



EPAS Fundamentals

リリース *18.1*

EnterpriseDB Corporation

2025 年 11 月 26 日

目次

第 1 章	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	データベースサーバーポストマスター	4
1.3.2	データベースクラスター	4
1.3.3	構成ファイル	4
第 2 章	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	20
2.3.3	Populating a table With rows	21
2.3.4	Querying a table	21
2.3.5	Executing joins between tables	24
2.3.6	Aggregating data	27
2.3.7	Updating a table	29
2.3.8	Deleting a table	29
2.3.9	The SQL language	31
2.4	Advanced concepts	31
2.4.1	Views	31
2.4.2	Foreign keys	32
2.4.3	The ROWNUM pseudo-column	33
2.4.4	Synonyms	35
2.4.5	Hierarchical queries	38
2.4.6	Hierarchical queries	39
2.4.7	Multidimensional analysis	49
2.4.8	Multidimensional analysis	50
2.5	Sample database description	70
2.5.1	使用可能なスクリプト	70
2.5.2	サンプルデータベースについて	70
第 3 章	Supported platforms	83
第 4 章	EDB Postgres Advanced Server release notes	85
4.1	コンポーネント認証	85
4.2	サポートのお知らせ	85
4.2.1	バックアップ&リカバリーツール BART の非互換性	85

EDB Postgres Advanced Server の主要な概念には、その機能、利点、用語が含まれます。SQL チュートリアルでは、サンプルデータベースを使用し、重要なデータベースの概念について説明します。

第1章 EDB Postgres Advanced Server fundamentals

EDB Postgres Advanced Server は、エンタープライズ規模のデータソリューションを管理するために使用できるツールを提供します。これらの重要な概念は、EDB Postgres Advanced Server の設計、実装、および管理に役立ちます。

1.1 Benefits of EDB Postgres Advanced Server

EDB Postgres Advanced Server は、オープンソースの PostgreSQL データベースに拡張機能を追加するスタンドアロン独自のデータベースサーバーです。

EDB Postgres Advanced Server は、コミュニティ Postgres に存在する機能に加えて、エンタープライズ機能を提供するように設計されています。これらの機能

- 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 には、Oracle から Postgres への移行を促進する大量の Oracle compatibility features もあります。

EDB Postgres Advanced Server には、Oracle アプリケーションでサポートされている構文の互換性を提供する拡張機能、ならびに互換性のあるプロシージャロジック、データ型、システムカタログビュー、および EDB の Oracle 互換コネクタ、EDB *Plus、EDB* Loader および他の多くのツールと機能。

EDB Postgres Advanced Server のメジャーバージョンは、PostgreSQL のメジャーバージョンリリースに合わせて毎年リリースされます。EDB Postgres Advanced Server は、通常 PostgreSQL メジャーバージョンリリースの 1〜2 か月後にリリースされます。

EDB Postgres Advanced Server は Linux x86-64、Linux では IBM Power および Windows x86-64 でサポートされています。

1.2 Database compatibility for Oracle developers

EDB Postgres Advanced Server は、Postgres のルック、フィール、および動作を Oracle のようにするため、移行時に書き換えるコードが少なく、すぐに稼働できます。Oracle 互換性機能を使用すると、Oracle 用に作成された多くのアプリケーションを最小限またはまったく変更せずに EDB Postgres Advanced Server で実行できます。

EDB Postgres Advanced Server は、Oracle 互換を提供します。

- Stored procedure language (SPL) ストアドプロシージャ、ファンクション、トリガー、パッケージのデータベースサーバーサイドアプリケーションロジックを作成する場合
- Data types that are compatible with Oracle databases
- Oracle SQL と互換性のある SQL statements
- SQL ステートメントおよびプロシージャルロジックで使用する System and built-in functions
- Oracle のデータディクショナリと互換性のある System catalog views
- Additional compatibility with Oracle MERGE。

!!!関連項目「詳細情報」- 互換性のある SQL 構文、データ型、およびビュー SQL reference。 - Built-in packages のビルトインパッケージの一部であるプロシージャとファンクションによって提供される互換性。 - 互換性のあるツールとユーティリティ EDB *Plus、EDB* Loader、DRITA、および EDB*Wrap Tools, utilities, and components の EDB Postgres Advanced Server インストールに含まれます。 - Oracle Call InterfaceOCI を使用して作成されたアプリケーションの場合、EDB の Open Client ライブラリーは、これらのアプリケーションとの相互運用性を提供します。 [EDB OCL Connector](#) を参照してください。

1.3 Terminology

以下の用語は、EDB Postgres Advanced Server の機能とそのコンポーネントを理解するために重要です。

1.3.1 データベースサーバーポストマスター

データを保存および管理できる主要機能を提供するサーバープロセス。データベースファイルを管理し、クライアントアプリケーションからデータベースへの接続を受け入れ、クライアントに代わってデータベースアクションを実行します。データベースサーバープロセスは *postgres* と呼ばれます。

1.3.2 データベースクラスター

データベースのコレクションを構成する一連のディスク上の構造。クラスターは、データベースサーバーの単一インスタンスによってサービスされます。データベースクラスターは **data** ディレクトリに保存されます。実稼働データベースの **data** ディレクトリを NFS ファイルシステムに保存しないでください。

1.3.3 構成ファイル

Postgres 構成ファイルにリストされているパラメーターを使用して、展開設定、セキュリティ設定、接続動作、およびロギング設定を管理できます。

第2章 SQL fundamentals

Structured Query Language SQL は、データベースにアクセスして操作するための標準言語です。これらの基本は、SQL を操作するための最も一般的なアクションの開始点を提供し、より詳細なドキュメントへのポインタを提供します。このチュートリアルでは、サンプルデータベースを使用し、主要な概念について説明します。

2.1 About the examples

ドキュメントに示されている例では、PSQL プログラムを使用しています。PSQL は、Postgres のターミナルベースのコマンドラインインターフェイスです。PSQL の使用時に通常表示されるプロンプトは、表示されているポイントをさらに明確にするために、これらの例では省略されています。

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

例に関連する次の重要なポイントを検討します。

- EDB Postgres Advanced Server のインストール中に、例と同じ結果を再現するには、Oracle データベースと互換性のある構成とデフォルトを選択する必要があります。PSQL で次のコマンドを発行し、表示された結果を返すことにより、デフォルトの互換性のある構成を確認できます。

```
SHOW edb_redwood_date;

edb_redwood_date
-----
on

SHOW datestyle;

DateStyle
-----
Redwood, DMY

SHOW edb_redwood_strings;

edb_redwood_strings
-----
on
```

- 例では、EDB Postgres Advanced Server のインストール時に作成およびロードされたサンプルテーブル dept、emp、および jobhist を使用します。emp テーブルには、例と同じ結果を再現するには無効にする必要があるトリガーがインストールされます。enterprisedb スーパーユーザーとして EDB Postgres Advanced Server にログインし、次のコマンドを発行してトリガーを無効にします。
-

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

後で、次のコマンドを使用して、emp テーブルのトリガーを再アクティブ化できます。

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

2.2 Conventions used

ここで説明するドキュメントで使用されている規則は、Linux システムと Windows システムの両方に適用されます。

ディレクトリパスは、スラッシュを使用した Linux 形式で表示されます。Windows システムで作業する場合、ディレクトリパスはドライブ文字とコロンで始まり、スラッシュの代わりにバックスラッシュを使用します。

この情報の一部は、PostgreSQL および EDB Postgres Advanced Server データベースシステムに同様に適用される場合があります。EDB Postgres Advanced Server は、EDB Postgres Advanced Server を参照するために使用されます。Postgres は、一般的に PostgreSQL と EDB Postgres Advanced Server の両方を参照するために使用されます。これら 2 つのデータベースシステムを区別する必要がある場合、特定の名前 PostgreSQL または EDB Postgres Advanced Server が使用されます。

PostgreSQL または EDB Postgres Advanced Server 製品のインストールディレクトリパスは、`POSTGRES_INSTALL_HOME` と呼ばれます。

- バージョン 10 以前の対話型インストーラーを使用して実行された EDB Postgres Advanced Server Linux インストールの場合、デフォルトは `/opt/edb/asx.x` です。
- RPM パッケージを使用して実行される EDB Postgres Advanced Server Linux インストールの場合、デフォルトは `/usr/edb/as<xx>` で、`<xx>` は EDB Postgres Advanced Server バージョン番号です。
- EDB Postgres Advanced Server Windows インストールの場合、デフォルトは `C:\Program Files\edb\as<xx>` です。製品バージョン番号は、`x.x` またはバージョン 10 以降の場合は `xx` で表されます。

2.3 SQL tutorial

EDB Postgres Advanced Server は、リレーショナルデータベース管理システム RDBMS です。これは、リレーションに格納されているデータを管理するためのシステムであることを意味します。リレーションとは、本質的にテーブルの数学用語です。各テーブルは、行の名前付けコレクションです。特定のテーブルの各行には、同じ名前付けの列のセットがあり、各列は特定のデータタイプです。列の各行の順序は固定されていますが、SQL はテーブル内の行の順序を決して保証しないことに注意していただき、表示のために明示的にソートすることはできます。

テーブルは **データベース** にグループ化され、単一の EDB Postgres Advanced Server インスタンスによって管理されるデータベースのコレクションは、**データベースクラスター** を構成します。

このチュートリアルでは、サンプルデータベースを使用して RDBMS の主な機能を説明します。

2.3.1 Sample database

このチュートリアルでは、サンプルデータベースを使用して、基本から高度なデータベースの概念を説明します。

Accessing the sample database

EDB Postgres Advanced Server がインストールされると、**edb** という名前のサンプルデータベースが自動的に作成されます。このサンプルデータベースには、このチュートリアルで `/usr/edb/as17/share` ディレクトリにあるスクリプト `edb-sample.sql` を実行した後に使用されるテーブルとプログラムが含まれています。

このスクリプトは次のことを行います。

- 現在接続しているデータベースにサンプルテーブルとプログラムを作成します
- テーブルに対するすべての権限を **PUBLIC** グループに付与

テーブルとプログラムは、現在のユーザーがテーブルとプロシージャを作成する権限がある検索パスの最初のスキーマに作成されます。次のコマンドを発行して、検索パスを表示できます。

```
SHOW SEARCH_PATH;
```

Postgres のターミナルベースのフロントエンドである **PSQL** のコマンドを使用して、検索パスを変更できます。

Overview of the sample database

サンプルデータベースは、組織内の従業員を表します。これには、従業員、部門、および従業員の履歴レコードの 3 種類のレコードが含まれています。

各従業員には、識別番号、名前、雇用日、給与、およびマネージャーがあります。一部の従業員は、給与に加えてコミッションを獲得しています。従業員関連の情報はすべて **emp** テーブルに保存されます。

サンプル企業は地域的に多様であるため、データベースは部門の場所を追跡します。各企業の従業員は部門に割り当てられます。各部門は、一意の部門番号と短い名前で識別されます。各部門は 1 つの場所に関連付けられます。部門関連のすべての情報は **dept** テーブルに保存されます。

同社は、従業員が就いている職務に関する情報も追跡しています。一部の従業員は会社に長く勤務し、さまざまな役職を歴任したり、昇給したり、部署を異動したりしてきました。従業員のステータスに変更が発生すると、会社は以前のポジションの終了日を記録します。開始日、新しい役職、部門、給与、およびステータス変更の理由を含む新しいジョブレコードが追加されます。すべての従業員履歴は **jobhist** テーブルに維持されます。

次は、サンプルデータベーステーブルのエンティティ関係図です。

`edb-sample.sql` スクリプトは次のとおりです。

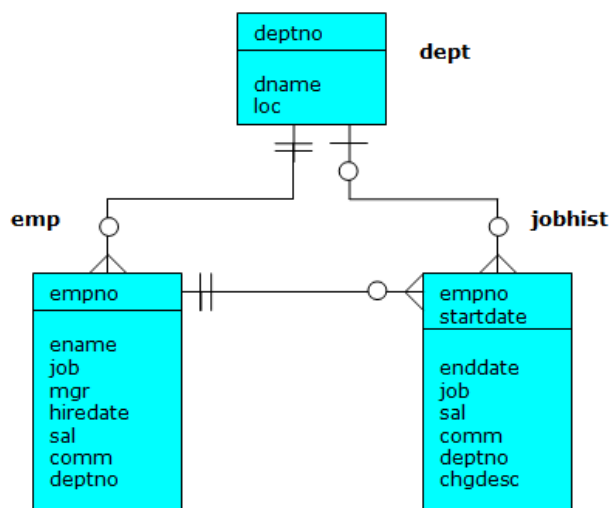


図 1: Sample Database Tables

```

--
-- Script that creates the sample tables, views, procedures,
-- functions, triggers, etc.
--
-- Start new transaction - commit all or nothing
--
BEGIN;
/
--
-- Create and load tables used in the documentation examples.
--
-- Create the dept table
--
CREATE TABLE dept (
    deptno      NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname       VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc         VARCHAR2(13)
);
--
-- Create the emp table
--
CREATE TABLE emp (
    empno       NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename       VARCHAR2(10),
    job         VARCHAR2(9),
    mgr         NUMBER(4),
    hiredate    DATE,

```

(次のページに続く)

(前のページからの続き)

```

    sal          NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm         NUMBER(7,2),
    deptno       NUMBER(2) CONSTRAINT emp_ref_dept_fk
                REFERENCES dept(deptno)
);
--
-- Create the jobhist table
--
CREATE TABLE jobhist (
    empno        NUMBER(4) NOT NULL,
    startdate    DATE NOT NULL,
    enddate      DATE,
    job          VARCHAR2(9),
    sal          NUMBER(7,2),
    comm         NUMBER(7,2),
    deptno       NUMBER(2),
    chgdesc      VARCHAR2(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
    CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
        REFERENCES emp(empno) ON DELETE CASCADE,
    CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
        REFERENCES dept (deptno) ON DELETE SET NULL,
    CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
);
--
-- Create the salesemp view
--
CREATE OR REPLACE VIEW salesemp AS
    SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = SALESMAN;
--
-- Sequence to generate values for function new_empno.
--
CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
--
-- Issue PUBLIC grants
--
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
GRANT ALL ON next_empno TO PUBLIC;
--
-- Load the dept table
--
INSERT INTO dept VALUES (10,ACCOUNTING,NEW YORK);

```

(次のページに続く)

(前のページからの続き)

```

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;

```

(次のページに続く)

(前のページからの続き)

```

ANALYZE emp;
ANALYZE jobhist;

--
-- Procedure that lists all employees numbers and names
-- from the emp table using a cursor.
--
CREATE OR REPLACE PROCEDURE list_emp
IS
    v_empno      NUMBER(4);
    v_ename      VARCHAR2(10);
    CURSOR emp_cur IS
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE(EMPNO      ENAME);
    DBMS_OUTPUT.PUT_LINE(-----);
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno ||      || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;
/
--
-- Procedure that selects an employee row given the employee
-- number and displays certain columns.
--
CREATE OR REPLACE PROCEDURE select_emp (
    p_empno      IN NUMBER
)
IS
    v_ename      emp.ename%TYPE;
    v_hiredate    emp.hiredate%TYPE;
    v_sal         emp.sal%TYPE;
    v_comm        emp.comm%TYPE;
    v_dname       dept.dname%TYPE;
    v_disp_date   VARCHAR2(10);
BEGIN
    SELECT ename, hiredate, sal, NVL(comm, 0), dname
        INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
        FROM emp e, dept d
        WHERE empno = p_empno
        AND e.deptno = d.deptno;
    v_disp_date := TO_CHAR(v_hiredate, MM/DD/YYYY);

```

(次のページに続く)

(前のページからの続き)

```

DBMS_OUTPUT.PUT_LINE(Number      : || p_empno);
DBMS_OUTPUT.PUT_LINE(Name        : || v_ename);
DBMS_OUTPUT.PUT_LINE(Hire Date   : || v_disp_date);
DBMS_OUTPUT.PUT_LINE(Salary      : || v_sal);
DBMS_OUTPUT.PUT_LINE(Commission: || v_comm);
DBMS_OUTPUT.PUT_LINE(Department: || v_dname);

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE(Employee || p_empno || not found);
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(The following is SQLERRM:);
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
    DBMS_OUTPUT.PUT_LINE(The following is SQLCODE:);
    DBMS_OUTPUT.PUT_LINE(SQLCODE);

END;
/
--
-- Procedure that queries the emp table based on
-- department number and employee number or name. Returns
-- employee number and name as IN OUT parameters and job,
-- hire date, and salary as OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query (
  p_deptno      IN      NUMBER,
  p_empno       IN OUT  NUMBER,
  p_ename       IN OUT  VARCHAR2,
  p_job         OUT    VARCHAR2,
  p_hiredate    OUT    DATE,
  p_sal         OUT    NUMBER
)
IS
BEGIN
  SELECT empno, ename, job, hiredate, sal
    INTO p_empno, p_ename, p_job, p_hiredate, p_sal
   FROM emp
  WHERE deptno = p_deptno
     AND (empno = p_empno
        OR ename = UPPER(p_ename));

END;
/
--
-- Procedure to call emp_query_caller with IN and IN OUT
-- parameters. Displays the results received from IN OUT and
-- OUT parameters.
--

```

(次のページに続く)

(前のページからの続き)

```

CREATE OR REPLACE PROCEDURE emp_query_caller
IS
    v_deptno      NUMBER(2);
    v_empno       NUMBER(4);
    v_ename       VARCHAR2(10);
    v_job         VARCHAR2(9);
    v_hiredate    DATE;
    v_sal         NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 0;
    v_ename  := Martin;
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE(Department : || v_deptno);
    DBMS_OUTPUT.PUT_LINE(Employee No: || v_empno);
    DBMS_OUTPUT.PUT_LINE(Name       : || v_ename);
    DBMS_OUTPUT.PUT_LINE(Job        : || v_job);
    DBMS_OUTPUT.PUT_LINE(Hire Date  : || v_hiredate);
    DBMS_OUTPUT.PUT_LINE(Salary     : || v_sal);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE(More than one employee was selected);
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE(No employees were selected);
END;
/
--
--  Function to compute yearly compensation based on semimonthly
--  salary.
--
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal      NUMBER,
    p_comm     NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
/
--
--  Function that gets the next number from sequence, next_empno,
--  and ensures it is not already in use as an employee number.
--
CREATE OR REPLACE FUNCTION new_empno RETURN NUMBER
IS

```

(次のページに続く)

(前のページからの続き)

```

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

```

(次のページに続く)

(前のページからの続き)

```

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

```

(次のページに続く)

```

        RAISE INFO The following is SQLERRM;;
        RAISE INFO %, SQLERRM;
        RAISE INFO The following is SQLSTATE;;
        RAISE INFO %, SQLSTATE;
        RETURN -1;
END;
$$ LANGUAGE plpgsql;
/
--
-- Rule to INSERT into view salesemp
--
CREATE OR REPLACE RULE salesemp_i AS ON INSERT TO salesemp
DO INSTEAD
    INSERT INTO emp VALUES (NEW.empno, NEW.ename, SALESMAN, 7698,
        NEW.hiredate, NEW.sal, NEW.comm, 30);
--
-- Rule to UPDATE view salesemp
--
CREATE OR REPLACE RULE salesemp_u AS ON UPDATE TO salesemp
DO INSTEAD
    UPDATE emp SET empno      = NEW.empno,
                  ename       = NEW.ename,
                  hiredate    = NEW.hiredate,
                  sal         = NEW.sal,
                  comm        = NEW.comm
    WHERE empno = OLD.empno;
--
-- Rule to DELETE from view salesemp
--
CREATE OR REPLACE RULE salesemp_d AS ON DELETE TO salesemp
DO INSTEAD
    DELETE FROM emp WHERE empno = OLD.empno;
--
-- After statement-level trigger that displays a message after
-- an insert, update, or deletion to the emp table. One message
-- per SQL command is displayed.
--
CREATE OR REPLACE TRIGGER user_audit_trig
AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action          VARCHAR2(24);
BEGIN
    IF INSERTING THEN
        v_action := added employee(s) on ;
    ELSIF UPDATING THEN

```

(前のページからの続き)

```

        v_action := updated employee(s) on ;
    ELSIF DELETING THEN
        v_action := deleted employee(s) on ;
    END IF;
    DBMS_OUTPUT.PUT_LINE(User || USER || v_action ||
TO_CHAR(SYSDATE,YYYY-MM-DD));
END;
/
--
-- Before row-level trigger that displays employee number and
-- salary of an employee that is about to be added, updated,
-- or deleted in the emp table.
--
CREATE OR REPLACE TRIGGER emp_sal_trig
    BEFORE DELETE OR INSERT OR UPDATE ON emp
    FOR EACH ROW
DECLARE
    sal_diff      NUMBER;
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE(Inserting employee || :NEW.empno);
        DBMS_OUTPUT.PUT_LINE(..New salary: || :NEW.sal);
    END IF;
    IF UPDATING THEN
        sal_diff := :NEW.sal - :OLD.sal;
        DBMS_OUTPUT.PUT_LINE(Updating employee || :OLD.empno);
        DBMS_OUTPUT.PUT_LINE(..Old salary: || :OLD.sal);
        DBMS_OUTPUT.PUT_LINE(..New salary: || :NEW.sal);
        DBMS_OUTPUT.PUT_LINE(..Raise      : || sal_diff);
    END IF;
    IF DELETING THEN
        DBMS_OUTPUT.PUT_LINE(Deleting employee || :OLD.empno);
        DBMS_OUTPUT.PUT_LINE(..Old salary: || :OLD.sal);
    END IF;
END;
/
--
-- Package specification for the emp_admin package.
--
CREATE OR REPLACE PACKAGE emp_admin
IS
    FUNCTION get_dept_name (
        p_deptno      NUMBER
    ) RETURN VARCHAR2;
    FUNCTION update_emp_sal (

```

(次のページに続く)

(前のページからの続き)

```

        p_empno          NUMBER,
        p_raise          NUMBER
    ) RETURN NUMBER;
PROCEDURE hire_emp (
    p_empno          NUMBER,
    p_ename          VARCHAR2,
    p_job            VARCHAR2,
    p_sal            NUMBER,
    p_hiredate        DATE,
    p_comm           NUMBER,
    p_mgr            NUMBER,
    p_deptno         NUMBER
);
PROCEDURE fire_emp (
    p_empno          NUMBER
);
END emp_admin;
/
--
-- Package body for the emp_admin package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
    --
    -- Function that queries the dept table based on the department
    -- number and returns the corresponding department name.
    --
    FUNCTION get_dept_name (
        p_deptno      IN NUMBER
    ) RETURN VARCHAR2
    IS
        v_dname       VARCHAR2(14);
    BEGIN
        SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
        RETURN v_dname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE(Invalid department number || p_deptno);
            RETURN ;
    END;
    --
    -- Function that updates an 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.

```

(次のページに続く)

(前のページからの続き)

```

--
FUNCTION update_emp_sal (
    p_empno      IN NUMBER,
    p_raise      IN NUMBER
) RETURN NUMBER
IS
    v_sal        NUMBER := 0;
BEGIN
    SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
    v_sal := v_sal + p_raise;
    UPDATE emp SET sal = v_sal WHERE empno = p_empno;
    RETURN v_sal;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE(Employee || p_empno || not found);
        RETURN -1;
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE(The following is SQLERRM:);
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE(The following is SQLCODE:);
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        RETURN -1;
END;
--
-- Procedure that inserts a new employee record into the emp table.
--
PROCEDURE hire_emp (
    p_empno      NUMBER,
    p_ename      VARCHAR2,
    p_job        VARCHAR2,
    p_sal        NUMBER,
    p_hiredate    DATE,
    p_comm       NUMBER,
    p_mgr        NUMBER,
    p_deptno     NUMBER
)
AS
BEGIN
    INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
        VALUES(p_empno, p_ename, p_job, p_sal,
            p_hiredate, p_comm, p_mgr, p_deptno);
END;
--
-- Procedure that deletes an employee record from the emp table based
-- on the employee number.

```

(次のページに続く)

(前のページからの続き)

```
--
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

テーブル名、すべての列名とその型を指定して、新しいテーブルを作成します。次は、テーブルを定義するために必要な最小限の情報を含む `emp` サンプルテーブルの単純化バージョンです。

```
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)
);
```

このコードは、改行を使用して PSQL に入力できます。PSQL は、コマンドがセミコロンまで終了しないことを認識します。

SQL コマンドでは、空白スペース、タブ、および改行を使用できます。必要に応じてコマンドを配置したり、すべてのテキストを 1 行に配置したりできます。

2つのダッシュ「`--`」はコメントを紹介します。これらに続くものは、行の最後まで無視されます。さらに、SQL、大文字と小文字を保持するために識別子が二重引用符で囲まれている場合を除き、キーワードと識別子について大文字と小文字を区別しません。

`VARCHAR2(10)` は、長さ 10 文字までの任意の文字列を格納できるデータ型を指定します。`NUMBER(7,2)` は、精度 7、スケール 2 の固定小数点数です。`NUMBER(4)` は、精度 4、スケール 0 の整数です。

EDB Postgres Advanced Server は、通常の SQL データ型 `INTEGER`、`SMALLINT`、`NUMBER`、`REAL`、`DOUBLE PRECISION`、`CHAR`、`VARCHAR2`、`DATE`、および `TIMESTAMP` およびこれらのタイプのさまざまな同義語をサポートしています。

テーブルが不要な場合、または再作成する場合は、次のコマンドを使用して削除できます。

```
DROP TABLE tablename;
```

2.3.3 Populating a table With rows

INSERT ステートメントを使用して、テーブルに行を設定します。

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

すべてのデータ型は、明らかな入力形式を使用します。単純な数値ではない定数は、例のように一重引用符' で囲まれます。DATE タイプは、受け入れるものにおいて柔軟です。ただし、このチュートリアルでは、ここに示している明確なフォーマットを使用します。

これまでに使用した構文では、列の順序を覚えておく必要があります。別の構文では、列を明示的にリストできます。

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

一部の列を省略または省略する場合、たとえば、手数料が不明な場合、別の順序で列をリストできます。

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

多くの開発者は、列を明示的にリストするほうが、暗黙的に順序に依存するよりも良いスタイルだと考えています。

2.3.4 Querying a table

テーブルからデータを取得するには、SQL SELECT ステートメントを使用して **クエリ**を実行します。ステートメントは次のように分かれています。

- 選択リスト (返す列をリストする部分)
- テーブルリストデータを取得するテーブルをリストした部分
- オプショナル修飾制限を指定する部分

次のクエリーは、テーブル内のすべての従業員のすべての列を特定の順序でリストします。

```
SELECT * FROM emp;
```

ここで、選択リストの “*” はすべての列を意味します。以下は、このクエリからの出力です。

```
__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
```

(次のページに続く)

(前のページからの続き)

```

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

```

選択リストには任意の式を指定できます。例、次のことができます。

```

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)

```

AS 句を使用して出力列のラベルを変更する方法に注意してください。AS 句はオプション。

必要な行を指定する WHERE 句を追加することにより、クエリーを修飾できます。WHERE 句には、ブール真値式が含まれます。ブール式が true である行のみが返されます。修飾では、通常のブール演算子 AND、OR、および NOT を使用できます。たとえば、次のコマンドは、部門 20 の給与が \$1000.00 を超える従業員を取得します。

```

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

```

(次のページに続く)

(前のページからの続き)

```

-----+-----+-----
JONES | 2975.00 |    20
SCOTT | 3000.00 |    20
ADAMS | 1100.00 |    20
FORD  | 3000.00 |    20
(4 rows)

```

クエリの結果をソートされた順序で返すように要求できます。

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

```
__OUTPUT__
```

```

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

```

クエリの結果から重複行を削除するように要求できます。

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

```
__OUTPUT__
```

```

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

```

2.3.5 Executing joins between tables

クエリは、一度に複数のテーブルにアクセスしたり、テーブルの複数の行が同時に処理されるように同じテーブルにアクセスしたりできます。同じまたは異なるテーブルの複数の行に一度にアクセスするクエリは、*join* クエリと呼ばれます。

結合クエリの実行

たとえば、すべての従業員レコードを、関連する部門の名前と場所とともにリストしたいとします。そのためには、**emp** テーブルの各行の **deptno** 列と、**dept** テーブルのすべての行の **deptno** 列を比較する必要があります。次に、これらの値が一致する行のペアを選択します。これは、次のクエリを使用して実行できます。

```
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
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)

結果セットに関する 2 つのことを観察します。

- 部門 40 の結果行はありません。これは、部門 40 の **emp** テーブルに一致するエントリがないため、結合は **dept** テーブルの不一致行を無視するためです。次のコードは、これを修正する方法を示しています。
- ***を使用するか、次のように修飾を省略するのではなく、テーブル名で修飾された出力列をリストすることがより望ましい。

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

すべての列の名前が異なるため修飾する必要がある **deptno** を除き、パーサーはそれらが属しているテーブルを見つけました。ただし、結合クエリでは列名を完全修飾することをお勧めします。

この種類の結合クエリは、次の代替フォームで書くこともできます。

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

この構文はあまり一般的に使用されるものではありませんが、次のトピックを理解するのに役立つようにここで示しています。

外部結合の実行

上記のすべての結合結果では、部門 40 に属する従業員は返されませんでした。その結果、部門 40 のレコードは表示されません。次に、一致する従業員がないという事実にもかかわらず、結果に部門 40 レコードを取得する方法を解決します。クエリは dept テーブルをスキャンし、行ごとに、一致する emp 行を見つける必要があります。一致する行が見つからない場合は、emp テーブルの列をいくつかの「空の」値に置き換えます。この種類のクエリは**アウタージョイン**と呼ばれます。これまでに見てきた結合は**内部結合**です。コマンドは次のようなものです。

```
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
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)

このクエリは**左外部結合**と呼ばれます。結合演算子の左側に記載されているテーブルには、出力に各行が少なくとも 1 回あります。右側のテーブルには、左側のテーブルの一部の行と一致する行出力のみが含まれます。右テーブルに一致するものがない左テーブルの行が選択されると、空の NULL 値が右テーブルの列に置き換えられます。

外部結合の代替構文は、WHERE 句の結合条件で外部結合演算子“(+)”を使用することです。外部結合演算子は、一致しない行を null 値に置き換えるテーブルの列名の後に配置されます。したがって、emp テーブルに一致する行がない dept テーブルのすべての行では、EDB Postgres Advanced Server は emp の列を含む選択リスト式に対して null を返します。したがって、以前の例を次のように書き換えることができます。

```
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)

自己結合の実行

テーブルを自分自身に対して結合することもできます。これは、**自己結合**と呼ばれます。例として、各従業員の名前とその従業員のマネージャーの名前を見つけたいとします。各 emp 行の mgr 列を、他のすべての emp 行の empno 列と比較する必要があります。

```
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
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
```

(次のページに続く)

(前のページからの続き)

SMITH works **for** FORD
(13 rows)

ここでは、`emp` テーブルが `e1` にラベル付け変更されて、選択リストと結合条件の従業員行を表します。また、選択リストと結合条件でマネージャーとして機能する一致する従業員行を表すために、`e2` としてもラベル付けされました。他のクエリでこれらの種類のエイリアスを使用して、入力の一部節約できます。例

```
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)

このスタイルの略語はよく使用されます。

2.3.6 Aggregating data

他のほとんどのリレーショナルデータベース製品と同様に、EDB Postgres Advanced Server は集計機能をサポートしています。集計関数は、複数の入力行から単一の結果を計算します。たとえば、一部の集計は、行セットにわたって COUNT、SUM、AVG 平均、MAX 最大、および MIN 最小を計算します。

例として、次のクエリーで最高給与と最低給与を見つけることができます。

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

__OUTPUT__

highest_salary	lowest_salary
5000.00	800.00

(1 row)

最も高い給与の従業員を見つけたい場合は、次のことを試したくなるかもしれません。

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

WHERE 句では集計関数 MAX を使用できないため、このアプローチは機能しません。この制限は、WHERE 句が集計ステージに入る行を決定するために存在します。したがって、集計関数が計算される前に評価する必要があります。ただし、サブクエリを使用することにより、クエリをリスタートして、意図した結果を達成できます。サブクエリーは、外部クエリーとは別に独自の結果を取得する独立した計算です。

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

集計は、GROUP BY 句と組み合わせて使用する場合にも役立ちます。たとえば、次のクエリーは、各部門で最高の給与を取得します。

```
SELECT deptno, MAX(sal) FROM emp GROUP BY deptno;
__OUTPUT__
deptno |  max
-----+-----
    10 | 5000.00
    20 | 3000.00
    30 | 2850.00
(3 rows)
```

このクエリーは、部門ごとに1つの出力行を生成します。各集計結果は、その部門に一致する行で計算されます。これらのグループ化された行は、HAVING 句を使用してフィルタリングできます。

```
SELECT deptno, MAX(sal) FROM emp GROUP BY deptno HAVING AVG(sal) > 2000;
__OUTPUT__
deptno |  max
-----+-----
    10 | 5000.00
    20 | 3000.00
(2 rows)
```

このクエリーは、平均給与が 2000 を超える部門にのみ同じ結果を示します。

最後に、次のクエリーは、各部門のアナリストである最も高給の従業員のみを考慮します。

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

(次のページに続く)

(前のページからの続き)

```

-----+-----
      20 | 3000.00
(1 row)

```

WHERE 句と HAVING 句には微妙な違いがあります。WHERE 句は、グループ化が発生し集計関数が適用される前に行をフィルタリングします。HAVING 句は、行がグループ化され、グループごとに集計関数が計算された後、結果にフィルタを適用します。

したがって、前の例では、アナリストである従業員のみが検討されます。このサブセットから、従業員は部門ごとにグループ化されます。グループ内のアナリストの平均給与が 2000 を超えるグループのみが最終結果に含まれます。これは、部門 20 のグループにのみ当てはまり、部門 20 のアナリストの最大給与は 3000.00 です。

2.3.7 Updating a table

UPDATE コマンドを使用して、既存の行の列値を変更できます。たとえば、次のコマンドシーケンスは、マネージャーである全員に 10% の昇給を与える前と後の結果を示しています。

```
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

DELETE コマンドを使用して、テーブルから行を削除できます。たとえば、次のコマンドシーケンスは、20 部門のすべての従業員を削除する前と後の結果を示しています。

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

```
__OUTPUT__
```

(次のページに続く)

(前のページからの続き)

ename	deptno
SMITH	20
ALLEN	30
WARD	30
JONES	20
MARTIN	30
BLAKE	30
CLARK	10
SCOTT	20
KING	10
TURNER	30
ADAMS	20
JAMES	30
FORD	20
MILLER	10

(14 rows)

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

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

__OUTPUT__

ename	deptno
ALLEN	30
WARD	30
MARTIN	30
BLAKE	30
CLARK	10
KING	10
TURNER	30
JAMES	30
MILLER	10

(9 rows)

警告

次のような `WHERE` 句を指定せずに `DELETE` コマンドを指定するときは注意してください。

```
DELETE FROM tablename;
```

このステートメントは、指定されたテーブルからすべての行を削除し、完全に空のままにします。システムは、これを行う前に確認を要求しません。!!!

2.3.9 The SQL language

EDB Postgres Advanced Server は、Oracle 構文と互換性のある SQL 言語および拡張機能の構文とコマンドをサポートしています。拡張機能は、Oracle のデータベース互換性を提供せず、Oracle スタイルのアプリケーションをサポートしません。

SQL reference は、以下に関する詳細情報を提供します。

- Compatible SQL syntax and language elements
- Data types
- Functions and operators for the built-in data types

2.4 Advanced concepts

高度な SQL 機能により、管理が簡素化され、データの損失または破損を防ぐことができます。

2.4.1 Views

ビューは、アプリケーションの進化に応じて変更される可能性のあるテーブルの構造の詳細をカプセル化する一貫したインターフェイスを提供します。ビューを最大限に活用することは、優れた SQL データベースデザインの重要な側面です。

次の SELECT コマンドを検討します。

```
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)

これが繰り返し使用する照会である場合、毎回 SELECT コマンド全体を再入力せずにこの照会を再利用する簡素な方法は、**ビュー**を作成することです。

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

ビュー名 `employee_pay` を通常のテーブル名のように使用して、クエリーを実行できるようになりました。

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

(14 rows)

ビューは、実際のテーブルを使用するほぼすべての場所で使用できます。他のビューに基づいてビューをビルドすることも一般的です。

2.4.2 Foreign keys

外部キーは、外部キーテーブルに格納されるデータを制御するために2つのデータベーステーブルのデータ間のリンクを確立および適用するために使用される1つ以上の列を表します。

すべての従業員が有効な部門に属していることを確認したい、つまり、データの**参照整合性**を維持したいとします。単純なデータベースシステムでは、最初に `dept` テーブルを参照して一致するレコードが存在するかどうかを確認し、新しい従業員レコードを挿入または拒否することにより、参照整合性を保証できます。

このアプローチには多くの問題があり、不便です。EDB Postgres Advanced Server、それが簡単になります。

Creating a new table で提供された `emp` テーブルの変更バージョンをここに示し、外部キー制約を追加します。変更した `emp` テーブルは次のようになります。

```
CREATE TABLE emp (
    empno          NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename          VARCHAR2(10),
    job            VARCHAR2(9),
    mgr            NUMBER(4),
```

(次のページに続く)

(前のページからの続き)

```

hiredate      DATE,
sal           NUMBER(7,2),
comm          NUMBER(7,2),
deptno        NUMBER(2) CONSTRAINT emp_ref_dept_fk
              REFERENCES dept(deptno)
);

```

サンプル emp テーブルで次の INSERT コマンドを発行しようとする、外部キー制約 emp_ref_dept_fk は、部門 50 が dept テーブルに存在することを保証します。そうでないため、コマンドは拒否されます。

```
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".

```

アプリケーションの外部キーの動作を微調整できます。外部キーを正しく使用すると、データベースアプリケーションの品質が向上します。それらについてもっと学ぶことを強くお勧めします。

2.4.3 The ROWNUM pseudo-column

ROWNUM は、クエリから行が取得された順序に基づいて、行ごとにインクリメンタルな一意の整数値が割り当てられる擬似列です。したがって、取得された最初の行には 1 の ROWNUM があり、2 番目の行には 2 の ROWNUM などがあります。

返される行の数を制限する

この機能を使用して、クエリによって取得される行数を制限できます。

```
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)

```

ROWNUM 値は、結果セットの並べ替えが発生する前に各行に割り当てられます。したがって、結果セットは ORDER BY 句で指定された順序で返されます。ただし、ROWNUM 値は昇順でない場合があります。

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

__OUTPUT__

```
rownum | empno | ename |  job
```

(次のページに続く)

(前のページからの続き)

```

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

```

テーブル内の行にシーケンス番号を追加する

次の例は、jobhist テーブルのすべての行にシーケンス番号を追加する方法を示しています。最初に seqno という名前の列をテーブルに追加します。次に、UPDATE コマンドで seqno を ROWNUM に設定します。

```

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

```

次の SELECT コマンドは、新しい seqno 値を示します。

```

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
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

シノニムは、SQL ステートメントで別のデータベースオブジェクトを参照するために使用できる識別子です。シノニムは、データベースオブジェクトが通常、SQL ステートメントで適切に参照されるようにスキーマ名による完全修飾が必要な場合に役立ちます。そのオブジェクトに定義されたシノニムは、単一の非修飾名前への参照を簡素化します。

EDB Postgres Advanced Server は、以下の同義語をサポートしています。

- テーブル
- ビュー
- マテリアライズドビュー
- シークエンス
- パッケージ
- プロシージャ
- ファンクション
- 種類
- データベースリンクを介してアクセスできるオブジェクト
- その他の同義語

シノニムを作成するときに、参照されるスキーマも参照されるオブジェクトも存在する必要はありません。シノニムは、存在しないオブジェクトまたはスキーマを参照することができます。参照されたオブジェクトまたはスキーマを削除すると、シノニムが無効になります。シノニムを削除するには、明示的にドロップする必要があります。

他のスキーマオブジェクトと同様に、EDB Postgres Advanced Server は検索パスを使用して非修飾のシノニム名を解決します。同じ名前の2つのシノニムがある場合、シノニムへの非修飾参照は、検索パス内の指定された名前の最初のシノニムに解決されます。`public` が検索パスにある場合、その名前を修飾せずにそのスキーマのシノニムを参照できます。

EDB Postgres Advanced Server が SQL コマンドを実行すると、現在のユーザーの特権がシノニムの基になるデータベースオブジェクトに対してチェックされます。ユーザーがそのオブジェクトに対する適切な権限を持っていない場合、SQL コマンドは失敗します。

シノニムの作成

CREATE SYNONYM コマンドを使用して、シノニムを作成します。構文は次のとおりです。

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

パラメーター

`syn_name`

`syn_name` は、シノニムの名前です。シノニム名はスキーマ内で一意である必要があります。

`schema`

`schema` は、シノニムが存在するスキーマの名前を指定します。スキーマ名を指定しない場合、シノニムは検索パスの最初の既存のスキーマに作成されます。

`object_name`

`object_name` は、オブジェクトの名前を指定します。

`object_schema`

`object_schema` は、オブジェクトが存在するスキーマの名前を指定します。

`dblink_name`

`dblink_name` は、ターゲットオブジェクトにアクセスできるデータベースリンクの名前を指定します。

`REPLACE` 句を含めて、既存の同義語定義を新しい同義語定義に置き換えます。

`PUBLIC` 句を含めて、`public` スキーマにシノニムを作成します。Oracle データベースと互換性があり、`CREATE PUBLIC SYNONYM` コマンドは、`public` スキーマにあるシノニムを作成します。

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

これは、次の速記方法です。

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

この例では、`enterprisedb.emp` テーブルを参照する `personnel` という名前のシノニムを作成します。

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

シノニムが `CREATE SYNONYM` コマンドでスキーマ修飾されていない限り、検索パスの最初の既存のスキーマに作成されます。次のコマンドを実行して、検索パスを表示できます。

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

例

この例では、development という名前のスキーマが存在しない場合、シノニムは accounting という名前のスキーマに作成されます。

これで、シノニム personnel を使用して、任意の SQL ステートメント DDL または DML で enterprisedb スキーマの emp テーブルを参照できます。

```
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)

シノニムの削除

シノニムを削除するには、DROP SYNONYM コマンドを使用します。構文は次のとおりです。

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

パラメーター

syn_name

syn_name は、シノニムの名前です。シノニム名はスキーマ内で一意である必要があります。

schema

schema は、シノニムが存在するスキーマの名前を指定します。

スキーマ修飾できる他のオブジェクトと同様に、検索パスに同じ名前の2つのシノニムを含めることができます。ドロップするシノニムの名前を明確にするには、スキーマ名を含めます。シノニムが DROP SYNONYM

コマンドでスキーマ修飾されていない限り、EDB Postgres Advanced Server は、検索パスで見つけたシノニムの最初のインスタンスを削除します。

オプションで PUBLIC 句を含めて、public スキーマにあるシノニムを削除できます。DROP PUBLIC SYNONYM コマンドは、Oracle データベースと互換性があり、public スキーマにあるシノニムをドロップします。

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

例

次の例では、シノニム personnel を削除します。

```
DROP SYNONYM personnel;
```

2.4.5 Hierarchical queries

階層クエリは、親子関係を形成するデータに基づいて、結果セットの行を階層順序で返すクエリのタイプです。階層は通常、逆さまのツリー構造で表されます。ツリーは、相互接続されたノードで構成されています。各ノードは、なし、1つ、または複数の子ノードに接続できます。各ノードは、親を持たないトップノードを除き、1つの親ノードに接続されます。このノードは root ノードです。各ツリーには、ルートノードが1つだけあります。

子を持たないノードはリーフノードと呼ばれます。ツリーには常に少なくとも1つのリーフノードがあります。たとえば、ツリーが単一のノードで構成される自明な場合。この場合、ノードはルートでありリーフでもあります。

階層クエリでは、結果セットの行は1つ以上のツリーのノードを表します。

注釈

単一の特定の行が複数のツリーに表示されるため、結果セットに複数回表示されます。

構文

クエリーの階層関係は CONNECT BY 句で記述され、結果セットで行が返される順序の基礎を形成します。この例は、SELECT コマンドに CONNECT BY 句とその関連オプション句が表示される場所のコンテキストを示しています。

```
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` は、結果セットのフィールドを構成する 1 つ以上の式です。`table_expression` は、結果セットの行の生成元の 1 つ以上のテーブルまたはビューです。`other` は、追加の有効な SELECT コマンド句です。

CONNECT BY は、階層の親レコードとその子レコードの関係を指定します。

- PRIOR キーワードは、CONNECT BY 句の条件内の式を 1 つだけ指定できます。階層クエリーで親行を参照するには、右または左の式で指定する必要があります。例

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

- CONNECT BY condition が複合の場合、1 つの条件のみが PRIOR キーワードを必要としますが、複数の PRIOR 条件が存在する場合があります。
- PRIOR キーワードを使用して、階層クエリーの親子関係にある親行を評価できます。
- PRIOR キーワードは単項式では使用できません。
- PRIOR キーワードを使用して、ターゲットリストの式に対応するレコードごとに擬似列を追加できます。擬似列は、PRIOR 式を使用して子クエリーで参照されるとき、親レコードの実際の値を表します。

CONNECT_BY_ROOT オペレーターを使用して、`CONNECT BY *PRIOR* の機能をさらに強化します。

例

この例では、CONNECT BY 句と PRIOR を使用して、従業員とマネージャー間の関係を定義します。LEVEL 値を使用して、従業員名がインデントされて、各行の階層の深さがさらに強調されます。

```
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;
```

このクエリからの出力は次のとおりです。

2.4.6 Hierarchical queries

Defining the parent/child relationship

特定の行の子を決定するには

1. 指定された行で `parent_expr` を評価します。
2. `table_expression` の評価の結果である他の行で `child_expr` を評価します。
3. `parent_expr = child_expr` の場合、この行は指定された親行の子ノードです。
4. `table_expression` の残りのすべての行に対してプロセスを繰り返します。手順 3 の方程式を満たすすべての行は、指定された親行の子ノードです。

i 注釈

行が子ノードであるかどうかを判断する評価プロセスは、*WHERE* 句が ``table_expression`` に適用される前に、*table_expression* によって返されるすべての行で発生します。

このプロセスを繰り返し、以前の手順で見つかった各子ノードを親として扱い、ノードの逆ツリーを構成します。子ノードの最終セットに自分自身の子がない場合、プロセスは完了します。これらはリーフノードです。

CONNECT BY 句を含む *SELECT* コマンドには、通常、*START WITH* 句が含まれます。*START WITH* 句は、ルートノードになる行、つまりアルゴリズムを適用する初期親ノードである行を決定します。

Selecting the root nodes

START WITH 句は、ルートノードとして使用する *table_expression* によって選択された行を決定します。*start_expression* が `true` と評価する *table_expression* によって選択されたすべての行が、ツリーのルートノードになります。したがって、結果セット内のポテンシャルツリーの数は一時的にルートノードの数と等しくなります。その結果、*START WITH* 句が省略されると、*table_expression* によって返されるすべての行は、独自のツリーのルートになります。

Organization tree in the sample application

組織ツリーの例を確認するには、サンプルアプリケーションの *emp* テーブルを検討します。*emp* テーブルの行は、従業員のマネージャーの従業員番号が含まれる *mgr* 列に基づいて階層を形成します。各従業員には最大で 1 人のマネージャーがいます。*KING* は会社の社長です。彼にはマネージャーがいないため、*KING* の *mgr* 列は `null` です。また、従業員が複数の従業員のマネージャーとして行動することも可能です。この関係は、一般的なツリー構造の階層組織図を形成します。

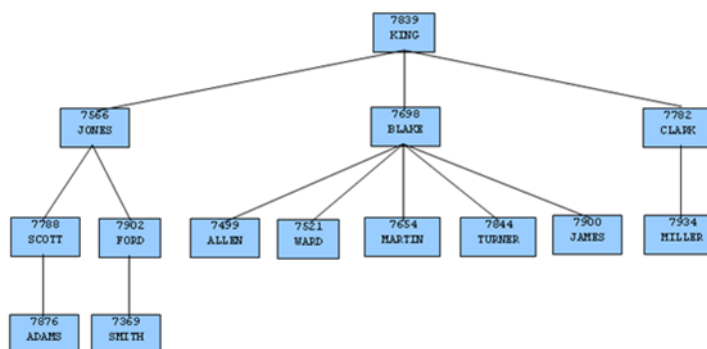


図 2: Employee Organization Hierarchy

この関係に基づいて階層クエリーを形成するために、*SELECT* コマンドに *CONNECT BY PRIOR empno = mgr* 句が含まれます。たとえば、社長が *KING*、従業員番号が 7839 であると、*mgr* 列が 7839 である従業員は *KING* に直接報告します。この関係は、*KING* の子ノードである *JONES*、*BLAKE*、および *CLARK* に当てはまります。同様に、従業員 *JONES* の場合、7566 と等しい *mgr* 列を持つ他の従業員は *JONES* の子ノードです。これらのノードは、例では *SCOTT* および *FORD* です。

組織図のトップは KING であるため、このツリーには 1 つのルートノードがあります。START WITH mgr IS NULL 句は、初期ルートノードとして KING のみを選択します。

完全な SELECT コマンドは次のとおりです。

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

クエリ出力の行は、ルートからリーフまで各ブランチを上から下、左から右の順序で移動します。

```
__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 は、SELECT コマンドで列を表示できる場所で使用できる擬似列です。LEVEL は、結果セットの各行に対して、この行が表すノードの階層の深さを指定するゼロ以外の整数値を返します。ルートノードの LEVEL は 1 です。ルートノードの直接の子の LEVEL は 2 などです。

例

次のクエリーは、LEVEL 擬似列を追加した以前のクエリーの変更です。さらに、LEVEL 値を使用して、従業員名がインデントされて、各行の階層の深さがさらに強調されます。

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

このクエリーからの出力は次のとおりです。

```

__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)

```

共通の親を共有し、同じレベルにあるノードは **兄弟**と呼ばれます。たとえば、上記の出力では、従業員 ALLEN、WARD、MARTIN、TURNER、および JAMES は、すべて親が BLAKE であるレベル 3 であるため、兄弟です。JONES、BLAKE、および CLARK は、レベル 2 であり、KING が共通の親であるため、兄弟です。

Ordering the siblings

ORDER SIBLINGS BY 句を使用して、選択した列値に基づいて兄弟が昇順または降順で表示されるように、結果セットを並べ替えることができます。これは、階層クエリーでのみ使用できる ORDER BY 句の特殊なケースです。

ORDER SIBLINGS BY ename ASC を追加することにより、前述のクエリーをさらに変更できます。

```

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;

```

以前のクエリからの出力が変更され、兄弟が名前の昇順で表示されます。兄弟 BLAKE、CLARK、および JONES が KING の下にアルファベット順に配置されるようになりました。兄弟 sALLEN、JAMES、MARTIN、TURNER、および WARD は、BLAKE の下にアルファベット順に配置されます。以下、同様に。

```

__OUTPUT__
level | employee | empno | mgr
-----+-----+-----+-----
1 | KING      | 7839 |
2 | BLAKE     | 7698 | 7839
3 | ALLEN     | 7499 | 7698

```

(次のページに続く)

(前のページからの続き)

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)

この最後の例は、WHERE 句を追加し、3つのルートノードで開始します。ノードツリーが構築された後、WHERE 句はツリー内の行をフィルタリングして結果セットを形成します。

```
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;
```

クエリーからの出力には、3つのルートノードレベル 1 BLAKE、CLARK、および JONES が表示されます。さらに、WHERE 句を満たさない行は出力から削除されました。

```
__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 は、列を修飾して、現在の行に対してルートノードと考えられる行の列の値を返すために使用できる単項演算子です。

注釈

*単項演算子*は、単一のオペランドを操作します。CONNECT_BY_ROOT の場合、CONNECT_BY_ROOT キーワードに続く列名です。

構文

SELECT リストの consql では、CONNECT_BY_ROOT 演算子は次のとおりです。

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

CONNECT_BY_ROOT 演算子に関する注意事項

- SELECT コマンドが階層照会用である限り、SELECT リスト、WHERE 句、GROUP BY 句、HAVING 句、ORDER BY 句、および ORDER SIBLINGS BY 句で CONNECT_BY_ROOT 演算子を使用できます。
- 階層クエリーの CONNECT BY 句または START WITH 句では、CONNECT_BY_ROOT 演算子は使用できません。
- CONNECT_BY_ROOT は、列を含む式に適用できます。これを行うには、式をカッコで囲む必要があります。

例

次のクエリーは、CONNECT_BY_ROOT 演算子を使用して、従業員 BLAKE、CLARK、および JONES で始まるツリーに基づいて、結果セットにリストされる各従業員のルートノードの従業員番号と従業員名を返すことを示しています。

```
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;
```

クエリーからの出力は、列 mgr empno および mgr ename のすべてのルートノードが、START WITH 句にリストされている従業員-BLAKE、CLARK、または JONES のいずれかであることを示しています。

```
__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
```

(次のページに続く)

(前のページからの続き)

```

 2 | SCOTT | 7788 | 7566 | 7566 | JONES
 3 | ADAMS | 7876 | 7788 | 7566 | JONES
(13 rows)

```

同様のクエリーですが、mgr 列が null である単一のトップレベルの従業員で始まるツリーを 1 つだけ生成します。

```

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;

```

次の出力では、列 mgr empno および mgr ename のすべてのルートノードが、このクエリーのルートとして KING を示しています。

```

__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)

```

対照的に、この例では START WITH 句を省略するため、14 本のツリーになります。

```

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;

```

リーフノードでさえ独自のツリーの上部を形成するため、クエリーからの出力では、各ノードは mgr empno

および mgr_ename 列の下の子ノードとして少なくとも 1 回表示されます。

__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
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)

CONNECT_BY_ROOT 演算子は、かっこで囲まれていない式に適用される場合、その直後の用語 ename にのみ

影響を与えます。後続の `|| ' manages ' ||` `ename` の連結は、`CONNECT_BY_ROOT` オペレーションの一部ではありません。したがって、`ename` の 2 回目の出現は現在処理されている行の値を結果ますが、`ename` の最初の出現はルートノードからの値を結果します。

```
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;
```

クエリーからの出力の `top mgr/employee` 列の下で生成された値に注意してください。

```
__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
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)
```

この例では、かっこで囲まれた式で `CONNECT_BY_ROOT` 演算子を使用します。

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

`ename` と `empno` の値は両方とも `CONNECT_BY_ROOT` 演算子の影響を受けます。その結果、それらは `top mgr/empno` 列の下でのルートノードから値を結果します。

```
__OUTPUT__
level | employee | empno | mgr | top mgr/empno
-----+-----+-----+-----+-----
```

(次のページに続く)

(前のページからの続き)

```

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 は、階層クエリーで動作し、現在のノードとルートノードの間で発生する指定された列の列値を取得するファンクションです。

構文

ファンクションのシグネチャは次のとおりです。

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

ファンクションは2つの引数を取ります。

- **column** は、ファンクションを呼び出している階層クエリーで指定されたテーブルにある列の名前です。
- **delimiter** は、指定された列の各エントリを分離する **varchar** 値です。

例

この例では、従業員名とそのマネージャーのリストを結果ます。マネージャーにマネージャーがいる場合、その名前が結果に追加されます。

```

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

```

(次のページに続く)

(前のページからの続き)

```

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)

```

結果セット内

- level 列には、クエリーが返したレベル数が表示されます。
- ename 列には、従業員名が表示されます。
- managers 列には、マネージャーの階層リストが含まれています。

SYS_CONNECT_BY_PATH の EDB Postgres Advanced Server 実装は、次の使用をサポートしていません。

- CONNECT_BY_PATH 内の SYS_CONNECT_BY_PATH
- SYS_CONNECT_BY_PATH 内 SYS_CONNECT_BY_PATH

2.4.7 Multidimensional analysis

多次元分析は、ディメンションのさまざまな組み合わせを使用してデータを検査するプロセスを指します。ディメンションは、時間、地域、会社の部門、製品ラインなどのデータを分類するために使用されるカテゴリです。このプロセスは、データウェアハウジングアプリケーションで一般的に使用されます。特定のディメンションセットに関連付けられた結果は、**ファクト**と呼ばれます。ファクトは通常、製品の売上、利益、数量、数などに関連する数値です。

リレーショナルデータベースシステムのディメンションセットに従ってこれらのファクトを取得するには、通常は **SQL 集計** を使用します。SQL 集計とは、基本的にデータが特定の基準ディメンションに従ってグループ化され、結果セットが各グループ内のデータの数、合計、平均などのファクトの集計で構成されることを意味します。

結果の集計

SQL SELECT コマンドの GROUP BY 句は、集計結果を生成するプロセスを簡素化する次の拡張機能をサポートしています。

- ROLLUP 拡張機能
- CUBE 拡張機能

- GROUPING SETS 拡張機能

さらに、SELECT リストまたは HAVING 句の GROUPING ファンクションと GROUPING_ID ファンクションを使用して、これらの拡張機能を使用する際の結果の解釈を支援できます。

注釈

サンプルの `dept` および `emp` テーブルは、このディスカッションで広範囲に使用され、使用例を提供します。より有益な結果を提供するために、次の変更がこれらのテーブルに適用されました。

```
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);
```

emp および dept テーブルの結合からの次の行が使用されます。

2.4.8 Multidimensional analysis

便利な拡張機能

ROLLUP および CUBE 拡張機能は、結果セットに追加レベルの小計を提供することにより、ベース集計行に追加します。

GROUPING SETS 拡張機能を使用すると、さまざまな種類のグループ化を単一の結果セットに結合できます。

GROUPING および GROUPING_ID ファンクションは、結果セットの解釈に役立ちます。

ROLLUP extension

ROLLUP 拡張機能は、各階層グループの小計と総計を含むグループの階層セットを生成します。階層の順序は、ROLLUP 式リストで指定された式の順序によって決まります。階層の最上位は、リストの最も左側の項目です。右に進む後続の各項目は、階層を下に移動します。一番右の項目が最下位レベルにあります。

構文

単一の ROLLUP の構文は次のとおりです。

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

各 expr は、結果セットのグループ化を決定する式です。(expr_1a, expr_1b, ...) として括弧で囲まれた場合、expr_1a および expr_1b によって返される値の組み合わせが、階層の単一のグループレベルを定義します。

結果セットで返される集計の基本レベルは、式リストによって返される値の一意の組み合わせごとにあります。

さらに、一意の値ごとに、リストの最初の項目 expr_1 または (expr_1a, expr_1b, ...) の組み合わせ、指定された方の小計が返されます。小計は、リストの 2 番目の項目 expr_2 または (expr_2a, expr_2b,

...) の組み合わせ、指定された方に対して、最初の項目の各グループの一意の値ごとに返されます。以下同様。最後に、結果セット全体の総計が返されます。

小計行の場合、小計が取得される項目については null が返されます。

GROUP BY 句のコンテキストで指定された ROLLUP 拡張機能は、次のとおりです。

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

select_list で指定された項目は、次のいずれかである必要があります。

- ROLLUP expression_list にも登場します
- COUNT、SUM、AVG、MIN、MAX などの集計ファンクション
- 戻り値がグループ内の個々の行から独立している定数またはファンクションたとえば SYSDATE ファンクション

GROUP BY 句を使用して、複数の ROLLUP 拡張機能、および他の GROUP BY 拡張機能と個々の式の複数のオカレンスを指定します。

ORDER BY 句を使用して、出力を階層またはその他の意味のある構造で表示します。この句を使用すると、結果セットの順序が保証されます。

グループ化レベルまたは合計の数は $n+1$ です。ここで、 n は ROLLUP 式リストの項目数を表します。括弧で囲まれたリストは 1 つの項目としてカウントされます。

例

次のクエリーは、列 loc、dname、および job の階層に基づいてロールアップを生成します。

```
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;
```

クエリーの結果には、loc、dname、および job の一意の組み合わせごとの従業員の数と、loc の一意の値ごとに loc および dname の一意の組み合わせごとの小計が表示されます。総計は最後の行に表示されます。

__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

(次のページに続く)

(前のページからの続き)

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)			

次のクエリーは、ROLLUP リストの項目を括弧内で組み合わせた効果を示しています。

```
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;
```

出力には、前の例のように、loc および dname の組み合わせの小計は含まれません。

__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)			

ROLLUP リストの最初の 2 列がかっこで囲まれている場合、小計レベルも同様に異なります。

```

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;

```

ここで、一意の loc と dname の組み合わせごとに小計がありますが、loc の一意の値の小計はありません。

```

__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

CUBE 拡張機能は ROLLUP 拡張機能に似ています。ただし、ROLLUP は、ROLLUP 式リストの項目の左から右のリストに基づいて、階層内のグループ化と結果を生成します。CUBE は、CUBE 式リスト内のすべての項目のすべての順列に基づいて、グループ化と小計を生成します。したがって、結果セットには、同じ式リストで実行された ROLLUP よりも多くの行が含まれます。

構文

単一の CUBE の構文は次のとおりです。

```

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

```

各 expr は、結果セットのグループ化を決定する式です。(expr_1a, expr_1b, ...) として括弧で囲まれた場合、expr_1a および expr_1b によって返される値の組み合わせが単一のグループを定義します。

結果セットで返される集計の基本レベルは、式リストによって返される値の一意の組み合わせごとにあり

ます。

さらに、一意の値ごとに、リストの最初の項目 `expr_1` または (`expr_1a`, `expr_1b`, ...) の組み合わせ、指定された方の小計が返されます。一意の値ごとに、リストの 2 番目の項目 `expr_2` または (`expr_2a`, `expr_2b`, ...) の組み合わせ、指定された方の小計が返されます。小計も、最初の項目と 2 番目の項目の一意の組み合わせごとに返されます。同様に、3 番目の項目がある場合、次の小計が返されます。

- 3 番目の項目の各一意の値
- 3 番目の項目と最初の項目の組み合わせの各一意の値
- 3 番目の項目と 2 番目の項目の組み合わせの各一意の値
- 3 番目の項目、2 番目の項目、および最初の項目の組み合わせの各一意の値

最後に、結果セット全体の総計が返されます。

小計行の場合、小計が取得される項目については `null` が返されます。

`GROUP BY` 句のコンテキストで指定された `CUBE` 拡張機能は、次のとおりです。

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

`select_list` で指定された項目は、次のいずれかである必要があります。

- `CUBE expression_list` にも登場します
- `COUNT`、`SUM`、`AVG`、`MIN`、または `MAX` などの集計ファンクション
- 戻り値がグループ内の個々の行から独立している定数またはファンクションたとえば `SYSDATE` ファンクション

`GROUP BY` 句を使用して、複数の `CUBE` 拡張機能、および他の `GROUP BY` 拡張機能と個々の式の複数のオカレンスを指定します。

`ORDER BY` 句を使用して、出力を意味のある構造で表示し、結果セットの順序を保証します。

グループ化レベルまたは合計の数は、2 の n 乗です。ここで、 n は `CUBE` 式リストの項目数を表します。括弧で囲まれたリストは 1 つの項目としてカウントされます。

例

次のクエリーは、列 `loc`、`dname`、および `job` の順列に基づいてキューブを生成します。

```
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;
```

以下はクエリーの結果です。 `loc`、`dname`、および `job` の組み合わせごとの従業員数の数と、次の小計が表示されます。

- `loc`、`dname` の各組み合わせ

- loc、job の各組み合わせ
- dname、job の各組み合わせ
- loc の各一意の値
- dname の各一意の値
- job の各一意の値

総計は最後の行に表示されます。

__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

(次のページに続く)

(前のページからの続き)

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

(50 rows)

次のクエリーは、CUBE リストの項目を括弧内で組み合わせた効果を示しています。

```
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;
```

出力には、loc と dname の組み合わせ、loc と job の組み合わせ、dname 自分自身、または job 自分自身。

__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

(次のページに続く)

(前のページからの続き)

	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)

次のクエリーは、最初の式が CUBE 拡張機能の外部で指定される別のバリエーションを示しています。

```
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;
```

この出力では、loc の各グループの dname および job の順列が実行されます。

__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

(次のページに続く)

(前のページからの続き)

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

GROUP BY 句で GROUPING SETS 拡張機能を使用すると、さまざまなグループ化に基づいた複数の結果セットの連結である 1 つの結果セットを生成する手段が提供されます。つまり、UNION ALL 操作は、複数のグループの結果セットを 1 つの結果セットに結合して実行されます。

UNION ALL オペレーション、つまり GROUPING SETS 拡張機能は、結合される結果セットから重複行を削除しません。

構文

単一の GROUPING SETS 拡張機能の構文は次のとおりです。

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

GROUPING SETS 拡張機能には、1 つ以上のコンマ区切りの式、かっこで囲まれた式のリスト、ROLLUP 拡張機能、および CUBE 拡張機能の任意の組み合わせを含めることができます。

GROUPING SETS 拡張子は、GROUP BY 句のコンテキストで指定されます。

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

select_list で指定された項目は、次のいずれかである必要があります。

- GROUPING SETS expression_list に登場
- COUNT, SUM, AVG, MIN, MAX などの集計ファンクション
- 戻り値がグループ内の個々の行から独立している定数またはファンクションたとえば SYSDATE ファンクション

GROUP BY 句を使用して、複数の GROUPING SETS 拡張機能、および他の GROUP BY 拡張機能と個々の式の複数のオカレンスを指定します。

ORDER BY 句を使用して、出力を意味のある構造で表示し、結果セットの順序を保証します。

例

次のクエリーは、列 loc、dname、および job で指定されたグループのユニオンを生成します。

```
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;
```

結果は次のとおりです。

```
__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)
```

これは、UNION ALL 演算子を使用する次のクエリーと同等です。

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

UNION ALL クエリーからの出力は、GROUPING SETS 出力と同じです。

__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)			

この例は、さまざまなタイプの GROUP BY 拡張機能を GROUPING SETS 式リストと一緒に使用できる方法を示しています。

```
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;
```

以下は、このクエリからの出力です。

__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

(次のページに続く)

(前のページからの続き)

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)

出力は、基本的に GROUP BY loc、GROUP BY ROLLUP (dname, job)、および GROUP BY CUBE (job, loc) の結果セットの連結です。これらの個々のクエリーは次のとおりです。

```
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;
```

GROUP BY loc 句の結果セットは次のとおりです。

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

次のクエリーは GROUP BY ROLLUP (dname, job) 句を使用します。

```
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;
```

GROUP BY ROLLUP (dname, job) 句の結果セットは次のとおりです。

```
__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)
```

次のクエリーは GROUP BY CUBE (job, loc) 句を使用します。

```
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;
```

GROUP BY CUBE (job, loc) 句の結果セットは次のとおりです。

```
__OUTPUT__
loc | dname | job | employees
-----+-----+-----+-----
BOSTON | | ANALYST | 3
BOSTON | | CLERK | 3
BOSTON | | MANAGER | 2
BOSTON | | | 8
CHICAGO | | CLERK | 1
```

(次のページに続く)

(前のページからの続き)

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)

前の3つのクエリーが UNION ALL 演算子と組み合わせると、3つの結果セットの連結が生成されます。

```

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;

```

出力は、GROUP BY GROUPING SETS (loc, ROLLUP (dname, job), CUBE (job, loc)) 句を使用した場合と同じです。

__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

(次のページに続く)

(前のページからの続き)

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

(38 rows)

GROUPING function

GROUP BY 句の ROLLUP、CUBE、または GROUPING SETS 拡張機能を使用する場合、拡張機能によって生成されたさまざまなレベルの小計と結果セットの基本集計行を区別することが難しい場合があります。GROUPING ファンクションは、この区別を行う方法を提供します。

構文

GROUPING ファンクションを使用するための一般的な構文は次のとおりです。

```
SELECT [ <expr> ...,] GROUPING( <col_expr> ) [ , <expr> ] ...
FROM ...
GROUP BY [...,]
```

(次のページに続く)

(前のページからの続き)

```
{ ROLLUP | CUBE | GROUPING SETS }( [...,] <col_expr>
[, ...] ) [, ...]
```

GROUPING ファンクションは、GROUP BY 句の ROLLUP、CUBE、または GROUPING SETS 拡張の式リストで指定されたディメンション列の式である単一のパラメーターを受け取ります。

GROUPING ファンクションの戻り値は 0 または 1 です。クエリーの結果セットでは

- 行がその列の複数の値の小計を表すため、GROUPING ファンクションで指定された列式が null の場合、GROUPING ファンクションは値 1 を返します。
- GROUPING ファンクションで指定された列の特定の値に基づいて行が結果を返す場合、GROUPING ファンクションは値 0 を返します。

後者の場合、列は null 以外の場合も同様に null になる場合があります。いずれの場合も、それはその列の特定の値に関するものであり、複数の値の小計ではありません。

例

次のクエリーは、GROUPING ファンクションの戻り値が小計行にどのように対応するかを示しています。

```
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;
```

GROUPING ファンクションの出力を表示する 3 つの右端の列では、対応する列の値にわたって小計が取得される場所に、小計行に 1 の値が表示されます。

__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

(次のページに続く)

(前のページからの続き)

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)

これらのインジケータは、特定の小計のスクリーニング基準として使用できます。たとえば、前述のクエリーを使用すると、HAVING 句で GROUPING ファンクションを使用することにより、loc および dname の組み合わせの小計のみを表示できます。

```
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;
```

このクエリーは次の結果を生成します。

__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)

GROUPING ファンクションを使用して、小計行を基本集計行または特定の小計行と区別できます。これらの小計行には、式の基になる列がテーブル内の 1 つ以上の行で null であるため、null を返す式リストに 1 つの項目があります。これは、列の小計を表すこととは反対です。

この点を説明するために、次の行を emp テーブルに追加します。これは、job 列に NULL 値を含む行を提供します。

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

次のクエリーは、明確にするために削減された行数を使用して発行されます。

```

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;

```

出力には、loc 列に BOSTON を含み、job 列テーブルの 4 番目および 5 番目のエントリにスペースを含む 2 つの行が含まれます。

```

__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)

```

5 行目は、job 列の GROUPING ファンクション (gf_job) が 1 を返す 5 行目で、これがすべてのジョブの小計であることを示しています。行の employees 列には、小計値 9 が含まれています。

4 行目では、job 列と loc 列の GROUPING ファンクションが 0 を返します。これは、これがすべての行の基本集計であり、loc が BOSTON であり、job が null であることを示します。これがこのために挿入された行です例。employees 列には、挿入されたこのような単一の行のカウントである 1 が含まれています。

9 行目最後から次の行では、job 列の GROUPING ファンクションは 0 を返しますが、loc 列の GROUPING ファンクションは 1 を返します。これは、これが job 列が null であるすべての場所の小計であることを示しています。この例で挿入された単一行の数です。

GROUPING_ID function

GROUPING_ID ファンクションは、GROUPING ファンクションの簡素化を提供し、ROLLBACK、CUBE、または GROUPING SETS 拡張機能からの結果セットの行の小計レベルを決定します。

GROUPING ファンクションは、列式を 1 つだけ取り、行が指定された列のすべての値の小計であるかどうかを示します。したがって、複数のグループ化列を含むクエリーの小計のレベルを解釈するには、複数の GROUPING ファンクションが必要になる場合があります。

GROUPING_ID ファンクションは、ROLLBACK、CUBE、または GROUPING SETS 拡張機能で使用された 1 つ以上の列式を受け入れます。これらの列のどの列で小計が集計されたかを判断するために使用できる単一の整数を返します。

構文

GROUPING_ID ファンクションの一般的な構文は次のとおりです。

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

GROUPING_ID ファンクションは、1 つ以上のパラメーターを取ります。これらは、GROUP BY 句の ROLLUP、CUBE、または GROUPING SETS 拡張の式リストで指定されたディメンション列の式である必要があります。

GROUPING_ID ファンクションは整数値を結果ます。この値は、GROUPING_ID ファンクションで指定されたパラメーターの順序と同じ左から右の順序で指定された一連の GROUPING ファンクションによって返される、連結された 1 と 0 で構成されるビットベクトルの base-10 解釈に対応します。

例

次のクエリーは、列 gid で表される GROUPING_ID ファンクションの戻り値が、列 loc および dname の 2 つの GROUPING ファンクションによって返される値にどのように対応するかを示しています。

```
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;
```

次の出力では、gid で指定された整数と比較した gf_loc 値と gf_dname 値で構成されるビットベクトルの関係に注意してください。

__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

(次のページに続く)

(前のページからの続き)

	SALES	6	1	0	2
		17	1	1	3
(12 rows)					

次の表は、出力の 4 行の GROUPING ファンクションの戻り値に基づいた GROUPING_ID ファンクション計算の具体例を提供します。

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

次の表は、GROUPING_ID ファンクションの戻り値が、集計が発生するグループ化列にどのように対応するかをまとめたものです。

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

dname によるこれらの小計のみを表示するには、GROUPING_ID ファンクションに基づいた HAVING 句で次の単純化されたクエリーを使用します。

```
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;
```

クエリーの結果は次のとおりです。

__OUTPUT__					
loc	dname	employees	gf_loc	gf_dname	gid
	ACCOUNTING	3	1	0	2

(次のページに続く)

(前のページからの続き)

OPERATIONS	3	1	0	2
RESEARCH	5	1	0	2
SALES	6	1	0	2
(4 rows)				

2.5 Sample database description

ドキュメントの例では、サンプルテーブル `dept`、`emp`、および `jobhist` を使用します。これらは EDB Postgres Advanced Server のインストール時に作成およびロードされます。

2.5.1 使用可能なスクリプト

次のスクリプトを実行することにより、サンプルデータベースのテーブルとプログラムがいつでも再作成できます。

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

<xx> は、EDB Postgres Advanced Server のバージョン番号です。

さらに、同じディレクトリのスクリプトには、Oracle データベースと互換性のある構文を使用して作成されたデータベースオブジェクトが含まれています。このスクリプトファイルは `edb-sample.sql` です。スクリプト

- 現在接続しているデータベースにサンプルテーブルとプログラムを作成します。
- テーブルに対するすべての権限を `PUBLIC` グループに付与。

テーブルとプログラムは、現在のユーザーがテーブルとプロシージャを作成する権限がある検索パスの最初のスキーマに作成されます。次のコマンドを使用して検索パスを表示できます。

```
SHOW SEARCH_PATH;
```

PSQL PostgreSQL のターミナルベースのインターフェイスコマンドを使用して、検索パスを変更できます。

2.5.2 サンプルデータベースについて

サンプルデータベースは、組織内の従業員を表します。これには、従業員、部門、および従業員の履歴レコードの 3 種類のレコードが含まれています。

各従業員には、識別番号、名前、雇用日、給与、およびマネージャーがあります。一部の従業員は、給与に加えてコミッションを獲得しています。従業員関連の情報はすべて `emp` テーブルに保存されます。

サンプル企業は地域的に多様であるため、部門の場所を追跡します。各企業の従業員は部門に割り当てられます。各部門は、一意の部門番号と短い名前で識別されます。各部門は 1 つの場所に関連付けられます。部門関連のすべての情報は `dept` テーブルに保存されます。

同社は、従業員が就いている職務に関する情報も追跡しています。一部の従業員は会社に長く勤務し、さまざまな役職を歴任したり、昇給したり、部署を異動したりしてきました。従業員のステータスに変更が

発生すると、会社は以前のポジションの終了日を記録します。開始日、新しい役職、部門、給与、およびステータス変更の理由を含む新しいジョブレコードが追加されます。すべての従業員履歴は jobhist テーブルに維持されます。

以下は pg-sample.sql スクリプトです。

```
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,
    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),
```

(次のページに続く)

```

    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;
--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);

```

(次のページに続く)

(前のページからの続き)

```

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;
--
-- 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;

```

(次のページに続く)

(前のページからの続き)

```

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;
    RAISE INFO Name        : %, v_ename;
    RAISE INFO Hire Date   : %, v_disp_date;
    RAISE INFO Salary      : %, v_sal;
    RAISE INFO Commission: %, v_comm;

```

(次のページに続く)

(前のページからの続き)

```

    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));

```

(次のページに続く)

(前のページからの続き)

```

END;
$$ LANGUAGE plpgsql;
--
--  Function to call emp_query_caller with IN and INOUT
--  parameters. Displays the results received from INOUT and
--  OUT parameters.
--
CREATE OR REPLACE FUNCTION emp_query_caller() RETURNS VOID
AS $$
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;

```

(次のページに続く)

(前のページからの続き)

```

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;
    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;

```

(次のページに続く)

(前のページからの続き)

```

    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;
$$ LANGUAGE plpgsql;
--
--  Function that adds a new salesman to table emp.
--
CREATE OR REPLACE FUNCTION hire_salesman (
    p_ename      VARCHAR,
    p_sal        NUMERIC,
    p_comm       NUMERIC
) RETURNS NUMERIC
AS $$
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;

```

(次のページに続く)

(前のページからの続き)

```

    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;
$$ LANGUAGE plpgsql;
--
-- Rule to INSERT into view salesemp
--
CREATE OR REPLACE RULE salesemp_i AS ON INSERT TO salesemp
DO INSTEAD
    INSERT INTO emp VALUES (NEW.empno, NEW.ename, SALESMAN, 7698,
        NEW.hiredate, NEW.sal, NEW.comm, 30);
--
-- Rule to UPDATE view salesemp
--
CREATE OR REPLACE RULE salesemp_u AS ON UPDATE TO salesemp
DO INSTEAD
    UPDATE emp SET empno      = NEW.empno,
                  ename       = NEW.ename,
                  hiredate    = NEW.hiredate,
                  sal         = NEW.sal,
                  comm        = NEW.comm
    WHERE empno = OLD.empno;
--
-- Rule to DELETE from view salesemp
--
CREATE OR REPLACE RULE salesemp_d AS ON DELETE TO salesemp
DO INSTEAD
    DELETE FROM emp WHERE empno = OLD.empno;
--
-- After statement-level trigger that displays a message after
-- an insert, update, or deletion to the emp table. One message
-- per SQL command is displayed.
--
CREATE OR REPLACE FUNCTION user_audit_trig() RETURNS TRIGGER
AS $$
DECLARE
    v_action          VARCHAR(24);

```

(次のページに続く)

```

    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;
    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;

```

(前のページからの続き)

```
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;
```


第3章 Supported platforms

EDB Postgres Advanced Server は、Linux および Windows プラットフォームへのインストールをサポートしています。

EDB Postgres Advanced Server のプラットフォームサポートについては、EDB ウェブサイトの [Platform Compatibility page](#) または Installing EDB Postgres Advanced Server の EDB Postgres Advanced Server のプラットフォームサポートを参照してください。

第4章 EDB Postgres Advanced Server release notes

EDB Postgres Advanced Server 18 は、コアデータベースのパフォーマンスと効率の向上に焦点を当てたさまざまな機能強化を導入し、リレーショナル機能と最新のセキュリティ統合を拡張します。

EDB Postgres Advanced Server ドキュメントでは、マイナーリリースとパッチを含む EDB Postgres Advanced Server 18 の最新バージョンについて説明しています。これらのリリースノートは、各リリースの新機能に関する情報を提供します。

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 コンポーネント認証

EDB Postgres Advanced Server でサポートされている EDB コンポーネントと拡張機能は、すべて EPAS 18 で検証および認定されています。EDB Postgres Advanced Server で使用可能な拡張機能の完全なリストについては、Postgres extensions available by deployment を参照してください。

4.2 サポートのお知らせ

4.2.1 バックアップ&リカバリツール BART の非互換性

EDB バックアップおよびリカバリツール BART は、EDB Postgres Advanced Server または PostgreSQL バージョン 14 以降ではサポートされていません。バックアップリカバリツールとしては Barman または PgBackRest に移動することを強くお勧めします。

EDB Postgres Advanced Server 18.1.0 release notes

リリース日 2025 年 11 月 25 日

EDB Postgres Advanced Server 18.1.0 には、次の機能強化とバグ修正が含まれています。

Type	Description	Addresses
Upstream merge	コミュニティ PostgreSQL 18.1 とマージされました。詳細については、 PostgreSQL 18.1 Release Notes を参照してください。	
Feature	<i>INACTIVE_ACCOUNT_TIME</i> オプションをパスワードプロファイルに追加しました。新しい自動非アクティブアカウントロック機能は、休眠ユーザーアカウントからのリスクを軽減することにより、データベースのセキュリティを強化します。これにより、セキュリティ管理者は、事前に定義された日数ログインしていないアカウントを自動的にロックするプロファイルたとえば <i>INACTIVE_ACCOUNT_TIME 60</i> を構成できます。これにより、古いアカウントの潜在的な悪用を防ぎ、コンプライアンスを合理化し、手動追跡を必要とせずにデータベースの攻撃面を削減します。	
Feature	サーバーログにパスワードを含むステートメントを編集する機能を追加しました。信頼性の低いパスワードリダクションに対処するために、私たちは生の文字列分析ではなく解析後コマンドツリーを使用する新しいアプローチを検討しています。この方法は、 <i>edb_log_redact_password_statements</i> のような GUC を使用して、すべてのパスワード変更コマンドたとえば <i>ALTER USER</i> と確実に一致します。これにより、悪意のあるコードが編集をバイパスするのを防ぎますが、コマンド文字列全体を編集する必要があるため、同時に実行される他のコマンドが隠される場合があります。	
Feature	<i>postgres_fdw</i> ユーザーマッピングにパスワードを難読化する機能を追加しました。 <i>PG_USER_MAPPING</i> に保存されているパスワードを難読化することにより、 <i>postgres_fdw</i> のセキュリティを強化しました。この変更により、パスワードは他のスーパーユーザーからでもカタログ内の平文で保存されなくなり、セキュリティ実装を既存の OCI dblink 標準と調整します。	

次のページに続く

表 2 – 前のページからの続き

Type	Description	Addresses nbsp;;
Feature	Oracle 同等の `NLSSORT()` のサポートを追加しました。 Oracle の互換性を強化するために、`NLSSORT()` ファンクション変換を実装しました。これにより、システムは Oracle の NLS パラメーターを正しく解釈し、PostgreSQL `Collate-Clause` に変換できます。この変更により、アプリケーションはロケールルールに基づいて照合対応の並べ替えを正しく実行できるようになり、移行の忠実度が大幅に向上します。	
Enhancement	Secure(TLS) および Authenticated(AUTH) メールのサポートを `UTL_SMTP` に追加しました。 Oracle から移行するデータベース管理者は、`AUTH` および `STARTTLS` の新しい `UTL_SMTP` サポートを使用して、安全な電子メール通信を保証できるようになりました。この実装では、標準の SMTP 認証が許可され、電子メールセッションの TLS / SSL 暗号化が有効になり、アプリケーションがセキュリティポリシーとコンプライアンスを維持するのに役立ちます。	
Enhancement	ネストされたステートメントを監査するサポートを edb_audit に追加しました。ネストされたステートメント監査ログを実装することにより、データベースのセキュリティが強化されました。新しい構成パラメーター `edb_audit_nested_statements` により、管理者はファンクションとプロシージャ内で実行されたステートメントをキャプチャできます。この機能強化により、完全な監査証跡が保証され、悪意のあるユーザーが隠しコードの実行を介してセキュリティログをバイパスするのを効果的に防止します。	
Enhancement	システムカタログテーブルの `edb_audit_group` 関連オプションを有効にしました。システムカタログテーブルの `edb_audit_group` リレーションオプションを有効にすることにより、セキュリティが強化されました。これにより、ユーザーは重要なシステムオブジェクトにきめの細かい監査ルールを適用できます機密テーブルの `SELECT` ステートメントの監視など。カタログテーブルでこのオプションを有効にすると、ユーザーは重要なシステムオブジェクトの非常に具体的な追跡ルールを定義できます。	

次のページに続く

表 2 – 前のページからの続き

Type	Description	Addresses sp;
Enhancement	Auditb_tran_5 コマンドのサポートを追加しました。`edb_audit`を強化して、`COPY`ステートメントオプションの`edb_audit_statement`へのロギングの明示的なサポートを含めました。この重要な更新により、ユーザーは機密テーブルへの`COPY`コマンドを監査できます。	
Enhancement	`DBMS_SCHEDULER`のカレンダーベースのスケジューリングが改善されました。新しいカレンダーベースのスケジューリング機能で`DBMS_SCHEDULER`を大幅に強化し、精度と柔軟性を向上させました。これには、`FREQ=SECONDLY`間隔サブ分ジョブの場合、`INTERVAL`キーワード、および負の`BYMONTHDAY`値月末に基づいてジョブをスケジュールするためのサポートが含まれます。	
Enhancement	Oracle 動的 SQL メソッド 4 のインジケータ配列のサポートを ecpg に追加しました。大量操作と Oracle ダイナミック SQL メソッド 4 のインジケータ配列をサポートするように、ECPG および ECPGPlus を強化しました。このアップグレードにより、開発者はデータ値の配列とともにインジケータ値の並列配列を使用することにより、`NULL`値を効率的に処理できます。これにより、埋め込み SQL を使用するアプリケーションでのデータバインディングと転送の柔軟性とパフォーマンスが大幅に向上します。	
Enhancement	非スクロールカーソル`FETCH`の累積フェッチ行数を ecpg に追加しました。ECPG の PROC モードを更新して、スクロール不可能なカーソルでの`FETCH`操作の`sqlca->sqlerrd[2]`の累積行数を維持します。これにより、その動作が Oracle の Pro *C と調整され、取得のたびにカウントがリセットされます。この機能強化により、処理される合計行に依存するアプリケーションの移行の互換性が向上します。	

次のページに続く

表 2 – 前のページからの続き

Type	Description	Addresses sp;
Enhancement	`ALTER [PUBLIC] DATABASE LINK` 構文のサポートを追加しました。データベース管理者は、サービスを中断せずに既存のデータベースリンクの詳細を変更する安全な方法を必要としていました。メンテナンスタスクを簡素化するために `ALTER [PUBLIC] DATABASE LINK` コマンドを導入しました。この新しいコマンドにより、インプレース更新が可能になり、複雑な依存オブジェクトを持つデータベースリンクを削除して再作成する必要がなくなり、安定性と操作の安全性が向上します。	
Enhancement	<code>pg_dump</code> が `SYS` スキーマオブジェクトの ACL 変更をダンプする機能を追加しました。`SYS` スキーマ内のシステムオブジェクトとユーザー定義オブジェクトの両方の ACL 変更を確実にキャプチャするように <code>pg_dump</code> を強化しました。システムオブジェクトが固定されるようになったため、この変更により、重要なシステムオブジェクトに設定されたカスタム権限がダンプおよびリストア操作中に完全に保持されます。	
Change	<code>pg_authid</code> の EDB Postgres Advanced Server 固有のブール列を整数ビットマスクに置き換えました。カタログ管理を簡素化し、将来の拡張機能をサポートするために、<code>pg_authid</code> のいくつかの EDB Postgres Advanced Server 固有のブール特権列が単一の整数ビットマスク列 <i>rolredwoodprivs</i> に統合されました。この機能強化により、列のスプロールが削減され、拡張性が向上しますが、新しい SQL ファンクションたとえば <code>has_privname_privilege()</code> により、ユーザー権限を簡単にチェックできます。	
Bug fix	列名の大文字と小文字の問題を修正して、リモートサーバーから <code>`resultset`</code> を正しく取得します。リモートクエリーを構築するときに外部テーブル定義で <code>`column_name`</code> オプションを利用するように <code>edb_dblink_oci</code> を強化しました。これにより、ユーザーはローカル列名をリモートテーブルの別の列名にマップし、潜在的な名前付けと大文字小文字の競合に対処できます。この変更により、正確なデータの取得が保証され、PostgreSQL と Oracle 互換環境間の相互運用性が向上します。	