With the `LOOP`, `EXIT`, `CONTINUE`, `WHILE`, and `FOR` statements, you can arrange for your SPL program to repeat a series of commands.

4.5.4.1 LOOP

```
LOOP
    statements
END LOOP;
```

`LOOP` defines an unconditional loop that is repeated indefinitely until terminated by an `EXIT` or `RETURN` statement.

4.5.4.2 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
```

4.5.4.3 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
```

4.5.4.4 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
```

4.5.4.5 FOR (integer variant)

```
    FOR name IN 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 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).

4.5.5 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`. But 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`, or if there is none 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 4-34 Exception Condition Names

Condition Name	Description
`CASE_NOT_FOUND`	None of the cases in a `CASE` statement evaluates to "true" and there is no `ELSE` condition.
`CURSOR_ALREADY_OPEN`	Attempt made to open a cursor that is already open.
`INVALID_CURSOR`	Attempt made to access an unopened cursor.
`NO_DATA_FOUND`	No rows satisfied the selection criteria.
`OTHERS`	Catches any exception that hasn't been caught by a prior condition in the exception section.
`TOO_MANY_ROWS`	More than one row satisfied the selection criteria where only one row is allowed to be returned.

Condition Name	Description
ZERO_DIVIDE	Division by zero was attempted.

4.5.6 Raise Application Error

The procedure, RAISE_APPLICATION_ERROR, provides the capability 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 4.5.5. 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);
```

error_number is an integer value or expression that is returned in a variable named, SQLCODE, when the procedure is executed. *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 4.11.

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 an Oracle compatible style of transaction control 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 1.3.3 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.

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
```

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 4.10.5.2 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 `FETCH`ed 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 4-35 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 of 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 then 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
```

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 CURSORs is the declaration section.

Note: *Strongly-typed* REF CURSORs 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 4-36 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 are examples of 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 also 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

Postgres Plus 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 object type (see Chapter 8 for a description of 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 parenthesis.

The most commonly known type of collection is an array. In Postgres Plus Advanced Server, the supported collection types are what was formerly called an *index-by table* in Oracle, now called an *associative array*, and a *nested table*.

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 in the array variable.
- The array does not have to be initialized - just start assigning values to array elements.
- The key can be any negative integer, positive integer, or zero if `INDEX BY BINARY_INTEGER` 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 with `SQLCODE 1403, access of uninitialized index`.

The `TYPE IS TABLE INDEX BY` statement is used to define an associative array type.

```
TYPE assoctype IS TABLE OF { datatype | rectype | objtype }
  INDEX BY { BINARY_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 declared 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 an integer. 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.

```
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 in 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;
```

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. In many respects, it is similar to an associative array.

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 in the nested table variable, or simply, "table" for short.
- The table does not have to be initialized - just start assigning values to table elements. **Note:** In Oracle, a nested table must be initialized with a constructor function. SPL does not currently support nested table constructors.

- The key is a positive integer.
- There is no pre-defined limit on the number of elements in the table - it grows dynamically as elements are added. **Note:** In Oracle, the constructor function must establish the number of elements in the table, or the `EXTEND` function must be used to add additional elements to the table. SPL does not currently support the constructor function or the `EXTEND` function.
- The table can be sparse - there may be gaps in the assignment of values to keys.
- An attempt to reference a table element that has not been assigned a value will result in an exception with `SQLCODE 1403, access of uninitialized index`.

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: The `CREATE TYPE` command can be used to define a nested table type that is available to all SPL programs in the database. See the

CREATE TYPE command.

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 declared nested table type.

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 example is a modification of the first example in the section on associative arrays that reads the first ten employee names from the `emp` table, stores them in a nested table, then displays the results from the table.

```
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
    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
```

Note: In order to run the above anonymous block in Oracle, the following assignment statement must be added as the first statement in the executable section. This statement executes the constructor function for `dname_tbl_typ`, allocating four elements in the table.

```
dname_tbl := dname_tbl_typ(NULL, NULL, NULL, NULL);
```

A modification of the prior example shows how a nested table of an object type can be used. See Chapter 8 for information on 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 declares a nested table whose element consists of the `dept_obj_typ` object type. The nested table is populated by the `dept` table, and then 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
    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;
```

The following is the output from the anonymous block.

```
DNAME          LOC
----------     ----------
ACCOUNTING     NEW YORK
OPERATIONS     BOSTON
RESEARCH       DALLAS
SALES          CHICAGO
```

Note: The following assignment statement must be added as the first executable statement in order to run the above anonymous block in Oracle. This statement executes the constructor function for `dept_tbl_typ`, allocating four elements in the table. Each table element requires the execution of the constructor function for the `dept_obj_typ` object type.

```
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)
);
```

4.10.3 Collection Methods

Collection methods are functions that provide useful information about a collection that can aid in the processing of data in the collection. The following sections discuss these methods.

4.10.3.1 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 identifier of a collection variable.

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 only the five populated elements are included in `COUNT`.

```
COUNT: 5
```

4.10.3.2 FIRST

`FIRST` is a method that returns the index of the first element in a collection. The syntax for using `FIRST` is as follows.

```
    collection.FIRST
```

`collection` is the identifier of a collection variable.

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
```

4.10.3.3 LAST

`LAST` is a method that returns the index of the last element in a collection. The syntax for using `LAST` is a follows.

```
    collection.LAST
```

`collection` is the identifier of a collection variable.

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
```

4.10.4 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 | update | delete };
```

`index` is the position in the collection given in the *`insert`*, *`update`*, or *`delete`* 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 Oracle compatible. 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 following example uses an `INSERT` command with the `FORALL` statement to insert three new employees into the `emp` table.

```
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.ename%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl    IS TABLE OF emp.ename%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_sal           SAL_TBL;
    t_deptno        DEPTNO_TBL;
BEGIN
    t_empno(1)   := 9001;
    t_ename(1)   := 'JONES';
    t_job(1)     := 'ANALYST';
    t_sal(1)    := 3200.00;
    t_deptno(1) := 40;
    t_empno(2)  := 9002;
    t_ename(2)  := 'LARSEN';
    t_job(2)    := 'CLERK';
    t_sal(2)    := 1400.00;
    t_deptno(2) := 40;
    t_empno(3)  := 9003;
    t_ename(3)  := 'WILSON';
    t_job(3)    := 'MANAGER';
    t_sal(3)    := 4000.00;
    t_deptno(3) := 40;
    FORALL i IN t_empno.FIRST..t_empno.LAST
        INSERT INTO emp (empno,ename,job,sal,deptno)
            VALUES (t_empno(i),t_ename(i),t_job(i),t_sal(i),t_deptno(i));
END;

SELECT * FROM emp WHERE empno > 9000;

 empno | ename  |   job   | mgr | hiredate |   sal   | comm | deptno
-------+--------+---------+-----+----------+---------+------+--------
  9001 | JONES  | ANALYST |     |          | 3200.00 |      |     40
  9002 | LARSEN | CLERK   |     |          | 1400.00 |      |     40
  9003 | WILSON | MANAGER |     |          | 4000.00 |      |     40
(3 rows)
```

The following example updates the salary of these three employees in a `FORALL` statement.

```
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 final example deletes these 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)
```

4.10.5 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` and `FETCH INTO` commands, and with the `RETURNING INTO` clause of the `DELETE`, `INSERT`, and `UPDATE` commands. Each of these is illustrated in the following sections.

4.10.5.1 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
```

4.10.5.2 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
```

4.10.5.3 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 `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 a Postgres Plus Advanced Server extension and is not Oracle compatible.)

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
```

4.11 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 Postgres Plus 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.

5.2 Types of Triggers

Postgres Plus 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.

5.3 Creating Triggers

The `CREATE TRIGGER` command defines and names a trigger that will be stored in the database.

```
CREATE [ OR REPLACE ] TRIGGER name
  { BEFORE | AFTER }
  { INSERT | UPDATE | DELETE } [ OR ...]
  ON table
[ FOR EACH ROW ]
[ DECLARE
    declarations ]
  BEGIN
    statements
  END;
```

`name` is the name of the trigger. If [`OR REPLACE`] is specified and a trigger with the same name already exists in the schema, the new trigger replaces the existing one. If [`OR REPLACE`] is not specified, the new trigger will not be allowed to replace an existing one with the same name in the same schema. If `BEFORE` is specified, the trigger is defined as a before trigger. If `AFTER` is specified, the trigger is defined as an after trigger. One of `INSERT`, `UPDATE`, or `DELETE` must be specified defining the triggering event as an insert, update, or deletion, respectively. One or both of the remaining triggering event keywords may also be specified separated by the keyword, `OR`, in which case these are also defined as triggering events. *`table`* is the name of the table on which a triggering event will cause the trigger to fire. If [`FOR EACH ROW`] is specified, the trigger is defined as a row-level trigger. If [`FOR EACH ROW`] is omitted, the trigger is defined as a statement-level trigger. *`declarations`* are variable, cursor, or type declarations. *`statements`* are SPL program statements. The `BEGIN` - `END` block may contain an `EXCEPTION` section.

See the

CREATE TRIGGER command for additional information on creating triggers.

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 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 affect 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.

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.

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 Postgres Plus 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 function 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 holds implementation details and private declarations which are invisible to the application. So we can debug, enhance or replace a package body without changing the specifications. Similarly we 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 following is the syntax of the package specification:

```
CREATE [ OR REPLACE ] PACKAGE package_name
  [ AUTHID { DEFINER | CURRENT_USER } ]
  { IS | AS }
  [ declaration; ] ...
  [ { PROCEDURE proc_name
      [ (parm1 [IN | IN OUT | OUT ] datatype1
      [, parm2 [IN | IN OUT | OUT ] datatype2 ] ...) ];
    |
      FUNCTION func_name
      [ (parm1 [IN | IN OUT | OUT ] datatype1
      [, parm2 [IN | IN OUT | OUT ] datatype2 ] ...) ]
      RETURN return_type; } ...]
END [ package_name ];
```

package_name is an identifier assigned to the package. If the `AUTHID` clause is omitted or `DEFINER` is specified, the rights and search path of the package owner are used to determine access privileges to database objects and resolve unqualified database object references, respectively. If `CURRENT_USER` is specified, the rights and search path of the current user executing a program in the package are used to determine access privileges and resolve unqualified object references. *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 none, one, or more public variables. Public variable definitions must come before procedure or function declarations. *declaration* can be any of the following:

- Variable Declaration (see Section 4.3)
- Record Declaration (see Section 4.3.4)
- Collection Declaration (see Section 4.10)
- `REF CURSOR` and Cursor Variable Declaration (see Section 4.9)
- `TYPE` Definitions for Records, Collections, and `REF CURSOR`s
- Object Variable Declaration (see Section 8.3)

proc_name is an identifier of a public procedure. Public procedures can be invoked from outside of the package using the syntax *package_name.proc_name[(...)]*. 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`.

func_name is an identifier of a public function. Public functions can be invoked from outside of the package using the syntax *package_name.func_name[(...)]*. 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`. *`return_type`* is the data type of the value the function returns. `IN` parameters can also be initialized with a default value which is used in place of any `IN` parameter you miss.

6.1.2 Package Body Syntax

The following is the syntax for the package body:

```
CREATE [ OR REPLACE ] PACKAGE BODY package_name
  { IS | AS }
  [ private_declaration; ] ...
  [ { PROCEDURE proc_name
        [ (parm1 [IN | IN OUT | OUT ] datatype1
        [, parm2 [IN | IN OUT | OUT ] datatype2 ] ...) ]
      { IS | AS }
      [ proc_declaration; ] ...
        BEGIN
          statement; ...
      [ EXCEPTION
          WHEN ... THEN
            statement; ...]
        END;
      |
        FUNCTION func_name
        [ (parm1 [IN | IN OUT | OUT ] datatype1
        [, parm2 [IN | IN OUT | OUT ] datatype2 ] ...) ]
        RETURN return_type
       {IS | AS }
       [ func_declaration; ]...
        BEGIN
          statement; ...
      [ EXCEPTION
          WHEN ... THEN
            statement; ...]
          END; }...]
      [ BEGIN
          init_statement; ...]
END [ 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 the same name.

`private_declaration` is an identifier of a private variable that can be accessed by any procedure or function within the package. There can be none, one, or more private variables. *`private_declaration`* can be any of the following:

- Variable Declaration (see Section 4.3)
- Record Declaration (see Section 4.3.4)

- Collection Declaration (see Section 4.10)
- `REF CURSOR` and Cursor Variable Declaration (see Section 4.9)
- `TYPE` Definitions for Records, Collections, and `REF CURSOR`s
- Object Variable Declaration (see Section 8.3)

If *`proc_name`* is the same as the identifier of a public procedure declared in the package specification and the signature of *`proc_name`* (i.e., formal parameter names (*`parm1`*, *`parm2`*,...), data types (*`datatype1`*, *`datatype2`*,...), parameter modes, order of formal parameters, and number of formal parameters) exactly matches the signature of the public procedure's declaration, then *`proc_name`* defines the body of this public procedure.

If the conditions described in the prior paragraph are not true, then *`proc_name`* defines a private procedure.

`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`. `IN` parameters can also be initialized with a default value which is used in place of any `IN` parameter you miss.

`proc_variable` is an identifier of a variable that can be accessed only from within procedure, *`proc_name`*. There can be none, one, or more variables. *`datatype`* is the data type of *`proc_variable`*. *`statement`* is an SPL program statement.

If *`func_name`* is the same as the identifier of a public function declared in the package specification and the signature of *`func_name`* (i.e., formal parameter names (*`parm1`*, *`parm2`*,...), data types (*`datatype1`*, *`datatype2`*,...), parameter modes, order of formal parameters, and number of formal parameters) exactly matches the signature of the public function's declaration, then *`func_name`* defines the body of this public function.

If the conditions described in the prior paragraph are not true, then *`func_name`* defines a private function.

`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`. *`return_type`* is the data type of the value returned by the function.

`func_variable` is an identifier of a variable that can be accessed only from within function, *`func_name`*. There can be none, one, or more variables. *`datatype`* is the data type of *`func_variable`*. *`statement`* is an SPL program statement.

`init_statement` is a statement in the initialization section of the package body. The initialization section, if specified, must contain at least one statement. The statements in

the initialization section are executed once per user's session when the package is first referenced.

6.2 Creating Packages

We will now try to create packages and store them in our database. One thing to remember here is that packages are not executable piece of code. Rather they are a repository of code that is used. When you use a package, you actually execute or make reference to an element in a package. This information is contained in the package specification.

6.2.1 Creating the Package Specification

The package specification contains definition of all the elements in the package that can be referenced from outside it. These are called the public elements of the package and act is the package interface. Following 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;
```

Here we have created 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.

```
--
--  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. See Section 4.3.4 and Section 4.9 for information on record types and `REF CURSOR`s, respectively.

```
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`. See Sections 4.8, 4.10, and 4.3.4 for information on static cursors, arrays, and record variables, respectively.

```
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 Postgres Plus 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.

The procedures and functions available in the `DBMS_ALERT` package are listed in the following table.

Table 7-37 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.

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:

```
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_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 7-38 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 7-39 DBMS_OUTPUT Public Variables

Public Variables	Data Type	Value	Description
`chararr`	`TABLE`		For message lines.

7.2.1 CHARARR

The `CHARARR` is for storing multiple message lines.

```
TYPE chararr IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

7.2.2 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;
```

7.2.3 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.

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;
```

7.2.4 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)
```

7.2.5 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 examples 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)
```

7.2.6 NEW_LINE

The `NEW_LINE` procedure writes an end-of-line character sequence in the message buffer.

```
NEW_LINE
```

7.2.7 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
```

7.2.8 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
```

7.2.9 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 Postgres Plus 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
```

7.3 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 7-40 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.

7.3.1 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

Create 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
```

Create 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
```

7.3.2 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 7-41.

Table 7-41 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 Oracle compatible. 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
```

7.3.3 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.

7.3.4 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
```

7.3.5 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 7-42 RECEIVE_MESSAGE Status Codes

Status Code	Description
0	Success
1	Time out
2	Message too large .for the buffer

7.3.6 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
```

7.3.7 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?
```

7.3.8 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 7-43 SEND_MESSAGE Status Codes

Status Code	Description
0	Success
1	Time out
3	Function interrupted

7.3.9 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
```

7.3.10 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.

7.3.11 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
```

7.4 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 7-44 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.

Function/Procedure	Return Type	Description
`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.
`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`*.

7.4.1 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.

7.4.2 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
```

7.4.3 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
```

7.4.4 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;
```

7.4.5 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 Oracle compatible - Oracle does count the end-of-line character(s).

`filetype`

> Variable of type `FILE_TYPE` containing the file handle of the opened file.

7.4.6 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
```

7.4.7 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
```

7.4.8 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
```

7.4.9 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.

7.4.10 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
```

7.4.11 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
```

7.4.12 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
```

7.4.13 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 Postgres Plus Advanced Server, a new line character must be specified with two consecutive backslashes instead of one - \\n. This characteristic is not Oracle compatible.

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 Oracle compatible.

```
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
```

7.5 DBMS_SQL

The DBMS_SQL package provides an Oracle compatible application interface 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 an Oracle compatible fashion without modifying your application.

DBMS_SQL assumes the privileges of the current user when executing dynamic SQL statements.

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.
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.

The following table lists the public variable available in the DBMS_SQL package.

Table 7-45 DBMS_SQL Public Variables

Public Variables	Data Type	Value	Description
`native`	`INTEGER`	1	Provided for Oracle syntax compatibility. See DBMS_SQL.PARSE for more information.
`V6`	`INTEGER`	2	Provided for Oracle syntax compatibility. See DBMS_SQL.PARSE for more information.
`V7`	`INTEGER`	3	Provided for Oracle syntax compatibility. See DBMS_SQL.PARSE for more information

7.5.1 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
```

7.5.2 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.

7.5.3 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.

7.5.4 CLOSE_CURSOR

The `CLOSE_CURSOR` procedure closes an open cursor. The resources allocated to the cursor are released and it cannot 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;
```

7.5.5 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. See the example for FETCH_ROWS for the complete anonymous block.

```
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;
```

7.5.6 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.

7.5.7 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.

7.5.8 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 ])
```

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;
```

7.5.9 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.

7.5.10 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.

7.5.11 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;
```

7.5.12 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
```

7.5.13 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
```

7.5.14 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.

7.5.15 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
```

7.5.16 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;
```

7.5.17 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 Oracle syntax compatibility. 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;
```

7.6 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 Postgres Plus 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.

You can find more information about configuring and starting the `pgAgent` service in the `pgAgent` README file in the doc subdirectory of the installation tree.

Table 7-46 DBMS_JOB Function/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.

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;
```

7.6.1 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;
```

7.6.2 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.

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;
```

7.6.3 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;
```

7.6.4 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;
```

7.6.5 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;
```

7.6.6 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;
```

7.6.7 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)
```

7.6.8 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;
```

7.7 DBMS_LOB

The `DBMS_LOB` package provides the capability to operate on large objects.

Table 7-47 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.
`CLOSE(`*`lob_loc`*` IN OUT)`	Procedure	n/a	Close an open large object.
`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`*.
`ISOPEN(`*`lob_loc`*`)`	Function	`INTEGER`	Check if the large object is open.
`OPEN(`*`lob_loc`*` IN OUT, `*`open_mode`*`)`	Procedure	n/a	Open a large object.
`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.

The following table lists the public variables available in the package.

Table 7-48 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.

7.7.1 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*.

7.7.2 CLOSE

Note: This procedure exists for compatibility only, and is ignored. Calls to this procedure have no effect.

```
CLOSE(lob_loc IN OUT { BLOB | CLOB })
```

Parameters

lob_loc

Large object locator of the large object to be closed.

7.7.3 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.

7.7.4 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.

7.7.5 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.

7.7.6 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.

7.7.7 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 BLOBs or with spaces for CLOBs. 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.

7.7.8 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.

7.7.9 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.

7.7.10 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*.

7.7.11 ISOPEN

This procedure exists for compatibility only, and is ignored. Calls to this procedure have no effect.

```
status INTEGER ISOPEN(lob_loc { BLOB | CLOB })
```

Parameters

lob_loc

Large object locator for the large object to be tested.

status

1 if the large object has been opened, 0 otherwise.

7.7.12 OPEN

The `OPEN` procedure opens a large object for either read-only or read-write modes. This procedure exists for compatibility only, and is ignored. Calls to this procedure have no effect.

```
OPEN(lob_loc IN OUT { BLOB | CLOB }, open_mode BINARY_INTEGER)
```

Parameters

lob_loc

Large object locator of the large object to be opened.

open_mode

Mode in which to open the large object. Set to 0 (`lob_readonly`) for read-only mode. Set to 1 (`lob_readwrite`) for read-write mode.

7.7.13 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.

7.7.14 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.

7.7.15 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.

7.7.16 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`.

7.7.17 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.

7.8 DBMS_UTILITY

The `DBMS_UTILITY` package provides various utility programs.

Table 7-49 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.
`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.

The following table lists the public variables available in the `DBMS_UTILITY` package.

Table 7-50 DBMS_UTILITY Public Variables

Public Variables	Data Type	Value	Description
`inv_error_on_restrictions`	`PLS_INTEGER`	1	Used by the `INVALIDATE` procedure.
`lname_array`	`TABLE`		For lists of long names.
`uncl_array`	`TABLE`		For lists of users and names.

7.8.1 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;
```

7.8.2 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;
```

7.8.3 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` or `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.

7.8.4 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
```

7.8.5 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`*. See LNAME_ARRAY or UNCL_ARRAY for a description of *`tab`*.

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
```

7.8.6 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(80);
    v_compat         VARCHAR2(80);
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: 8.3.0.106
Compatibility: 8.3.0.106
```

7.8.7 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;
```

7.8.8 GET_CPU_TIME

The `GET_CPU_TIME` function returns the CPU time in 100th's of a second from some arbitrary point in time.

```
cputime NUMBER GET_CPU_TIME
```

Parameters

cputime

Number of 100th's of a second of CPU time.

Examples

The following `SELECT` command retrieves the current CPU time.

```
SELECT DBMS_UTILITY.GET_CPU_TIME FROM DUAL;

get_cpu_time
--------------
          603
```

7.8.9 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.

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()
```

7.8.10 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
```

7.8.11 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
```

7.8.12 GET_TIME

The `GET_TIME` function provides the capability to return the current time in 100th's of a second.

```
time NUMBER GET_TIME
```

Parameters

time

Number of 100th's 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
```

7.8.13 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
```

7.8.14 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. See LNAME_ARRAY or UNCL_ARRAY for a description of *`tab`*.

`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
```

7.9 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 7-51 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.

7.9.1 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;
```

7.9.2 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.

7.9.3 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.

7.10 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 7-52 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.

The following table lists the public variables available in the `UTL_SMTP` package.

Table 7-53 UTL_SMTP Public Variables

Public Variables	Data Type	Value	Description
`connection`	`RECORD`		Description of an SMTP connection.
`reply`	`RECORD`		SMTP reply line.

7.10.1 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
);
```

7.10.2 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;
```

7.10.3 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.

7.10.4 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 REPLY/REPLIES.

7.10.5 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 REPLY/REPLIES.

7.10.6 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.

7.10.7 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.

7.10.8 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.

7.10.9 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 REPLY/REPLIES.

7.10.10 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.

7.10.11 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 REPLY/REPLIES.

7.10.12 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.

7.10.13 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.

7.10.14 QUIT

The `QUIT` procedure closes the session with an SMTP server.

```
QUIT(c IN OUT CONNECTION)
```

Parameters

c

SMTP connection to be terminated.

7.10.15 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.

7.10.16 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.

7.10.17 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 REPLY/REPLIES.

7.10.18 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.

7.10.19 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');
```

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 and object type as used in this chapter. The prior usage of these terms is in a general sense to mean the entities that can be created in a database such as tables, views, indexes, users, etc. Within the context of this chapter, object and object type refer to specific data structures and code that are well-defined by the SPL programming language.

As was stated at the beginning of this chapter, an object type is a description or definition of something. 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 remainder of this chapter delves into the creation and usage of object types and objects in SPL.

Note: Implementation of SPL object types and objects is following a phased approach. As of this release, support of methods along with certain other features of most object-oriented programming languages have not yet been implemented. This chapter documents only those features that have currently been implemented.

8.1 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.

Note: Only the object type specification and attributes are supported at this time.

8.1.1 Object Type Specification Syntax

The following is the syntax of the object type specification:

```
CREATE [ OR REPLACE ] TYPE name { IS | AS } OBJECT
  ({ attribute { datatype | objtype } } [, ...])
```

`name` is an identifier assigned to the object type. `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.

8.2 Creating Object Types

The `CREATE TYPE AS OBJECT` command is used to create the object type specification. The following example creates the `addr_obj_typ` object type.

```
CREATE OR REPLACE TYPE addr_obj_typ AS OBJECT (
    street          VARCHAR2(30),
    city            VARCHAR2(20),
    state           CHAR(2),
    zip             NUMBER(5)
);
```

The following object type specification creates the `emp_obj_typ` object type. In this example, the `addr` attribute is defined by the `addr_obj_typ` object type.

```
CREATE TYPE emp_obj_typ AS OBJECT (
    empno           NUMBER(4),
    ename           VARCHAR2(20),
    addr            ADDR_OBJ_TYP
);
```

8.3 Creating Object Instances

Creating instances of an object type requires the following steps.

- Declare an *object variable* of the object type
- Initialize the declared object variable with initial values

The syntax for declaring an object variable is as follows.

```
object objtype
```

`object` is an identifier assigned to the object variable. `objtype` is the identifier of a previously defined object type.

The next step is to initialize the object variable with values. The following is the syntax of an object initialization expression.

```
[ 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. `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 `NULL` is specified, the corresponding object attribute is set to null. If an attribute is of an object type, then the corresponding expression can be null or an object initialization expression.

Note: In Oracle, the initialization process is done with a constructor function that takes the name of the object type. The constructor function name takes the place of the `ROW` keyword in the initialization expression. Constructor functions are not supported in SPL at this time.

In Oracle the syntax is the following:

```
objtype  ( { expr1 | NULL } [, { expr2 | NULL } ] [, ...] )
```

The following anonymous block declares a variable of type `emp_obj_typ` named `v_emp`, and initializes it.

```
DECLARE
    v_emp           EMP_OBJ_TYP;
BEGIN
    v_emp := (9001,'JONES',
        ('123 MAIN STREET','EDISON','NJ',08817));
END;
```

Note: In Oracle the assignment statement in the anonymous block would take the following form:

```
    v_emp := emp_obj_typ (9001,'JONES',
        addr_obj_typ('123 MAIN STREET','EDISON','NJ',08817));
```

8.4 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 := (9001,'JONES',
        ('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 || ' ' || v_emp.addr.zip);
END;
```

The following is the output from this anonymous block.

```
Employee No    : 9001
Name           : JONES
Street         : 123 MAIN STREET
City/State/Zip: EDISON, NJ 8817
```

8.5 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, these nested object types must be dropped last.

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;
```

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 a Postgres Plus Advanced Server or an Oracle database with minimal to no changes to the application code. The Open Client Library was written in C code from scratch.

9.1 Comparison with Oracle Call Interface

The following diagram compares the Open Client Library and Oracle Call Interface application stacks.

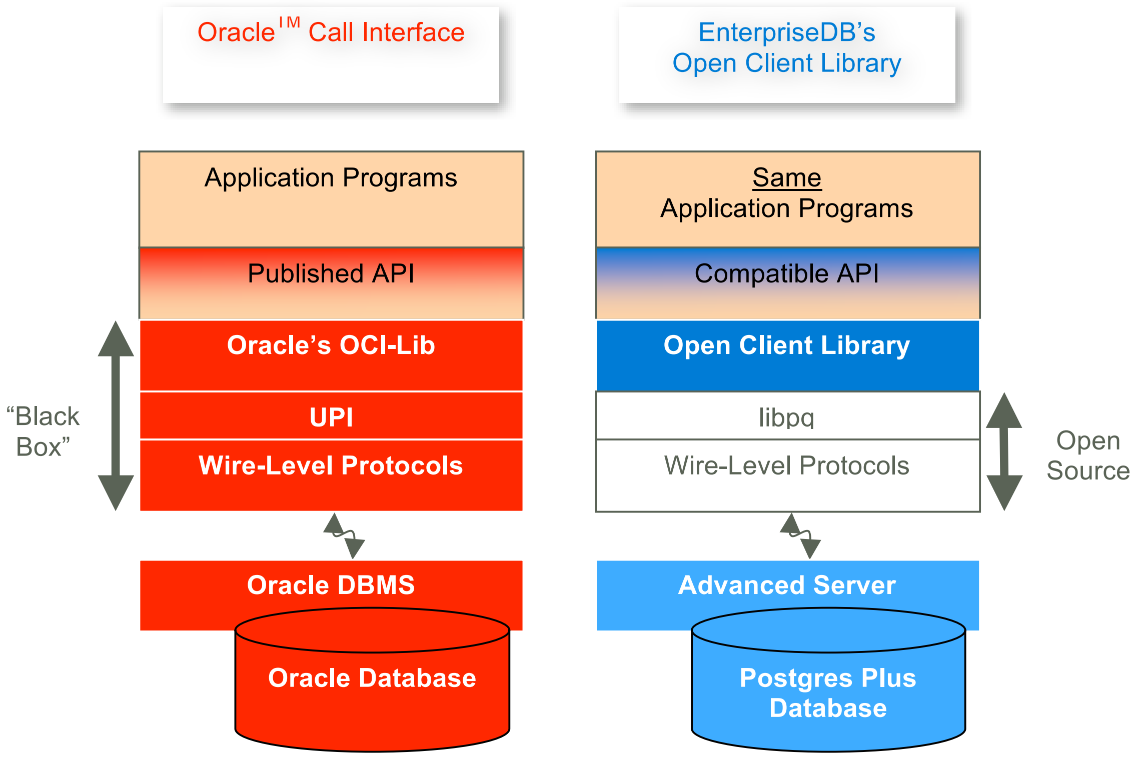


Figure 6 Open Client Library

9.2 OCL 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. Postgres Plus Advanced Server does not supply any such files.

Table 9-54 Connect, Authorize, and Initialize Functions

Function	Description
`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.
`OCIServerAttach`	Establish an access path to a data source.
`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.

Table 9-55 Handle and Descriptor Functions

Function	Description
`OCIAttrGet`	Get handle attributes.
`OCIAttrSet`	Set handle attributes.
`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.

Table 9-56 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.
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.

Table 9-57 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.

Table 9-58 Transaction Functions

Function	Description
OCITransCommit	Commit a transaction.
OCITransRollback	Roll back a transaction.

Table 9-59 Miscellaneous Functions

Function	Description
OCIClientVersion	Return client library version.
OCIErrorGet	Return error message.
OCIPasswordChange	Change password.
OCIPing	Confirm that the connection and server are active.
OCIServerVersion	Get the Oracle version string.

Table 9-60 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.
`OCIDateTimeToArray`	Convert an `OCIDateTime` descriptor to an array.
`OCIDateTimeToText`	Convert the given date to a string according to the specified format.

Table 9-61 NUMBER Functions

Function	Description
`OCINumberAbs`	Compute the absolute value.
`OCINumberAdd`	Adds `NUMBER`s.
`OCINumberArcCos`	Compute the arc cosine.
`OCINumberArcSin`	Compute the arc sine.

Function	Description
OCINumberArcTan	Compute the arc tangent.
OCINumberArcTan2	Compute the arc tangent of two NUMBERs.
OCINumberAssign	Assign one NUMBER to another.
OCINumberCeil	Compute the ceiling of NUMBER.
OCINumberCmp	Compare NUMBERs.
OCINumberCos	Compute the cosine.
OCINumberDec	Decrement a NUMBER.
OCINumberDiv	Divide two NUMBERs.
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 NUMBERs.
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 NUMBERs.
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.

Table 9-62 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.

Table 9-63 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.

Table 9-64 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

Function	Description
`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 on Oracle compatible database objects in a manner compatible with the Oracle data dictionary views found in an Oracle database.

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.
`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.

Name	Type	Description
`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 (`Y` or `N`).
`deferred`	`BOOLEAN`	Specifies if the constraint has been deferred (`Y` or `N`).
`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.
`schema_name`	`TEXT`	Name of the schema in which the link belongs.
`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.

Name	Type	Description
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	CHAR(1)	Sorted order of the column on disk. 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.
index_schema	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	CHAR(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	Whether or not the state of the object is valid. (VALID or INVALID).
partitioned	CHAR(3)	Indicates that the index is partitioned. Currently, always set to NO.
temporary	CHAR(1)	Indicates that an index is on a temporary table. Always set to N; included for compatibility only.
secondary	CHAR(1)	Included for compatibility only. Always set to N.
join_index	CHAR(3)	Included for compatibility only. Always set to NO.
dropped	CHAR(3)	Included for compatibility only. Always set to NO.

10.7 ALL_OBJECTS

The ALL_OBJECTS view provides information on the following database objects – tables, indexes, sequences, views, triggers, functions, procedures, packages, and package bodies. Note that only SPL triggers, functions, procedures, packages, and package bodies are shown – PL/pgSQL triggers and functions do not appear in the ALL_OBJECTS view.

Name	Type	Description
owner	VARCHAR2	User name of the object's owner.
schemaname	VARCHAR2	Name of the schema in which the object belongs.
object_name	VARCHAR2	Name of the object.
object_type	VARCHAR2	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	VARCHAR2	Whether or not the state of the object is valid. Currently, always set to VALID.

10.8 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	VARCHAR2	User name of the program's owner.
schemaname	VARCHAR2	Name of the schema in which the program belongs.
name	VARCHAR2	Name of the program.
type	VARCHAR2	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	VARCHAR2	Line of source code text.

10.9 ALL_SYNONYMS

The ALL_SYNONYMS view provides information on all synonyms that may be referenced by the current user.

Name	Type	Description
owner	VARCHAR2	User name of the synonym's owner.
schemaname	VARCHAR2	Name of the schema in which the synonym belongs.
name	VARCHAR2	Name of the program.
type	VARCHAR2	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	VARCHAR2	Line of source code text.

10.10 ALL_TAB_COLUMNS

The ALL_TAB_COLUMNS view provides information on all columns in all user-defined tables.

Name	Type	Description
owner	VARCHAR2	User name of the table's owner.

Name	Type	Description
schemaname	VARCHAR2	Name of the schema in which the table belongs.
table_name	VARCHAR2	Name of the table.
column_name	VARCHAR2	Name of the column.
data_type	VARCHAR2	Data type of the column.
data_length	INTEGER	Length of text columns.
data_precision	INTEGER	Precision (number of digits) for NUMBER columns.
data_scale	INTEGER	Scale of NUMBER columns.
column_id	INTEGER	Relative position of the column within the table.
nullable	CHARACTER	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
data_default	VARCHAR2	Default value assigned to the column.

10.11ALL_TABLES

The ALL_TABLES view provides information on all user-defined tables.

Name	Type	Description
owner	VARCHAR2	User name of the table's owner.
schemaname	VARCHAR2	Name of the schema in which the table belongs.
table_name	VARCHAR2	Name of the table.
table_space	VARCHAR2	Name of the tablespace in which the table resides if other than the default tablespace.
status	VARCHAR2	Whether or not the state of the table is valid. Currently, always set to VALID.

10.12ALL_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.
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

Name	Type	Description
		(NOTVALID).
description	TEXT	Included for compatibility only. Value will always be SEE TRIGGER BODY FOR TEXT.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

10.13 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.

10.14 ALL_USERS

The ALL_USERS view provides information on all user names.

Name	Type	Description
username	VARCHAR2	Name of the user.
user_id	VARCHAR2	Numeric user id assigned to the user.
created	TIMESTAMP	Always NULL; Included for compatibility only.

10.15 ALL_VIEW_COLUMNS

The ALL_VIEW_COLUMNS view provides information on all columns in all user-defined views.

Name	Type	Description
Owner	VARCHAR2	User name of the view's owner.
schemaname	VARCHAR2	Name of the schema in which the view belongs.
view_name	VARCHAR2	Name of the view.
column_name	VARCHAR2	Name of the column.
data_type	VARCHAR2	Data type of the column.
data_length	INTEGER	Length of text columns.
data_precision	INTEGER	Precision (number of digits) for NUMBER columns.

Name	Type	Description
data_scale	INTEGER	Scale of NUMBER columns.
column_id	INTEGER	Relative position of the column within the view.
nullable	CHARACTER	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
data_default	VARCHAR2	Default value assigned to the column.

10.16 ALL_VIEWS

The ALL_VIEWS view provides information on all user-defined views.

Name	Type	Description
Owner	VARCHAR2	User name of the view's owner.
schemaname	VARCHAR2	Name of the schema in which the view belongs.
view_name	VARCHAR2	Name of the view.
status	VARCHAR2	Whether or not the state of the view is valid. Currently, always set to VALID.

10.17 DBA_ALL_TABLES

The DBA_ALL_TABLES view provides information about all tables in the database.

Name	Type	Description
owner	VARCHAR2	User name of the table's owner.
schema_name	VARCHAR2	Name of the schema in which the table belongs.
table_name	VARCHAR2	Name of the table.
tablespace_name	VARCHAR2	Name of the tablespace in which the table resides if other than the default tablespace.
status	VARCHAR2	Whether or not the state of the table is valid. Currently, always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

10.18 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.

Name	Type	Description
constraint_def	TEXT	The definition of the constraint.

10.19 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 (Y or N).
deferred	BOOLEAN	Specifies if the constraint has been deferred (Y or N).
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.

10.20 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.
schema_name	TEXT	Name of the schema in which the link belongs.
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.

10.21 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	CHAR(1)	Sorted order of the column on disk. Always set to Y (descending); included for compatibility only.

10.22 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.
index_schema	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. Currently, always set to TABLE.
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHAR(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. Currently always set to LOGGING.
status	TEXT	Whether or not the state of the object is valid. (VALID or INVALID).
partitioned	CHAR(3)	Indicates that the index is partitioned. Currently, always set to NO.
temporary	CHAR(1)	Indicates that an index is on a temporary table. Currently, always set to N.

Name	Type	Description
secondary	CHAR(1)	Included for compatibility only. Currently always set to N.
join_index	CHAR(3)	Included for compatibility only. Currently always set to NO.
dropped	CHAR(3)	Included for compatibility only. Currently always set to NO.

10.23 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.
What	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
nls_env	VARCHAR2 (4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

10.24 DBA_OBJECTS

The `DBA_OBJECTS` view provides information about all objects in the database.

Name	Type	Description
owner	VARCHAR2	User name of the object's owner.
schema_name	VARCHAR2	Name of the schema in which the object belongs.
object_name	VARCHAR2	Name of the object.
object_type	VARCHAR2	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.

Name	Type	Description
status	VARCHAR2	Whether or not the state of the object is valid. Currently, always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

10.25 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	VARCHAR2	User name to whom the role was granted.
granted_role	VARCHAR2	Name of the role granted to the grantee.
admin_option	VARCHAR2	YES if the role was granted with the admin option, NO otherwise.
default_role	VARCHAR2	YES if the role is automatically enabled when the grantee creates a session, NO otherwise. Based on rolinherit in pg_roles. If rolinherit is TRUE, default_role is YES. If rolinherit is FALSE, default_role is NO.

10.26 DBA_ROLES

The DBA_ROLES view provides information on all roles with the NOLOGIN attribute (groups).

Name	Type	Description
role	VARCHAR2	Name of a role having the NOLOGIN attribute – i.e., a group.
password_required	VARCHAR2	Whether or not a password is required to use the role. Always N. Included for compatibility only.

10.27 DBA_SOURCE

The DBA_SOURCE view provides the source code listing of all objects in the database.

Name	Type	Description
owner	VARCHAR2	User name of the program's owner.
schemaname	VARCHAR2	Name of the schema in which the program belongs.
name	VARCHAR2	Name of the program.
type	VARCHAR2	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	VARCHAR2	Line of source code text.

10.28 DBA_SYNONYMS

The `DBA_SYNONYM` view provides information about all synonyms in the database.

Name	Type	Description
`owner`	`VARCHAR2`	User name of the synonym's owner.
`schema_name`	`TEXT`	Name of the schema in which the program belongs.
`synonym_name`	`VARCHAR2`	Name of the synonym.
`table_owner`	`VARCHAR2`	User name of the table's owner on which the synonym is defined.
`table_name`	`VARCHAR2`	Name of the table on which the synonym is defined.
`db_link`	`VARCHAR2`	Name of any associated database link.

10.29 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.
`schemaname`	`TEXT`	Name of the schema in which the table belongs.
`table_name`	`TEXT`	Name of the table.
`table_space`	`TEXT`	Name of the tablespace in which the table resides if other than the default tablespace.
`status`	`CHAR(5)`	Whether or not the state of the table is valid. Currently, always set to `VALID`.
`temporary`	`CHAR(1)`	Y if the table is temporary; N if the table is permanent.

10.30 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.
`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`.

Name	Type	Description
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only. Value will always be SEE TRIGGER BODY FOR TEXT.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

10.31 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.

10.32 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	VARCHAR2(30)	The password (encrypted) of the user.
account_status	VARCHAR2(32)	The current status of the account. Possible values are: EXPIRED & LOCKED OPEN LOCKED
lock_date	TIMESTAMP W/O ZONE	Included for compatibility only. The value is always NULL.
expiry_date	TIMESTAMP W/O ZONE	The expiration date of the account.
default_tablespace	VARCHAR2(30)	The default tablespace associated with the account.
temporary_tablespace	VARCHAR2(30)	Included for compatibility only. The value will always be '' (an empty string).
created	TIMESTAMP W/O ZONE	Included for compatibility only. The value is always NULL.
profile	VARCHAR2(30)	Included for compatibility only. The value is always NULL.
initial_rsrc_consumer_group	VARCHAR2(30)	Included for compatibility only. The value is always

Name	Type	Description
		NULL.
external_name	VARCHAR2 (4000)	Included for compatibility only. The value is always NULL.

10.33 DBA_VIEWS

The DBA_VIEWS view provides information about all views in the database.

Name	Type	Description
owner	VARCHAR2	User name of the view's owner.
schema_name	VARCHAR2	Name of the schema in which the view belongs.
view_name	VARCHAR2	Name of the view.
text	TEXT	The text of the SELECT statement that defines the view.

10.34 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	VARCHAR2(5)	Whether or not the state of the table is valid. Currently, always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

10.35 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.

10.36 USER_CONSTRAINTS

The `USER_CONSTRAINTS` view provides information about all constraints placed on tables that are owned by the current user.

Name	Type	Description
`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 (`Y` or `N`).
`deferred`	`BOOLEAN`	Specifies if the constraint has been deferred (`Y` or `N`).
`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.

10.37 USER_DB_LINKS

The `USER_DB_LINKS` view provides information about all database links that are owned by the current user.

Name	Type	Description
`schema_name`	`TEXT`	Name of the schema in which the link belongs.
`db_link`	`TEXT`	The name of the database link.
`type`	`VARCHAR2`	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.

10.38 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_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`	`CHAR(1)`	Sorted order of the column on disk. Always set to `Y` (descending); included for compatibility only.

10.39 USER_INDEXES

The `USER_INDEXES` view provides information about all indexes on tables that are owned by the current user.

Name	Type	Description
`owner`	`TEXT`	User name of the index's owner.
`index_schema`	`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. Currently, always set to `TABLE`.
`uniqueness`	`TEXT`	Indicates if the index is `UNIQUE` or `NONUNIQUE`.
`compression`	`CHAR(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. Currently always set to `LOGGING`.
`status`	`TEXT`	Whether or not the state of the object is valid. (`VALID` or `INVALID`).
`partitioned`	`CHAR(3)`	Indicates that the index is partitioned. Currently, always set to `NO`.
`temporary`	`CHAR(1)`	Indicates that an index is on a temporary table. Currently, always set to `N`.
`secondary`	`CHAR(1)`	Included for compatibility only. Currently always set to `N`.
`join_index`	`CHAR(3)`	Included for compatibility only. Currently always set to `NO`.

Name	Type	Description
dropped	CHAR(3)	Included for compatibility only. Currently always set to NO.

10.40 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	integer	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.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
nls_env	VARCHAR2(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

10.41 USER_OBJECTS

The USER_OBJECTS view provides information about all objects that are owned by the current user.

Name	Type	Description
owner	VARCHAR2	User name of the object's owner.
schemaname	VARCHAR2	Name of the schema in which the object belongs.
object_name	VARCHAR2	Name of the object.
object_type	VARCHAR2	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	VARCHAR2	Whether or not the state of the object is valid. Currently,

Name	Type	Description
		always set to VALID.

10.42 USER_SOURCE

The USER_SOURCE view provides information about all programs owned by the current user.

Name	Type	Description
schema_name	VARCHAR2	Name of the schema in which the program belongs.
name	VARCHAR2	Name of the program.
type	VARCHAR2	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	VARCHAR2	Line of source code text.

10.43 USER_SYNONYMS

The

ALL_SYNONYMS view provides information about all synonyms owned by the current user.

Name	Type	Description
owner	TEXT	User name of the synonym's owner.
synonym_name	TEXT	Name of the synonym.
object_owner	TEXT	User name of the table's owner on which the synonym is defined.
object_name	TEXT	Name of the table on which the synonym is defined.
synacl	ACLITEM[]	The access control list for the synonym.
status	VARCHAR2(5)	Always VALID; included for compatibility only.
db_link	TEXT	Name of any associated database link.

10.44 USER_TAB_COLUMNS

The USER_TAB_COLUMNS view displays information about all columns in tables owned by the current user.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schemaname	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
column_name	TEXT	Name of the column.
data_type	VARCHAR2	Data type of the column.
data_length	INTEGER	Length of text columns.

Name	Type	Description
data_precision	INTEGER	Precision (number of digits) for NUMBER columns.
data_scale	INTEGER	Scale of NUMBER columns.
column_id	INTEGER	Relative position of the column within the table.
nullable	BPCHAR	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
data_default	VARCHAR2	Default value assigned to the column.

10.45 USER_TABLES

The USER_TABLES view displays information about all tables owned by the current user.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schemaname	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
table_space	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	VARCHAR2(5)	Whether or not the state of the table is valid. Currently, always set to VALID.

10.46 USER_TRIGGERS

The USER_TRIGGERS view displays information about all triggers on tables owned by the current user.

Name	Type	Description
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. Value will always be SEE TRIGGER BODY FOR TEXT.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

10.47 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.

10.48 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`	`VARCHAR2(32)`	The current status of the account. Possible values are: `EXPIRED & LOCKED` `OPEN` `LOCKED`
`lock_date`	`TIMESTAMP W/O ZONE`	Included for compatibility only. The value is always `NULL`.
`expiry_date`	`TIMESTAMP W/O ZONE`	The expiration date of the account.
`default_tablespace`	`VARCHAR2(30)`	The default tablespace associated with the account.
`temporary_tablespace`	`VARCHAR2(30)`	Included for compatibility only. The value will always be '' (an empty string).
`created`	`TIMESTAMP W/O ZONE`	Included for compatibility only. The value will always be `NULL`.
`initial_rsrc_consumer_group`	`VARCHAR2(30)`	Included for compatibility only. The value will always be `NULL`.
`external_name`	`VARCHAR2 (4000)`	Included for compatibility only. The value will always be `NULL`.

10.49 USER_VIEW_COLUMNS

The `USER_VIEW_COLUMNS` view provides information about all columns in views owned by the current user.

Name	Type	Description
`Owner`	`VARCHAR2`	User name of the view's owner.

Name	Type	Description
`schemaname`	`VARCHAR2`	Name of the schema in which the view belongs.
`view_name`	`VARCHAR2`	Name of the view.
`column_name`	`VARCHAR2`	Name of the column.
`data_type`	`VARCHAR2`	Data type of the column.
`data_length`	`INTEGER`	Length of text columns.
`data_precision`	`INTEGER`	Precision (number of digits) for `NUMBER` columns.
`data_scale`	`INTEGER`	Scale of `NUMBER` columns.
`column_id`	`INTEGER`	Relative position of the column within the view.
`nullable`	`CHARACTER`	Whether or not the column is nullable – possible values are: `Y` – column is nullable; `N` – column does not allow null.
`data_default`	`VARCHAR2`	Default value assigned to the column.

10.50 USER_VIEWS

The `USER_VIEWS` view provides information about all views owned by the current user.

Name	Type	Description
`Owner`	`VARCHAR2`	User name of the view's owner.
`schemaname`	`VARCHAR2`	Name of the schema in which the view belongs.
`view_name`	`VARCHAR2`	Name of the view.
`status`	`VARCHAR2`	Whether or not the state of the view is valid. Currently, always set to `VALID`.

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 Postgres Plus 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
- Recording output

The following section describes how to connect to an Postgres Plus Advanced Server database using EDB*Plus. The final section provides a summary of the EDB*Plus commands.

11.1.1 Starting EDB*Plus

EDB*Plus can be started by selecting it from the application menu or by running the EDB*Plus program directly from the operating system command line. For the latter, the EDB*Plus program is invoked by running `edbplus` from the `edbplus` subdirectory located under the Postgres Plus Advanced Server home directory as follows:

```
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* }]

username is a database username with which to connect to the database. *password* is the password for *username*. If *password* is omitted, but a password is required, EDB*Plus will prompt for the password. Either @*connectstring* or @*variable* may be specified where *connectstring* is the database connection string or *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 Postgres Plus Advanced Server home directory.) In either case, the database connection string takes the following format. (There must be no white space within the connection string.)

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`

If `/NOLOG` is specified, EDB*Plus is started 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:\EnterpriseDB\8.3\edbplus>edbplus enterprisedb/password
Connected to EnterpriseDB 8.3.0.10 (localhost:5444/edb) AS enterprisedb

EDB*Plus: Release 8.3 - Beta (Build 12)
Copyright (c) 2008, 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:\EnterpriseDB\8.3\edbplus>edbplus enterprisedb/password@localhost:5445/edb
Connected to EnterpriseDB 8.3.0.10 (localhost:5445/edb) AS enterprisedb

EDB*Plus: Release 8.3 - Beta (Build 12)
Copyright (c) 2008, 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:\EnterpriseDB\8.3\edbplus>edbplus enterprisedb/password@hr_5445
Connected to EnterpriseDB 8.3.0.10 (localhost:5445/hr) AS enterprisedb

EDB*Plus: Release 8.3 - Beta (Build 12)
Copyright (c) 2008, 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:\EnterpriseDB\8.3\edbplus>edbplus enterprisedb/password @dept_query
Connected to EnterpriseDB 8.3.0.10 (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
```

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 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
```

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 formatting of output. 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:

- No other options
- `CLEAR`
- 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:

```
An
```

`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 10-65 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.

```
CONNECT username[/password][@{connectstring | variable } ]
```

The `CONNECT` command parameters have the same meaning as those of the *`login`* parameter for starting up EDB*Plus from the command line. See the description of the *`login`* parameter in Section 11.1.1.

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
Connected to EnterpriseDB 8.3.0.10 (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
Connected to EnterpriseDB 8.3.0.10 (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 Postgres Plus 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.

The following are the meanings of the parameters.

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, data types, and lengths for a table or view; a list of parameters for a procedure or function; or a list of procedures and functions and their respective parameters for a package.

```
DESCRIBE [ 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.

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 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`.

11.1.2.13 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`.

11.1.2.14 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.

11.1.2.15 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.

11.1.2.16 INPUT

`INPUT` is a line editor command that 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
```

11.2 EDB*Loader

EDB*Loader is a high-performance bulk data loader that provides an Oracle compatible interface for Postgres Plus Advanced Server. The EDB*Loader command-line utility loads data from a file into one (or more) tables, using a subset of the parameters offered by SQL*Loader.

EDB*Loader includes support for conventional path data loading. Conventional path data loading is slower than direct path loading, but is fully recoverable. All constraints are enforced during conventional path loading. Rules, triggers and inheritance based partitions are enforced during conventional loading.

EDB*Loader also supports direct path loading; direct path loading is faster than conventional path loading, but is non-recoverable. `UNIQUE` and `NOT NULL` constraints are enforced during direct loading. The following features are *not* supported during direct path loading:

- Foreign key constraints
- Rules, triggers and inheritance based partitions
- Loading into a non empty table

EDB*Loader can load delimiter-separated values, or fixed-width entries.

11.2.1 Invoking EDB*Loader

Use the following command to invoke EBD*Loader from the command line:

```
edbldr -d dbname -p port userid={user[/passwd]|/} control=control_file_path
log=log_file_path bad=bad_file_path parfile=param_file_path skip=skip_count
skip_index_maintenance={true|false} direct={true|false} errors=error_count
```

Parameters:

`-d` *`dbname`*

Specifies the database name.

`-p` *`port`*

Specifies the port number.

`userid={`*`username`*`[/`*`password`*`]|/}`

Specifies the username and password to use when connecting to the database.

If omitted, the EDB*Loader utility will prompt for the username and or password. If "/" is included, EDB*Loader will attempt to connect to the database using operating system authentication.

`control=`*`control_ file_name`*

Specifies the name of the control file. The default extension is `.ctl`.

`log=`*`log_file_name`*

Specifies the name of the log file. The log file is written to the same directory location as the control file. The default extension is `.log`.

`direct={TRUE|FALSE}`

Indicates the technique to use for the load. A value of `FALSE` results in a conventional based path load. A value of `TRUE` specifies direct path loading. The default value is `FALSE`.

`errors=`*`error_count`*

Specifies a limit on the number of errors that can be permitted before aborting the load operation. The default value for `error_count` is `50`.

`skip=`*`skip_count`*

Specifies the number of initial rows that should be skipped from the input data file from the load.

`skip_index_maintenance={TRUE|FALSE}`

If this parameter is `TRUE`, index maintenance will not be performed as part of the load for direct path loads. Any indexes related to the involved table will be marked as invalid. To validate the indexes, execute a `REINDEX` command for each table.

`parfile=`*`param_file_name`*

Specifies the name of the parameter file. This file does not have a default extension. All parameters that are accepted on the command line may be included in the parameter file.

`bad=`*`bad_file_name`*

Specifies the name of the bad-record file. It is written to the same directory location as the control file. The default extension is `.bad`.

discard=*discard_file_name*

Specifies the name of the discard file. The discard file is written to the same directory location as the control file. The default extension is .dsc. Only those records which cannot be loaded into *any* of the involved tables, because of failure to match the corresponding WHEN clause, will be logged into this file. The discard file contains those records that fail to satisfy the WHERE clause.

11.2.2 The EDB*Loader Control File

When you invoke EDB*Loader, the list of arguments must include the name of a control file. The control file includes the instructions that EDB*Loader uses to build the table (or tables) from the input file; it includes information such as:

- The fully qualified name of the input file
- The name of the table or tables
- The name of the columns within the table or tables
- The delimiters or other selection criteria used to choose the column content
- The fully qualified names of the bad and discard files

The following code snippets demonstrate code that might be included as part of a control file for the EDB*Loader utility.

The first example loads data from a file named /tmp/mydata.csv into a table named emp. The data within the input file is delimited by a comma; each comma tells EDB*Loader to place the next piece of data into the next column. The column names are specified within parentheses: (empno, empname, sal, deptno).

```
LOAD DATA
INFILE      '/tmp/mydata.csv'
BADFILE     '/tmp/mydata.bad'
DISCARDFILE '/tmp/mydata.dsc'
INSERT INTO TABLE emp
  FIELDS TERMINATED BY "," ENCLOSED BY '”'
  (empno, empname, sal, deptno)
```

The following code snippet demonstrates column syntax specification, using the FILLER keyword. EDB* loader ignores columns that include the FILLER keyword.

```
LOAD DATA
INSERT INTO TABLE table_1
    FIELDS TERMINATED BY ','
  (field1, field2 FILLER, field3)
```

The following example demonstrates fixed-width column specification. EDB*Loader determines the content placed within each column by its position within the input file.

```
LOAD DATA
INSERT INTO TABLE table_1
  (field1 POSITION (1:2),
  field2 FILLER POSITION (4:8),
  field3 POSITION (60:80)
)
```

This example demonstrates multiple-table specification. EDB*Loader can load multiple tables from the same input file using selective loading.

```
LOAD DATA
INFILE '/tmp/mydata.csv'
APPEND INTO TABLE emp1 WHEN (1:3) = '100'
  (empno POSITION (1:3), sal POSITION (5:7), deptno POSITION (9:11))
APPEND INTO TABLE emp2 WHEN (1:3) = '200'
  (empno POSITION (1:3), sal POSITION (5:7), deptno POSITION (9:11))
```

11.2.3 Notes

TRAILING NULLCOLS support:

If TRAILING NULLCOLS is specified, any trailing columns that cannot be populated from the input data file are assumed to be NULL.

```
    LOAD DATA
    INSERT INTO TABLE emp TRAILING NULLCOLS
    (empno, sal, deptno)
```

For example, if an input row contains the following:

```
     7824,$1200
```

The deptno column is assigned a NULL value because the input row only contains two values.

TRUNCATE/REPLACE keyword support:

If the TRUNCATE or REPLACE keywords are used, the involved table will be truncated before loading.

```
    LOAD DATA TRUNCATE INTO TABLE table_1 ..
    LOAD DATA REPLACE INTO TABLE table_1..
```

Unsupported Features:

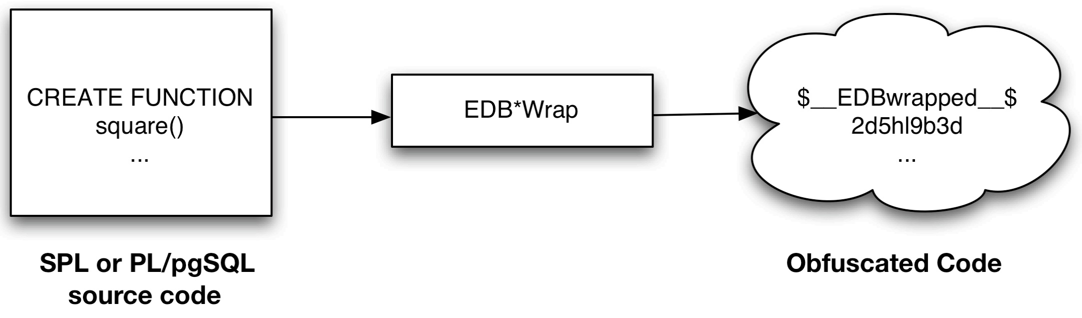
This release does not support specification of data within the control file, or columns that include default value expressions.

*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 protoypes 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 Postgres Studio or `edb-psql`:

```
[bash] edb-psql edb
Welcome to edb-psql 8.3.0.104, 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 8.3.0.104

Copyright (c) 2004-2009 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 8.3.0.104, 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 backup 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
```

11.4 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 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.

DRITA consumes minimal system resources.

11.4.1 Initialization Parameters

DRITA includes a configuration parameter, `timed_statistics`, to control the collection of timing data. This is a dynamic parameter that can be set in the `postgresql.conf` file or while a session is in progress. The valid values are `TRUE` or `FALSE`; the default value is `FALSE`.

11.4.2 Setting up and Using DRITA

To use DRITA, you must first create a small set of tables and functions. To create the tables and functions that store and report information, run the following scripts:

```
snap_tables.sql
snap_functions.sql
```

After creating the required tables and functions, take a beginning snapshot. The beginning snapshot will be compared to a later snapshot to gauge system performance. To take a beginning snapshot:

```
SELECT * from edbsnap()
```

Then, run the workload that you would like to evaluate; when the workload has completed (or at a strategic point during the workload), take an ending snapshot:

```
SELECT * from edbsnap()
```

11.4.3 DRITA Functions

11.4.3.1 get_snaps()

The `get_snaps()` function returns a list of snapshot ID's; you can use the snapshot ID's to run one or more reporting functions. To view a list of snapshot ID's and the time they were taken, enter the following command:

```
SELECT * FROM get_snaps();

          get_snaps
------------------------------
 1 15-JUN-09 17:43:50.072733
 5 15-JUN-09 18:18:15.792194
 6 16-JUN-09 09:55:03.969197
 7 16-JUN-09 11:00:01.083305
 8 16-JUN-09 11:07:59.481583
 9 16-JUN-09 11:34:45.338325
 10 16-JUN-09 11:38:05.415392
 11 16-JUN-09 11:42:31.551796
 12 16-JUN-09 11:49:44.698102
 13 16-JUN-09 11:53:11.371272
 14 16-JUN-09 11:53:32.627307
 15 16-JUN-09 12:49:38.718433
 16 16-JUN-09 14:20:00.781601
 17 16-JUN-09 14:35:17.584266
 18 16-JUN-09 14:42:22.257647
 19 16-JUN-09 14:43:07.621677
(16 rows)
```

11.4.3.2 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:

```
SELECT * FROM sys_rpt(18, 19, 10);

                                    sys_rpt
-----------------------------------------------------------------------------
 WAIT NAME                               COUNT      WAIT TIME        % WAIT
 ---------------------------------------------------------------------------
 db file read                            31         0.187628         80.75
 query plan                              20         0.027784         11.96
 infinitecache read                      63         0.004523         1.95
 wal flush                               6          0.004067         1.75
 wal write                               1          0.004063         1.75
 wal file sync                           1          0.003664         1.58
 infinitecache write                     5          0.000548         0.24
 db file write                           5          0.000082         0.04
 wal write lock acquire                  0          0.000000         0.00
 bgwriter communication lock acquire     0          0.000000         0.00
(12 rows)
```

11.4.3.3 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 write          5     0.000082   0.04        0.04
 17373 enterprise wal write lock acquire 0     0.000000   0.00        0.00
 17373 enterprise bgwriter comm lock ac  0     0.000000   0.00        0.00
(12 rows)
```

11.4.3.4 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 write          5     0.000082  0.04        0.04
 17373 enterprise wal write lock acquire 0     0.000000  0.00        0.00
 17373 enterprise bgwriter comm lock ac  0     0.000000  0.00        0.00
(12 rows)
```

11.4.3.5 sesshist_rpt()

The `sesshist_rpt()` function returns session wait information for a specified backend. The signature is:

```
sesshist_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 example demonstrates a call to the `sesshist_rpt()` function:

```
SELECT * FROM sesshist_rpt (18, 17373);

                                sesshist_rpt
 ----------------------------------------------------------------------------
 ID    USER        SEQ   WAIT NAME
    ELAPSED(ms)   File   Name                    # of Blk    Sum of Blks
 ----------------------------------------------------------------------------
 17373 enterprise 1     infinitecache read
    84            1249   pg_attribute            44          1
 17373 enterprise 2     query plan
    12            0      N/A                     0           0
 17373 enterprise 3     infinitecache read
    110           1255   pg_proc                 64          1
 17373 enterprise 4     db file read
    3326          16421  session_waits_pk        2           1
 17373 enterprise 5     db file read
    4201          16421  session_waits_pk        3           1
 17373 enterprise 6     db file read
    5386          16421  session_waits_pk        0           1
 17373 enterprise 7     db file read
    13414         16416  edb$session_waits       3           1
 17373 enterprise 8     db file read
    4609          1260   pg_authid               0           1
 17373 enterprise 9     query plan
    12842         0      N/A                     0           0
 17373 enterprise 10    infinitecache read
    50            2619   pg_statistic            10          1
 17373 enterprise 11    infinitecache read
    51            2696   pg_statistic_relid_a 1              1
 17373 enterprise 12    infinitecache read
```

```
    51            1249   pg_attribute         8           1
 17373 enterprise 13     infinitecache read
    65            2654   pg_amop_opr_opc_inde 1           1
 17373 enterprise 14     infinitecache read
    77            2654   pg_amop_opr_opc_inde 3           1
 17373 enterprise 15     infinitecache read
    81            2696   pg_statistic_relid_a 4           1
 17373 enterprise 16     db file read
    11915         2696   pg_statistic_relid_a 3           1
 17373 enterprise 17     infinitecache read
    32            2696   pg_statistic_relid_a 3           1
 17373 enterprise 18     query plan
    12            0      N/A                  0           0
 17373 enterprise 19     infinitecache read
    50            1249   pg_attribute         12          1
 17373 enterprise 20     infinitecache read
    52            2659   pg_attribute_relid_a 2           1
 17373 enterprise 21     infinitecache read
    52            1255   pg_proc              2           1
 17373 enterprise 22     infinitecache read
    58            2617   pg_operator          3           1
 17373 enterprise 23     infinitecache read
    52            2690   pg_proc_oid_index    5           1
 17373 enterprise 24     infinitecache read
    58            1255   pg_proc              28          1
 17373 enterprise 25     infinitecache read
    50            2618   pg_rewrite           4           1
(27 rows)
```

11.4.3.6 truncsnap()

Use the `truncsnap()` function to purge all records from the snapshot tables:

```
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 purged from the snapshot tables:

```
SELECT * FROM get_snaps
 get_snaps
-----------
(0 rows)
```

11.4.3.7 purgesnap()

The `purgesnap()` function purges a range of snapshots within the snap tables. Pass the snapshot ID's for the start of the range and the end of the range to purge:

```
SELECT * FROM purgesnap(6, 9);

             purgesnap
------------------------------------
```

```
 Snapshots in range 6 to 9 deleted.
(1 row)
```

A call to the `get_snaps()` function after calling the `purgesnap()` function shows that columns `6` through `9` have been purged from the snapshot tables:

```
 SELECT * FROM get_snaps
         get_snaps
------------------------------
 1 15-JUN-09 17:43:50.072733
 5 15-JUN-09 18:18:15.792194
 10 16-JUN-09 11:38:05.415392
 11 16-JUN-09 11:42:31.551796
 12 16-JUN-09 11:49:44.698102
 13 16-JUN-09 11:53:11.371272
 14 16-JUN-09 11:53:32.627307
 15 16-JUN-09 12:49:38.718433
 16 16-JUN-09 14:20:00.781601
 17 16-JUN-09 14:35:17.584266
 18 16-JUN-09 14:42:22.257647
 19 16-JUN-09 14:43:07.621677
(12 rows)
```

11.5 Simulating Statspack AWR Reports

The snapshot tables and 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.

Catalog Table	New DRITA Table	Reporting Function
`pg_stat_database`	`edb$stat_database`	`stat_db_rpt()`
`pg_stat_all_tables`	`edb$stat_all_tables`	`stat_tables_rpt()`
`pg_stat_io_tables`	`edb$statio_all_tables`	`statio_tables_rpt()`
`Pgstat_all_indexes`	`edb$stat_all_indexes`	`stat_indexes_rpt()`
`pg_statio_all_indexes`	`edb$statio_all_indexes`	`statio_indexes_rpt()`

The reporting functions can be executed individually or you can execute all five functions by calling the `edbreport()` function.

11.5.1.1 edbreport()

The `edbreport()` function includes data from the other reporting functions, plus additional system information. The signature is:

```
edb_report(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 code sample demonstrates a call to the `edbreport()` function:

```
SELECT * FROM edbreport(18, 19);

                                edbreport
---------------------------------------------------------------------------
EnterpriseDB Report for database edb        16-JUN-09
Version: EnterpriseDB 8.3.0.106 on i686-pc-linux-gnu, compiled by GCC gcc
(GCC) 4.1.0

      Begin snapshot: 18 at 16-JUN-09 14:42:22.257647
      End snapshot:   19 at 16-JUN-09 14:43:07.621677
```

```
Size of database edb is 2124 MB
     Tablespace: pg_default Size: 2136 MB Owner: enterprisedb
     Tablespace: pg_global Size: 283 kB Owner: enterprisedb

Schema: public                 Size: 2114 MB         Owner: enterprisedb


               Top 10 Relations by pages

TABLE                                            RELPAGES
---------------------------------------------------------------------------
accounts                                          222231
history                                           513
pg_proc                                           92
edb$statio_all_indexes                            86
edb$stat_all_indexes                              86
pg_depend                                         56
tellers                                           53
edb$stat_all_tables                               51
edb$statio_all_tables                             49
pg_attribute                                      43



               Top 10 Indexes by pages

INDEX                                            RELPAGES
---------------------------------------------------------------------------
accounts_pkey                                     46127
pg_proc_proname_args_nsp_index                    81
pg_depend_reference_index                         48
pg_depend_depender_index                          46
edb$stat_idx_pk                                   40
edb$statio_idx_pk                                 40
pg_attribute_relid_attnam_index                   33
pg_operator_oprname_l_r_n_index                   20
edb$statio_tab_pk                                 19
edb$stat_tab_pk                                   19


               Top 10 Relations by DML

SCHEMA          RELATION                       UPDATES   DELETES   INSERTS
---------------------------------------------------------------------------
public          accounts                       7399697   0         7000000
public          tellers                        199699    0         700
public          branches                       199699    0         70
public          history                        0         150000    199699
sys             edb$stat_all_indexes           0         336       2128
sys             edb$statio_all_indexes         0         336       2128
sys             edb$stat_all_tables            0         264       1672
sys             edb$statio_all_tables          0         264       1672
sys             edb$session_wait_history       0         75        525
sys             edb$session_waits              0         9         125


  DATA from pg_stat_database

DATABASE NUMBACKENDS XACT COMMIT XACT ROLLBACK  BLKS READ BLKS HIT HIT RATIO
---------------------------------------------------------------------------
edb      0           5           0              59         2538      97.73
```

```
  DATA from pg_buffercache

RELATION                            BUFFERS
---------------------------------------------------------------------------
accounts                            21884
pg_proc                             34
pg_proc_proname_args_nsp_index      27
edb$statio_all_indexes              24
edb$stat_all_indexes                24
pg_attribute                        23
pg_operator                         19
edb$statio_all_tables               17
edb$stat_all_tables                 17
edb$stat_idx_pk                     14


   DATA from pg_stat_all_tables ordered by seq scan

SCHEMA     RELATION      SEQ       REL     READ    IDX
                         SCAN      TUP     SCAN    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        wait_history   0         0       0       0         25     0   0
sys        session_waits  0         0       0       0         0      10  0





    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


  DATA from pg_statio_all_tables

SCHEMA       RELATION
                          HEAP  HEAP  IDX  IDX  TOAST  TOAST  TIDX  TIDX
                          READ  HIT   READ HIT  READ   HIT    READ  HIT
----------------------------------------------------------------------------
pg_catalog   pg_class
                          0     137   3    104  0      0      0     0
pg_catalog   pg_attribute
                          1     121   1    264  0      0      0     0
sys          edb$stat_all_indexes
                          5     111   5    225  0      0      0     0
sys          edb$statio_all_indexes
                          5     111   5    225  0      0      0     0
sys          edb$stat_all_tables
                          4     87    4    175  0      0      0     0
sys          edb$statio_all_tables
                          4     87    4    175  0      0      0     0
pg_catalog   pg_opclass
                          0     38    1    5    0      0      0     0
pg_catalog   pg_proc
                          0     37    0    92   0      0      0     0
pg_catalog   pg_index
                          1     30    1    22   0      0      0     0
sys          edb$session_wait_history
                          1     24    0    48   0      0      0     0


  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
       140        21             21
pg_catalog   pg_attribute     pg_attribute_relid_attnum_index
       134        303            303
pg_catalog   pg_class         pg_class_oid_index
       48         48             48
pg_catalog   pg_proc          pg_proc_oid_index
       44         44             44
pg_catalog   pg_class         pg_class_relname_nsp_index
       30         17             17
pg_catalog   pg_statistic     pg_statistic_relid_att_index
       21         10             10
pg_catalog   pg_rewrite       pg_rewrite_rel_rulename_index
       15         15             15
pg_catalog   pg_index         pg_index_indrelid_index
       13         18             18
sys          edb$system_waits system_waits_pk
       12         38             6
pg_catalog   pg_index         pg_index_indexrelid_index
       10         10             10


  DATA from pg_statio_all_indexes

SCHEMA      RELATION                  INDEX
   IDX BLKS READ   IDX BLKS HIT
----------------------------------------------------------------------------
```

```
 pg_catalog  pg_attribute               pg_attribute_relid_attnum_index
    1              264
 sys         edb$stat_all_indexes       edb$stat_idx_pk
    5              225
 sys         edb$statio_all_indexes     edb$statio_idx_pk
    5              225
 sys         edb$stat_all_tables        edb$stat_tab_pk
    4              175
 sys         edb$statio_all_tables      edb$statio_tab_pk
    4              175
 pg_catalog  pg_cast                    pg_cast_source_target_index
    0              139
 pg_catalog  pg_proc                    pg_proc_oid_index
    0              82
 pg_catalog  pg_class                   pg_class_relname_nsp_index
    3              56
 pg_catalog  pg_class                   pg_class_oid_index
    0              48
 sys         edb$session_wait_history   session_waits_hist_pk
    0              48


    System Wait Information

 WAIT NAME                                COUNT      WAIT TIME       % WAIT
 ---------------------------------------------------------------------------
 db file read                             31         0.187628        80.75
 query plan                               20         0.027784        11.96
 infinitecache read                       63         0.004523        1.95
 wal flush                                6          0.004067        1.75
 wal write                                1          0.004063        1.75
 wal file sync                            1          0.003664        1.58
 infinitecache write                      5          0.000548        0.24
 db file write                            5          0.000082        0.04
 wal write lock acquire                   0          0.000000        0.00
 bgwriter communication lock acquire      0          0.000000        0.00


    Database Parameters from postgresql.conf

 PARAMETER                       SETTING      CONTEXT     MINVAL   MAXVAL
 ----------------------------------------------------------------------------
 add_missing_from                off          user
 allow_system_table_mods         off          postmaster
 archive_command                              sighup
 archive_timeout                 0            sighup      0        2147483647
 array_nulls                     on           user
 authentication_timeout          10           sighup      1        600
 autovacuum                      on           sighup
 autovacuum_analyze_scale_factor 0.1          sighup      0        100
 autovacuum_analyze_threshold    250          sighup      0        2147483647
 autovacuum_freeze_max_age       200000000    postmaster  10000000 2000000000
 autovacuum_naptime              60           sighup      1        2147483647
 autovacuum_vacuum_cost_delay    -1           sighup      -1       1000
 autovacuum_vacuum_cost_limit    -1           sighup      -1       10000
 autovacuum_vacuum_scale_factor  0.2          sighup      0        100
 autovacuum_vacuum_threshold     1000         sighup      0        2147483647

  ...

(384 rows)
```

11.5.1.2 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(18, 19);

                                stat_db_rpt
-------------------------------------------------------------------------------
   DATA from pg_stat_database

 DATABASE NUMBACKENDS XACT COMMIT XACT ROLLBACK  BLKS READ BLKS HIT HIT RATIO
 ------------------------------------------------------------------------------
 edb      0            5           0               59        2538     97.73
(5 rows)
```

11.5.1.3 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 the scope of the statistics returned (`ALL`, `USER` or `SYS`).

The following code sample demonstrates the `stat_tables_rpt()` function:

```
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

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)
```

11.5.1.4 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 the scope of the statistics returned (`ALL`, `USER` or `SYS`).

The following example demonstrates the `statio_tables_rpt()` function:

```
SELECT * FROM statio_tables_rpt(18, 19, 10, 'ALL');

                             statio_tables_rpt
-----------------------------------------------------------------------------
   DATA from pg_statio_all_tables

SCHEMA           RELATION
    HEAP     HEAP     IDX      IDX      TOAST      TOAST      TIDX       TIDX
    READ     HIT      READ     HIT      READ       HIT        READ       HIT
 ----------------------------------------------------------------------------
 pg_catalog       pg_class
    0        137      3        104      0          0          0          0
 pg_catalog       pg_attribute
    1        121      1        264      0          0          0          0
 sys              edb$stat_all_indexes
    5        111      5        225      0          0          0          0
 sys              edb$statio_all_indexes
    5        111      5        225      0          0          0          0
 sys              edb$stat_all_tables
    4        87       4        175      0          0          0          0
 sys             edb$statio_all_tables
    4        87       4        175       0          0          0          0
 pg_catalog      pg_opclass
    0        38       1        5         0          0          0          0
 pg_catalog      pg_proc
    0        37       0        92        0          0          0          0
 pg_catalog      pg_index
    1        30       1        22        0          0          0          0
 sys             edb$session_wait_history
    1        24       0        48        0          0          0          0
(15 rows)
```

11.5.1.5 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 the scope of the statistics returned (`ALL`, `USER` or `SYS`).

The following code sample demonstrates the `stat_indexes_rpt()` function:

```
SELECT * FROM stat_indexes_rpt(18, 19, 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
    140        21           21
 pg_catalog           pg_attribute             pg_attribute_relid_attnum_index
    134        303          303
 pg_catalog          pg_class               pg_class_oid_index
    48          48          48
 pg_catalog          pg_proc                pg_proc_oid_index
    44          44          44
 pg_catalog          pg_class               pg_class_relname_nsp_index
    30          17          17
 pg_catalog          pg_statistic           pg_statistic_relid_att_index
    21          10          10
 pg_catalog          pg_rewrite             pg_rewrite_rel_rulename_index
    15          15          15
 pg_catalog          pg_index               pg_index_indrelid_index
    13          18          18
 sys                 edb$system_waits       system_waits_pk
    12          38          6
 pg_catalog          pg_index               pg_index_indexrelid_index
    10          10          10
(14 rows)
```

11.5.1.6 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 the scope of the statistics returned (`ALL`, `USER` or `SYS`).

The following example demonstrates the `statio_indexes_rpt()` function:

```
SELECT * FROM statio_indexes_rpt(18, 19, 10, 'ALL');

                              statio_indexes_rpt
 ----------------------------------------------------------------------------
```

```
   DATA from pg_statio_all_indexes

 SCHEMA              RELATION                  INDEX
              IDX BLKS READ   IDX BLKS HIT
 ----------------------------------------------------------------------------
 pg_catalog          pg_attribute              pg_attribute_relid_attnum_index
              1               264
 sys                 edb$stat_all_indexes      edb$stat_idx_pk
              5               225
 sys                 edb$statio_all_indexes    edb$statio_idx_pk
              5               225
 sys                 edb$stat_all_tables       edb$stat_tab_pk
              4               175
 sys                 edb$statio_all_tables     edb$statio_tab_pk
              4               175
 pg_catalog          pg_cast                   pg_cast_source_target_index
              0               139
 pg_catalog          pg_proc                   pg_proc_oid_index
              0               82
 pg_catalog          pg_class                  pg_class_relname_nsp_index
              3               56
 pg_catalog          pg_class                  pg_class_oid_index
              0               48
 sys                 edb$session_wait_history  session_waits_hist_pk
              0               48
(14 rows)
```

11.6 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.

11.7 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.

11.8 Catalog Views

The DRITA catalog views provide access to performance information relating to system waits.

11.8.1 edb$system_waits

The `edb$system_waits` view summarizes the number of waits and the total wait time per session for each wait named. It also displays the average and max wait times. `edb$system_waits` summarizes the following information:

```
   Column    |      Type       | Modifiers |    Definition
------------+----------------+-----------+------------------
 edb_id      | numeric         |           |identifier
 dbname      | text            |           |database name
 wait_name   | text            |           |name of the event
 wait_count  | numeric         |           |number of times the event occurs
 avg_wait    | numeric(50,6)   |           |average wait time in microseconds
 max_wait    | numeric         |           |maximum wait time in microseconds
 total_wait  | numeric         |           |total wait time in microseconds
 wait_name   | text            |           |name of the event
```

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
```

11.8.2 edb$session_waits

The `edb$session_waits` view summarizes the number of waits and the total wait time per session for each wait named and identified by backend ID. It also displays the average and max wait times. `edb$session_waits` summarizes the following information:

```
    Column        |      Type       | Modifiers |Definition
-----------------+----------------+-----------+----------------
 backend_id       | bigint          |           |session identifier
 wait_count       | bigint          |           |number of times the event
                                                 occurs
 avg_wait_time    | numeric         |           |average wait time in
                                                 microseconds
```

```
  max_wait_time   | numeric(50,6) |            |maximum wait time in
                                                microseconds
  total_wait_time | numeric(50,6) |            |total wait time in
                                                microseconds
  wait_name       | text          |            |name of the event
```

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)
```

11.8.3 edb$session_wait_history

The `edb$session_wait_history` view contains the last 25 wait events for each backend ID active during the session. The `edb$session_wait_history` view includes the following information:

```
   Column    |  Type   | Modifiers |   Definition
 ------------+---------+-----------+--------------------------
  edb_id     | numeric|           |identifier
  dbname     | text    |           |database name
  backend_id | bigint |           |session identifier
  seq        | bigint |           |number between 1 and 25
  wait_name  | text    |           |name of the event
  elapsed    | bigint |           |elapsed time in microseconds
  p1         | bigint |           |variable #1- meaning dependent on
                                               event
  p2         | bigint |           |variable #2- meaning dependent on
                                               event
  p3         | bigint |           |variable #3- meaning dependent on
                                               event
```

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)
```

12 Acknowledgements

The PostgreSQL 8.3 Documentation provided the baseline for portions of this Oracle Compatibility Developer's Guide that is 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