



EPAS Fundamentals

Release 18.1

EnterpriseDB Corporation

Nov 26, 2025

Contents

1	EDB Postgres Advanced Server fundamentals	3
1.1	Benefits of EDB Postgres Advanced Server	3
1.2	Database compatibility for Oracle developers	4
1.3	Terminology	4
1.3.1	Database server (postmaster)	4
1.3.2	Database cluster	4
1.3.3	Configuration files	4
2	SQL fundamentals	5
2.1	About the examples	5
2.2	Conventions used	6
2.3	SQL tutorial	6
2.3.1	Sample database	7
2.3.2	Creating a new table	18
2.3.3	Populating a table With rows	19
2.3.4	Querying a table	19
2.3.5	Executing joins between tables	21
2.3.6	Aggregating data	24
2.3.7	Updating a table	26
2.3.8	Deleting a table	26
2.3.9	The SQL language	27
2.4	Advanced concepts	28
2.4.1	Views	28
2.4.2	Foreign keys	29
2.4.3	The ROWNUM pseudo-column	29
2.4.4	Synonyms	31
2.4.5	Hierarchical queries	34
2.4.6	Hierarchical queries	35
2.4.7	Multidimensional analysis	44
2.4.8	Multidimensional analysis	44
2.5	Sample database description	61
2.5.1	Available scripts	62
2.5.2	About the sample database	62
3	Supported platforms	73
4	EDB Postgres Advanced Server release notes	75

4.1	Component certification	75
4.2	Support announcements	75
4.2.1	Backup and Recovery Tool (BART) incompatibility	75

The key concepts of EDB Postgres Advanced Server include its features, benefits, and terminology. The SQL tutorial uses a sample database and explains important database concepts.

EDB Postgres Advanced Server fundamentals

EDB Postgres Advanced Server provides tools you can use to administer enterprise-scale data solutions. These essential concepts help you to design, implement, and manage EDB Postgres Advanced Server.

1.1 Benefits of EDB Postgres Advanced Server

EDB Postgres Advanced Server is a stand-alone proprietary database server that adds extended functionality to the open-source PostgreSQL database.

EDB Postgres Advanced Server is designed to provide enterprise capabilities on top of the features present in community Postgres. These capabilities:

- Help database administrators to maintain and operate EDB Postgres Advanced Server databases
- Include enhanced SQL functionality and other features that add flexibility and convenience
- Extend Postgres security with features designed to limit unauthorized access
- Assist with performance monitoring and analysis
- Include multiple features designed to increase application programmer productivity
- [Advanced replication support \(version 14 and higher\)](#) .

EDB Postgres Advanced Server also has a significant amount of Oracle compatibility features which facilitate Oracle to Postgres migrations.

EDB Postgres Advanced Server includes extended functionality that provides compatibility for syntax supported by Oracle applications, as well as compatible procedural logic, data types, system catalog views and other features that enable EDB's Oracle compatible connectors, EDB*Plus, EDB*Loader and many other tools and functionality.

Major versions of EDB Postgres Advanced Server are released yearly in line with the major version release of PostgreSQL. EDB Postgres Advanced Server is normally released one to two months after PostgreSQL major version releases.

EDB Postgres Advanced Server is supported on Linux x86-64 , Linux on IBM Power and Windows x86-64 .

1.2 Database compatibility for Oracle developers

EDB Postgres Advanced Server makes Postgres look, feel, and operate more like Oracle, so when you migrate, there is less code to rewrite, and you can be up and running quickly. The Oracle compatibility features allow you to run many applications written for Oracle in EDB Postgres Advanced Server with minimal to no changes.

EDB Postgres Advanced Server provides Oracle compatible:

- Stored procedure language (SPL) when creating database server-side application logic for stored procedures, functions, triggers, and packages
- Data types that are compatible with Oracle databases
- SQL statements that are compatible with Oracle SQL
- System and built-in functions for use in SQL statements and procedural logic
- System catalog views that are compatible with Oracle's data dictionary
- Additional compatibility with Oracle MERGE .

!!!SeeAlso "Further reading" - Compatible SQL syntax, data types, and views in SQL reference . - Compatibility offered by the procedures and functions that are part of the built-in packages in Built-in packages . - Compatible tools and utilities (EDB*Plus, EDB*Loader, DRITA, and EDB*Wrap) that are included with an EDB Postgres Advanced Server installation in Tools, utilities, and components . - For applications written using the Oracle Call Interface (OCI), EDB's Open Client Library (OCL) provides interoperability with these applications. See [EDB OCL Connector](#) .

1.3 Terminology

The terminology that follows is important for understanding EDB Postgres Advanced Server's functionality and its components.

1.3.1 Database server (postmaster)

The server process that provides the key functionality that allows you to store and manage data. It manages the database files, accepts connections to the database from client applications, and performs database actions on behalf of the clients. The database server process is called *postgres*.

1.3.2 Database cluster

A set of on-disk structures that comprise a collection of databases. A cluster is serviced by a single-instance of the database server. A database cluster is stored in the `data` directory. Don't store the `data` directory of a production database on an NFS file system.

1.3.3 Configuration files

You can use the parameters listed in Postgres configuration files to manage deployment preferences, security preferences, connection behaviors, and logging preferences.

Structured Query Language (SQL) is a standard language for accessing and manipulating databases. These fundamentals provide a starting point for the most common actions for working with SQL, with pointers to more in-depth documentation. The tutorial uses a sample database and explains key concepts.

2.1 About the examples

The examples shown in the documentation use the PSQL program. PSQL is a terminal-based command line interface for Postgres. The prompt that normally appears when using PSQL is omitted in these examples to provide extra clarity for the point being shown.

Examples and output from examples are shown in a fixed-width, white font on a dark background.

Consider the following key points related to the examples:

- During installation of the EDB Postgres Advanced Server, you must make selections for configuration and defaults compatible with Oracle databases to reproduce the same results as the examples. You can verify a default compatible configuration by issuing the following commands in PSQL and returning the results shown:

```
SHOW edb_redwood_date;

edb_redwood_date
-----
on

SHOW datestyle;

DateStyle
-----
Redwood, DMY

SHOW edb_redwood_strings;
```

(continues on next page)

(continued from previous page)

```
edb_redwood_strings
```

```
on
```

- The examples use the sample tables, `dept`, `emp`, and `jobhist`, created and loaded when EDB Postgres Advanced Server was installed. The `emp` table is installed with triggers that you must disable to reproduce the same results as the examples. Log in to EDB Postgres Advanced Server as the `enterprisedb` superuser, and disable the triggers by issuing the following command:

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

You can later reactivate the triggers on the `emp` table with the following command:

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

2.2 Conventions used

The conventions used in the documentation described here apply to both Linux and Windows systems.

Directory paths are presented in the Linux format with forward slashes. When working on Windows systems, start the directory path with the drive letter followed by a colon, and substitute back slashes for forward slashes.

Some of this information might apply interchangeably to the PostgreSQL and EDB Postgres Advanced Server database systems. The term *EDB Postgres Advanced Server* is used to refer to EDB Postgres Advanced Server. The term *Postgres* is used to generically refer to both PostgreSQL and EDB Postgres Advanced Server. When a distinction must be made between these two database systems, the specific names-PostgreSQL or EDB Postgres Advanced Server-are used.

The installation directory path of the PostgreSQL or EDB Postgres Advanced Server products is referred to as *POSTGRES_INSTALL_HOME*.

- For EDB Postgres Advanced Server Linux installations performed using the interactive installer for version 10 and earlier, the default is `/opt/edb/asx.x`.
- For EDB Postgres Advanced Server Linux installations performed using an RPM package, the default is `/usr/edb/as<xx>` where `<xx>` is the EDB Postgres Advanced Server version number.
- For EDB Postgres Advanced Server Windows installations, the default is `C:\Program Files\edb\as<xx>`. The product version number is represented by `x.x` or by `xx` for version 10 and later.

2.3 SQL tutorial

EDB Postgres Advanced Server is a *relational database management system* (RDBMS), which means it's a system for managing data stored in *relations*. A relation is essentially a mathematical term for a *table*. 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's important to remember that SQL doesn't guarantee the order of the rows in the table in any way (although you can explicitly sort them for display).

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

This tutorial walks you through the main features of an RDBMS using a sample database.

2.3.1 Sample database

Throughout this tutorial, we work with a sample database to help explain database concepts, from basic to advanced.

Accessing the sample database

When EDB Postgres Advanced Server is installed, a sample database named `edb` is automatically created. This sample database contains the tables and programs used in this tutorial after you execute the script `edb-sample.sql`, located in the `/usr/edb/as17/share` 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 are 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;
```

You can alter the search path using commands in PSQL, a terminal-based front end to Postgres.

Overview of the sample database

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, and so on. 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.

The `edb-sample.sql` script is:

```
--
-- 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 (
```

(continues on next page)

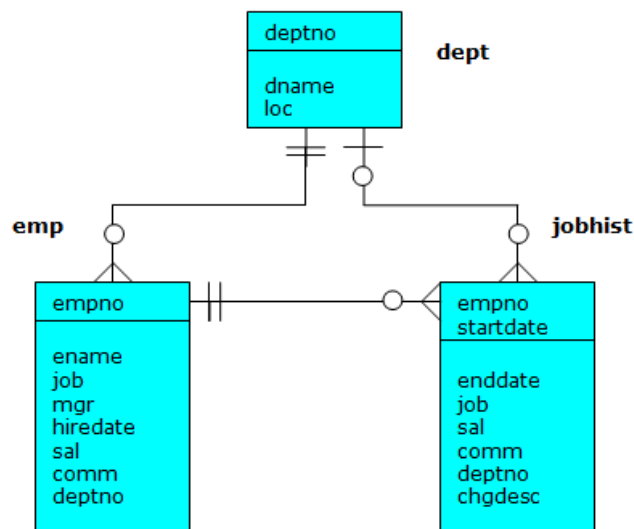


Fig. 1: Sample Database Tables

(continued from previous page)

```

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),

```

(continues on next page)

(continued from previous page)

```

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

```

(continues on next page)

(continued from previous page)

```

--
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 (

```

(continues on next page)

(continued from previous page)

```

    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

```

(continues on next page)

(continued from previous page)

```

        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.
--

```

(continues on next page)

(continued from previous page)

```

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;

```

(continues on next page)

(continued from previous page)

```

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

```

(continues on next page)

(continued from previous page)

```

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

```

(continues on next page)

(continued from previous page)

```

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.

```

(continues on next page)

(continued from previous page)

```

--
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 employees 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,

```

(continues on next page)

(continued from previous page)

```

    p_mgr          NUMBER,
    p_deptno       NUMBER
)
AS
BEGIN
    INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
        VALUES (p_empno, p_ename, p_job, p_sal,
                p_hiredate, p_comm, p_mgr, p_deptno);
END;
--
-- Procedure that deletes an employee record from the emp table based
-- on the employee number.
--
PROCEDURE fire_emp (
    p_empno        NUMBER
)
AS
BEGIN
    DELETE FROM emp WHERE empno = p_empno;
END;
END;
/
COMMIT;

```

2.3.2 Creating a new table

Create a new table 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 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 code into PSQL with line breaks. PSQL recognizes that the command isn't terminated until the semicolon.

You can use white space (spaces, tabs, and newlines) in SQL commands. You can align the command as you want or even put all the text all on one line.

Two dashes (“--”) introduce comments. Whatever follows them is ignored up to the end of the line. In addition, SQL is not case sensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case.

`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.

EDB Postgres 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 or want to recreate it, you can remove it using the following command:

```
DROP TABLE tablename;
```

2.3.3 Populating a table With rows

Use the `INSERT` statement to populate a table with rows:

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

All data types use obvious input formats. Constants that aren't simple numeric values usually are surrounded by single quotes ('), as in the example. The `DATE` type is flexible in what it accepts. However, for this tutorial we use 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 want or even omit some columns, for example, 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.

2.3.4 Querying a table

To retrieve data from a table, you execute a *query* using an SQL `SELECT` statement. The statement is divided into:

- A select list (the part that lists the columns to return)
- A table list (the part that lists the tables from which to retrieve the data)
- 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.

__OUTPUT__							
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

(continues on next page)

(continued from previous page)

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

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

__OUTPUT__

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.

You can qualify a query by adding a WHERE clause that specifies the rows you want. The WHERE clause contains a Boolean (truth value) expression. 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;
```

__OUTPUT__

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

You can request to return the results of a query in sorted order:

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

__OUTPUT__

ename	sal	deptno
-----+-----+-----		
ADAMS	1100.00	20

(continues on next page)

(continued from previous page)

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 to remove duplicate rows from the result of a query:

```
SELECT DISTINCT job FROM emp;
```

```
__OUTPUT__
```

```
job
```

```
-----
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

(5 rows)

2.3.5 Executing joins between tables

Queries can access multiple tables at once or access the same table in such a way that multiple rows of the table are 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.

Executing a join query

For example, suppose you want to list all the employee records together with the name and location of the associated department. To do that, you need to compare the `deptno` column of each row of the `emp` table with the `deptno` column of all rows in the `dept` table. Then select the pairs of rows where these values match. You can accomplish this using the following query:

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

```
__OUTPUT__
```

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

(continues on next page)

(continued from previous page)

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's no result row for department 40. That's because there's no matching entry in the `emp` table for department 40, so the join ignores the unmatched rows in the `dept` table. The code that follows shows how to fix this.
- It's 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 found the table they belong to. However, it's best practice to fully qualify column names in join queries.

You can also write this kind of join queries 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, but we show it here to help you understand the following topics.

Executing an outer join

In all the above results for joins, no employees were returned that belonged to department 40. As a consequence, the record for department 40 never appears. Next, resolve how to get the department 40 record in the results despite the fact that there are no matching employees. The query must scan the `dept` table and, for each row, find the matching `emp` row. If no matching row is found, we want to substitute some "empty" values for the `emp` table's columns. This kind of query is called an *outer join*. (The joins you 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;
```

__OUTPUT__

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

(continues on next page)

(continued from previous page)

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*. The table mentioned on the left of the join operator has each of its rows in the output at least once. The table on the right has only 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 in the *WHERE* clause. The outer join operator is placed after the column name of the table for which you substitute null values for unmatched rows. So for all the rows in the *dept* table that have no matching rows in the *emp* table, EDB Postgres Advanced Server returns null for any select list expressions containing columns of *emp*. Hence you can rewrite the earlier example as:

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

__OUTPUT__

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)

Executing a self join

You can also join a table against itself, which is called a *self join*. As an example, suppose you want to find the name of each employee and the name of that employee's manager. You 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;
```

__OUTPUT__

Employees and their Managers

```
-----+-----
FORD works for JONES
SCOTT works for JONES
WARD works for BLAKE
```

(continues on next page)

(continued from previous page)

```

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 was relabeled as e1 to represent the employee row in the select list and in the join condition. It was also relabeled as e2 to represent the matching employee row acting as manager in the select list and in the join condition. You can use these kinds of aliases 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;
__OUTPUT__

```

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 is used often.

2.3.6 Aggregating data

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

As an example, you can find the highest and lowest salaries with the following query:

```

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

```

highest_salary	lowest_salary
5000.00	800.00

```

(1 row)

```

If you want to find the employee with the largest salary, you might be tempted to try:

```
SELECT ename FROM emp WHERE sal = MAX(sal);
__OUTPUT__
ERROR: aggregates not allowed in WHERE clause
```

This approach doesn't work because you can't use the aggregate function `MAX` in the `WHERE` clause. This restriction exists because the `WHERE` clause determines the rows that go into the aggregation stage. Hence, it has to be evaluated before aggregate functions are computed. However, you can restart the query to accomplish the intended result by using a *subquery*. The subquery is an independent computation that obtains its own result separately from the outer query.

```
SELECT ename FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);
__OUTPUT__
ename
-----
KING
(1 row)
```

Aggregates are also 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;
__OUTPUT__
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;
__OUTPUT__
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;
__OUTPUT__
deptno |    max
-----+-----
      20 | 3000.00
(1 row)
```

There's 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 are grouped.

and aggregate functions are 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. Only those groups where the average salary of analysts in the group is greater than 2000 are in the final result. This is true only of the group for department 20, and the maximum analyst salary in department 20 is 3000.00.

2.3.7 Updating a table

You can change the column values of existing rows 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;
```

__OUTPUT__

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;
```

__OUTPUT__

ename	sal
JONES	3272.50
BLAKE	3135.00
CLARK	2695.00

(3 rows)

2.3.8 Deleting a table

You can remove rows 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;
```

__OUTPUT__

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

(continues on next page)

(continued from previous page)

```
FORD      |      20
MILLER    |      10
(14 rows)
```

```
DELETE FROM emp WHERE deptno = 20;
```

```
SELECT ename, deptno FROM emp;
```

```
__OUTPUT__
ename    | deptno
-----+-----
ALLEN    |     30
WARD     |     30
MARTIN   |     30
BLAKE    |     30
CLARK    |     10
KING     |     10
TURNER   |     30
JAMES    |     30
MILLER   |     10
(9 rows)
```

Warning

Be careful when giving a *DELETE* command without a *WHERE* clause such as the following:

```
DELETE FROM tablename;
```

This statement removes all rows from the given table, leaving it completely empty. The system doesn't request confirmation before doing this. !!!

2.3.9 The SQL language

EDB Postgres Advanced Server supports SQL language that's compatible with Oracle syntax as well as syntax and commands for extended functionality. Extended functionality doesn't provide database compatibility for Oracle or support Oracle-styled applications.

SQL reference provides detailed information about:

- `:ref:`Compatible SQL syntax and language elements <../../../../reference/sql_reference/01_sql_syntax/>``
- `:ref:`Data types <../../../../reference/sql_reference/02_data_types/>``
- `:ref:`Functions and operators for the built-in data types <../../../../reference/sql_reference/03_functions_and_operators/>``

2.4 Advanced concepts

Advanced SQL features can simplify management and prevent loss or corruption of your data.

2.4.1 Views

A *view* provides a consistent interface that encapsulates details of the structure of your tables that might change as your application evolves. Making liberal use of views is a key aspect of good SQL database design.

Consider the following `SELECT` command:

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

__OUTPUT__

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 you use repeatedly, a shorthand method of reusing this query without retyping the entire `SELECT` command each time is to create a *view*:

```
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;
```

__OUTPUT__

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

(continues on next page)

(continued from previous page)

JAMES	950.00	22800.00	30
FORD	3000.00	72000.00	20
MILLER	1300.00	31200.00	10

(14 rows)

You can use views in almost any place that you use a real table. Building views on other views is also common.

2.4.2 Foreign keys

A *foreign key* represents one or more than one column used to establish and enforce a link between data in two database tables for controlling data stored in the foreign key table.

Suppose you want to make sure all employees belong to a valid department, that is, you want to maintain the *referential integrity* of your data. In simplistic database systems, you can ensure referential integrity 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 inconvenient. EDB Postgres Advanced Server can make it easier for you.

A modified version of the emp table presented in *Creating a new table* is shown here 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 you try to issue the following INSERT command in the sample emp table, the foreign key constraint emp_ref_dept_fk is meant to ensure that department 50 exists in the dept table. Since it doesn't, 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".
```

You can finely tune the behavior of foreign keys to your application. Making correct use of foreign keys improves the quality of your database applications. We strongly encourage you to learn more about them.

2.4.3 The ROWNUM pseudo-column

ROWNUM is a pseudo-column that's 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 has ROWNUM of 1, the second row has ROWNUM of 2, and so on.

Limiting the number or rows returned

You can use this feature to limit the number of rows retrieved by a query:

```
SELECT empno, ename, job FROM emp WHERE ROWNUM < 5;
__OUTPUT__
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. However, the ROWNUM values might not be in ascending order:

```
SELECT ROWNUM, empno, ename, job FROM emp WHERE ROWNUM < 5 ORDER BY ename;
__OUTPUT__
rownum | empno | ename | job
-----+-----+-----+-----
2      | 7499  | ALLEN | SALESMAN
4      | 7566  | JONES | MANAGER
1      | 7369  | SMITH | CLERK
3      | 7521  | WARD  | SALESMAN
(4 rows)
```

Adding a sequence number to rows in a table

The following example shows how you can add a sequence number to every row in the jobhist table. First add a column named seqno to the table. Then set seqno 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;
__OUTPUT__
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
```

(continues on next page)

(continued from previous page)

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)

2.4.4 Synonyms

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

EDB Postgres Advanced Server supports synonyms for:

- Tables
- Views
- Materialized views
- Sequences
- Packages
- Procedures
- Functions
- Types
- Objects that you can access through a database link
- Other synonyms

Neither the referenced schema or referenced object must exist when you create the synonym. A synonym can refer to a nonexistent object or schema. A synonym becomes invalid if you drop the referenced object or schema. You must explicitly drop a synonym to remove it.

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

When EDB Postgres Advanced Server executes an SQL command, the privileges of the current user are checked against the synonym's underlying database object. If the user doesn't have the proper permissions for that object, the SQL command fails.

Creating a synonym

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

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

Parameters

`syn_name`

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

`schema`

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

`object_name`

`object_name` specifies the name of the object.

`object_schema`

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

`dblink_name`

`dblink_name` specifies the name of the database link through which you can access a target object.

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

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

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

This is a shorthand way to write:

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

This example creates a synonym named `personnel` that refers to the `enterprisedb.emp` table:

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

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

```
SHOW SEARCH_PATH;
__OUTPUT__
  search_path
-----
development,accounting
(1 row)
```

Example

In the example, if a schema named `development` doesn't exist, the synonym are created in the schema named `accounting`.

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

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

```
SELECT * FROM personnel;
```

```
__OUTPUT__
```

empno	ename	job	mgr	hiredate	sal	comm	deptno
7369	SMITH	CLERK	7902	17-DEC-80 00:00:00	800.00		20
7499	ALLEN	SALESMAN	7698	20-FEB-81 00:00:00	1600.00	300.00	30
7521	WARD	SALESMAN	7698	22-FEB-81 00:00:00	1250.00	500.00	30
7566	JONES	MANAGER	7839	02-APR-81 00:00:00	2975.00		20
7654	MARTIN	SALESMAN	7698	28-SEP-81 00:00:00	1250.00	1400.00	30
7698	BLAKE	MANAGER	7839	01-MAY-81 00:00:00	2850.00		30
7782	CLARK	MANAGER	7839	09-JUN-81 00:00:00	2450.00		10
7788	SCOTT	ANALYST	7566	19-APR-87 00:00:00	3000.00		20
7839	KING	PRESIDENT		17-NOV-81 00:00:00	5000.00		10
7844	TURNER	SALESMAN	7698	08-SEP-81 00:00:00	1500.00	0.00	30
7876	ADAMS	CLERK	7788	23-MAY-87 00:00:00	1100.00		20
7900	JAMES	CLERK	7698	03-DEC-81 00:00:00	950.00		30
7902	FORD	ANALYST	7566	03-DEC-81 00:00:00	3000.00		20
7934	MILLER	CLERK	7782	23-JAN-82 00:00:00	1300.00		10
8142	ANDERSON	CLERK	7902	17-DEC-06 00:00:00	1300.00		20

```
(15 rows)
```

Deleting a synonym

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

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

Parameters

`syn_name`

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

`schema`

`schema` specifies the name of the schema in which the synonym resides.

Like any other object that can be schema qualified, you can have two synonyms with the same name in your search path. To disambiguate the name of the synonym that you're dropping, include a schema name. Unless a synonym is schema qualified in the `DROP SYNONYM` command, EDB Postgres Advanced Server deletes the first instance of the synonym it finds in your search path.

You can optionally include the `PUBLIC` clause to drop a synonym that resides in the `public` schema. Compatible with Oracle databases, the `DROP PUBLIC SYNONYM` command drops a synonym that resides in the `public` schema:

```
DROP PUBLIC SYNONYM <syn_name>;
```

Example

The following example drops the synonym `personnel`:

```
DROP SYNONYM personnel;
```

2.4.5 Hierarchical queries

A *hierarchical query* is a type of query that returns the rows of the result set in a hierarchical order based on data forming a parent-child relationship. A hierarchy is typically represented by an inverted tree structure. The tree is made up of interconnected *nodes*. Each node can 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, for example, the trivial case where the tree is made up of a single node. In this case, the node 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

A single given row can appear in more than one tree and thus appear more than once in the result set.

Syntax

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. This example shows the context of where the `CONNECT BY` clause and its associated optional clauses appear in the `SELECT` command:

```
SELECT <select_list> FROM <table_expression> [ WHERE ...]
[ START WITH <start_expression> ]
[ { CONNECT BY <condition> } ]
[ ORDER SIBLINGS BY <column1> [ ASC | DESC ]
  [, <column2> [ ASC | DESC ] ] ...
[ GROUP BY ...]
[ HAVING ...]
[ <other> ...]
```

`select_list` is one or more expressions that make up 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.

`CONNECT BY` specifies the relationship between a parent record and its child records of the hierarchy.

- You can specify the `PRIOR` keyword with only one expression in a condition in the `CONNECT BY` clause. You must specify it with either right or left expression to refer to a parent row in a hierarchical query. For example:

```
PRIOR expr = expr
or
expr = PRIOR expr
```

- If the `CONNECT BY condition` is compound, only one condition requires the `PRIOR` keyword, although there can be multiple `PRIOR` conditions.
- You can use the `PRIOR` keyword to evaluate the parent row in a parent-child relationship in a hierarchical query.
- You can't use `PRIOR` keyword with unary expressions.
- You can use the `PRIOR` keyword to add a pseudo-column for each record corresponding to an expression in a target list. The pseudo-column represents a parent record's actual value when referenced in a child query using a `PRIOR` expression.

Use the `CONNECT_BY_ROOT` operator to further enhance the functionality of the `'CONNECT BY PRIOR'`.

Examples

This example uses the `CONNECT BY` clause with `PRIOR` to define a relationship between employees and managers. Using the `LEVEL` value, the employee names are indented to further emphasize the depth in the hierarchy of each row.

```
SELECT LPAD(ename,length(ename)+2*(LEVEL-1), ) AS employee, empno, mgr, LEVEL
FROM emp
CONNECT BY PRIOR empno = mgr AND PRIOR emp.deptno = emp.deptno;
```

The output from this query is:

2.4.6 Hierarchical queries

Defining the parent/child relationship

To determine the children for any given row:

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`.

Iteratively repeating this process, treating each child node found in the prior steps as a parent, constructs an inverted tree of nodes. The process is complete when the final set of child nodes has no children of its 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, that is, the rows that are the initial parent nodes upon which to apply the algorithm.

Selecting the root nodes

The `START WITH` clause determines the rows selected by `table_expression` to use 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.

Organization tree in the sample application

To see an example of an organizational tree, consider the `emp` table of the sample application. The rows of the `emp` table form a hierarchy based on 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. He has no manager, therefore KING's `mgr` column is null. Also, it's possible for an employee to act as a manager for more than one employee. This relationship forms a typical tree-structured hierarchical organization chart:

To form a hierarchical query based on this relationship, the `SELECT` command includes the clause `CONNECT BY PRIOR empno = mgr`. For example, given the company president, KING, with employee number 7839, any employee whose `mgr` column is 7839 reports directly to KING. This relationship is true for JONES, BLAKE, and CLARK, the child nodes

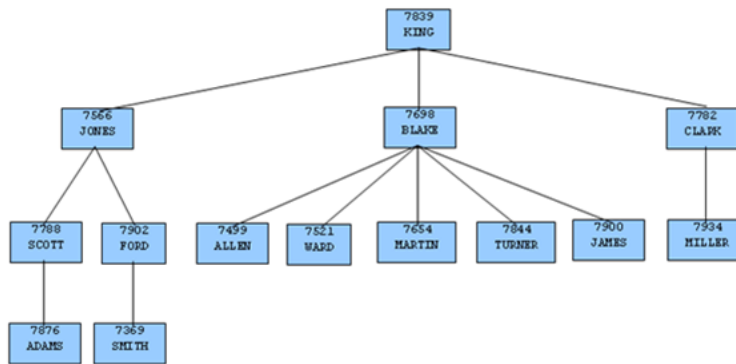


Fig. 2: Employee Organization Hierarchy

of KING . Similarly, for employee JONES , any other employee with `mgr` column equal to 7566 is a child node of JONES . These nodes are SCOTT and FORD in the 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:

```

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:

```

__OUTPUT__
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)

```

Designating the node level

LEVEL is a pseudo-column that you can use 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.

Example

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 is:

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

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

Ordering the siblings

You can order the result set so the siblings appear in ascending or descending order by selected column values using the ORDER SIBLINGS BY clause. This is a special case of the ORDER BY clause that you can use only with hierarchical queries.

You can further modify the previous query by adding 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.

```

__OUTPUT__
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 don't satisfy the `WHERE` clause were eliminated from the output.

```

__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)

```

Retrieving the root node with `CONNECT_BY_ROOT`

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

Note

A *unary operator* operates on a single operand. In the case of `CONNECT_BY_ROOT` , it's the column name following the `CONNECT_BY_ROOT` keyword.

Syntax

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

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

Some points to note about the `CONNECT_BY_ROOT` operator:

- You can use the `CONNECT_BY_ROOT` operator in the `SELECT` list, `WHERE` clause, `GROUP BY` clause, `HAVING` clause, `ORDER BY` clause, and `ORDER SIBLINGS BY` clause as long as the `SELECT` command is for a hierarchical query.
- You can't use the `CONNECT_BY_ROOT` operator in the `CONNECT BY` clause or the `START WITH` clause of the hierarchical query.
- You can apply `CONNECT_BY_ROOT` to an expression involving a column. To do so, you must enclose the expression in parentheses.

Examples

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

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

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

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

A similar query but producing only one tree starting with the single, top-level employee where the `mgr` column is null:

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

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

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

By contrast, this example omits the START WITH clause, thereby resulting in 14 trees:

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

In the output from the query, each node appears at least once as a root node under the mgr empno and mgr ename columns, since even the leaf nodes form the top of their own trees.

```
__OUTPUT__
level | employee | empno | mgr | mgr empno | mgr ename
-----+-----+-----+-----+-----+-----
1 | ADAMS | 7876 | 7788 | 7876 | ADAMS
1 | ALLEN | 7499 | 7698 | 7499 | ALLEN
1 | BLAKE | 7698 | 7839 | 7698 | BLAKE
2 | ALLEN | 7499 | 7698 | 7698 | BLAKE
2 | JAMES | 7900 | 7698 | 7698 | BLAKE
2 | MARTIN | 7654 | 7698 | 7698 | BLAKE
2 | TURNER | 7844 | 7698 | 7698 | BLAKE
2 | WARD | 7521 | 7698 | 7698 | BLAKE
1 | CLARK | 7782 | 7839 | 7782 | CLARK
```

(continues on next page)

(continued from previous page)

2	MILLER	7934	7782	7782	CLARK
1	FORD	7902	7566	7902	FORD
2	SMITH	7369	7902	7902	FORD
1	JAMES	7900	7698	7900	JAMES
1	JONES	7566	7839	7566	JONES
2	FORD	7902	7566	7566	JONES
3	SMITH	7369	7902	7566	JONES
2	SCOTT	7788	7566	7566	JONES
3	ADAMS	7876	7788	7566	JONES
1	KING	7839		7839	KING
2	BLAKE	7698	7839	7839	KING
3	ALLEN	7499	7698	7839	KING
3	JAMES	7900	7698	7839	KING
3	MARTIN	7654	7698	7839	KING
3	TURNER	7844	7698	7839	KING
3	WARD	7521	7698	7839	KING
2	CLARK	7782	7839	7839	KING
3	MILLER	7934	7782	7839	KING
2	JONES	7566	7839	7839	KING
3	FORD	7902	7566	7839	KING
4	SMITH	7369	7902	7839	KING
3	SCOTT	7788	7566	7839	KING
4	ADAMS	7876	7788	7839	KING
1	MARTIN	7654	7698	7654	MARTIN
1	MILLER	7934	7782	7934	MILLER
1	SCOTT	7788	7566	7788	SCOTT
2	ADAMS	7876	7788	7788	SCOTT
1	SMITH	7369	7902	7369	SMITH
1	TURNER	7844	7698	7844	TURNER
1	WARD	7521	7698	7521	WARD

(39 rows)

When applied to an expression that isn't enclosed in parentheses, the `CONNECT_BY_ROOT` operator affects only the term `ename` immediately following it. The subsequent concatenation of `|| ' manages ' || ename` isn't part of the `CONNECT_BY_ROOT` operation. Hence the second occurrence of `ename` results in the value of the currently processed row, while the first occurrence of `ename` results in the value from the root node.

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

Note the values produced under the `top mgr/employee` column in the output from the query:

```
__OUTPUT__
level | employee | empno | mgr | top mgr/employee
-----+-----+-----+-----+-----
1 | BLAKE    | 7698  | 7839 | BLAKE manages BLAKE
2 | ALLEN    | 7499  | 7698 | BLAKE manages ALLEN
2 | JAMES    | 7900  | 7698 | BLAKE manages JAMES
```

(continues on next page)

(continued from previous page)

2	MARTIN	7654	7698	BLAKE manages MARTIN
2	TURNER	7844	7698	BLAKE manages TURNER
2	WARD	7521	7698	BLAKE manages WARD
1	CLARK	7782	7839	CLARK manages CLARK
2	MILLER	7934	7782	CLARK manages MILLER
1	JONES	7566	7839	JONES manages JONES
2	FORD	7902	7566	JONES manages FORD
3	SMITH	7369	7902	JONES manages SMITH
2	SCOTT	7788	7566	JONES manages SCOTT
3	ADAMS	7876	7788	JONES manages ADAMS

(13 rows)

This example uses the `CONNECT_BY_ROOT` operator on an expression enclosed in parentheses:

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

The values of both `employee` and `empno` are affected by the `CONNECT_BY_ROOT` operator. As a result, they return the values from the root node under the `top mgr/empno` column:

__OUTPUT__				
level	employee	empno	mgr	top mgr/empno
-----+-----+-----+-----+-----				
1	BLAKE	7698	7839	Manager BLAKE is emp # 7698
2	ALLEN	7499	7698	Manager BLAKE is emp # 7698
2	JAMES	7900	7698	Manager BLAKE is emp # 7698
2	MARTIN	7654	7698	Manager BLAKE is emp # 7698
2	TURNER	7844	7698	Manager BLAKE is emp # 7698
2	WARD	7521	7698	Manager BLAKE is emp # 7698
1	CLARK	7782	7839	Manager CLARK is emp # 7782
2	MILLER	7934	7782	Manager CLARK is emp # 7782
1	JONES	7566	7839	Manager JONES is emp # 7566
2	FORD	7902	7566	Manager JONES is emp # 7566
3	SMITH	7369	7902	Manager JONES is emp # 7566
2	SCOTT	7788	7566	Manager JONES is emp # 7566
3	ADAMS	7876	7788	Manager JONES is emp # 7566

(13 rows)

Retrieving a path with `SYS_CONNECT_BY_PATH`

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

Syntax

The signature of the function is:

```
SYS_CONNECT_BY_PATH (<column>, <delimiter>)
```

The function takes two arguments:

- `column` is the name of a column that resides in a table specified in the hierarchical query that's calling the function.
- `delimiter` is the `varchar` value that separates each entry in the specified column.

Examples

This example returns a list of employee names and their managers. If the manager has a manager, that name is appended to the result.

```
edb=# SELECT level, ename , SYS_CONNECT_BY_PATH(ename, '/') managers
      FROM emp
      CONNECT BY PRIOR empno = mgr
      START WITH mgr IS NULL
      ORDER BY level, ename, managers;
__OUTPUT__
 level | ename | managers
-----+-----+-----
 1 | KING | /KING
 2 | BLAKE | /KING/BLAKE
 2 | CLARK | /KING/CLARK
 2 | JONES | /KING/JONES
 3 | ALLEN | /KING/BLAKE/ALLEN
 3 | FORD | /KING/JONES/FORD
 3 | JAMES | /KING/BLAKE/JAMES
 3 | MARTIN | /KING/BLAKE/MARTIN
 3 | MILLER | /KING/CLARK/MILLER
 3 | SCOTT | /KING/JONES/SCOTT
 3 | TURNER | /KING/BLAKE/TURNER
 3 | WARD | /KING/BLAKE/WARD
 4 | ADAMS | /KING/JONES/SCOTT/ADAMS
 4 | SMITH | /KING/JONES/FORD/SMITH
(14 rows)
```

In the result set:

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

The EDB Postgres Advanced Server implementation of `SYS_CONNECT_BY_PATH` doesn't support use of:

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

2.4.7 Multidimensional analysis

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

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

Aggregating results

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

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

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

Note

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

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

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

2.4.8 Multidimensional analysis

Useful extensions

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

The `GROUPING SETS` extension lets you combine different types of groupings into a single result set.

The `GROUPING` and `GROUPING_ID` functions help you to interpret the result set.

ROLLUP extension

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

Syntax

The syntax for a single ROLLUP is:

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

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

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

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

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

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

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

The items specified in `select_list` must either:

- Also appear in the ROLLUP `expression_list`
- Be aggregate functions such as COUNT, SUM, AVG, MIN, or MAX
- Be constants or functions whose return values are independent of the individual rows in the group (for example, the SYSDATE function)

Use the GROUP BY clause to specify multiple ROLLUP extensions as well as multiple occurrences of other GROUP BY extensions and individual expressions.

Use the ORDER BY clause to display the output in a hierarchical or other meaningful structure. Using this clause guarantees the order of the result set.

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

Examples

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

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

The result of the query displays a count of the number of employees for each unique combination of `loc`, `dname`, and `job`, as well as subtotals for each unique combination of `loc` and `dname` for each unique value of `loc`. A grand total appears on the last line:

```
__OUTPUT__
loc      | dname    | job      | employees
```

(continues on next page)

(continued from previous page)

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

The following query shows the effect of combining items in the ROLLUP list in parentheses:

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

The output doesn't include subtotals for loc and dname combinations as in the prior example:

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

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

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

Now there's a subtotal for each unique `loc` and `dname` combination but none for unique values of `loc`:

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

CUBE extension

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

Syntax

The syntax for a single `CUBE` is:

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

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

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

In addition, a subtotal is returned for the first item in the list (`expr_1` or the combination of (`expr_1a`, `expr_1b`, ...), whichever is specified) for each unique value. A subtotal is returned for the second item in the list (`expr_2` or the combination of (`expr_2a`, `expr_2b`, ...), whichever is specified) for each unique value. A subtotal is also

returned for each unique combination of the first item and the second item. Similarly, if there's a third item, a subtotal is returned for:

- Each unique value of the third item
- Each unique value of the third item and first item combination
- Each unique value of the third item and second item combination
- Each unique value of the third item, second item, and first item combination

Finally, a grand total is returned for the entire result set.

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

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

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

The items specified in select_list must either:

- Also appear in the CUBE expression_list
- Be aggregate functions such as COUNT, SUM, AVG, MIN, or MAX
- Be constants or functions whose return values are independent of the individual rows in the group (for example, the SYSDATE function)

Use the GROUP BY clause to specify multiple CUBE extensions as well as multiple occurrences of other GROUP BY extensions and individual expressions.

Use the ORDER BY clause to display the output in a meaningful structure and to guarantee the order of the result set.

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

Examples

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

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

The following is the result of the query. It displays a count of the number of employees for each combination of loc, dname, and job, as well as subtotals for:

- Each combination of loc and dname
- Each combination of loc and job
- Each combination of dname and job
- Each unique value of loc
- Each unique value of dname
- Each unique value of job

A grand total appears on the last line.

__OUTPUT__			
loc	dname	job	employees
BOSTON	OPERATIONS	ANALYST	1
BOSTON	OPERATIONS	CLERK	1
BOSTON	OPERATIONS	MANAGER	1
BOSTON	OPERATIONS		3
BOSTON	RESEARCH	ANALYST	2
BOSTON	RESEARCH	CLERK	2
BOSTON	RESEARCH	MANAGER	1
BOSTON	RESEARCH		5
BOSTON		ANALYST	3
BOSTON		CLERK	3
BOSTON		MANAGER	2
BOSTON			8
CHICAGO	SALES	CLERK	1
CHICAGO	SALES	MANAGER	1
CHICAGO	SALES	SALESMAN	4
CHICAGO	SALES		6
CHICAGO		CLERK	1
CHICAGO		MANAGER	1
CHICAGO		SALESMAN	4
CHICAGO			6
NEW YORK	ACCOUNTING	CLERK	1
NEW YORK	ACCOUNTING	MANAGER	1
NEW YORK	ACCOUNTING	PRESIDENT	1
NEW YORK	ACCOUNTING		3
NEW YORK		CLERK	1
NEW YORK		MANAGER	1
NEW YORK		PRESIDENT	1
NEW YORK			3
	ACCOUNTING	CLERK	1
	ACCOUNTING	MANAGER	1
	ACCOUNTING	PRESIDENT	1
	ACCOUNTING		3
	OPERATIONS	ANALYST	1
	OPERATIONS	CLERK	1
	OPERATIONS	MANAGER	1
	OPERATIONS		3
	RESEARCH	ANALYST	2
	RESEARCH	CLERK	2
	RESEARCH	MANAGER	1
	RESEARCH		5
	SALES	CLERK	1
	SALES	MANAGER	1
	SALES	SALESMAN	4
	SALES		6
		ANALYST	3
		CLERK	5
		MANAGER	4
		PRESIDENT	1
		SALESMAN	4
			17

(continues on next page)

(continued from previous page)

(50 rows)

The following query shows the effect of combining items in the CUBE list in parentheses:

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

The output has no subtotals for permutations involving loc and dname combinations, loc and job combinations, for dname by itself, or for job by itself.

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

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

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

In this output, the permutations are performed for dname and job in each grouping of loc :

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

GROUPING SETS extension

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

A `UNION ALL` operation, and therefore the `GROUPING SETS` extension, doesn't eliminate duplicate rows from the result sets that are being combined.

Syntax

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

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

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

The `GROUPING SETS` extension is specified in the context of the `GROUP BY` clause:

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

The items specified in `select_list` must also either:

- Appear in the `GROUPING SETS` `expression_list`
- Be aggregate functions such as `COUNT`, `SUM`, `AVG`, `MIN`, or `MAX`
- Be constants or functions whose return values are independent of the individual rows in the group (for example, the `SYSDATE` function)

Use the `GROUP BY` clause to specify multiple `GROUPING SETS` extensions as well as multiple occurrences of other `GROUP BY` extensions and individual expressions.

Use the `ORDER BY` clause to display the output in a meaningful structure and to guarantee the order of the result set.

Examples

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

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

The result is as follows:

__OUTPUT__			
loc	dname	job	employees
BOSTON			8
CHICAGO			6
NEW YORK			3
	ACCOUNTING		3
	OPERATIONS		3
	RESEARCH		5
	SALES		6
		ANALYST	3
		CLERK	5
		MANAGER	4
		PRESIDENT	1
		SALESMAN	4

(12 rows)

This is equivalent to the following query, which uses the `UNION ALL` operator:

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

(continues on next page)

(continued from previous page)

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

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

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

This example shows how you can use various types of GROUP BY extensions together in a GROUPING SETS expression list:

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

The following is the output from this query:

__OUTPUT__			
loc	dname	job	employees
-----+-----+-----+-----			
BOSTON		ANALYST	3
BOSTON		CLERK	3
BOSTON		MANAGER	2
BOSTON			8
BOSTON			8
CHICAGO		CLERK	1
CHICAGO		MANAGER	1
CHICAGO		SALESMAN	4
CHICAGO			6
CHICAGO			6
NEW YORK		CLERK	1
NEW YORK		MANAGER	1
NEW YORK		PRESIDENT	1
NEW YORK			3

(continues on next page)

(continued from previous page)

NEW YORK			3
	ACCOUNTING	CLERK	1
	ACCOUNTING	MANAGER	1
	ACCOUNTING	PRESIDENT	1
	ACCOUNTING		3
	OPERATIONS	ANALYST	1
	OPERATIONS	CLERK	1
	OPERATIONS	MANAGER	1
	OPERATIONS		3
	RESEARCH	ANALYST	2
	RESEARCH	CLERK	2
	RESEARCH	MANAGER	1
	RESEARCH		5
	SALES	CLERK	1
	SALES	MANAGER	1
	SALES	SALESMAN	4
	SALES		6
		ANALYST	3
		CLERK	5
		MANAGER	4
		PRESIDENT	1
		SALESMAN	4
			17
			17

(38 rows)

The output is basically a concatenation of the result sets from GROUP BY loc, GROUP BY ROLLUP (dname, job), and GROUP BY CUBE (job, loc). These individual queries are:

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

The result set from the GROUP BY loc clause is:

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

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

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

The result set from the GROUP BY ROLLUP (dname, job) clause is:

```

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

```

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

```

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

```

The result set from the GROUP BY CUBE (job, loc) clause is:

```

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

```

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

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

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

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

(continues on next page)

(continued from previous page)

		ANALYST	3
		CLERK	5
		MANAGER	4
		PRESIDENT	1
		SALESMAN	4
			17
			17
(38 rows)			

GROUPING function

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

Syntax

The general syntax for use of the `GROUPING` function is:

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

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

The return value of the `GROUPING` function is either a 0 or 1. In the result set of a query:

- If the column expression specified in the `GROUPING` function is null because the row represents a subtotal over multiple values of that column, then the `GROUPING` function returns a value of 1.
- If the row returns results based on a particular value of the column specified in the `GROUPING` function, then the `GROUPING` function returns a value of 0.

In the latter case, the column can be null as well as non-null. In any case, it's for a particular value of that column, not a subtotal across multiple values.

Examples

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

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

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

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

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

```

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

```

This query produces the following result:

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

You can use the `GROUPING` function to distinguish a subtotal row from a base aggregate row or from certain subtotal rows. These subtotal rows have one item in the expression list that returns null as a result of the column on which the expression is based being null for one or more rows in the table. This is opposed to representing a subtotal over the

column.

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

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

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

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

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

__OUTPUT__				
loc	job	employees	gf_loc	gf_job
BOSTON	ANALYST	3	0	0
BOSTON	CLERK	3	0	0
BOSTON	MANAGER	2	0	0
BOSTON		1	0	0
BOSTON		9	0	1
	ANALYST	3	1	0
	CLERK	3	1	0
	MANAGER	2	1	0
		1	1	0
		9	1	1

(10 rows)

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

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

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

GROUPING_ID function

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

The `GROUPING` function takes only one column expression and returns an indication of whether a row is a subtotal over all values of the given column. Thus, you might need multiple `GROUPING` functions to interpret the level of subtotals for queries with multiple grouping columns.

The `GROUPING_ID` function accepts one or more column expressions that were used in the `ROLLBACK`, `CUBE`, or `GROUPING SETS` extensions. It returns a single integer that you can use to determine over which of these columns a

subtotal was aggregated.

Syntax

The general syntax for the GROUPING_ID function is:

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

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

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

Examples

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

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

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

__OUTPUT__					
loc	dname	employees	gf_loc	gf_dname	gid
BOSTON	OPERATIONS	3	0	0	0
BOSTON	RESEARCH	5	0	0	0
CHICAGO	SALES	6	0	0	0
NEW YORK	ACCOUNTING	3	0	0	0
BOSTON		8	0	1	1
CHICAGO		6	0	1	1
NEW YORK		3	0	1	1
	ACCOUNTING	3	1	0	2
	OPERATIONS	3	1	0	2
	RESEARCH	5	1	0	2
	SALES	6	1	0	2
		17	1	1	3

(12 rows)

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

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

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

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

To display only those subtotals by `dname`, use the following simplified query with a `HAVING` clause based on the `GROUPING_ID` function:

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

The result of the query is:

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

2.5 Sample database description

The examples in the documentation use the sample tables `dept`, `emp`, and `jobhist`, which are created and loaded when EDB Postgres Advanced Server is installed.

2.5.1 Available scripts

You can re-create the tables and programs in the sample database at any time by executing the following script:

```
/usr/edb/as<xx>/share/pg-sample.sql
```

Where <xx> is the EDB Postgres Advanced Server version number.

In addition, a script in the same directory contains the database objects created using syntax compatible with Oracle databases. This script file is `edb-sample.sql`. The script:

- 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 are 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 using the command:

```
SHOW SEARCH_PATH;
```

You can use PSQL (a terminal-based interface for PostgreSQL) commands to modify the search path.

2.5.2 About the sample database

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 it tracks the locations of its 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, and so on. 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 the `pg-sample.sql` script:

```
SET datestyle TO iso, dmy;

--
-- Script that creates the sample tables, views
-- 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          NUMERIC(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
```

(continues on next page)

(continued from previous page)

```

    dname          VARCHAR(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc            VARCHAR(13)
);
--
-- Create the emp table
--
CREATE TABLE emp (
    empno          NUMERIC(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename          VARCHAR(10),
    job            VARCHAR(9),
    mgr            NUMERIC(4),
    hiredate       DATE,
    sal            NUMERIC(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm           NUMERIC(7,2),
    deptno         NUMERIC(2) CONSTRAINT emp_ref_dept_fk
                    REFERENCES dept (deptno)
);
--
-- Create the jobhist table
--
CREATE TABLE jobhist (
    empno          NUMERIC(4) NOT NULL,
    startdate      TIMESTAMP(0) NOT NULL,
    enddate        TIMESTAMP(0),
    job            VARCHAR(9),
    sal            NUMERIC(7,2),
    comm           NUMERIC(7,2),
    deptno         NUMERIC(2),
    chgdesc        VARCHAR(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;

```

(continues on next page)

(continued from previous page)

```

-- 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;

```

(continues on next page)

(continued from previous page)

```

--
--  Function that lists all employees numbers and names
--  from the emp table using a cursor.
--
CREATE OR REPLACE FUNCTION list_emp() RETURNS VOID
AS $$
DECLARE
    v_empno      NUMERIC(4);
    v_ename      VARCHAR(10);
    emp_cur CURSOR FOR
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    RAISE INFO EMPNO ENAME;
    RAISE INFO -----;
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN NOT FOUND;
        RAISE INFO % %, v_empno, v_ename;
    END LOOP;
    CLOSE emp_cur;
    RETURN;
END;
$$ LANGUAGE plpgsql;
--
--  Function that selects an employee row given the employee
--  number and displays certain columns.
--
CREATE OR REPLACE FUNCTION select_emp (
    p_empno      NUMERIC
) RETURNS VOID
AS $$
DECLARE
    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  VARCHAR(10);
BEGIN
    SELECT INTO
        v_ename, v_hiredate, v_sal, v_comm, v_dname
        ename, hiredate, sal, COALESCE(comm, 0), dname
    FROM emp e, dept d
    WHERE empno = p_empno
        AND e.deptno = d.deptno;
    IF NOT FOUND THEN
        RAISE INFO Employee % not found, p_empno;
        RETURN;
    END IF;
    v_disp_date := TO_CHAR(v_hiredate, MM/DD/YYYY);
    RAISE INFO Number      : %, p_empno;

```

(continues on next page)

(continued from previous page)

```

    RAISE INFO Name      : %, v_ename;
    RAISE INFO Hire Date : %, v_disp_date;
    RAISE INFO Salary    : %, v_sal;
    RAISE INFO Commission: %, v_comm;
    RAISE INFO Department: %, v_dname;
    RETURN;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO The following is SQLERRM : %, SQLERRM;
        RAISE INFO The following is SQLSTATE: %, SQLSTATE;
        RETURN;
END;
$$ LANGUAGE plpgsql;
--
-- A RECORD type used to format the return value of
-- function, emp_query.
--
CREATE TYPE emp_query_type AS (
    empno      NUMERIC,
    ename       VARCHAR(10),
    job         VARCHAR(9),
    hiredate    DATE,
    sal         NUMERIC
);
--
-- Function that queries the emp table based on
-- department number and employee number or name. Returns
-- employee number and name as INOUT parameters and job,
-- hire date, and salary as OUT parameters. These are
-- returned in the form of a record defined by
-- RECORD type, emp_query_type.
--
CREATE OR REPLACE FUNCTION emp_query (
    IN p_deptno      NUMERIC,
    INOUT p_empno     NUMERIC,
    INOUT p_ename     VARCHAR,
    OUT p_job         VARCHAR,
    OUT p_hiredate    DATE,
    OUT p_sal         NUMERIC
)
AS $$
BEGIN
    SELECT INTO
        p_empno, p_ename, p_job, p_hiredate, p_sal
        empno, ename, job, hiredate, sal
    FROM emp
    WHERE deptno = p_deptno
        AND (empno = p_empno
            OR ename = UPPER(p_ename));
END;
$$ LANGUAGE plpgsql;
--

```

(continues on next page)

(continued from previous page)

```

-- Function to call emp_query_caller with IN and INOUT
-- parameters. Displays the results received from INOUT and
-- OUT parameters.
--
CREATE OR REPLACE FUNCTION emp_query_caller() RETURNS VOID
AS $$
DECLARE
    v_deptno      NUMERIC;
    v_empno       NUMERIC;
    v_ename       VARCHAR;
    v_rows        INTEGER;
    r_emp_query    EMP_QUERY_TYPE;
BEGIN
    v_deptno := 30;
    v_empno := 0;
    v_ename := Martin;
    r_emp_query := emp_query(v_deptno, v_empno, v_ename);
    RAISE INFO Department : %, v_deptno;
    RAISE INFO Employee No: %, (r_emp_query).empno;
    RAISE INFO Name       : %, (r_emp_query).ename;
    RAISE INFO Job        : %, (r_emp_query).job;
    RAISE INFO Hire Date  : %, (r_emp_query).hiredate;
    RAISE INFO Salary     : %, (r_emp_query).sal;
    RETURN;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO The following is SQLERRM : %, SQLERRM;
        RAISE INFO The following is SQLSTATE: %, SQLSTATE;
        RETURN;
END;
$$ LANGUAGE plpgsql;

--
-- Function to compute yearly compensation based on semimonthly
-- salary.
--
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal      NUMERIC,
    p_comm     NUMERIC
) RETURNS NUMERIC
AS $$
BEGIN
    RETURN (p_sal + COALESCE(p_comm, 0)) * 24;
END;
$$ LANGUAGE plpgsql;

--
-- 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() RETURNS INTEGER
AS $$
DECLARE
    v_cnt      INTEGER := 1;

```

(continues on next page)

(continued from previous page)

```

    v_new_empno      INTEGER;
BEGIN
    WHILE v_cnt > 0 LOOP
        SELECT INTO v_new_empno nextval(next_empno);
        SELECT INTO v_cnt COUNT(*) FROM emp WHERE empno = v_new_empno;
    END LOOP;
    RETURN v_new_empno;
END;
$$ LANGUAGE plpgsql;
--
-- Function that adds a new clerk to table emp.
--
CREATE OR REPLACE FUNCTION hire_clerk (
    p_ename          VARCHAR,
    p_deptno         NUMERIC
) RETURNS NUMERIC
AS $$
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, CLERK, 7782,
        CURRENT_DATE, 950.00, NULL, p_deptno);
    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 : %, SQLERRM;
        RAISE INFO The following is SQLSTATE: %, SQLSTATE;
        RETURN -1;
END;
$$ LANGUAGE plpgsql;
--
-- Function that adds a new salesman to table emp.

```

(continues on next page)

(continued from previous page)

```

--
CREATE OR REPLACE FUNCTION hire_salesman (
    p_ename          VARCHAR,
    p_sal            NUMERIC,
    p_comm           NUMERIC
) RETURNS NUMERIC
AS $$
DECLARE
    v_empno          NUMERIC(4);
    v_ename          VARCHAR(10);
    v_job            VARCHAR(9);
    v_mgr            NUMERIC(4);
    v_hiredate       DATE;
    v_sal            NUMERIC(7,2);
    v_comm           NUMERIC(7,2);
    v_deptno         NUMERIC(2);
BEGIN
    v_empno := new_empno();
    INSERT INTO emp VALUES (v_empno, p_ename, SALESMAN, 7698,
        CURRENT_DATE, p_sal, p_comm, 30);
    SELECT INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        empno, ename, job, mgr, hiredate, sal, comm, deptno
    FROM emp WHERE empno = v_empno;
    RAISE INFO Department : %, v_deptno;
    RAISE INFO Employee No: %, v_empno;
    RAISE INFO Name       : %, v_ename;
    RAISE INFO Job        : %, v_job;
    RAISE INFO Manager    : %, v_mgr;
    RAISE INFO Hire Date  : %, v_hiredate;
    RAISE INFO Salary     : %, v_sal;
    RAISE INFO Commission : %, v_comm;
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO The following is SQLERRM : %, SQLERRM;
        RAISE INFO The following is SQLSTATE: %, SQLSTATE;
        RETURN -1;
END;
$$ 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

```

(continues on next page)

(continued from previous page)

```

    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 FUNCTION user_audit_trig() RETURNS TRIGGER
AS $$
DECLARE
    v_action      VARCHAR(24);
    v_text        TEXT;
BEGIN
    IF TG_OP = INSERT THEN
        v_action := added employee(s) on ;
    ELSIF TG_OP = UPDATE THEN
        v_action := updated employee(s) on ;
    ELSIF TG_OP = DELETE THEN
        v_action := deleted employee(s) on ;
    END IF;
    v_text := User || USER || v_action || CURRENT_DATE;
    RAISE INFO %, v_text;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER user_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH STATEMENT EXECUTE PROCEDURE user_audit_trig();
--
-- 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 FUNCTION emp_sal_trig() RETURNS TRIGGER
AS $$
DECLARE
    sal_diff      NUMERIC(7,2);
BEGIN
    IF TG_OP = INSERT THEN
        RAISE INFO Inserting employee %, NEW.empno;
        RAISE INFO ..New salary: %, NEW.sal;
        RETURN NEW;

```

(continues on next page)

(continued from previous page)

```
END IF;
IF TG_OP = UPDATE THEN
    sal_diff := NEW.sal - OLD.sal;
    RAISE INFO Updating employee %, OLD.empno;
    RAISE INFO ..Old salary: %, OLD.sal;
    RAISE INFO ..New salary: %, NEW.sal;
    RAISE INFO ..Raise : %, sal_diff;
    RETURN NEW;
END IF;
IF TG_OP = DELETE THEN
    RAISE INFO Deleting employee %, OLD.empno;
    RAISE INFO ..Old salary: %, OLD.sal;
    RETURN OLD;
END IF;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER emp_sal_trig
    BEFORE DELETE OR INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_sal_trig();
COMMIT;
```


CHAPTER 3

Supported platforms

EDB Postgres Advanced Server supports installations on Linux and Windows platforms.

To learn about the platform support for EDB Postgres Advanced Server, you can refer either to the platform support for EDB Postgres Advanced Server on the [Platform Compatibility page](#) on the EDB website or Installing EDB Postgres Advanced Server .

EDB Postgres Advanced Server release notes

EDB Postgres Advanced Server 18 introduces diverse enhancements focused on improving core database performance and efficiency, while expanding relational features and modern security integration.

The EDB Postgres Advanced Server documentation describes the latest version of EDB Postgres Advanced Server 18 including minor releases and patches. These release notes provide information on what was new in each release.

Version	Release date	Upstream merges
<i>EDB Postgres Advanced Server 18.1.0 release notes</i>	25 Nov 2025	<i>EDB Postgres Advanced Server 18.1.0 release notes</i>

4.1 Component certification

The EDB components and extensions supported by EDB Postgres Advanced Server have all been verified and certified with EPAS 18. For a complete list of available extensions for EDB Postgres Advanced Server, see Postgres extensions available by deployment .

4.2 Support announcements

4.2.1 Backup and Recovery Tool (BART) incompatibility

The EDB Backup and Recovery Tool (BART) isn’t supported by EDB Postgres Advanced Server or PostgreSQL version 14 and later. We strongly recommend that you move to Barman or PgBackRest as your backup recovery tool.

EDB Postgres Advanced Server 18.1.0 release notes

Released: 25 Nov 2025

EDB Postgres Advanced Server 18.1.0 includes the following enhancements and bug fixes:

Type	Description	Addresses
Upstream merge	Merged with community PostgreSQL 18.1. See the PostgreSQL 18.1 Release Notes for more information.	
Feature	Added the <i>INACTIVE_ACCOUNT_TIME</i> option to password profiles. The new Automated Inactive Account Locking feature enhances database security by mitigating risks from dormant user accounts. It allows security administrators to configure a profile (for example, <i>INACTIVE_ACCOUNT_TIME 60</i>) that automatically locks accounts that haven't logged in for a predefined number of days. This prevents potential misuse of stale accounts, streamlining compliance and reducing the database's attack surface without requiring manual tracking.	
Feature	Added the ability to redact statements containing passwords in the server logs. To address unreliable password redaction, we are exploring a new approach that uses the post-parsing command tree instead of raw string analysis. This method would reliably match all password-changing commands (for example, <i>ALTER USER</i>) using a GUC like <i>edb_log_redact_password_statements</i> . While this prevents malicious code from bypassing redaction, it requires redacting the entire command string, which may hide other commands executed simultaneously.	
Feature	Added the ability to obfuscate passwords in <i>postgres_fdw</i> user mappings. We've enhanced security for <i>postgres_fdw</i> by obfuscating passwords stored in <i>PG_USER_MAPPING</i> . This change ensures that passwords are no longer stored in plaintext within the catalog, even from other superusers, and aligns the security implementation with our existing OCI dblink standard.	
Feature	Added support for the Oracle-equivalent <i>NLSSORT()</i> . To enhance Oracle compatibility, we've implemented the <i>NLSSORT()</i> function transformation. This allows the system to correctly interpret Oracle's NLS parameters and translate them into a PostgreSQL <i>Collate Clause</i> . This change ensures applications correctly perform collation-aware sorting based on locale rules, significantly improving migration fidelity.	
Enhancement	Added support for Secure(TLS) and Authenticated(AUTH) email to <i>UTL_SMTP</i> . Database Administrators migrating from Oracle can now ensure secure email communications with the new <i>UTL_SMTP</i> support for <i>AUTH</i> and <i>STARTTLS</i> . This implementation allows for standard SMTP authentication and enables TLS/SSL encryption of email sessions, helping your applications maintain security policies and compliance.	

continues on next page

Table 2 – continued from previous page

Type	Description	Addresses
Enhancement	Added support to audit nested statements to <code>edb_audit</code> . We've enhanced database security by implementing nested statement audit logging. A new configuration parameter, <code>edb_audit_nested_statements</code> , allows administrators to capture statements executed within functions and procedures. This enhancement ensures complete audit trails, effectively preventing malicious users from bypassing security logs through hidden code execution.	
Enhancement	Enabled an <code>edb_audit_group</code> related option for system catalog tables. We've enhanced security by enabling the <code>edb_audit_group</code> relation option for system catalog tables. This allows users to apply fine-grained audit rules for critical system objects (like monitoring <code>SELECT</code> statements on sensitive tables). Enabling this option for catalog tables allows users to define highly specific tracking rules for critical system objects.	
Enhancement	Added support to audit <code>COPY</code> commands. We've enhanced <code>edb_audit</code> to include explicit support for logging the <code>COPY</code> statement option into <code>edb_audit_statement</code> . This critical update allows users to audit <code>COPY</code> commands to sensitive tables.	
Enhancement	Improved calendar-based scheduling in <code>DBMS_SCHEDULER</code> . We've significantly enhanced <code>DBMS_SCHEDULER</code> with new calendar-based scheduling features for greater precision and flexibility. This includes support for the <code>FREQ=SECONDLY</code> interval (for sub-minute jobs), the <code>INTERVAL</code> keyword, and negative <code>BYMONTHDAY</code> values (to schedule jobs based on the end of the month).	
Enhancement	Added Indicator Array Support for Oracle Dynamic SQL Method 4 to <code>ecpg</code> . We've enhanced <code>ECPG</code> and <code>ECPGPlus</code> to support indicator arrays for bulk operations and Oracle Dynamic SQL Method 4. This upgrade allows developers to efficiently handle <code>NULL</code> values by using parallel arrays of indicator values alongside arrays of data values. This significantly improves the flexibility and performance of data binding and transfer in applications using embedded SQL.	
Enhancement	Added a cumulative fetch row count for non-scroll cursor <code>FETCH</code> to <code>ecpg</code> . We've updated <code>ECPG</code> 's PROC mode to maintain a cumulative row count in <code>sqlca->sqlerrd[2]</code> for <code>FETCH</code> operations on non-scrollable cursors. This aligns its behavior with Oracle's Pro*C, preventing the count from being reset with each fetch. This enhancement improves migration compatibility for applications that rely on the total rows processed.	

continues on next page

Table 2 – continued from previous page

Type	Description	Addresses fix
Enhancement	Added support for <i>ALTER [PUBLIC] DATABASE LINK</i> syntax. Database Administrators required a safe way to modify existing database link details without service disruption. We've introduced the <i>ALTER [PUBLIC] DATABASE LINK</i> command to simplify maintenance tasks. This new command allows for in-place updates, preventing the need to drop and recreate database links that have complex dependent objects, thereby enhancing stability and operational safety.	
Enhancement	Added the ability for <i>pg_dump</i> to dump ACL changes for <i>SYS</i> schema objects. We've enhanced <i>pg_dump</i> to reliably capture ACL (permission) changes for both system and user-defined objects within the <i>SYS</i> schema. Since system objects are now pinned, this change ensures that custom permissions set on crucial system objects are fully preserved during dump and restore operations.	
Change	Replaced EDB Postgres Advanced Server-specific boolean columns in <i>pg_authid</i> with the integer bitmask. To simplify catalog management and support future extensions, several EDB Postgres Advanced Server-specific boolean privilege columns in <i>pg_authid</i> have been consolidated into a single integer bitmask column (<i>rolredwoodprivs</i>). This enhancement reduces column sprawl and improves extensibility, while new SQL functions, for example <i>has_priv-name_privilege()</i> , allow easy checking of user privileges.	
Bug fix	Fixed column name casing issue to fetch <i>result-set</i> correctly from the remote server. We've enhanced <i>edb_dblink_oci</i> to utilize the <i>column_name</i> option in foreign table definitions when building remote queries. This allows users to map local column names to different column names on the remote table, addressing potential naming and case conflicts. The change ensures accurate data retrieval and improves interoperability between PostgreSQL and Oracle-compatible environments.	