



Database Compatibility for Oracle Developers Guide

リリース *14.1.0*

EnterpriseDB

2021 年 12 月 12 日

目次

第 1 章	Configuration Parameters Compatible with Oracle Databases	3
1.1	edb_redwood_date	4
1.2	edb_redwood_raw_names	4
1.3	edb_redwood_strings	5
1.4	edb_stmt_level_tx	7
1.5	oracle_home	8
第 2 章	SQL Tutorial	9
2.1	Sample Database	9
2.1.1	Sample Database Installation	9
2.1.2	Sample Database Description	10
2.2	Creating a New Table	23
2.3	Populating a Table With Rows	23
2.4	Querying a Table	24
2.5	Joins Between Tables	26
2.6	Aggregate Functions	30
2.7	Updates	31
2.8	Deletions	32
2.9	The SQL Language	33
第 3 章	Advanced Concepts	35
3.1	Views	35
3.2	Foreign Keys	36
3.3	The ROWNUM Pseudo-Column	37
3.4	Synonyms	39
3.5	Hierarchical Queries	42
3.5.1	Defining the Parent/Child Relationship	46
3.5.2	Selecting the Root Nodes	46
3.5.3	Organization Tree in the Sample Application	47
3.5.4	Node Level	48
3.5.5	Ordering the Siblings	49
3.5.6	Retrieving the Root Node with CONNECT_BY_ROOT	50
3.5.7	Retrieving a Path with SYS_CONNECT_BY_PATH	55
3.6	Multidimensional Analysis	56
3.6.1	ROLLUP Extension	58
3.6.2	CUBE Extension	61
3.6.3	GROUPING SETS Extension	65
3.6.4	GROUPING Function	71
3.6.5	GROUPING_ID Function	74

第 4 章	Profile Management	77
4.1	Creating a New Profile	78
4.1.1	Creating a Password Function	81
4.2	Altering a Profile	83
4.3	Dropping a Profile	84
4.4	Associating a Profile with an Existing Role	85
4.5	Unlocking a Locked Account	87
4.6	Creating a New Role Associated with a Profile	88
4.7	Backing up Profile Management Functions	89
第 5 章	Optimizer Hints	91
5.1	Default Optimization Modes	92
5.2	Access Method Hints	94
5.3	Specifying a Join Order	98
5.4	Joining Relations Hints	99
5.5	Global Hints	102
5.6	Using the APPEND Optimizer Hint	105
5.7	Parallelism Hints	106
5.8	Conflicting Hints	110
第 6 章	dblink_ora	111
6.1	dblink_ora Functions and Procedures	112
6.1.1	dblink_ora_connect()	112
6.1.2	dblink_ora_status()	113
6.1.3	dblink_ora_disconnect()	113
6.1.4	dblink_ora_record()	114
6.1.5	dblink_ora_call()	114
6.1.6	dblink_ora_exec()	114
6.1.7	dblink_ora_copy()	115
6.2	Calling dblink_ora Functions	115
第 7 章	Open Client Library	117
第 8 章	Oracle Catalog Views	119
第 9 章	Tools and Utilities	121
第 10 章	ECPGPlus	123
第 11 章	System Catalog Tables	125

</div>

オラクルのデータベース互換性とは、アプリケーションコードをほとんどまたはまったく変更せずに、アプリケーションがオラクル環境および EDB Postgres Advanced Server（Advanced Server）環境で実行されることを意味します。Advanced Server でオラクルデータベースと互換性のあるアプリケーションを開発するには、アプリケーションの構築で使用する機能に特に注意する必要があります。例、互換性のあるアプリケーションを開発するということは、互換性のあるものを選択することを意味します。

SQL ステートメントおよび手続き型ロジックで使用するシステムおよびビルトイン関数。- ストアドプロシージャ、関数、トリガー、およびパッケージソフトのデータベースサーバ側アプリケーションロジックを作成するときのストアドプロシージャ言語（SPL）。オラクルデータベースと互換性のあるデータ型 オラクル SQL と互換性のある SQL ステートメント - Oracle のデータディクショナリと互換性のあるシステムカタログビュー

互換性のある SQL 構文、データ型、およびビューの詳細については、* オラクル Developers のデータベース互換性 SQL ガイド*を参照してください。

ビルトインパッケージソフトのパートであるプロシージャと関数によって提供される互換性は、* オラクル Developers ビルトインパッケージガイドのデータベース互換性*に記載されています。

Advanced Server のインストールに含まれる互換性のあるツールおよびユーティリティ（EDB * Plus、EDB * Loader、DRITA および EDB * Wrap）の使用の詳細は、* オラクル Developers Tools and Utilities Guide のデータベース互換性*を参照してください。

オラクル Call Interface（OCI）を使用して記述されたアプリケーションの場合、EDB の Open Client Library（OCL）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。Open Client Library の使用に関する詳細については、* EDB Postgres Advanced Server OCL Connector Guide *を参照してください。

Advanced Server には、PostgreSQL またはオラクル用のデータベースアプリケーションの開発を可能にする豊富な機能セットが含まれています。Advanced Server のすべての機能の詳細文書は、EDB ウェブサイトで入手可能なユーザマニュアルを参照してください。

Advanced Server の文書は次の場所から入手できます。

<https://www.enterprisedb.com/docs>

第1章 Configuration Parameters Compatible with Oracle Databases

</ div>

EDB Postgres Advanced Server は、PostgreSQL およびオラクルと互換性のあるアプリケーションの開発と実行をサポートしています。一部のシステム動作は、より多くの PostgreSQL またはよりオラクルに準拠した方法で動作するように変更できます。これらの動作は、構成パラメーターによって制御されます。`postgresql.conf` ファイルのパラメーターを変更すると、クラスター内のすべてのデータベースの動作が変更されますが、ユーザまたはグループはコマンドラインでパラメータ値を SET でき、セッションのみに影響を与えます。これらのパラメーターは次のとおりです。

- `edb_redwood_date` – 時間コンポーネントを DATE 列に格納するかどうかを制御します。オラクルデータベースと互換性のある動作を行うには、`edb_redwood_date` を TRUE に設定します。`edb_redwood_date` を参照してください。
- `edb_redwood_raw_names` – オラクルシステムカタログから表示するときに、データベースオブジェクト名を大文字で表示するか小文字で表示するかを制御します。オラクルデータベースと互換性のある動作の場合、`edb_redwood_raw_names` はデフォルト値の FALSE に設定されます。PostgreSQL システムカタログに実際に保存されているデータベースオブジェクト名をビューするには、`edb_redwood_raw_names` を TRUE に設定します。`edb_redwood_raw_names` を参照してください。
- `edb_redwood_strings` – 文字列結合操作のために、NULL を空の文字列と同等にします。オラクルデータベースと互換性のある動作を行うには、`edb_redwood_strings` を TRUE に設定します。`edb_redwood_strings` を参照してください。
- `edb_stmt_level_tx` – 中止された SQL コマンドの自動ロールバックをステートメントレベルのロールバックのみに分離します。デフォルトの PostgreSQL 動作の場合のように、現在のトランザクション全体は自動的にロールバックされません。オラクルデータベースと互換性のある動作を行うには、`edb_stmt_level_tx` を TRUE に設定します。ただし、絶対に必要な場合にのみ使用してください。`edb_stmt_level_tx` を参照してください。
- `oracle_home` – Advanced Server が正しいオラクルインストールディレクトリを指すようにします。`oracle_home` を参照してください。

1.1 edb_redwood_date

</ div>

DATE がコマンドの列のデータタイプとして表示される場合、設定パラメータ `edb_redwood_date` が TRUE に設定されている場合、テーブル定義がデータベースに格納されるときに、TIMESTAMP に変換されます。したがって、時間コンポーネントも日付とともに列に格納されます。これは、Oracle の DATE データタイプと一致しています。

`edb_redwood_date` が FALSE に設定されている場合、CREATE TABLE または ALTER TABLE コマンドの列のデータタイプは、PostgreSQL のネイティブ DATE データタイプのままであり、データベースにそのまま格納されます。PostgreSQL の DATE データタイプは、列に時間コンポーネントのない日付のみを格納します。

`edb_redwood_date` の設定に関係なく、コンテキストが SPL 宣言セクションの変数のデータタイプデータタイプ、SPL プロシージャまたは SPL ファンクションの仮パラメータのデータタイプ、または SPL ファンクションの戻り値の型は、常に内部的に TIMESTAMP に変換されるため、時間コンポーネントが存在する場合はハンドルできます。

日付/時刻データ型の詳細については、『オラクル開発者向けデータベース互換性リファレンスガイド』を参照してください。

1.2 edb_redwood_raw_names

</ div>

`edb_redwood_raw_names` がデフォルト値 FALSE に設定されている場合、オラクルカタログから表示すると、テーブル名、列名、トリガー名、プログラム名、ユーザ名などのデータベースオブジェクト名が大文字で表示されます（サポートされているカタログの完全なリスト）ビューについては、「* オラクル Developers カatalog Views のデータベース互換性ガイド*」を参照してください。さらに、引用符は、引用符で囲まれて作成された名前を囲み引用符。

`edb_redwood_raw_names` が TRUE に設定されている場合、データベースオブジェクト名は、オラクルカタログから表示したときに PostgreSQL システムカタログに保存されているとおりに表示されます。したがって、引用符で囲まずに作成された名前は、PostgreSQL で期待されるように小文字で表示されます。引用符で囲んで作成された名前は、作成されたとおりに表示されますが、引用符は表示されません。

例、次のユーザ名前が作成され、そのユーザとのセッションが開始されます。

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

reduser としてデータベースに接続すると、次のテーブルが作成されます。

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

edb_redwood_raw_names をデフォルト値 FALSE に設定してオラクルカタログ USER_TABLES から表示すると、引用符名前を除き、名前は大文字で表示されます。

```
edb=> SELECT * FROM USER_TABLES;
```

schema_name	table_name	tablespace_name	status	temporary
REDUSER	ALL_LOWER		VALID	N
REDUSER	ALL_UPPER		VALID	N
REDUSER	"Mixed_Case"		VALID	N

(3 rows)

edb_redwood_raw_names を TRUE に設定して表示すると、Mixed_Case 名前を除き、名前は小文字で表示され引用符。

```
edb=> SET edb_redwood_raw_names TO true;
SET
edb=> SELECT * FROM USER_TABLES;
```

schema_name	table_name	tablespace_name	status	temporary
reduser	all_lower		VALID	N
reduser	all_upper		VALID	N
reduser	Mixed_Case		VALID	N

(3 rows)

これらの名前は、PostgreSQL pg_tables カタログから見ると大文字と小文字がマッチするようになりました。

```
edb=> SELECT schemaname, tablename, tableowner FROM pg_tables WHERE
tableowner = 'reduser';
```

schemaname	tablename	tableowner
reduser	all_lower	reduser
reduser	all_upper	reduser
reduser	Mixed_Case	reduser

(3 rows)

1.3 edb_redwood_strings

</ div>

オラクルでは、文字列が null 変数または null 列と連結されると、結果はオリジナルの文字列になります。ただし、PostgreSQL では、文字列と null 変数または null 列を結合すると、null の結果が得られます。edb_redwood_strings パラメータが TRUE に設定されている場合、前述の結合オペレーションの結果、オラクルで行われたオリジナルの文字列が生成されます。edb_redwood_strings が FALSE に設定されている場合、ネイティブの PostgreSQL の動作が維持されます。

次の例は、違いを示しています。

次のセクションで紹介するサンプルアプリケーションには、従業員のテーブルが含まれています。このテーブルには、ほとんどの従業員にとって null である comm という名前付けの列があります。次のクエリーは、edb_redwood_string を FALSE に設定して実行されます。null 列と空でない文字列を結合すると、最終結果 null が生成されるため、クエリー結果には歩合のある従業員のみが表示されます。他のすべての従業員の出力行は null です。

```
SET edb_redwood_strings TO off;

SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' ||
TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;
*      EMPLOYEE COMPENSATION
-----
* ALLEN      1,600.00      300.00
WARD         1,250.00      500.00
* MARTIN     1,250.00     1,400.00
* TURNER     1,500.00        .00

(14 rows)
```

以下は、edb_redwood_strings が TRUE に設定されている場合に実行される同じクエリーです。ここでは、null 列の値は空の文字列として扱われます。空の文字列と空でない文字列を結合すると、空でない文字列が生成され文字列。この結果は、同じクエリーに対してオラクルによって生成された結果と一致しています。

```
SET edb_redwood_strings TO on;

SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' ||
TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;
*      EMPLOYEE COMPENSATION
-----
SMITH         800.00
ALLEN         1,600.00      300.00
WARD          1,250.00      500.00
JONES         2,975.00
MARTIN        1,250.00     1,400.00
BLAKE         2,850.00
CLARK         2,450.00
SCOTT         3,000.00
KING          5,000.00
TURNER        1,500.00        .00
ADAMS         1,100.00
JAMES         950.00
FORD          3,000.00
MILLER        1,300.00

(14 rows)
```

1.4 edb_stmt_level_tx

</div>

オラクルでは、SQL コマンドでランタイムエラーが発生すると、その単一のコマンドによって引き起こされたデータベース上のすべての更新がロールバックされます。これは、**文レベルのトランザクションの分離**と呼ばれます。例、1つの UPDATE コマンドが5行を正常に更新したが、6行を更新しようすると例外が発生する場合、この UPDATE コマンドによって行われた6行すべての更新はロールバックされます。まだコミットまたはロールバックされていない以前の SQL コマンドの効果は、COMMIT または ROLLBACK コマンドが実行されるまで保留されます。

PostgreSQL では、SQL コマンドの実行中に例外が発生すると、トランザクションのスタート以降のデータベース上のすべての更新がロールバックされます。さらに、トランザクションは中止状態のままになり、別のトランザクションを開始する前に COMMIT または ROLLBACK コマンドを発行する必要があります。

edb_stmt_level_tx が TRUE に設定されている場合、例外は以前のコミットされていないデータベース更新を自動的にロールバックせず、オラクルの動作をエミュレートします。edb_stmt_level_tx が FALSE に設定されている場合、例外はコミットされていないデータベース更新をロールバックします。

!!! Note * edb_stmt_level_tx を TRUE に設定するのは、どうしても必要な場合のみにしてください。パフォーマンスにインパクトを与える可能性があります。

次の PSQL での実行例は、edb_stmt_level_tx が FALSE である場合、2 番目の INSERT コマンドのアボートによって最初の INSERT コマンドもロールバックされることを示しています。PSQL ではコマンド \set AUTOCOMMIT off を発行する必要があります。そうしないと、すべてのステートメントが自動的にコミットし、edb_stmt_level_tx の効果のこのデモの目的に反します。

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

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

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

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

次の例では、edb_stmt_level_tx が TRUE に設定されているため、2 番目の INSERT コマンドでエラーが発生した後、最初の INSERT コマンドはロールバックされていません。この時点で、最初の INSERT コマンドはコミットまたはロールバックできます。

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

(次のページに続く)

(前のページからの続き)

```

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

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

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

COMMIT;
```

COMMIT コマンドの代わりに ROLLBACK コマンドが発行された場合、従業員番号 9001 の挿入もロールバックされます。

1.5 oracle_home

</div>

オラクルサーバーへのリンクを作成する前に、Advanced Server を正しいオラクルホームディレクトリに移動する必要があります。Linux の LD_LIBRARY_PATH 環境変数（または Windows の PATH）をオラクルクライアントインストールディレクトリの lib ディレクトリに設定します。

Windows の場合のみ、代わりに postgresql.conf ファイルの oracle_home 設定パラメータの値を設定できます。oracle_home 設定パラメータで指定された値は、Windows の PATH 環境変数をオーバーライドします。

Linux では LD_LIBRARY_PATH 環境環境変数（WindowsPATH 環境変数または oracle_home 設定パラメータ）は、Advanced Server をスタートするたびに適切に設定する必要があります。

Linux サービススクリプトを使用して Advanced Server をスタートする場合は、スクリプトが pg_ctl ユーティリティを呼び出して Advanced Server をスタートするときに有効になるように、サービススクリプト内で LD_LIBRARY_PATH が設定されていることを確認してください。

**** Windows のみ **: postgresql.conf ファイルで oracle_home 設定パラメータを設定するには、ファイルを編集し、次の行を追加します。**

```
oracle_home = 'lib_directory'
```

lib_directory を oci.dll を含む Windows ディレクトリの名前に置き換えます。

oracle_home 設定パラメータを設定した後、変更を有効にするにはサーバーをリスタートする必要があります。Windows サービスコンソールからサーバーを再起動します。

第2章 SQL Tutorial

</ div>

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。つまり、リレーションに保存されたデータを管理するためのシステムです。リレーションは、本質的に**テーブル**の数学用語です。データをテーブルに格納するという概念は今日では非常に一般的であるため、本質的に明白に思えるかもしれませんが、データベースを編成する方法は他にもたくさんあります。Unix ライクなオペレーティングシステム上のファイルとディレクトリは、階層フォームデータベースの例です。より近代的な開発は、オブジェクト指向データベースです。

各テーブルは、*** rows の名前付けコレクション**です。特定の**テーブルの行**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。列は行で固定されたオーダーを持っていますが、SQL はテーブル内の行のオーダーを保証しません（表示用に明示的に並べ替えることはできますが）。

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

2.1 Sample Database

</ div>

この文書全体を通して、サンプルデータベースを使用して、基本レベルから高度なレベルのデータベースの概念を説明しヘルプ。

2.1.1 Sample Database Installation

</ div>

Advanced Server をインストールすると、**edb** 名前付け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、**/usr/edb/as14/share** ディレクトリにあるスクリプト **edb-sample.sql** を実行することにより、このドキュメント全体で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。

- 現在接続され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
- テーブルのすべての権限を **PUBLIC** グループに付与します

テーブルとプログラムは、現在のユーザがテーブルとプロシージャを作成するパーミッションを持つ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;
```

検索パスの変更は、PSQL のコマンドを使用して実行できます。

2.1.2 Sample Database Description

</ div>

サンプルデータベースは、組織内の従業員を表します。

従業員、部門、および従業員の履歴レコードの 3 つのタイプ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、およびマネージャがあります。一部の従業員は、給与に加えて手数料を受け取ります。すべての従業員関連情報は、**emp** テーブルに保存されます。

サンプル会社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。各会社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縮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ています。すべての部門関連情報は **dept** テーブルに保存されます。

同社はまた、従業員が抱える仕事に関する情報も追跡しています。一部の従業員は、長期間にわたって会社に勤務しており、さまざまな位置に就いたり、昇給を受けたり、部門を切り替えたりしました。スタート日、新しいジョブ、部門、給与、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。すべての従業員の履歴は、**jobhist** テーブルで維持されます。

以下は、サンプルデータベーステーブルのエンティティリレーション図です。

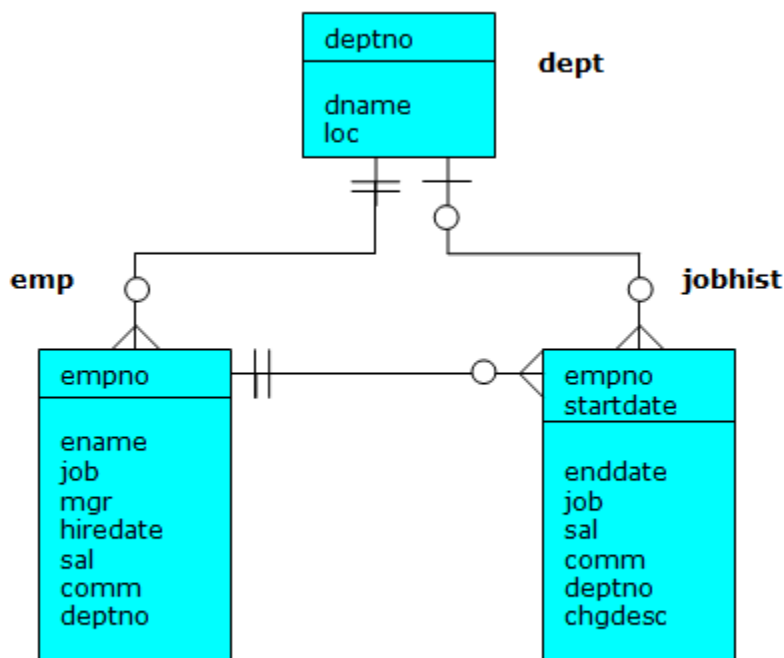


図 1: Sample Database Tables

図 1：データベーステーブルの例</ div>

以下は、edb-sample.sql スクリプトです。

```
--
-- Script that creates the 'sample' tables, views, procedures,
-- functions, triggers, etc.
--
-- Start new transaction - commit all or nothing
--
BEGIN;
/
--
-- Create and load tables used in the documentation examples.
--
-- Create the 'dept' table
--
CREATE TABLE dept (
  * deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  * dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
  * loc             VARCHAR2(13)
);
--
-- Create the 'emp' table
--
CREATE TABLE emp (
  * empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  * ename           VARCHAR2(10),
  * job             VARCHAR2(9),
  * mgr             NUMBER(4),
  * hiredate        DATE,
  * sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  * comm            NUMBER(7,2),
  * deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
  *                REFERENCES dept(deptno)
);
--
-- Create the 'jobhist' table
--
CREATE TABLE jobhist (
  * empno           NUMBER(4) NOT NULL,
  * startdate       DATE NOT NULL,
  * enddate         DATE,
  * job             VARCHAR2(9),
  * sal             NUMBER(7,2),
  * comm            NUMBER(7,2),
  * deptno          NUMBER(2),
```

(次のページに続く)

(前のページからの続き)

```

*   chgdesc          VARCHAR2(80),
*   CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
*   CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
*       REFERENCES emp(empno) ON DELETE CASCADE,
*   CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
*       REFERENCES dept (deptno) ON DELETE SET NULL,
*   CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
);
--
-- Create the 'salesemp' view
--
CREATE OR REPLACE VIEW salesemp AS
    *   SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
--
-- Sequence to generate values for function 'new_empno'.
--
CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
--
-- Issue PUBLIC grants
--
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
GRANT ALL ON next_empno TO PUBLIC;
--
-- Load the 'dept' table
--
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');
--
-- Load the 'emp' table
--
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-81',2975,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-81',1250,1400,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-81',2850,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-81',2450,NULL,10);
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-87',3000,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'17-NOV-81',5000,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,0,30);

```

(次のページに続く)

(前のページからの続き)

```

INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3000,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);
--
-- Load the 'jobhist' table
--
INSERT INTO jobhist VALUES (7369,'17-DEC-80',NULL,'CLERK',800,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,'New Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,'New Hire');
INSERT INTO jobhist VALUES (7566,'02-APR-81',NULL,'MANAGER',2975,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7654,'28-SEP-81',NULL,'SALESMAN',1250,1400,30,'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,'New Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-88','CLERK',1000,NULL,20,'New
→Hire');
INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-89','CLERK',1040,NULL,20,'Raise
→');
INSERT INTO jobhist VALUES (7788,'05-MAY-90',NULL,'ANALYST',3000,NULL,20,'Promoted to
→Analyst');
INSERT INTO jobhist VALUES (7839,'17-NOV-81',NULL,'PRESIDENT',5000,NULL,10,'New Hire
→');
INSERT INTO jobhist VALUES (7844,'08-SEP-81',NULL,'SALESMAN',1500,0,30,'New Hire');
INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-83','CLERK',950,NULL,10,'New Hire
→');
INSERT INTO jobhist VALUES (7900,'15-JAN-83',NULL,'CLERK',950,NULL,30,'Changed to
→Dept 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,'New Hire');
--
-- Populate statistics table and view (pg_statistic/pg_stats)
--
ANALYZE dept;
ANALYZE emp;
ANALYZE jobhist;
--
-- Procedure that lists all employees' numbers and names
-- from the 'emp' table using a cursor.
--
CREATE OR REPLACE PROCEDURE list_emp
IS
*   v_empno          NUMBER(4);
*   v_ename          VARCHAR2(10);
*   CURSOR emp_cur IS

```

(次のページに続く)

(前のページからの続き)

```

*      SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
*      OPEN emp_cur;
*      DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
*      DBMS_OUTPUT.PUT_LINE('-----      -----');
*      LOOP
*          FETCH emp_cur INTO v_empno, v_ename;
*          EXIT WHEN emp_cur%NOTFOUND;
*          DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || v_ename);
*      END LOOP;
*      CLOSE emp_cur;
END;
/
--
-- Procedure that selects an employee row given the employee
-- number and displays certain columns.
--
CREATE OR REPLACE PROCEDURE select_emp (
*      p_empno          IN  NUMBER
)
IS
*      v_ename          emp.ename%TYPE;
*      v_hiredate       emp.hiredate%TYPE;
*      v_sal            emp.sal%TYPE;
*      v_comm           emp.comm%TYPE;
*      v_dname          dept.dname%TYPE;
*      v_disp_date      VARCHAR2(10);
BEGIN
*      SELECT ename, hiredate, sal, NVL(comm, 0), dname
*          INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
*          FROM emp e, dept d
*          WHERE empno = p_empno
*          AND e.deptno = d.deptno;
*      v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
*      DBMS_OUTPUT.PUT_LINE('Number      : ' || p_empno);
*      DBMS_OUTPUT.PUT_LINE('Name        : ' || v_ename);
*      DBMS_OUTPUT.PUT_LINE('Hire Date   : ' || v_disp_date);
*      DBMS_OUTPUT.PUT_LINE('Salary      : ' || v_sal);
*      DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
*      DBMS_OUTPUT.PUT_LINE('Department: ' || v_dname);
EXCEPTION
*      WHEN NO_DATA_FOUND THEN
*          DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
*      WHEN OTHERS THEN
*          DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');

```

(次のページに続く)

(前のページからの続き)

```

*      DBMS_OUTPUT.PUT_LINE(SQLERRM);
*      DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
*      DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
/
--
-- Procedure that queries the 'emp' table based on
-- department number and employee number or name. Returns
-- employee number and name as IN OUT parameters and job,
-- hire date, and salary as OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query (
*   p_deptno      IN      NUMBER,
*   p_empno       IN OUT NUMBER,
*   p_ename       IN OUT VARCHAR2,
*   p_job         OUT     VARCHAR2,
*   p_hiredate    OUT     DATE,
*   p_sal         OUT     NUMBER
)
IS
BEGIN
*   SELECT empno, ename, job, hiredate, sal
*       INTO p_empno, p_ename, p_job, p_hiredate, p_sal
*       FROM emp
*       WHERE deptno = p_deptno
*             AND (empno = p_empno
*                  OR  ename = UPPER(p_ename));
END;
/
--
-- Procedure to call 'emp_query_caller' with IN and IN OUT
-- parameters. Displays the results received from IN OUT and
-- OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query_caller
IS
*   v_deptno      NUMBER(2);
*   v_empno       NUMBER(4);
*   v_ename       VARCHAR2(10);
*   v_job         VARCHAR2(9);
*   v_hiredate    DATE;
*   v_sal         NUMBER;
BEGIN
*   v_deptno := 30;
*   v_empno  := 0;

```

(次のページに続く)

(前のページからの続き)

```

* v_ename := 'Martin';
* emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
* DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
* DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
* DBMS_OUTPUT.PUT_LINE('Name      : ' || v_ename);
* DBMS_OUTPUT.PUT_LINE('Job      : ' || v_job);
* DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_hiredate);
* DBMS_OUTPUT.PUT_LINE('Salary   : ' || v_sal);
EXCEPTION
* WHEN TOO_MANY_ROWS THEN
*     DBMS_OUTPUT.PUT_LINE('More than one employee was selected');
* WHEN NO_DATA_FOUND THEN
*     DBMS_OUTPUT.PUT_LINE('No employees were selected');
END;
/
--
-- Function to compute yearly compensation based on semimonthly
-- salary.
--
CREATE OR REPLACE FUNCTION emp_comp (
* p_sal          NUMBER,
* p_comm         NUMBER
) RETURN NUMBER
IS
BEGIN
* RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
/
--
-- Function that gets the next number from sequence, 'next_empno',
-- and ensures it is not already in use as an employee number.
--
CREATE OR REPLACE FUNCTION new_empno RETURN NUMBER
IS
* v_cnt          INTEGER := 1;
* v_new_empno    NUMBER;
BEGIN
* WHILE v_cnt > 0 LOOP
*     SELECT next_empno.nextval INTO v_new_empno FROM dual;
*     SELECT COUNT(*) INTO v_cnt FROM emp WHERE empno = v_new_empno;
* END LOOP;
* RETURN v_new_empno;
END;
/
--

```

(次のページに続く)

(前のページからの続き)

```

-- EDB-SPL function that adds a new clerk to table 'emp'.  This function
-- uses package 'emp_admin'.
--
CREATE OR REPLACE FUNCTION hire_clerk (
    *   p_ename          VARCHAR2,
    *   p_deptno         NUMBER
) RETURN NUMBER
IS
    *   v_empno         NUMBER(4);
    *   v_ename         VARCHAR2(10);
    *   v_job           VARCHAR2(9);
    *   v_mgr           NUMBER(4);
    *   v_hiredate      DATE;
    *   v_sal           NUMBER(7,2);
    *   v_comm          NUMBER(7,2);
    *   v_deptno        NUMBER(2);
BEGIN
    *   v_empno := new_empno;
    *   INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
    *       TRUNC(SYSDATE), 950.00, NULL, p_deptno);
    *   SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
    *       v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
    *       FROM emp WHERE empno = v_empno;
    *   DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    *   DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    *   DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    *   DBMS_OUTPUT.PUT_LINE('Job       : ' || v_job);
    *   DBMS_OUTPUT.PUT_LINE('Manager   : ' || v_mgr);
    *   DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    *   DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
    *   DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
    *   RETURN v_empno;
EXCEPTION
    *   WHEN OTHERS THEN
    *       DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
    *       DBMS_OUTPUT.PUT_LINE(SQLERRM);
    *       DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
    *       DBMS_OUTPUT.PUT_LINE(SQLCODE);
    *       RETURN -1;
END;
/
--
-- PostgreSQL PL/pgSQL function that adds a new salesman
-- to table 'emp'.
--

```

(次のページに続く)

(前のページからの続き)

```

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

```

(次のページに続く)

(前のページからの続き)

```

CREATE OR REPLACE RULE salesemp_i AS ON INSERT TO salesemp
DO INSTEAD
    * INSERT INTO emp VALUES (NEW.empno, NEW.ename, 'SALESMAN', 7698,
    *     NEW.hiredate, NEW.sal, NEW.comm, 30);
--
-- Rule to UPDATE view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_u AS ON UPDATE TO salesemp
DO INSTEAD
    * UPDATE emp SET empno      = NEW.empno,
    *             ename        = NEW.ename,
    *             hiredate     = NEW.hiredate,
    *             sal          = NEW.sal,
    *             comm         = NEW.comm
    * WHERE empno = OLD.empno;
--
-- Rule to DELETE from view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_d AS ON DELETE TO salesemp
DO INSTEAD
    * DELETE FROM emp WHERE empno = OLD.empno;
--
-- After statement-level trigger that displays a message after
-- an insert, update, or deletion to the 'emp' table. One message
-- per SQL command is displayed.
--
CREATE OR REPLACE TRIGGER user_audit_trig
    * AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    * v_action          VARCHAR2(24);
BEGIN
    * IF INSERTING THEN
    *     v_action := ' added employee(s) on ';
    * ELSIF UPDATING THEN
    *     v_action := ' updated employee(s) on ';
    * ELSIF DELETING THEN
    *     v_action := ' deleted employee(s) on ';
    * END IF;
    * DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
    * TO_CHAR(SYSDATE, 'YYYY-MM-DD'));
END;
/
--
-- Before row-level trigger that displays employee number and
-- salary of an employee that is about to be added, updated,

```

(次のページに続く)

(前のページからの続き)

```

-- or deleted in the 'emp' table.
--
CREATE OR REPLACE TRIGGER emp_sal_trig
* BEFORE DELETE OR INSERT OR UPDATE ON emp
* FOR EACH ROW
DECLARE
* sal_diff      NUMBER;
BEGIN
* IF INSERTING THEN
*   DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
*   DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
* END IF;
* IF UPDATING THEN
*   sal_diff := :NEW.sal - :OLD.sal;
*   DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
*   DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
*   DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
*   DBMS_OUTPUT.PUT_LINE('..Raise      : ' || sal_diff);
* END IF;
* IF DELETING THEN
*   DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
*   DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
* END IF;
END;
/
--
-- Package specification for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE emp_admin
IS
* FUNCTION get_dept_name (
*   p_deptno      NUMBER
* ) RETURN VARCHAR2;
* FUNCTION update_emp_sal (
*   p_empno       NUMBER,
*   p_raise       NUMBER
* ) RETURN NUMBER;
* PROCEDURE hire_emp (
*   p_empno       NUMBER,
*   p_ename       VARCHAR2,
*   p_job         VARCHAR2,
*   p_sal         NUMBER,
*   p_hiredate    DATE,
*   p_comm        NUMBER,
*   p_mgr         NUMBER,

```

(次のページに続く)

(前のページからの続き)

```

*      p_deptno      NUMBER
* );
* PROCEDURE fire_emp (
*      p_empno      NUMBER
* );
END emp_admin;
/
--
-- Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
* --
* -- Function that queries the 'dept' table based on the department
* -- number and returns the corresponding department name.
* --
* FUNCTION get_dept_name (
*      p_deptno      IN NUMBER
* ) RETURN VARCHAR2
* IS
*      v_dname      VARCHAR2(14);
* BEGIN
*      SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
*      RETURN v_dname;
* EXCEPTION
*      WHEN NO_DATA_FOUND THEN
*          DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
*          RETURN '';
* END;
* --
* -- Function that updates an employee's salary based on the
* -- employee number and salary increment/decrement passed
* -- as IN parameters. Upon successful completion the function
* -- returns the new updated salary.
* --
* FUNCTION update_emp_sal (
*      p_empno      IN NUMBER,
*      p_raise      IN NUMBER
* ) RETURN NUMBER
* IS
*      v_sal      NUMBER := 0;
* BEGIN
*      SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
*      v_sal := v_sal + p_raise;
*      UPDATE emp SET sal = v_sal WHERE empno = p_empno;

```

(次のページに続く)

```

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

```

2.2 Creating a New Table

</ div>

新しいテーブルを作成するには、テーブル名前とすべての列名とそのタイプを指定します。以下は、**emp** サンプルテーブルの単純化されたバージョンで、テーブルの定義に必要な最小限の情報のみが含まれています。

```
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)
);
```

これを改行で PSQL に入力できます。PSQL は、セミコロンまでコマンドが終了しないことを認識します。

スペース（つまり、スペース、タブ、改行）は、SQL コマンドで自由に使用できます。つまり、上記とは異なる方法で整列したコマンドを入力することも、1 行ですべてを入力することもできます。2 つのダッシュ（“-”）はコメントを示します。それらに続くものはすべて、行末まで無視されます。SQL、大文字と小文字を区別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（上記では行われません）、キーと識別子について大文字と小文字は区別されません。

VARCHAR2(10) は、最大 10 文字の任意の文字列を格納できるデータタイプを指定し長さ。**NUMBER(7,2)** は、精度 7 およびスケール 2 の固定小数点数です。**NUMBER(4)** は、精度 4 およびスケール 0 の整数です。

Advanced Server は、通常の SQL データ型 **INTEGER**、**SMALLINT**、**NUMBER**、**REAL**、**DOUBLE PRECISION**、**CHAR**、**VARCHAR2**、**DATE**、および **TIMESTAMP** と、これらの型のさまざまな同義語をサポートしています。

テーブルが不要になった場合、または別の方法で再作成する場合は、次のコマンドを使用して削除できます。

```
DROP TABLE tablename;
```

2.3 Populating a Table With Rows

</ div>

INSERT ステートメントは、テーブルに行を入力するために使用されます。

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

すべてのデータ型は、かなり明白な入力形式を使用することに注意してください。例のように、シンプル数値ではない定数は通常、シングルクォート（'）で囲む必要があります。例。**DATE** 型は、実際に受け入れるものが非常に柔軟ですが、このチュートリアルでは、ここに示す明確なフォーマットに固執します。

これまでに使用された構文では、列のオーダーを覚えておく必要があります。別の構文を使用すると、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);
```

希望する場合は列を別のオーダーでリストしたり、一部の列を省略したりすることもできます（例：手数料が不明な場合）。

```
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、暗黙的にオーダーに依存するよりも、列を明示的にリストする方がスタイルが優れていると考えています。

2.4 Querying a Table

</div>

テーブルからデータを取得するために、テーブルはクエリされます。これを行うには、SQL SELECT ステートメントを使用します。ステートメントは、選択リスト（返される列をリストするパート）、テーブルリスト（データを取得するテーブルをリストするパート）、およびオプションの修飾（制限を指定するパート）に分かれています。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オーダーでリストします。

```
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、このクエリーからの出力です。

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

(14 rows)

選択リストで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;
```

* 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目してください。(AS 句はオプションル。)

クエリーは、必要な行を指定する WHERE 句を追加することで修飾できます。WHERE 句にはブール (真理値) 式が含まれ、ブール式が true である行のみが返されます。修飾には通常のブール演算子 (AND、OR、および NOT) を使用できます。例、次の例では、給与が 1000.00 ドルを超える部門 20 の従業員を取得します。

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

* ename	sal	deptno
JONES	2975.00	20
SCOTT	3000.00	20
ADAMS	1100.00	20
FORD	3000.00	20

(4 rows)

クエリーの結果がソートされたオーダーで返されるようにリクエストできます。

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

* ename	sal	deptno
ADAMS	1100.00	20
ALLEN	1600.00	30
BLAKE	2850.00	30
CLARK	2450.00	10
FORD	3000.00	20
JAMES	950.00	30

(次のページに続く)

(前のページからの続き)

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)				

クエリーの結果から重複する行を削除するように要求できます。

SELECT DISTINCT job FROM emp;				
* job				

ANALYST				
CLERK				
MANAGER				
PRESIDENT				
SALESMAN				
(5 rows)				

次のセクションでは、1つのクエリーで複数のテーブルから行を取得する方法を示します。

2.5 Joins Between Tables

</div>

これまで、クエリは一度に1つのテーブルにしかアクセスしませんでした。クエリは、一度にマルチプルのテーブルにアクセスするか、テーブルのマルチプル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できます。同じテーブルまたは異なるテーブルのマルチプルの行に一度にアクセスするクエリーは、*結合*クエリーと呼ばれます。例、すべての従業員レコードを関連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 テーブルの行の deptno 列を dept テーブルのすべての行の deptno 列と比較し、これらの値がマッチする行のペアを選択する必要があります。これは、次のクエリーによって実現されます。

SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM emp, dept				
WHERE emp.deptno = dept.deptno;				
* ename sal deptno dname loc				
-----+-----+-----+-----+-----				
MILLER		1300.00		10 ACCOUNTING NEW YORK
CLARK		2450.00		10 ACCOUNTING NEW YORK
KING		5000.00		10 ACCOUNTING NEW YORK
SCOTT		3000.00		20 RESEARCH DALLAS

(次のページに続く)

(前のページからの続き)

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)

結果セットについて次の2つのことを確認してください。

- 部門 40 の結果行はありません。これは、部門 40 の **emp** テーブルにマッチングエントリがないため、**dept** テーブルの一致しない行を結合が無視するためです。すぐにこれを修正する方法を確認します。
- *を使用したり、次のように修飾を省略したりするのではなく、テーブル名前で修飾された出力列をリストの方が望ましいです。

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

すべての列の名前は異なるため（修飾する必要がある **deptno** を除く）、パーサはどのテーブルに属しているかを自動的に検出しますが、結合クエリで列名を完全に修飾するのが良いスタイルです。

これまで見た種類の結合クエリは、この代替フォームでも記述できます。

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

この構文は上記の構文ほど一般的には使用されませんが、次のトピックを理解するのにヘルプようにここで示します。

結合の上記のすべての結果で、部門 40 に属する従業員が返されなかったため、部門 40 のレコードが表示されないことに気付くでしょう。次に、マッチング従業員がないという事実にもかかわらず、結果に部門 40 のレコードを取得する方法を見つけます。クエリで実行するのは、**dept** テーブルをスキャンし、行でマッチング **emp** 行を見つけることです。マッチング行が見つからない場合は、**emp** テーブルの列を「空の」値に置き換える必要があります。この種のクエリは**外部結合**と呼ばれます。（これまで見てきた結合は**内部結合**です。）コマンドは次のようになります。

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM dept LEFT
OUTER JOIN emp ON emp.deptno = dept.deptno;
   * ename | sal | deptno | dname | loc
-----+-----+-----+-----+-----
MILLER | 1300.00 | 10 | ACCOUNTING | NEW YORK
```

(次のページに続く)

(前のページからの続き)

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)

このクエリーは、**左外部結合**と呼ばれます。これは、結合演算子の左側にあるテーブルの出力に各行が少なくとも 1 回含まれるのに対し、右側のテーブルには、ある行にマッチする行出力のみがあるためです左のテーブルの。右テーブルのマッチがない左テーブル行が選択されると、空（NULL）値が右テーブル列に置き換えられます。

外部結合の代替構文は、WHERE 句内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 値で置き換える必要があるテーブルの列名前の後に配置されます。そのため、emp テーブルにマッチング行がない dept テーブルのすべての行について、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、上記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;
  * 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
```

(次のページに続く)

(前のページからの続き)

```
*          |          |      40 | OPERATIONS | BOSTON
(15 rows)
```

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の名前を検索するとします。したがって、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;
```

```
* Employees and their Managers
```

```
-----
FORD works for JONES
SCOTT works for JONES
WARD works for BLAKE
TURNER works for BLAKE
MARTIN works for BLAKE
JAMES works for BLAKE
ALLEN works for BLAKE
MILLER works for CLARK
ADAMS works for SCOTT
CLARK works for KING
BLAKE works for KING
JONES works for KING
SMITH works for FORD
```

```
(13 rows)
```

ここで、emp テーブルは、選択リストと結合条件の従業員行を表す e1 として、また選択リストと結合条件のマネージャとして機能マッピング従業員行を表す e2 としてラベルが変更されました。これらの種類のエイリアスを他のクエリで使用して、例入力を節約できます。

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

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

(次のページに続く)

(前のページからの続き)

```

BLAKE | 7839 |      30 | SALES      | CHICAGO
MARTIN | 7698 |      30 | SALES      | CHICAGO
JAMES  | 7698 |      30 | SALES      | CHICAGO
(14 rows)

```

このスタイルの省略は非常に頻繁に発生します。

2.6 Aggregate Functions

</div>

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、Advanced Server は集計関数をサポートしています。集約ファンクションは、マルチプルの入力行から単一の結果を計算します。例、行のセットで COUNT、SUM、AVG（平均）、MAX（最大）、および MIN（最小）を計算する集計があります。

例として、最高給与と最低給与は次のクエリーで見つけることができます。

```

SELECT MAX(sal) highest_salary, MIN(sal) lowest_salary FROM emp;
* highest_salary | lowest_salary
-----+-----
*      5000.00   |          800.00
(1 row)

```

給与が最も高い従業員を見つけたい場合は、次のことを試してみてください。

```

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

ERROR: aggregates not allowed in WHERE clause

```

集計ファンクション MAX は WHERE 句では使用できないため、これは機能しません。この制約が存在するのは、WHERE 句が集計ステージに入る行を決定し、集計関数を計算する前に評価する必要があるためです。ただし、サブクエリを使用して、目的の結果を達成するためにクエリーを修正できます。

```

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

```

サブクエリは、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。

集計は、GROUP BY 句と組み合わせて非常に便利です。例、次のクエリーは各部門の最高給与を取得します。

```

SELECT deptno, MAX(sal) FROM emp GROUP BY deptno;
* deptno | max
-----+-----

```

(次のページに続く)

(前のページからの続き)

```

*      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、その部門にマッチング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;
* deptno |    max
-----+-----
*      10 | 5000.00
*      20 | 3000.00
(2 rows)

```

このクエリーは、平均給与が 2000 をより大きい部門についてのみ同じ結果を返します。

最後に、次のクエリーでは、各部門のアナリストである最も給与の高い従業員のみがアカウントされます。

```

SELECT deptno, MAX(sal) FROM emp WHERE job = 'ANALYST' GROUP BY deptno HAVING
→AVG(sal) > 2000;
* deptno |    max
-----+-----
*      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て集計関数が適用される前に行をフィルターで除外します。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、結果にフィルターを適用します。

したがって、前の例では、アナリストである従業員のみが考慮されます。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、グループ内のアナリストの平均給与が 2000 をより大きいグループのみが最終結果に含まれます。これは、部門 20 のグループのみに当てはまり、部門 20 の最大アナリスト給与は 3000.00 です。

2.7 Updates

</div>

既存の行の列値は、UPDATE コマンドを使用して変更できます。例、次のシーケンスのコマンドは、マネージャである全員に 10 % の昇給を行った前後の結果を示しています。

```

SELECT ename, sal FROM emp WHERE job = 'MANAGER';
* ename |    sal
-----+-----
JONES  | 2975.00

```

(次のページに続く)

(前のページからの続き)

```
BLAKE | 2850.00
CLARK | 2450.00
(3 rows)

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

SELECT ename, sal FROM emp WHERE job = 'MANAGER';
   * ename |    sal
-----+-----
JONES | 3272.50
BLAKE | 3135.00
CLARK | 2695.00
(3 rows)
```

2.8 Deletions

</ div>

DELETE コマンドを使用して、テーブルから行を削除できます。例、次のコマンドシーケンスは、部門 20 のすべての従業員を削除した前後の結果を示しています。

```
SELECT ename, deptno FROM emp;
   * ename | deptno
-----+-----
SMITH | 20
ALLEN | 30
WARD | 30
JONES | 20
MARTIN | 30
BLAKE | 30
CLARK | 10
SCOTT | 20
KING | 10
TURNER | 30
ADAMS | 20
JAMES | 30
FORD | 20
MILLER | 10
(14 rows)

DELETE FROM emp WHERE deptno = 20;

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

(次のページに続く)

(前のページからの続き)

```

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

```

次のような WHERE 句なしで DELETE コマンドを指定する場合は、十分に注意してください。

```
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。これを行う前に、システムは確認を要求しません。

2.9 The SQL Language

</ div>

Advanced Server は、オラクル構文と互換性のある SQL 言語と、拡張機能（オラクルのデータベース互換性を提供しない、または Oracle スタイルのアプリケーションをサポートしない機能）の構文とコマンドをサポートします。

- オラクル Developers SQL ガイドのデータベース互換性*には、以下に関する詳細情報が記載されています。
- 互換性のある SQL 構文と言語要素
- データ型
- サポートされている SQL コマンド構文

ガイドのコピーを確認するには、Advanced Server ウェブサイトにアクセスしてください。

<https://www.enterprisedb.com/docs>

第3章 Advanced Concepts

</div>

前のセクションでは、SQL を使用して Advanced Server のデータを保存およびアクセスする基本について説明しました。このセクションでは、管理を簡素化し、データの損失または破損を防ぐことができる、より高度な SQL 機能について説明します。

3.1 Views

</div>

次の SELECT コマンドを検討してください。

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

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

(14 rows)

これが繰り返し使用されるクエリーの場合、毎回 SELECT コマンド全体を再入力せずにこのクエリーを再利用する簡単なメソッドは、以下に示すように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;
```

ビュー名前 `employee_pay` は、クエリーを実行するために通常のテーブル名前のように使用できるようになりました。

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

ビューを自由に使用することは、優れた SQL データベースデザインのキー側面です。Views は、テーブル構造の詳細をカプセル化する一貫したインタフェースを提供します。これは、アプリケーションの進化に応じて変化する可能性があります。

Views は、実際のテーブルを使用できるほぼすべての場所で使用できます。他のビューの上にビューを構築することは珍しくありません。

3.2 Foreign Keys

</ div>

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とします。これは、データの参照整合性の維持と呼ばれます。単純なデータベースシステムでは、これは（もしあれば）実装され、最初に dept テーブルを見てマッチングレコードが存在するかどうかを確認し、次に新しい従業員レコードを挿入または拒否します。このアプローチには多くの問題があり、非常に不便です。Advanced Server は、それをより簡単にします。

このセクションでは、Creating a New Table に示されている emp テーブルの修正バージョンを、外部キー制約を追加して示します。変更された 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),
  *   hiredate       DATE,
  *   sal            NUMBER(7,2),
```

(次のページに続く)

(前のページからの続き)

```

*   comm          NUMBER(7,2),
*   deptno        NUMBER(2) CONSTRAINT emp_ref_dept_fk
*                REFERENCES dept(deptno)
);

```

サンプルの **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** により、**dept** テーブルに部門 50 が存在することが保証されます。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".

```

外部キーの動作は、アプリケーションに合わせて微調整できます。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確実に向上するため、外部キーについてさらに学ぶことを強くお勧めします。

3.3 The ROWNUM Pseudo-Column

</ div>

ROWNUM は、クエリーから行が取得されたオーダーに基づいて、行に増分の一意の整数値が割り当てられる擬似列です。したがって、取得される最初の行の ROWNUM は 1 になります。2 番目の行には、2 の ROWNUM などが含まれます。

この機能を使用して、クエリーによって取得される行の数を制限できます。これを次の例で示します。

```

SELECT empno, ename, job FROM emp WHERE ROWNUM < 5;
* empno | ename |  job
-----+-----+-----
7369   | SMITH |  CLERK
7499   | ALLEN |  SALESMAN
7521   | WARD  |  SALESMAN
7566   | JONES |  MANAGER
(4 rows)

```

ROWNUM 値は、結果セットの並べ替えが行われる前に行に割り当てられます。したがって、結果セットは ORDER BY 句で指定されたオーダーで返されますが、次の例に示すように、ROWNUM 値は必ずしもオーダーであるとは限りません。

```

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

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 名前付けの新しい列がテーブルに追加され、次に UPDATE コマンドで seqno が ROWNUM に設定されます。

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

次の SELECT コマンドは、新しい seqno 値を示しています。

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

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

(17 rows)

3.4 Synonyms

</ div>

- synonym *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リファレンス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被参照されるためにスキーマ名による完全修飾を必要とする場合に役立ちます。そのオブジェクトに対して定義された同義語は、単一の非修飾名へのリファレンスを単純化します。

Advanced Server は、次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
- シーケンス
- 手順
- 機能
- タイプ
- データベースリンクを介してアクセス可能なオブジェクト
- 他の同義語

シノニムを作成する時点では、被参照スキーマも被参照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指す場合があります。被参照オブジェクトまたはスキーマを削除すると、シノニムは無効になります。シノニムを明示的に削除して削除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、Advanced Server は検索パスを使用して、修飾されていない同義語名を解決します。名前の同義語が2つある場合、同義語への非修飾リファレンスは、検索パス内の指定された名前を持つ最初の同義語に解決されます。public が検索パスにある場合、その名前を修飾せずに、そのスキーマ内の同義語を参照できます。

Advanced Server が SQL コマンドを実行すると、現在のユーザの権限がシノニムの基になるデータベースオブジェクトと照合されます。ユーザ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同義語を作成します。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：

syn_name

syn_name は同義語の名前です。シノニム名前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。スキーマ名前を指定しない場合、検索パスの最初の既存のスキーマにシノニムが作成されます。

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へのアクセスに使用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 スキーマに同義語を作成するには、PUBLIC 句を含めます。オラクルデータベースと互換性がある CREATE PUBLIC SYNONYM コマンドは、public スキーマに存在する同義語を作成します。

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

これは書くための簡単な方法です：

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

次の例では、enterprisedb.emp テーブルを参照する personnel 名前付けのシノニムを作成します。

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

シノニムが CREATE SYNONYM コマンドで修飾されたスキーマでない限り、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、検索パスをビューできます。

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

この例では、development 名前付けのスキーマが存在しない場合、同義語は accounting 名前付けのスキーマに作成されます。

現在、enterprisedb スキーマの emp テーブルは、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）で被参照できます。

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

SELECT * FROM personnel;
* empno | ename   | job      | mgr | hiredate       | sal  | comm | deptno
```

(次のページに続く)

(前のページからの続き)

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、コマンド `DROP SYNONYM` 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：

syn_name

syn_name は同義語の名前です。シノニム名前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。

スキーマで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 つのシノニムがある場合があります。削除するシノニムの名前を明確にするには、スキーマ名前を含めます。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、Advanced Server は検索パスで見つかっ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` スキーマにある同義語を削除できます。オラクルデータベースと互換性がある `DROP PUBLIC SYNONYM` コマンドは、`public` スキーマにある同義語を削除します。

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

次の例では、同義語 `personnel` を削除します。

```
DROP SYNONYM personnel;
```

3.5 Hierarchical Queries

</ div>

- 階層クエリーは、親子リレーションを形成するデータに基づいて、結果セットの行を階層オーダーに返すクエリーの一種です。通常、階層は逆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、またはマルチプルの子ノードに接続できます。各ノードは、親を持たない最上位ノードを除き、1つの親ノードに接続されます。このノードはルートノードです。各ツリーには、1つのルートノードがあります。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ている些細なケースです。この場合、ルートとリーフの両方です。

階層クエリーでは、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。

!!! Note * 1つの指定された行が複数のツリーに表示され、結果セットに複数回表示される可能性があります。

クエリーの階層リレーションは、結果セットで行が返されるオーダーの基礎となる `CONNECT BY` 句によって記述されます。`CONNECT BY` 句とそれに関連するオプションの句が `SELECT` コマンドで現れる場所のコンテキストを以下に示します。

```
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、結果セットの行の元になる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間のリレーションを指定します。

- `PRIOR` キーワードは、`CONNECT BY` 句の条件内の1つの式のみで指定できます。階層クエリーの親行を参照するには、右式または左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、`PRIOR` キーワードが必要な条件は1つだけですが、`PRIOR` 条件はマルチプル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リレーションの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、ターゲットリスト内の式に対応する各レコードに疑似列を追加できます。疑似列は、`PRIOR` 式を使用して子クエリーで被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 演算子を使用して、CONNECT BY [PRIOR] 条件の機能をさらに強化できます。この演算子は、直接の親行だけでなく、階層内のすべての祖先行も返します。詳細については、[Retrieving the Root Node with CONNECT_BY_ROOT](#) を参照してください。

例

次の例では、CONNECT BY 句と PRIOR を使用して、従業員とマネージャーのリレーションを定義します。LEVEL 値を使用して、従業員名をインデントして、行の階層の深さをさらに強調します。

```
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;
```

このクエリーの出力は次のとおりです。

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

(28 rows)

次の例では、ORDER BY 句と、CONNECT BY 句に AND 演算子を含む複合条件を追加して、従業員とマネージャの階層を表示します。

```
SELECT LPAD(ename,length(ename)+2 *(LEVEL - 1),' ') AS employee, empno, mgr, LEVEL
FROM emp
CONNECT BY PRIOR empno = mgr AND LEVEL <= 3
ORDER BY LEVEL, ename;
```

このクエリーの出力は次のとおりです。

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

(次のページに続く)

(前のページからの続き)

*	SMITH		7369		7902		3
*	TURNER		7844		7698		3
*	WARD		7521		7698		3
(37 rows)							

次の例では、PRIOR キーワードを使用して、各従業員の従業員名前とマネージャ名前を結果ます。

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

このクエリーの出力は次のとおりです。

```
ename      | mgr
-----+-----
ADAMS      | SCOTT
ALLEN      | BLAKE
BLAKE      | KING
CLARK      | KING
FORD       | JONES
JAMES      | BLAKE
JONES      | KING
KING       |
MARTIN     | BLAKE
MILLER     | CLARK
SCOTT      | JONES
SMITH      | FORD
TURNER     | BLAKE
WARD       | BLAKE
(14 rows)
```

次の例では、オプションの PRIOR ステートメントとともに CONNECT BY 句を使用して、レベル値を表示します。

```
SELECT LEVEL
FROM DUAL
CONNECT BY LEVEL <= 5;
```

このクエリーの出力は次のとおりです。

```
level
-----
*      1
*      2
*      3
*      4
*      5
```

(次のページに続く)

(前のページからの続き)

(5 rows)

階層クエリ、START WITH、CONNECT BY、および ORDER SIBLINGS BY に関連する句については、次のセクションで説明します。

3.5.1 Defining the Parent/Child Relationship

</ div>

特定の行について、そ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、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。

!!! Note * WHERE 句が `table_expression` に適用される前に、行で行が子ノードであるかどうかを判断する評価プロセスが発生します。

前のステップで見つかった各子ノードを親として処理するこのプロセスを繰り返し繰り返すことにより、ノードの逆ツリーが構築されます。子ノードの最終セットにそれ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、前述のアルゴリズムが適用される最初の親ノードである行を決定します。これについては、次のセクションでさらに説明します。

3.5.2 Selecting the Root Nodes

</ div>

START WITH 節は、ルートノードとして使用される `table_expression` によって選択された行を決定するために使用されます。`start_expression` が true と評価される `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、結果セット内の潜在的なツリーの数、ルートノードの数に等しくなります。結果として、START WITH 句が省略された場合、`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。

3.5.3 Organization Tree in the Sample Application

</ div>

サンプルアプリケーションの **emp** テーブルを検討してください。 **emp** テーブルの行は、従業員のマネージャの従業員番号を含む **mgr** 列に基づいて階層をフォームします。各従業員には、最大で 1 人のマネージャがいます。 **KING** は会社の社長であるため、マネージャがないため、 **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として行動することも可能です。このリレーションは、以下に示すように、典型的なツリー構造の階層組織図を形成します。

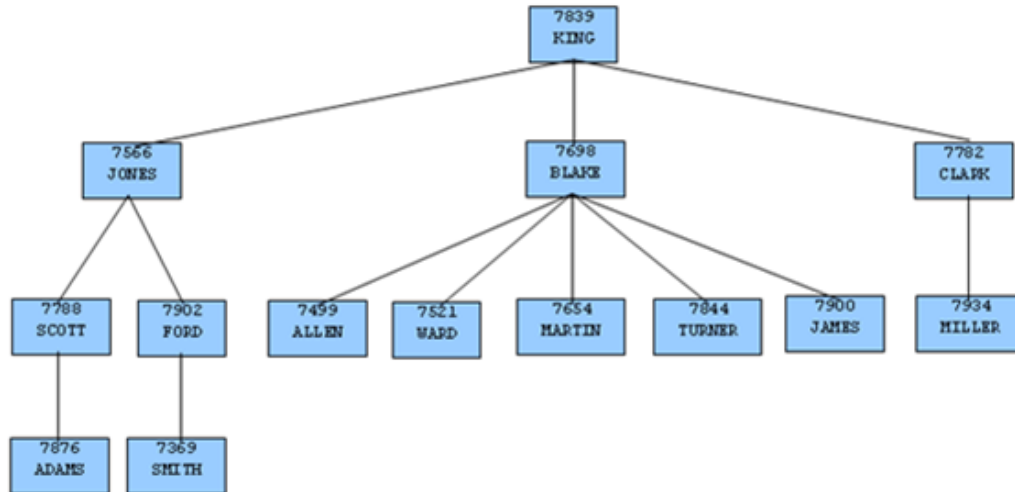


図 1: Employee Organization Hierarchy

図 1：従業員の組織階層</ div>

このリレーションに基づいて階層クエリーを作成するには、**SELECT** コマンドに **CONNECT BY PRIOR empno = mgr** 句を含めます。例、従業員番号 7839 の社長である **KING** が与えられた場合、**mgr** 列が 7839 である従業員は、**JONES**、**BLAKE** および **CLARK**（これらは **KING** の子ノード）に該当する **KING** に直接報告します。同様に、従業員 **JONES** の場合、**mgr** 列が 7566 に等しい他の従業員は **JONES** の子ノードです。この例では、これらは **SCOTT** と **FORD** です。

組織図の最上部は **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、**KING** のみを初期ルートノードとして選択します。

完全な **SELECT** コマンドを以下に示します。

```

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

```

クエリー出力の行は、ルートからリーフへの各ブランチを上から下、左から右のオーダーに移動します。以下は、このクエリーからの出力です。

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)		

3.5.4 Node Level

</ div>

LEVEL は疑似列であり、SELECT コマンドで列を表示できる場所であればどこでも使用できます。結果セットの行について、LEVEL は、この行で表される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。

次のクエリーは、LEVEL 疑似列が追加された前のクエリーの修正です。さらに、LEVEL 値を使用して、従業員名をインデントして、行の階層の深さをさらに強調します。

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

このクエリーの出力は次のとおりです。

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

(次のページに続く)

(前のページからの続き)

* 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。例、上記の出力では、従業員 ALLEN, WARD, MARTIN, TURNER および JAMES は兄弟であり、それらはすべてレベル 3 であり、親 BLAKE. JONES, BLAKE および CLARK はレベル 2 であり、KING は共通の親であるため兄弟です。

3.5.5 Ordering the Siblings

</div>

結果セットは、ORDER SIBLINGS BY 句を使用して選択した列の値によって兄弟がオーダーまたは降順で表示されるように順序付けることができます。これは、階層クエリでのみ使用できる ORDER BY 句の特殊なケースです。

前のクエリーは、ORDER SIBLINGS BY ename ASC を追加してさらに変更されています。

```
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;
```

前のクエリーの出力が変更され、兄弟が名前のオーダーで表示されるようになりました。兄弟 BLAKE, CLARK および JONES は、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 ALLEN, 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。

level		employee		empno		mgr
-----+-----+-----+-----						
* 1		KING		7839		
* 2		BLAKE		7698		7839
* 3		ALLEN		7499		7698
* 3		JAMES		7900		7698
* 3		MARTIN		7654		7698
* 3		TURNER		7844		7698
* 3		WARD		7521		7698
* 2		CLARK		7782		7839
* 3		MILLER		7934		7782
* 2		JONES		7566		7839
* 3		FORD		7902		7566
* 4		SMITH		7369		7902

(次のページに続く)

(前のページからの続き)

```
* 3 | SCOTT | 7788 | 7566
* 4 | ADAMS | 7876 | 7788
(14 rows)
```

この最後の例では、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フォームします。

```
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。

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

3.5.6 Retrieving the Root Node with CONNECT_BY_ROOT

</div>

CONNECT_BY_ROOT は単項演算子で、現在の行リレーションルートノードと見なされる行の列の値を結果オーダーに列を修飾するために使用できます。

!!! Note * 単項演算子は、CONNECT_BY_ROOT の場合は CONNECT_BY_ROOT キーワードに続く列名前である単一のオペランドで動作します。

SELECT リストのコンテキストでは、CONNECT_BY_ROOT 演算子は次のように表示されます。

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

CONNECT_BY_ROOT 演算子に関するノート点を次に示します。

- SELECT コマンドが階層クエリーである限り、CONNECT_BY_ROOT 演算子は、SELECT リスト、WHERE 句、GROUP BY 句、HAVING 句、ORDER BY 句、および ORDER SIBLINGS BY 句で使用できます。

- CONNECT_BY_ROOT 演算子は、階層クエリーの CONNECT BY 句または START WITH 句では使用できません。
- CONNECT_BY_ROOT を列を含む式に適用することは可能ですが、そのためには式をカッコで囲む必要があります。

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ている各従業員のルートノードの従業員番号と従業員名前を結果方法を示しています。

```
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ることに注意してください。

level	employee	empno	mgr	mgr empno	mgr ename
* 1	BLAKE	7698	7839	7698	BLAKE
* 2	ALLEN	7499	7698	7698	BLAKE
* 2	JAMES	7900	7698	7698	BLAKE
* 2	MARTIN	7654	7698	7698	BLAKE
* 2	TURNER	7844	7698	7698	BLAKE
* 2	WARD	7521	7698	7698	BLAKE
* 1	CLARK	7782	7839	7782	CLARK
* 2	MILLER	7934	7782	7782	CLARK
* 1	JONES	7566	7839	7566	JONES
* 2	FORD	7902	7566	7566	JONES
* 3	SMITH	7369	7902	7566	JONES
* 2	SCOTT	7788	7566	7566	JONES
* 3	ADAMS	7876	7788	7566	JONES

(13 rows)

以下は同様のクエリーですが、mgr 列が null である単一の最上位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は、この特定のクエリーのルートとして KING を示しています。

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、次の例では `START WITH` 句を省略し、14 のツリーを作成しています。

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

以下は、クエリーからの出力です。リーフノードも独自のツリーの最上部をフォームするため、各ノードは、`mgr empno` 列と `mgr ename` 列の下にルートノードとして少なくとも 1 回表示されます。

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

(次のページに続く)

(前のページからの続き)

*	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)

以下は、CONNECT_BY_ROOT の単項演算子の効果を示しています。この例に示すように、カッコで囲まれていない式に適用すると、CONNECT_BY_ROOT 演算子は、直後の用語 `ename` のみに影響します。|| 'manages' || `ename` の後続の結合は CONNECT_BY_ROOT オペレーションのパートではないため、`ename` の 2 回目の出現は現在処理されている行の値になり、`ename` の最初の出現はルートノードからの値になります。

```
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;
```

以下は、クエリーからの出力です。top mgr/employee 列の下に生成された値に注意してください。

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

(次のページに続く)

(前のページからの続き)

*	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)

次の例では、カッコで囲まれた式で CONNECT_BY_ROOT 演算子を使用しています。

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || 'ename "employee", empno, mgr,
CONNECT_BY_ROOT ('Manager ' || ' || ' 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;
```

以下は、クエリーの出力です。ename と empno の両方の値は CONNECT_BY_ROOT 演算子の影響を受けるため、top mgr/empno 列の下に示されているようにルートノードから値を結果ことに注意してください。

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

(13 rows)

3.5.7 Retrieving a Path with SYS_CONNECT_BY_PATH

</ div>

SYS_CONNECT_BY_PATH は、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階層クエリー内で機能するファンクションです。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>)

このファンクションは 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内にある列の名前です。

delimiter は、指定された列の各エントリーを区切る **varchar** 値です。

次の例では、従業員名とそのマネージャーのリストを返します。マネージャにマネージャがある場合、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;
* level | ename |      managers
-----+-----+-----
*    1 | KING  | /KING
*    2 | BLAKE | /KING/BLAKE
*    2 | CLARK | /KING/CLARK
*    2 | JONES | /KING/JONES
*    3 | ALLEN | /KING/BLAKE/ALLEN
*    3 | FORD  | /KING/JONES/FORD
*    3 | JAMES | /KING/BLAKE/JAMES
*    3 | MARTIN | /KING/BLAKE/MARTIN
*    3 | MILLER | /KING/CLARK/MILLER
*    3 | SCOTT | /KING/JONES/SCOTT
*    3 | TURNER | /KING/BLAKE/TURNER
*    3 | WARD  | /KING/BLAKE/WARD
*    4 | ADAMS | /KING/JONES/SCOTT/ADAMS
*    4 | SMITH | /KING/JONES/FORD/SMITH
(14 rows)
```

結果セット内：

- **level** 列には、クエリーが返したレベルの数が表示されます。
- **ename** 列には従業員名前が表示されます。
- **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 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 内の SYS_CONNECT_BY_PATH

3.6 Multidimensional Analysis

</ div>

- 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調べる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るプロセスを指します。ディメンションは、時間、地理、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。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、カウント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一連のディメンションに従ってこれらのファクトを取得オーダーには、通常、SQL 集計が使用されます。* SQL 集計*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のデータのカウン、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。

SQL SELECT コマンドの GROUP BY 句は、集計結果の生成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
- CUBE 拡張機能
- GROUPING SETS 拡張

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

!!! Note * この説明では、サンプルの dept および emp テーブルを広範囲に使用して、使用例を示します。これらのテーブルには、より有益な結果を提供するために、次の変更が適用されました。

```
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 テーブルと dept テーブルの結合からの次の行が使用されます。

```
SELECT loc, dname, job, empno FROM emp e, dept d
WHERE e.deptno = d.deptno
ORDER BY 1, 2, 3, 4;
*   loc      |   dname      |   job      | empno
-----+-----+-----+-----
BOSTON      | OPERATIONS  | ANALYST    | 9002
BOSTON      | OPERATIONS  | CLERK      | 9001
BOSTON      | OPERATIONS  | MANAGER    | 9003
```

(次のページに続く)

(前のページからの続き)

BOSTON	RESEARCH	ANALYST	7788
BOSTON	RESEARCH	ANALYST	7902
BOSTON	RESEARCH	CLERK	7369
BOSTON	RESEARCH	CLERK	7876
BOSTON	RESEARCH	MANAGER	7566
CHICAGO	SALES	CLERK	7900
CHICAGO	SALES	MANAGER	7698
CHICAGO	SALES	SALESMAN	7499
CHICAGO	SALES	SALESMAN	7521
CHICAGO	SALES	SALESMAN	7654
CHICAGO	SALES	SALESMAN	7844
NEW YORK	ACCOUNTING	CLERK	7934
NEW YORK	ACCOUNTING	MANAGER	7782
NEW YORK	ACCOUNTING	PRESIDENT	7839

(17 rows)

loc, dname および job 列は、例で使用される SQL 集計のディメンションに使用されます。結果の集計のファクトは、COUNT(*) ファンクションを使用して取得した従業員の数です。

loc, dname 列と job 列をグループ化する基本的なクエリーは、次のとおりです。

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

拡張のない基本的な GROUP BY 句を使用したこの結果セットの行は、* base aggregate *行と呼ばれます。

loc	dname	job	employees
BOSTON	OPERATIONS	ANALYST	1
BOSTON	OPERATIONS	CLERK	1
BOSTON	OPERATIONS	MANAGER	1
BOSTON	RESEARCH	ANALYST	2
BOSTON	RESEARCH	CLERK	2
BOSTON	RESEARCH	MANAGER	1
CHICAGO	SALES	CLERK	1
CHICAGO	SALES	MANAGER	1
CHICAGO	SALES	SALESMAN	4
NEW YORK	ACCOUNTING	CLERK	1
NEW YORK	ACCOUNTING	MANAGER	1
NEW YORK	ACCOUNTING	PRESIDENT	1

(12 rows)

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

GROUPING SETS 拡張機能は、異なるタイプのグループ化を単一の結果セットに結合する機能を提供します。

GROUPING および GROUPING_ID 関数は、結果セットの解釈に役立ちます。

これらの拡張機能によって提供される追加機能については、以降のセクションで詳しく説明します。

3.6.1 ROLLUP Extension

</div>

ROLLUP 拡張機能は、各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。階層のオーダーは、ROLLUP 式リストで指定された式のオーダーによって決まります。階層の最上部はリストの一番左のアイテムです。右に進むそれぞれの連続するアイテムは、右端のアイテムが最下位レベルである階層を下に移動します。

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ている場合、**expr_1a** と **expr_1b** によって返される値の組み合わせは、階層の単一のグループ化レベルを定義します。

結果セットで返される集約の基本レベルは、式リストによって返される値の一意の組み合わせごとです。

さらに、一意の値ごとに、リストの最初のアイテム (**expr_1** または (**expr_1a**, **expr_1b**, ...) の組み合わせのいずれかが指定されている方) の小計が返されます。リスト内の 2 番目のアイテム (**expr_2** または (**expr_2a**, **expr_2b**, ...) の組み合わせのいずれかが指定された方) の小計は、最初のアイテムの各グループ化内など、一意の値ごとに返されます。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、小計が取得されるアイテムに対して null が返されます。

GROUP BY 句のコンテキスト内で指定された ROLLUP 拡張機能は、次のように表示されます。

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

select_list で指定されたアイテムは、ROLLUP **expression_list** にも表示される必要があります。または、COUNT, SUM, AVG, MIN, や MAX などの集計関数でなければなりません。または、結果値がグループ内の個々の行に依存しない定数または関数でなければなりません (例、SYSDATE ファンクション)。

GROUP BY 句では、マルチプルの ROLLUP 拡張機能と、他の GROUP BY 拡張機能および個々の式のマルチプルのオカレンスを指定できます。

出力を階層構造またはその他の意味のある構造で表示する場合は、ORDER BY 句を使用する必要があります。ORDER BY 句が指定されていない場合、結果セットのオーダーに関する保証はありません。

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 式リスト内のアイテムの数を表します。括弧で囲まれたリストは 1 つのアイテムとしてカウントされます。

次のクエリーは、列 **loc**, **dname**、次に **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;

```

以下はクエリーの結果です。loc, dname と job の一意の組み合わせごとの従業員数のカウントと、loc と dname の一意の組み合わせごとの小計、loc の一意の各値、および最終行に表示される総計があります。

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

(20 rows)

次のクエリーは、ROLLUP リスト内のアイテムをカッコ内で結合した結果を示しています。

```

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;

```

出力では、前の例のように loc と dname の組み合わせの小計がないことにノートしてください。

loc	dname	job	employees
BOSTON	OPERATIONS	ANALYST	1
BOSTON	OPERATIONS	CLERK	1
BOSTON	OPERATIONS	MANAGER	1

(次のページに続く)

(前のページからの続き)

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 列が括弧で囲まれている場合、小計レベルも異なります。

```
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;
```

現在、一意の loc と dname の組み合わせごとに小計がありますが、loc の一意の値には小計がありません。

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)

3.6.2 CUBE Extension

</div>

CUBE 拡張機能は、ROLLUP 拡張機能に似ています。ただし、ROLLUP は、グループを作成し、ROLLUP 式リストのアイテムの左から右のリストに基づいて階層を生成しますが、CUBE は、CUBE 式リストのすべてのアイテムのすべての順列に基づいてグループ化および小計を生成します。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。

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ている場合、**expr_1a** と **expr_1b** によって返される値の組み合わせは、単一のグループを定義します。

結果セットで返される集約の基本レベルは、式リストによって返される値の一意の組み合わせごとです。

さらに、一意の値ごとに、リストの最初のアイテム (**expr_1** または (**expr_1a**, **expr_1b**, ...)) の組み合わせのいずれかが指定されている方) の小計が返されます。リスト内の 2 番目のアイテム (**expr_2** または (**expr_2a**, **expr_2b**, ...)) の組み合わせのいずれかが指定された方) の一意の値ごとに小計が返されます。最初のアイテムと 2 番目のアイテムの一意の組み合わせごとに小計も返されます。同様に、3 番目のアイテムの一意の各値、3 番目のアイテムの一意の値と 1 番目のアイテムの組み合わせ、3 番目のアイテムの一意の値と 2 番目のアイテムの組み合わせ、および一意の値ごとに小計が返されアイテム 3 番目のアイテム、2 番目のアイテム、1 番目のアイテムの組み合わせ。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、小計が取得されるアイテムに対して null が返されます。

GROUP BY 句のコンテキスト内で指定された CUBE 拡張機能は、次のように表示されます。

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

select_list で指定されたアイテムは、CUBE **expression_list** にも表示される必要があります。または、COUNT, SUM, AVG, MIN や MAX などの集計関数でなければなりません。または、結果値がグループ内の個々の行に依存しない定数または関数でなければなりません (例、SYSDATE ファンクション)。

GROUP BY 句では、マルチプルの CUBE 拡張機能と、他の GROUP BY 拡張機能および個々の式のマルチプルのオカレンスを指定できます。

出力を意味のある構造で表示する場合は、ORDER BY 句を使用する必要があります。ORDER BY 句が指定されていない場合、結果セットのオーダーに関する保証はありません。

グループ化レベルまたは合計の数は 2^n の累乗です。ここで、 n は CUBE 式リスト内のアイテムの数を表します。括弧で囲まれたリストは 1 つのアイテムとしてカウントされます。

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

(次のページに続く)

(前のページからの続き)

```
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 と job の各組み合わせの従業員数のカウントと、loc と dname の各組み合わせの小計、loc と job の各組み合わせ、dname と job の各組み合わせ、各 unique loc の固有値、dname の固有値、job の固有値、および最後の行に表示される総計。

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

(次のページに続く)

(前のページからの続き)

*	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 リスト内のアイテムをかつこ内で結合した結果を示しています。

```
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;
```

出力では、ノートと dname の組み合わせ、loc と job の組み合わせ、または dname 自分自身、または job 自分自身の組み合わせに関する小計はありません。

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

(次のページに続く)

(前のページからの続き)

*	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 拡張の外側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 の各グループ化内の dname および job に対して置換が実行されます。

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

(次のページに続く)

(前のページからの続き)

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)

3.6.3 GROUPING SETS Extension

</ div>

GROUP BY 句内で GROUPING SETS 拡張機能を使用すると、実際には異なるグループ化に基づいてマルチプルの結果セットを結合した 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 オペレーションは、マルチプルのグループ化の結果セットを 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、したがって GROUPING SETS 拡張では、結合されている結果セットから重複行が削除されないことに注意してください。

単一の GROUPING SETS 拡張の構文は次のとおりです。

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

GROUPING SETS 拡張には、1 つ以上のコンマ区切り式、括弧で囲まれた式のリスト、ROLLUP 拡張、および CUBE 拡張の任意の組み合わせを含めることができます。

GROUPING SETS 拡張機能は、次のように GROUP BY 句のコンテキスト内で指定されます。

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

select_list で指定されたアイテムは、GROUPING SETS expression_list にも表示される必要があります。または、COUNT、SUM、AVG、MIN や MAX などの集計関数でなければなりません。または、結果値がグループ内の個々の行に依存しない定数または関数でなければなりません（例、SYSDATE ファンクション）。

GROUP BY 句では、マルチプルの GROUPING SETS 拡張機能と、他の GROUP BY 拡張機能および個々の式のマルチプルのオカレンスを指定できます。

出力を意味のある構造で表示する場合は、ORDER BY 句を使用する必要があります。ORDER BY 句が指定されていない場合、結果セットのオーダーに関する保証はありません。

次のクエリーは、loc、dname 列と job 列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。

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 の出力と同じです。

loc	dname	job	employees
BOSTON			8
CHICAGO			6
NEW YORK			3

(次のページに続く)

(前のページからの続き)

* *	ACCOUNTING		3
* *	OPERATIONS		3
* *	RESEARCH		5
* *	SALES		6
* *		ANALYST	3
* *		CLERK	5
* *		MANAGER	4
* *		PRESIDENT	1
* *		SALESMAN	4

(12 rows)

次の例は、さまざまなタイプの GROUP BY 拡張機能を GROUPING SETS 式リスト内で一緒に使用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、このクエリーからの出力です。

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

(次のページに続く)

(前のページからの続き)

*	*		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) から個別に生成される結果セットの結合です。これらの個々のクエリを以下に示します。

```
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;
```

以下は、GROUP BY loc 句の結果セットです。

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;
```

以下は、GROUP BY ROLLUP (dname, job) 句の結果セットです。

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;
```

以下は、GROUP BY CUBE (job, loc) 句の結果セットです。

loc	dname	job	employees
-----+-----+-----+-----			
BOSTON		ANALYST	3
BOSTON		CLERK	3
BOSTON		MANAGER	2
BOSTON			8
CHICAGO		CLERK	1
CHICAGO		MANAGER	1
CHICAGO		SALESMAN	4
CHICAGO			6
NEW YORK		CLERK	1
NEW YORK		MANAGER	1
NEW YORK		PRESIDENT	1
NEW YORK			3
*		ANALYST	3
*		CLERK	5

(次のページに続く)

(前のページからの続き)

*		MANAGER		4
*		PRESIDENT		1
*		SALESMAN		4
*				17
(18 rows)				

前の 3 つのクエリを UNION ALL 演算子と組み合わせると、3 つの結果セットの結合が作成されます。

```

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;

```

出力は次のとおりです。これは、GROUP BY GROUPING SETS (loc, ROLLUP (dname, job), CUBE (job, loc)) 句を使用した場合と同じです。

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

(次のページに続く)

(前のページからの続き)

* *	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)

3.6.4 GROUPING Function

</ div>

GROUP BY 句に ROLLUP, CUBE, or 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る小計のさまざまなレベルと、結果セットのベース集計行を区別することが困難な場合があります。GROUPING ファンクションは、この区別を行う手段を提供します。

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

GROUPING ファンクションは、ROLLUP, CUBE、または GROUP BY 句の 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る単一のパラメータを取ります。

GROUPING ファンクションの結果値は 0 または 1 です。クエリーの結果セットで、GROUPING ファンクションで指定された列式が null の場合、行はその列のマルチプル値の小計を表すため、GROUPING ファンクションは戻ります。1 の値。行が 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 でも非 null でもかまいま

せん。、いずれにしても、その列の特定の値に対するものであり、マルチプルの値にわたる小計ではありません。

次のクエリーは、GROUPING ファンクションの結果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 関数の出力を表示する右端の 3 つの列では、対応する列の値にわたって小計が取られている場合、小計行に値 1 が表示されます。

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

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。例、前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して、loc と 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",
```

(次のページに続く)

(前のページからの続き)

```

* 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。

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、小計行をベース集計行または式リストのアイテムの 1 つが式の基になっている列の結果として null を返す特定の小計行と区別することができ null。列の小計を表すのではなく、表の行。

この点を説明するために、次の行が emp テーブルに追加されます。これにより、job 列に NULL 値を持つ行が提供されます。

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

次のクエリーは、わかりやすくするために行数を減らして発行されています。

```

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 列の BOSTON 列と job 列のスペース（表の 4 番目と 5 番目のエントリ）を含む 2 行が含まれていることに注意してください。

loc	job	employees	gf_loc	gf_job
BOSTON	ANALYST	3	0	0
BOSTON	CLERK	3	0	0

(次のページに続く)

(前のページからの続き)

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)				

job 列 (gf_job) の GROUPING ファンクションが 1 を返す 5 行は、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 列に小計値 9 が含まれていることに注意してください。

job 列と loc 列の GROUPING ファンクションが 0 を返す 4 番目の行は、これが loc が BOSTON で job が null であるすべての行のベース集約であることを示し例。employees 列には 1 が含まれます。これは、挿入されたそのような単一の行のカウントです。

また、9 番目の行（最後の隣）では、ノート列の GROUPING ファンクションが 0 を返し、loc 列の GROUPING ファンクションが 1 を返します。これは、これが job 列が null この例で挿入された単一行のカウント。

3.6.5 GROUPING_ID Function

</ div>

GROUPING_ID ファンクションは、ROLLBACK, CUBE または GROUPING SETS 拡張からの結果セット内の行の小計レベルを決定するオーダーに、GROUPING ファンクションの単純化を提供します。

GROUPING ファンクションは 1 つの列式のみを取り、行が特定の列のすべての値の小計であるかどうかの表示を返します。したがって、マルチプルグループ化列をマルチプルクエリの小計のレベルを解釈するには、複数の GROUPING 関数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ている 1 つ以上の列式を受け入れ、これらの列の小計が集計されているかどうかを判別するために使用できる単一の整数を返します。

GROUPING_ID 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_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、ROLLUP, CUBE、または GROUP BY 句の 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る 1 つ以上のパラメーターを取ります。

GROUPING_ID ファンクションは整数値を返します。この値は、GROUPING_ID で指定されたパラメーターの順序付けと同じ左から右のオーダーで指定されたシリーズの GROUPING 関数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 10 進法の解釈に対応します。ファンクション

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 関数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、`gf_loc` 値と `gf_dname` 値で構成されるビットベクトルと、`gid` で指定された整数とのリレーションにノートしてください。

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

(12 rows)

次の表は、出力の 4 行の GROUPING ファンクションの結果値に基づく GROUPING_ID ファンクションの計算の具体例を示しています。

header 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 ファンクションの結果値が、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。

header 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

そのため、dname による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できます。

```
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;
```

以下はクエリーの結果です。

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

第4章 Profile Management

</div>

Advanced Server では、データベーススーパーユーザが名前付けプロファイル*を作成できます。各プロファイルは、password および md5 認証を強化するパスワード管理のルールを定義します。プロファイル内のルールは次のことができます。

- 失敗したログイン試行をカウントする
- ログイン試行の過剰な失敗によりアカウントをロックする
- パスワードに有効期限をマーク
- パスワードの有効期限後の猶予期間を定義する
- パスワードの複雑さのルールを定義する
- パスワードの再利用を制限するルールを定義する

プロファイルは、同等の認証要件を共有するロールのグループを簡単に管理できるパスワード属性の名前付けセットです。パスワード要件が変更された場合、プロファイルを変更して、そのプロファイルに関連付けられている各ユーザに新しい要件を適用できます。

プロファイルを作成した後、プロファイルを 1 人以上のユーザーに関連付けることができます。ユーザがサーバーに接続すると、サーバーはログインロールに関連付けられたプロファイルを適用します。プロファイルはクラスター内のすべてのデータベースで共有されますが、各クラスターにはマルチプルのプロファイルが存在する場合があります。マルチプルのデータベースにアクセスできる 1 人のユーザは、クラスター内の各データベースに接続するときに同じプロファイルを使用します。

Advanced Server は、代替プロファイルが指定されていない限り、ロールの作成時に新しいロールに関連付けられる default という名前付けのプロファイルを作成します。以前のサーババージョンから Advanced Server にアップグレードすると、既存のロールが default プロファイルに自動的に割り当てられます。default プロファイルを削除することはできません。

default プロファイルは、次の属性を指定します。

<i>FAILED_LOGIN_ATTEMPTS</i>	<i>UNLIMITED</i>	
<i>PASSWORD_LOCK_TIME</i>	<i>UNLIMITED</i>	
<i>PASSWORD_LIFE_TIME</i>	<i>UNLIMITED</i>	
<i>PASSWORD_GRACE_TIME</i>	<i>UNLIMITED</i>	
<i>PASSWORD_REUSE_TIME</i>	<i>UNLIMITED</i>	
<i>PASSWORD_REUSE_MAX</i>	<i>UNLIMITED</i>	
<i>PASSWORD_VERIFY_FUNCTION</i>	<i>NULL</i>	
<i>PASSWORD_ALLOW_HASHED</i>	<i>TRUE</i>	

データベーススーパーユーザは、ALTER PROFILE コマンドを使用して、default プロファイルで指定された値を変更できます。プロファイルの変更の詳細については、[Altering a Profile](#) を参照してください。

4.1 Creating a New Profile

</div>

CREATE PROFILE コマンドを使用して、新しいプロファイルを作成します。構文は次のとおりです。

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

LIMIT 句と 1 つ以上のスペースで区切られた parameter / value のペアを含めて、Advanced Server によって実施されるルールを指定します。

パラメータ：

profile_name は、プロファイルの名前を指定します。

parameter は、プロファイルによって制限される属性を指定します。

value は、パラメータの制限を指定します。

Advanced Server は、各 parameter に対して以下に示す value をサポートしています。

FAILED_LOGIN_ATTEMPTS は、サーバーが PASSWORD_LOCK_TIME で指定された長さ、ユーザをアカウントからロックアウト make までにユーザが試行できるログイン試行の失敗回数を指定します。サポートされる値は次のとおりです。

- 0 より大きい INTEGER 値。
- DEFAULT-DEFAULT プロファイルで指定された FAILED_LOGIN_ATTEMPTS の値。

- **UNLIMITED** –接続ユーザは、ログイン試行の失敗回数に制限はありません。

PASSWORD_LOCK_TIME は、**FAILED_LOGIN_ATTEMPTS** によってロックされたアカウントをサーバーがロック解除するまでに経過する必要がある時間の長さを指定します。サポートされる値は次のとおりです。

- 0 以上の **NUMERIC** 値。日の 10 進数部分を指定するには、小数値を指定します。例、値 4.5 を使用して、4 日、12 時間を指定します。
- **DEFAULT-DEFAULT** プロファイルで指定された **PASSWORD_LOCK_TIME** の値。
- **UNLIMITED** –アカウントは、データベーススーパーユーザによって手動でロック解除されるまでロックされます。

PASSWORD_LIFE_TIME は、ユーザが新しいパスワードの入力を求められるまでに現在のパスワードを使用できる日数を指定します。**PASSWORD_LIFE_TIME** 句を使用して、パスワードによる有効期限が切れてからロールによる接続が拒否されるまでの日数を指定する場合は、**PASSWORD_GRACE_TIME** 句を含めます。**PASSWORD_GRACE_TIME** が指定されていない場合、パスワードは **PASSWORD_GRACE_TIME** のデフォルト値で指定された日にエキスパイアし、ユーザは新しいパスワードが提供されるまでコマンドの実行を許可されません。サポートされる値は次のとおりです。

- 0 以上の **NUMERIC** 値。日の 10 進数部分を指定するには、小数値を指定します。例、値 4.5 を使用して、4 日、12 時間を指定します。
- **DEFAULT-DEFAULT** プロファイルで指定された **PASSWORD_LIFE_TIME** の値。
- **UNLIMITED** –パスワードには有効期限がありません。

PASSWORD_GRACE_TIME は、パスワードの有効期限が切れてからユーザがパスワードの変更を強制されるまでの猶予期間の長さを指定します。猶予期間が終了すると、ユーザは接続を許可されますが、期限切れのパスワードを更新するまでコマンドの実行は許可されません。サポートされる値は次のとおりです。

- 0 以上の **NUMERIC** 値。日の 10 進数部分を指定するには、小数値を指定します。例、値 4.5 を使用して、4 日、12 時間を指定します。
- **DEFAULT-DEFAULT** プロファイルで指定された **PASSWORD_GRACE_TIME** の値。
- **UNLIMITED** –猶予期間は無限です。

PASSWORD_REUSE_TIME は、ユーザがパスワードを再利用するまで待機する日数を指定します。**PASSWORD_REUSE_TIME** パラメーターと **PASSWORD_REUSE_MAX** パラメーターは、一緒に使用することを目的としています。これらのパラメーターの一方に **UNLIMITED** を指定して、一方のパラメーターに有限値を指定すると、古いパスワードを再利用できなくなります。両方のパラメーターが **UNLIMITED** に設定されている場合、パスワードの再利用に関する制限はありません。サポートされる値は次のとおりです。

- 0 以上の **NUMERIC** 値。日の 10 進数部分を指定するには、小数値を指定します。例、値 4.5 を使用して、4 日、12 時間を指定します。
- **DEFAULT-DEFAULT** プロファイルで指定された **PASSWORD_REUSE_TIME** の値。
- **UNLIMITED** –パスワードは制限なしで再利用できます。

PASSWORD_REUSE_MAX は、パスワードを再利用する前に発生する必要があるパスワード変更の数を指定します。**PASSWORD_REUSE_TIME** パラメーターと **PASSWORD_REUSE_MAX** パラメーターは、一緒に使用することを目的としています。これらのパラメーターの一方に **UNLIMITED** を指定して、一方のパラメーターに有

限値を指定すると、古いパスワードを再利用できなくなります。両方のパラメーターが **UNLIMITED** に設定されている場合、パスワードの再利用に関する制限はありません。サポートされる値は次のとおりです。

- 0 以上の **INTEGER** 値。
- **DEFAULT-DEFAULT** プロファイルで指定された **PASSWORD_REUSE_MAX** の値。
- **UNLIMITED** –パスワードは制限なしで再利用できます。

PASSWORD_VERIFY_FUNCTION はパスワードの複雑さを指定します。サポートされる値は次のとおりです。

- **PL/SQL** ファンクションの名前。
- **DEFAULT-DEFAULT** プロファイルで指定された **PASSWORD_VERIFY_FUNCTION** の値。
- **NULL**

PASSWORD_ALLOW_HASHED は、暗号化されたパスワードの使用を許可するかどうかを指定します。値を **TRUE** として指定すると、システムは、クライアントサイドでハッシュ計算された暗号化パスワードを指定することにより、ユーザがパスワードを変更できるようにします。ただし、値を **FALSE** として指定する場合、効果的に検証オーダーにはパスワードをプレーンテキストフォームで指定する必要があります。そうしないと、サーバーが暗号化されたパスワードを受信した場合にエラーがスローされます。サポートされる値は次のとおりです。

- **BOOLEAN** 値 **TRUE/ON/YES/1** または **FALSE/OFF/NO/0**。
- **DEFAULT-DEFAULT** プロファイルで指定された **PASSWORD_ALLOW_HASHED** の値。

!!! Note * **-PASSWORD_ALLOW_HASHED** は Oracle 互換パラメータではありません。 **-DROP PROFILE** コマンドを使用して、プロファイルを削除します。

例

次のコマンドは、**acctg** 名前付けのプロファイルを作成します。このプロファイルは、ユーザが 5 回の試行で正しいパスワードで認証されなかった場合、アカウントが 1 日間ロックされることを指定します。

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

次のコマンドは、**sales** 名前付けのプロファイルを作成します。このプロファイルは、ユーザが 90 日ごとにパスワードを変更する必要があることを指定しています。

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

プロファイルで指定された 90 日が経過する前にユーザがパスワードを変更していない場合、ログインにワーニングが発行されます。3 日間の猶予期間が過ぎると、アカウントはパスワードを変更するまでコマンドの呼び出しを許可されません。

次のコマンドは、**accts** 名前付けのプロファイルを作成します。このプロファイルは、ユーザーがパスワードを最後に使用してから 180 日以内にパスワードを再ユーザできないことを指定し、パスワードを再使用する前に少なくとも 5 回パスワードを変更する必要があるパスワード。

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

次のコマンドは、resources 名前付けのプロファイルを作成します。プロファイルは password_rules という名前付けのユーザー定義ファンクションを呼び出し、指定されたパスワードが複雑さの基準を満たしていることを確認します。

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

4.1.1 Creating a Password Function

</div>

PASSWORD_VERIFY_FUNCTION を指定する場合、ユーザーがパスワードを変更したときに適用されるセキュリティルールを指定するカスタマイズされたファンクションを提供できます。例、新しいパスワードの長さは少なくとも *n* 文字である必要があり、特定の値が含まれていないことを規定するルールを指定できます。

パスワードファンクションには次のシグネチャがあります。

```
<function_name> (<user_name> VARCHAR2,
*                <new_password> VARCHAR2,
*                <old_password> VARCHAR2) RETURN boolean
```

場所：

- user_name はユーザの名前です。
- new_password は新しいパスワード。
- old_password は、ユーザーの以前のパスワード。ファンクション内でこのパラメータをリファレンスする場合：

データベーススーパーユーザがパスワードを変更すると、3 番目のパラメータは常に NULL になります。

CREATEROLE 属性を持つユーザがパスワードを変更すると、ステートメントに REPLACE 句が含まれている場合、パラメータは前のパスワードを渡します。REPLACE 句は、CREATEROLE 権限を持つユーザのオプションの構文であることに注意してください。

データベーススーパーユーザではなく、CREATEROLE 属性を持たないユーザーがパスワードを変更すると、3 番目のパラメータにはロールの以前のパスワードが含まれます。

ファンクションはブール値を返します。ファンクションが true を返し、例外を発生させない場合、パスワードは受け入れられます。ファンクションが false を返すか、例外を発生させた場合、パスワードは拒否されます。ファンクションが例外を発生させた場合、指定されたエラーメッセージがユーザに表示されます。ファンクションが例外を発生させず、false を返す場合、次のエラーメッセージが表示されます。

ERROR: password verification for the specified password failed

ファンクションはデータベーススーパーユーザが所有し、sys スキーマに存在する必要があります。

例：

次の例では、プロファイルとカスタムファンクションを作成します。次に、ファンクションはプロファイルに関連付けられます。次の CREATE PROFILE コマンドは、acctg_pwd_profile 名前付けのプロファイルを作成します。

```
CREATE PROFILE acctg_pwd_profile;
```

次のコマンドは、verify_password 名前付けの（スキーマ修飾）ファンクションを作成します。

```
CREATE OR REPLACE FUNCTION sys.verify_password(user_name varchar2,
new_password varchar2, old_password varchar2)
RETURN boolean IMMUTABLE
IS
BEGIN
    * IF (length(new_password) < 5)
    * THEN
    *     raise_application_error(-20001, 'too short');
    * END IF;
    * IF substring(new_password FROM old_password) IS NOT NULL
    * THEN
    *     raise_application_error(-20002, 'includes old password');
    * END IF;
    * RETURN true;
END;
```

このファンクションは、最初にパスワードの長さが少なくとも 5 文字であることを確認してから、新しいパスワードと古いパスワードを比較します。新しいパスワードに含まれる文字が 5 文字未満の場合、または古いパスワードが含まれる場合、ファンクションはエラーを発生させます。

次の文は、verify_password ファンクションの所有権を enterprisedb データベーススーパーユーザに設定します。

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

次に、verify_password ファンクションがプロファイルに関連付けられます。

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

次のステートメントは、最初にテストユーザ (alice) を作成し、次に無効および有効なパスワードを彼女のルールに関連付けようとすることで、ファンクションが機能していることを確認します。

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

次に、alice がデータベースに接続してパスワードを変更しようとする、彼女はプロファイルファンクションによって確立されたルールを順守する必要があります。CREATEROLE を持たない非スーパーユーザーは、パスワードを変更するときに REPLACE 句を含める必要があります。

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

新しいパスワードは少なくとも 5 文字の長さが必要です。

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

新しいパスワードが受け入れられる場合、コマンドはエラーなしで完了します。

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

alice がパスワードを変更する場合、新しいパスワードに古いパスワードを含めることはできません。

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

検証ファンクションを削除するには、password_verify_function を NULL に設定します。

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

次に、すべてのパスワードのコンストレインが解除されます。

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

4.2 Altering a Profile

</div>

ALTER PROFILE コマンドを使用して、ユーザー定義のプロファイルを変更します。Advanced Server は、コマンドの 2 つの形式をサポートしています。

```
ALTER PROFILE <profile_name> RENAME TO <new_name>;

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

LIMIT 句とスペースで区切られた 1 つ以上の parameter/value のペアを含めて Advanced Server によって適用されるルールを指定するか、ALTER PROFILE...RENAME TO を使用してプロファイルの名前を変更します。

パラメータ：

profile_name は、プロファイルの名前を指定します。

new_name は、プロファイルの新しい名前を指定します。

parameter は、プロファイルによって制限される属性を指定します。

value は、パラメータの制限を指定します。

受け入れられるパラメータ/値のペアの完全なリストについては、[Creating a New Profile](#) の表を参照してください。

例

次の例では、acctg_profile 名前付けのプロファイルを変更します。

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

ログインロールがサーバーへの接続を試みた場合、acctg_profile は失敗した接続試行をカウントします。このプロファイルは、ユーザが 3 回の試行で正しいパスワードで認証されなかった場合、アカウントが 1 日間ロックされることを指定します。

次の例では、acctg_profile の名前を payables_profile に変更します。

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

4.3 Dropping a Profile

</div>

DROP PROFILE コマンドを使用して、プロファイルを削除します。構文は次のとおりです。

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

指定されたプロファイルが存在しない場合にエラーをスローしないようにサーバーに指示するには、IF EXISTS 句を含めます。プロファイルが存在しない場合、サーバーは通知を発行します。

オプションの CASCADE 句を含めて、現在プロファイルに関連付けられているユーザーを default プロファイルに再割り当てしてから、プロファイルをドロップします。オプションの RESTRICT 句を含めて、ロールに関連付けられているプロファイルを削除しないようにサーバーに指示します。これがデフォルトの動作です。

パラメータ

profile_name

削除されるプロファイルの名前。

例

次の例では、acctg_profile 名前付けのプロファイルを削除します。

```
DROP PROFILE acctg_profile CASCADE;
```

このコマンドは、最初に acctg_profile プロファイルに関連付けられたロールを default プロファイルに再度関連付けてから、acctg_profile プロファイルをドロップします。

次の例では、acctg_profile 名前付けのプロファイルを削除します。

```
DROP PROFILE acctg_profile RESTRICT;
```

コマンドの RESTRICT 句は、プロファイルに関連付けられたロールがある場合、acctg_profile を削除しないようにサーバーに指示します。

4.4 Associating a Profile with an Existing Role

</div>

プロファイルを作成したら、ALTER USER... PROFILE または ALTER ROLE... PROFILE コマンドを使用して、プロファイルをロールに関連付けることができます。プロファイル管理機能に関連するコマンド構文は次のとおりです。

```
ALTER USER|ROLE <name> [[WITH] option[...]]
```

option には、次の互換性のある句を指定できます。

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

または option には、次の非互換句を指定できます。

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

Advanced Server でサポートされている ALTER USER または ALTER ROLE コマンドの管理句については、次の URL で入手可能な PostgreSQL コア文書を参照してください。

< [https://www.postgresql.org/docs/current/ 静的/sql-commands.html](https://www.postgresql.org/docs/current/静的/sql-commands.html) >

データベーススーパーユーザ管理者のみが、プロファイル管理を実施する ALTER USER|ROLE 句を使用できます。句は、次の動作を強制します。

- PROFILE 句と profile_name を含めて、定義済みプロファイルをロールに関連付けるか、ユーザに関連付けられている定義済みプロファイルを変更します。

- ACCOUNT 句と LOCK または UNLOCK キーワードを含めて、ユーザアカウントをロックまたはロック解除状態にすることを指定します。
- LOCK TIME 'timestamp' 句と日付/時刻値を含めて、指定された時間にロールをロックし、このロールに割り当てられたプロファイルの PASSWORD_LOCK_TIME パラメータで示された時間にロールをロック解除します。LOCK TIME が ACCOUNT LOCK 句とともに使用される場合、ロールのロックを解除できるのは、ACCOUNT UNLOCK 句を持つデータベーススーパーユーザのみです。
- PASSWORD EXPIRE 句に AT 'timestamp' キーワードを含めて、ロールに関連付けられたパスワードの有効期限がエキスパイア日付/時刻を指定します。AT 'timestamp' キーワードを省略すると、パスワードはすぐにエキスパイアになります。
- PASSWORD SET AT 'timestamp' キーワードを含めて、パスワード変更日を指定した時間に設定します。
- STORE PRIOR PASSWORD {'password' 'timestamp'} [, ...] 句を含めてパスワード履歴を変更し、新しいパスワードとパスワードが設定された時刻を追加します。

各ログインロールに設定できるプロファイルは1つだけです。現在ログインロールに関連付けられているプロファイルを検出するには、DBA_USERS ビューの profile 列をクエリーします。

パラメータ

name

指定されたプロファイルが関連付けられるロールの名前。

password

ロールに関連付けられたパスワード。

profile_name

ロールに関連付けられるプロファイルの名前。

timestamp

条項が施行される日時。timestamp の値を指定するときは、値を単一引用符で囲みます。

例

次のコマンドは、ALTER USER... PROFILE コマンドを使用して、acctg 名前付けのプロファイルを john 名前付けのユーザに関連付けます。

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

次のコマンドは、ALTER ROLE... PROFILE コマンドを使用して、acctg 名前付けのプロファイルを john 名前付けのユーザに関連付けます。

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

4.5 Unlocking a Locked Account

</div>

データベーススーパーユーザは、ALTER USER|ROLE... コマンドの句を使用して、ロールをロックまたはロック解除できます。構文は次のとおりです。

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

ロールをすぐにロックするには、ACCOUNT LOCK 句を含めます。ロックされると、ロールの LOGIN 機能は無効になります。LOCK TIME 句なしで ACCOUNT LOCK 句を指定すると、スーパーユーザが ACCOUNT UNLOCK 句を使用してロールをロック解除するまで、ロールの状態は変更されません。

ACCOUNT UNLOCK 句を使用して、ロールのロックを解除します。

LOCK TIME 'timestamp' 句を使用して、このロールに関連付けられたプロファイルの PASSWORD_LOCK_TIME パラメータで指定された長さ、指定されたタイムスタンプで指定された時間にアカウントをロックするようサーバーに指示します。

LOCK TIME 'timestamp' 句と ACCOUNT LOCK 句を組み合わせ、ACCOUNT UNLOCK 句を呼び出すスーパーユーザによってアカウントがロック解除されるまで、指定された時間にアカウントをロックします。

パラメータ

name

ロックまたはロック解除されるロールの名前。

timestamp

ロールがロックされる日付と時刻。timestamp の値を指定するときは、値を単一引用符で囲みます。

!!! Note * このコマンド（Advanced Server でのみ使用可能）は、Oracle スタイルのプロファイル管理をサポートするために実装されています。

例

次の例では、ACCOUNT LOCK 句を使用して、john 名前付けのロールをロックします。アカウントは、ACCOUNT UNLOCK 句でロック解除されるまでロックされたままになりアカウント。

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

次の例では、ACCOUNT UNLOCK 句を使用して、john 名前付けのロールをロック解除します。

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

次の例では、LOCK TIME 'timestamp' 句を使用して、2015 年 9 月 4 日に john 名前付けのロールをロックします。

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

ロールは、PASSWORD_LOCK_TIME パラメータで指定された長さロックされたままになります。

次の例では、LOCK TIME 'timestamp' 句と ACCOUNT LOCK 句を組み合わせて、2015 年 9 月 4 日に john 名前付けのロールをロックします。

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

データベーススーパーユーザが ACCOUNT UNLOCK コマンドを使用してロールをロック解除するまで、ロールはロックされたままになります。

4.6 Creating a New Role Associated with a Profile

</div>

データベーススーパーユーザは、CREATE USER|ROLE コマンドの句を使用して、ロールの作成時に名前付けプロファイルをロールに割り当てるか、ロールのプロファイル管理の詳細を指定できます。プロファイル管理機能に関連するコマンド構文は次のとおりです。

```
CREATE USER|ROLE <name> [[WITH] <option> [...]]
```

option には、次の互換性のある句を指定できます。

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

または option には、次の非互換句を指定できます。

```
| LOCK TIME '<timestamp>'
```

Advanced Server でサポートされている CREATE USER または CREATE ROLE コマンドの管理句については、次の URL で入手可能な PostgreSQL コア文書を参照してください。

< <https://www.postgresql.org/docs/current/ 静的/sql-commands.html> >

CREATE ROLE|USER... PROFILE は、関連するプロファイルを持つ新しいロールを Advanced Server データベースクラスタに追加します。

CREATE USER コマンドで作成されたロールは（デフォルトでは）ログインロールです。CREATE ROLE コマンドで作成されたロールは（デフォルトでは）ログインロールではありません。CREATE ROLE コマンドでログインアカウントを作成するには、LOGIN キーワードを含める必要があります。

データベーススーパーユーザ管理者のみが、プロファイル管理を実施する CREATE USER|ROLE 句を使用できます。これらの句は、次の動作を強制します。

- PROFILE 句と profile_name を含めて、定義済みプロファイルをロールに関連付けるか、ユーザに関連付けられている定義済みプロファイルを変更します。
- ACCOUNT 句と LOCK または UNLOCK キーワードを含めて、ユーザアカウントをロックまたはロック解除状態にすることを指定します。

- `LOCK TIME 'timestamp'` 句と日付/時刻値を含めて、指定された時間にロールをロックし、このロールに割り当てられたプロファイルの `PASSWORD_LOCK_TIME` パラメータで示された時間にロールをロック解除します。`LOCK TIME` が `ACCOUNT LOCK` 句とともに使用される場合、ロールのロックを解除できるのは、`ACCOUNT UNLOCK` 句を持つデータベーススーパーユーザのみです。
- オプションの `AT 'timestamp'` キーワードとともに `PASSWORD EXPIRE` 句を含めて、ロールに関連付けられたパスワードの有効期限がエキスパイア日付/時刻を指定します。`AT 'timestamp'` キーワードを省略すると、パスワードはすぐにエキスパイアになります。

各ログインロールに設定できるプロファイルは1つだけです。現在ログインロールに関連付けられているプロファイルを検出するには、`DBA_USERS` ビューの `profile` 列をクエリーします。

パラメータ

name

ロールの名前。

profile_name

ロールに関連付けられたプロファイルの名前。

timestamp

条項が施行される日時。`timestamp` の値を指定するときは、値を単一引用符で囲みます。

例

次の例では、`CREATE USER` を使用して、`acctg_profile` プロファイルに関連付けられている `john` 名前付けのログインロールを作成します。

```
CREATE USER john PROFILE acctg_profile IDENTIFIED BY "1safepwd" ;
```

`john` は、パスワード `1safepwd` を使用してサーバーにログできます。

次の例では、`CREATE ROLE` を使用して、`acctg_profile` プロファイルに関連付けられている `john` 名前付けのログインロールを作成します。

```
CREATE ROLE john PROFILE acctg_profile LOGIN PASSWORD "1safepwd" ;
```

`john` は、パスワード `1safepwd` を使用してサーバーにログできます。

4.7 Backing up Profile Management Functions

</div>

プロファイルには、Advanced Server によって強制される動作を指定するユーザー定義ファンクションを参照する `PASSWORD_VERIFY_FUNCTION` 句を含めることができます。プロファイルはグローバルオブジェクトです。それらはクラスター内のすべてのデータベースで共有されます。プロファイルはグローバルオブジェクトが、ユーザー定義関数はデータベースオブジェクト。

`-g` または `-r` オプションで `pg_dumpall` を呼び出すと、既存のプロファイルの定義を再作成するスクリプトが作成されますが、`PASSWORD_VERIFY_FUNCTION` 句で参照されるユーザー定義関数は再作成されません。

`pg_dump` ユーティリティを使用して、これらの関数が存在するデータベースを明示的にダンプ（および後で復元）する必要があります。

`pg_dump` によって作成されたスクリプトには、句とファンクション名前を含むコマンドが含まれます。

```
ALTER PROFILE... LIMIT PASSWORD_VERIFY_FUNCTION <function_name>
```

復元されたファンクションを、以前に関連付けられていたプロファイルに関連付けます。

`PASSWORD_VERIFY_FUNCTION` 句が `DEFAULT` または `NULL` に設定されている場合、`pg_dumpall -g` または `pg_dumpall -r` コマンドによって生成されたスクリプトによって動作が複製されます。

第5章 Optimizer Hints

</div>

DELETE, INSERT, SELECT または UPDATE コマンドを呼び出すと、サーバーは一連の実行プランを生成します。これらの実行計画を分析した後、サーバーは、最短で結果セットを（一般に）結果計画を選択します。サーバーの計画の選択は、いくつかの要因に依存します。

- データハンドリング操作の推定実行コスト。
- postgresql.conf ファイルの Query Tuning セクションのパラメーターに割り当てられたパラメーター値。
- ANALYZE コマンドによって収集された列統計。

ルールとして、クエリープランナは最も安価なプランを選択します。オブティマイザヒントを使用して、クエリプランを選択するときにサーバーに影響を与えることができます。オブティマイザヒントは、DELETE, INSERT, SELECT または UPDATE コマンドの直後に続くコメントのような構文に埋め込みれたディレクティブ（またはマルチプルの指示子）です。コメント内のキーワードは、結果セットを作成するときに特定の計画を採用または回避するようにサーバーに指示します。

概要

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

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

オブティマイザーのヒントは、上記の形式のいずれかに含めることができます。両方の形式で、プラス記号 (+) が /* または -- の開始コメント記号の直後にスペースを入れずに続けなければなりません。そうしないと、サーバーは次のトークンをヒントとして解釈しません。

最初のフォームを使用している場合、ヒントとオプションのコメントはマルチプル行にわたる場合があります。2 番目のフォームでは、すべてのヒントとコメントが 1 行に収まる必要があります。ステートメントの残りは新しい行でスタートする必要があります。

説明

注意してください：

- データベースサーバは、可能な限り、指定されたヒントを常に使用しようとします。
- 特定のプランタイプを無効にするようにプランナメソッドパラメータが設定されている場合、プランナに他の可能なオプションがない限り、このプランはヒントで指定されていても使用されません。

プランナメソッドパラメータの例は、`enable_indexscan`, `enable_seqscan`, `enable_hashjoin`, `enable_mergejoin`, `enable_nestloop` です。これらはすべてブールパラメータです。

- ヒントがコメントに埋め込みれていることを忘れないでください。結果として、ヒントのスペルが間違っている場合、またはビュー、テーブル、列名前などのヒントのパラメータのスペルが間違っている場合、または SQL コマンドに存在しない場合、何らかのエラーが発生したことは示されません。構文エラーは発生せず、ヒント全体が単に無視されます。SQL コマンドのテーブル名またはビュー名前に別名が使用されている場合、オリジナルのオブジェクト名前ではなく別名名前をヒントで使用する必要があります。例、`accounts` の別名である `SELECT /*+ FULL(acct) */ * FROM accounts acct ...`, `acct` コマンドでは、テーブル名前 `accounts` ではなく、`FULL` ヒントで指定する必要があります。

`EXPLAIN` コマンドを使用して、ヒントが正しく形成され、プランナがヒントを使用していることを保証します。`EXPLAIN` コマンドの詳細については、Advanced Server の文書セットを参照してください。

一般的に、オプティマイザのヒントは、稼動アプリケーション（アプリケーションの存続期間中にテーブルデータが変更される）では使用しないでください。動的な列が頻繁に `ANALYZED` になるようにすることで、列の統計が更新されて値の変更が反映され、プランナはその情報を使用して特定のコマンド実行の最小コスト計画を作成します。オプティマイザのヒントを使用すると、このプロセスの目的が損なわれ、テーブルデータの変更方法に関係なく同じプランが作成されます。

パラメータ

hint

オプティマイザのヒントディレクティブ。

comment

追加情報を含む文字列。コメントに含めることができる文字には制限があることに注意してください。通常、`comment` は、アルファベット、数字、アンダースコア、ドル記号、シャープ記号、スペース文字のみで構成されます。これらは識別子の構文にも従わなければなりません。コメントがこのフォームにないなら、どんなその後のヒントも無視されるでしょう。

statement_body

`DELETE`, `INSERT`, `SELECT` または `UPDATE` コマンドの残り。

以下のセクションでは、オプティマイザヒント指示子について詳しく説明します。

5.1 Default Optimization Modes

</ div>

Advanced Server データベースクラスタのデフォルト設定として選択できる最適化モードがいくつかあります。この設定は、オプティマイザヒント内の `ALTER SESSION` コマンドと個々の `DELETE`, `SELECT` および `UPDATE` コマンドを使用して、セッションごとに変更することもできます。これらのデフォルトモードを制御する設定パラメータの名前付けは `OPTIMIZER_MODE` です。

次の表は、可能な値を示しています。

Hint	Description
<i>ALL_ROWS</i>	結果セットのすべての行を取得するために最適化します。
<i>CHOOSE</i>	結果セットから取得されると想定される行数に基づくデフォルトの最適化は行われません。これがデフォルト。
<i>FIRST_ROWS</i>	結果セットの最初の行のみを取得するように最適化します。
<i>FIRST_ROWS_10</i>	結果セットの最初の 10 行の取得用に最適化します。
<i>FIRST_ROWS_100</i>	結果セットの最初の 100 行の取得用に最適化します。
<i>FIRST_ROWS_1000</i>	結果セットの最初の 1000 行の取得用に最適化します。
<i>FIRST_ROWS_n</i>	結果セットの最初の * n * 行の取得用に最適化します。このフォームは、`ALTER SESSION SET OPTIMIZER_MODE` コマンドのオブジェクトとして使用することはできません。SQL コマンドのヒントのフォームでのみ使用できます。
	NA

これらの最適化モードは、SQL コマンドを送信するクライアントが結果セットの最初の「n」行のみを表示し、結果セットの残りを破棄するという前提に基づいています。クエリーに割り当てられたリソースは、そのように調整されます。

例

現在のセッションを変更して、結果セットの最初の 10 行の取得を最適化します。

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

OPTIMIZER_MODE パラメータの現在の値は、SHOW コマンドを使用して表示できます。このコマンドはユーティリティに依存するコマンドであることに注意してください。PSQL では、SHOW コマンドは次のように使用されます。

```
SHOW OPTIMIZER_MODE;
```

```
optimizer_mode
```

```
-----
```

```
first_rows_10
```

```
(1 row)
```

オラクルデータベースと互換性のある SHOW コマンドの構文は次のとおりです。

```
SHOW PARAMETER OPTIMIZER_MODE;
```

```
NAME
```

```
-----
```

```
VALUE
```

```
-----
```

```
optimizer_mode
```

```
first_rows_10
```

次の例は、SELECT コマンドでヒントとして使用される最適化モードを示しています。

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

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

(14 rows)

5.2 Access Method Hints

</ div>

次のヒントは、オプティマイザがリレーションにアクセスして結果セットを作成する方法に影響します。

Hint	Description
<i>FULL(table)</i>	`table`で完全なシーケンシャルスキャンを実行します。
<i>INDEX(table [index] [...])</i>	`table`で`index`を使用して、リレーションにアクセスします。
<i>NO_INDEX(table [index] [...])</i>	リレーションにアクセスするために`table`で`index`を使用しないでください。
	NA

さらに、ALL_ROWS、FIRST_ROWS および FIRST_ROWS(n) ヒントを使用できます。

例

サンプルアプリケーションには、オプティマイザヒントの効果を示すのに十分なデータがないため、このセクションの残りの例では、Advanced Server bin サブディレクトリにある **pgbench** アプリケーションによって作成された銀行データベースを使用します。

次の手順では、bank、pgbench_accounts、pgbench_branches、pgbench_tellers、pgbench_history の各表が移入される名前付けのデータベースを作成します。-s 20 オプションは、スケーリングファクタ 20 を指定します。これにより、それぞれ 100,000 アカウントの 20 のブランチが作成され、pgbench_accounts テーブルに合計 2,000,000 行、pgbench_branches テーブルに 20 行が作成されます。10 個のテラーが各ブランチに割り当てられ、pgbench_tellers テーブルに合計 200 行が作成されます。

以下は、bank データベースの pgbench アプリケーションを初期化します。

```
createdb -U enterprisedb bank
CREATE DATABASE

pgbench -i -s 20 -U enterprisedb bank

NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 2000000 tuples (5%) done (elapsed 0.11 s, remaining 2.10 s)
200000 of 2000000 tuples (10%) done (elapsed 0.22 s, remaining 1.98 s)
300000 of 2000000 tuples (15%) done (elapsed 0.33 s, remaining 1.84 s)
400000 of 2000000 tuples (20%) done (elapsed 0.42 s, remaining 1.67 s)
500000 of 2000000 tuples (25%) done (elapsed 0.52 s, remaining 1.57 s)
600000 of 2000000 tuples (30%) done (elapsed 0.62 s, remaining 1.45 s)
700000 of 2000000 tuples (35%) done (elapsed 0.73 s, remaining 1.35 s)
800000 of 2000000 tuples (40%) done (elapsed 0.87 s, remaining 1.31 s)
900000 of 2000000 tuples (45%) done (elapsed 0.98 s, remaining 1.19 s)
1000000 of 2000000 tuples (50%) done (elapsed 1.09 s, remaining 1.09 s)
1100000 of 2000000 tuples (55%) done (elapsed 1.22 s, remaining 1.00 s)
1200000 of 2000000 tuples (60%) done (elapsed 1.36 s, remaining 0.91 s)
1300000 of 2000000 tuples (65%) done (elapsed 1.51 s, remaining 0.82 s)
1400000 of 2000000 tuples (70%) done (elapsed 1.65 s, remaining 0.71 s)
1500000 of 2000000 tuples (75%) done (elapsed 1.78 s, remaining 0.59 s)
1600000 of 2000000 tuples (80%) done (elapsed 1.93 s, remaining 0.48 s)
1700000 of 2000000 tuples (85%) done (elapsed 2.10 s, remaining 0.37 s)
1800000 of 2000000 tuples (90%) done (elapsed 2.23 s, remaining 0.25 s)
1900000 of 2000000 tuples (95%) done (elapsed 2.37 s, remaining 0.12 s)
2000000 of 2000000 tuples (100%) done (elapsed 2.48 s, remaining 0.00 s)
vacuum...
set primary keys...
done.
```

その後、合計 500,00 のトランザクションが処理されます。これにより、pgbench_history テーブルに 500,000 行が入力されます。

```
pgbench -U enterprisedb -t 500000 bank

starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 20
query mode: simple
number of clients: 1
```

(次のページに続く)

(前のページからの続き)

```

number of threads: 1
number of transactions per client: 500000
number of transactions actually processed: 500000/500000
latency average: 0.000 ms
tps = 1464.338375 (including connections establishing)
tps = 1464.350357 (excluding connections establishing)

```

表の定義は次のとおりです。

```

\d pgbench_accounts
*   Table "public.pgbench_accounts"
* Column |      Type      | Modifiers
-----+-----+-----
aid      | integer        | not null
bid      | integer        |
abalance | integer        |
filler   | character(84)  |
Indexes:
*   "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)

\d pgbench_branches
*   Table "public.pgbench_branches"
* Column |      Type      | Modifiers
-----+-----+-----
bid      | integer        | not null
bbalance | integer        |
filler   | character(88)  |
Indexes:
*   "pgbench_branches_pkey" PRIMARY KEY, btree (bid)

\d pgbench_tellers
*   Table "public.pgbench_tellers"
* Column |      Type      | Modifiers
-----+-----+-----
tid      | integer        | not null
bid      | integer        |
tbalance | integer        |
filler   | character(84)  |
Indexes:
*   "pgbench_tellers_pkey" PRIMARY KEY, btree (tid)

\d pgbench_history
*           Table "public.pgbench_history"
Column |      Type      | Modifiers
-----+-----+-----

```

(次のページに続く)

(前のページからの続き)

tid	integer	
bid	integer	
aid	integer	
delta	integer	
mtime	timestamp without time zone	
filler	character(22)	

EXPLAIN コマンドは、クエリープランナによって選択されたプランを表示します。次の例では、aid がプライマリキー列であるため、インデックス付き検索がインデックス pgbench_accounts_pkey で使用されます。

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE aid = 100;
*
                                QUERY PLAN
-----
Index Scan using pgbench_accounts_pkey on pgbench_accounts (cost=0.43..8.45
rows=1 width=97)
*   Index Cond: (aid = 100)
(2 rows)
```

以下に示すように、インデックスを使用する代わりに、FULL ヒントを使用して完全シーケンシャルスキャンを強制します。

```
EXPLAIN SELECT /*+ FULL(pgbench_accounts) */ * FROM pgbench_accounts WHERE
aid = 100;
*
                                QUERY PLAN
-----
Seq Scan on pgbench_accounts (cost=0.00..58781.69 rows=1 width=97)
*   Filter: (aid = 100)
(2 rows)
```

NO_INDEX ヒントは、次のようにインデックスを使用する代わりに、強制的に並列シーケンシャルスキャンを実行します。

```
EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_pkey) */ *
FROM pgbench_accounts WHERE aid = 100;
*
                                QUERY PLAN
-----
Gather (cost=1000.00..45094.80 rows=1 width=97)
*   Workers Planned: 2
*   -> Parallel Seq Scan on pgbench_accounts (cost=0.00..44094.70 rows=1
*   width=97)
*       Filter: (aid = 100)
(4 rows)
```

前の例で示した **EXPLAIN** コマンドの使用に加えて、プランナがヒントを使用したかどうかに関する詳細情報を取得するには、**trace_hints** 設定パラメータを次のように設定します。

```
SET trace_hints TO on;
```

NO_INDEX ヒント付きの **SELECT** コマンドを以下に繰り返して、**trace_hints** 構成パラメーターが設定されている場合に生成される追加情報を示します。

```
EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_pkey) */ *
FROM pgbench_accounts WHERE aid = 100;

INFO: [HINTS] Index Scan of [pgbench_accounts].[pgbench_accounts_pkey]
rejected due to NO_INDEX hint.

*                                QUERY PLAN
-----
-----
Gather (cost=1000.00..45094.80 rows=1 width=97)
*   Workers Planned: 2
*   -> Parallel Seq Scan on pgbench_accounts (cost=0.00..44094.70 rows=1
width=97)
*       Filter: (aid = 100)
(4 rows)
```

ヒントが無視される場合、**INFO: [HINTS]** 行は表示されないことに注意してください。これは、インデックス名前のスペルが間違っている次の例に示すように、ヒントに構文エラーまたは他のスペルミスがあることを示している可能性があります。

```
EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_xxx) */ * FROM
pgbench_accounts WHERE aid = 100;

*                                QUERY PLAN
-----
-----
Index Scan using pgbench_accounts_pkey on pgbench_accounts (cost=0.43..8.45
rows=1 width=97)
*   Index Cond: (aid = 100)
(2 rows)
```

5.3 Specifying a Join Order

</div>

ORDERED ディレクティブを含めて、**FROM** 句にリストされているオーダーでテーブルを結合ようにクエリーオプティマイザに指示します。**ORDERED** キーワードを含めない場合、クエリーオプティマイザはテーブルを結合オーダーを選択します。

例、次のコマンドを使用すると、オプティマイザは **FROM** 句にリストされているテーブルを結合オーダーを選択できます。

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

次のコマンドは、指定された順序でテーブルを結合ようオプティマイザに指示します。

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

ORDERED バージョンのコマンドでは、Advanced Server は最初に **emp e** を **dept d** に結合してから、結果を **jobhist h** に結合します。ORDERED ディレクティブがない場合、結合オーダーはクエリーオプティマイザによって選択されます。

!!! Note * ORDERED ディレクティブは、Oracle スタイルの外部結合（「+」記号を含む結合）では機能しません。

5.4 Joining Relations Hints

</div>

2つのテーブルを結合する場合、結合の実行に使用できる3つの計画があります。

- **ネストされたループ結合**-テーブルは、結合された他のテーブルのすべての行に対して1回スキャンされます。
- **Merge Sort Join** *-各テーブルは、結合が開始される前に結合属性でソートされます。次に、2つのテーブルが並行してスキャンされ、マッチング行が結合されて結合行がフォームされます。
- **ハッシュ結合**-テーブルがスキャンされ、その結合属性がハッシュキーとしてハッシュハッシュテーブルにロードされます。次に、もう1つの結合テーブルがスキャンされ、その結合属性がハッシュキーとして使用されて、最初のテーブルからマッチング行が特定されます。

次の表に、プランナが1つのタイプの結合プランを別のタイプのプランに使用するように影響を与えるために使用できるオプティマイザのヒントを示します。

Hint	Description
<code>USE_HASH(table [...])</code>	`table` にはハッシュ結合を使用します。
<code>NO_USE_HASH(table [...])</code>	`table` にはハッシュ結合を使用しないでください。
<code>USE_MERGE(table [...])</code>	`table` にはマージソート結合を使用します。
<code>NO_USE_MERGE(table [...])</code>	`table` にはマージソート結合を使用しないでください。
<code>USE_NL(table [...])</code>	`table` にはネステッドループ結合を使用します。
<code>NO_USE_NL(table [...])</code>	`table` にネステッドループ結合を使用しないでください。
	NA

例

次の例では、pgbench_branches ヒントと pgbench_accounts テーブルの結合に USE_HASH ヒントが使用されます。クエリプランは、pgbench_branches テーブルの結合属性からハッシュテーブルを作成することにより、ハッシュ結合が使用されることを示しています。

```
EXPLAIN SELECT /*+ USE_HASH(b) */ b.bid, a.aid, abalance FROM
pgbench_branches b, pgbench_accounts a WHERE b.bid = a.bid;
*
                                QUERY PLAN
-----
* Hash Join  (cost=21.45..81463.06 rows=2014215 width=12)
*   Hash Cond: (a.bid = b.bid)
*   -> Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=12)
*   -> Hash  (cost=21.20..21.20 rows=20 width=4)
*       -> Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
(5 rows)
```

次に、NO_USE_HASH(a b) ヒントにより、プランナはハッシュテーブル以外のアプローチを使用するように強制されます。結果はマージ結合です。

```
EXPLAIN SELECT /*+ NO_USE_HASH(a b) */ b.bid, a.aid, abalance FROM
pgbench_branches b, pgbench_accounts a WHERE b.bid = a.bid;
*
                                QUERY PLAN
-----
* Merge Join  (cost=333526.08..368774.94 rows=2014215 width=12)
*   Merge Cond: (b.bid = a.bid)
*   -> Sort  (cost=21.63..21.68 rows=20 width=4)
*       Sort Key: b.bid
*       -> Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
*   -> Materialize  (cost=333504.45..343575.53 rows=2014215 width=12)
*       -> Sort  (cost=333504.45..338539.99 rows=2014215 width=12)
*           Sort Key: a.bid
*           -> Seq Scan on pgbench_accounts a  (cost=0.00..53746.15,
→rows=2014215 width=12)
(9 rows)
```

最後に、USE_MERGE ヒントはプランナにマージ結合の使用を強制します。

```
EXPLAIN SELECT /*+ USE_MERGE(a) */ b.bid, a.aid, abalance FROM
pgbench_branches b, pgbench_accounts a WHERE b.bid = a.bid;
*
                                QUERY PLAN
-----
* Merge Join  (cost=333526.08..368774.94 rows=2014215 width=12)
*   Merge Cond: (b.bid = a.bid)
*   -> Sort  (cost=21.63..21.68 rows=20 width=4)
*       Sort Key: b.bid
```

(次のページに続く)

(前のページからの続き)

```

*          -> Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
*  -> Materialize  (cost=333504.45..343575.53 rows=2014215 width=12)
*          -> Sort  (cost=333504.45..338539.99 rows=2014215 width=12)
*              Sort Key: a.bid
*          -> Seq Scan on pgbench_accounts a  (cost=0.00..53746.15
->rows=2014215 width=12)
(9 rows)

```

この3つのテーブルの結合の例では、プランナはまず pgbench_branches および pgbench_history テーブルでハッシュ結合ハッシュ結合を実行し、最後に結果と pgbench_accounts テーブルでハッシュ結合を実行します。

```

EXPLAIN SELECT h.mtime, h.delta, b.bid, a.aid FROM pgbench_history h, pgbench_
->branches b,
pgbench_accounts a WHERE h.bid = b.bid AND h.aid = a.aid;
*
                                QUERY PLAN
-----
Hash Join  (cost=86814.29..123103.29 rows=5000000 width=20)
* Hash Cond: (h.aid = a.aid)
* -> Hash Join  (cost=21.45..15081.45 rows=5000000 width=20)
*     Hash Cond: (h.bid = b.bid)
*     -> Seq Scan on pgbench_history h  (cost=0.00..8185.00 rows=5000000
->width=20)
*         -> Hash  (cost=21.20..21.20 rows=20 width=4)
*             -> Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
* -> Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
*     -> Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215
->width=4)
(9 rows)

```

この計画は、ヒントを使用してマージソート結合とハッシュ結合の組み合わせを強制することにより変更されます。

```

EXPLAIN SELECT /*+ USE_MERGE(h b) USE_HASH(a) */ h.mtime, h.delta, b.bid, a.aid FROM
pgbench_history h, pgbench_branches b, pgbench_accounts a WHERE h.bid = b.bid AND h.
->aid = a.aid;
*
                                QUERY PLAN
-----
Hash Join  (cost=152583.39..182562.49 rows=5000000 width=20)
* Hash Cond: (h.aid = a.aid)
* -> Merge Join  (cost=65790.55..74540.65 rows=5000000 width=20)
*     Merge Cond: (b.bid = h.bid)
*     -> Sort  (cost=21.63..21.68 rows=20 width=4)

```

(次のページに続く)

(前のページからの続き)

```

*          Sort Key: b.bid
*          -> Seq Scan on pgbench_branches b (cost=0.00..21.20 rows=20
↪width=4)
*          -> Materialize (cost=65768.92..68268.92 rows=500000 width=20)
*          -> Sort (cost=65768.92..67018.92 rows=500000 width=20)
*          Sort Key: h.bid
*          -> Seq Scan on pgbench_history h (cost=0.00..8185.00
↪rows=500000 width=20)
*          -> Hash (cost=53746.15..53746.15 rows=2014215 width=4)
*          -> Seq Scan on pgbench_accounts a (cost=0.00..53746.15 rows=2014215
↪width=4)
(13 rows)

```

5.5 Global Hints

</div>

これまで、SQL コマンドで被参照されるテーブルにヒントが直接適用されてきました。ビューが SQL コマンドで被参照されるときに、ビューに表示されるテーブルにヒントを適用することもできます。ヒントはビュー自分自身には表示されませんが、ビューを参照する SQL コマンドに表示されます。

ビュー内のテーブルに適用するヒントを指定する場合、ビュー名とテーブル名はヒント引数リスト内のドット表記で示されます。

概要

```
<hint>(<view>.<table>)
```

パラメータ

hint

表のヒントのいずれか。

view

table を含むビューの名前。

table

ヒントが適用されるテーブル。

例

tx 名前付けのビューは、[Joining Relations Hints](#) の最後の例に示されている pgbench_history, pgbench_branches と pgbench_accounts の 3 つのテーブルの結合から作成されます。

```
CREATE VIEW tx AS SELECT h.mtime, h.delta, b.bid, a.aid FROM pgbench_history
h, pgbench_branches b, pgbench_accounts a WHERE h.bid = b.bid AND h.aid =
a.aid;
```

このビューから選択して作成されたクエリプランを以下に示します。

```
EXPLAIN SELECT * FROM tx;

*
          QUERY PLAN
-----
Hash Join  (cost=86814.29..123103.29 rows=5000000 width=20)
*   Hash Cond: (h.aid = a.aid)
*   -> Hash Join  (cost=21.45..15081.45 rows=5000000 width=20)
*       Hash Cond: (h.bid = b.bid)
*       -> Seq Scan on pgbench_history h  (cost=0.00..8185.00 rows=5000000
->width=20)
*       -> Hash  (cost=21.20..21.20 rows=20 width=4)
*           -> Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20
->width=4)
*   -> Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
*       -> Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215
->width=4)
(9 rows)
```

Joining Relations Hints の最後でこの結合に適用されたのと同じヒントを、次のようにビューに適用できます。

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

*
          QUERY PLAN
-----
Hash Join  (cost=152583.39..182562.49 rows=5000000 width=20)
*   Hash Cond: (h.aid = a.aid)
*   -> Merge Join  (cost=65790.55..74540.65 rows=5000000 width=20)
*       Merge Cond: (b.bid = h.bid)
*       -> Sort  (cost=21.63..21.68 rows=20 width=4)
*           Sort Key: b.bid
*           -> Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20
->width=4)
*       -> Materialize  (cost=65768.92..68268.92 rows=5000000 width=20)
*           -> Sort  (cost=65768.92..67018.92 rows=5000000 width=20)
*               Sort Key: h.bid
*               -> Seq Scan on pgbench_history h  (cost=0.00..8185.00
->rows=5000000 width=20)
*   -> Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
*       -> Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215
->width=4)
```

(次のページに続く)

(前のページからの続き)

(13 rows)

ストアドビュー内のテーブルにヒントを適用することに加えて、次の例に示すように、サブクエリ内のテーブルにヒントを適用できます。サンプルアプリケーションの **emp** テーブルに対するこのクエリーでは、**emp** テーブルを、別名 **b** で識別される **emp** テーブルのサブクエリと結合することにより、従業員とそのマネージャーがリストされます。

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

* empno	ename	mgr empno	mgr ename
* 7369	SMITH	7902	FORD
* 7499	ALLEN	7698	BLAKE
* 7521	WARD	7698	BLAKE
* 7566	JONES	7839	KING
* 7654	MARTIN	7698	BLAKE
* 7698	BLAKE	7839	KING
* 7782	CLARK	7839	KING
* 7788	SCOTT	7566	JONES
* 7844	TURNER	7698	BLAKE
* 7876	ADAMS	7788	SCOTT
* 7900	JAMES	7698	BLAKE
* 7902	FORD	7566	JONES
* 7934	MILLER	7782	CLARK

(13 rows)

クエリープランナが選択したプランを以下に示します。

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

```
*
          QUERY PLAN
-----
Hash Join  (cost=1.32..2.64 rows=13 width=22)
*   Hash Cond: (a.mgr = emp.empno)
*   -> Seq Scan on emp a  (cost=0.00..1.14 rows=14 width=16)
*   -> Hash  (cost=1.14..1.14 rows=14 width=11)
*       -> Seq Scan on emp  (cost=0.00..1.14 rows=14 width=11)
```

(5 rows)

サブクエリ内の **emp** テーブルにヒントを適用して、テーブルスキャンの代わりに、インデックス **emp_pk** でインデックススキャンを実行できます。クエリープランの違いに注意してください。

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

(次のページに続く)

(前のページからの続き)

```

*
      QUERY PLAN
-----
* Merge Join  (cost=4.17..13.11 rows=13 width=22)
*   Merge Cond: (a.mgr = emp.empno)
*     -> Sort  (cost=1.41..1.44 rows=14 width=16)
*         Sort Key: a.mgr
*         -> Seq Scan on emp a  (cost=0.00..1.14 rows=14 width=16)
*     -> Index Scan using emp_pk on emp  (cost=0.14..12.35 rows=14 width=11)
(6 rows)

```

5.6 Using the APPEND Optimizer Hint

</div>

デフォルトでは、Advanced Server は新しいデータをテーブル内の最初の使用可能な空き領域に追加します (バキュームされたレコードによって空にされます)。INSERT または SELECT コマンドの後に APPEND ディレクティブを含めて、テーブルに中間テーブルのフリースペースをバイパスし、テーブルの最後に新しい行を付加するように指示します。このオプティマイザのヒントは、データをバルクローディングするときに特に役立ちます。

構文は次のとおりです。

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

例、オラクルデータベースと互換性のある次のコマンドは、INSERT ステートメントのデータを sales テーブルの最後に追加するようにサーバーに指示します。

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

Advanced Server は、単一の INSERT ステートメントにマルチプルの行を追加するときに APPEND ヒントをサポートすることに注意してください。

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

APPEND ヒントは、INSERT INTO ステートメントの SELECT 句にも含めることができます。

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

5.7 Parallelism Hints

</ div>

PARALLEL オプティマイザヒントは、並列スキャンを強制するために使用されます。

NO_PARALLEL オプティマイザヒントは、パラレルスキャンの使用を防ぎます。

概要

```
PARALLEL (<table> [ <parallel_degree> | DEFAULT ])
```

```
NO_PARALLEL (<table>)
```

説明

- 並列スキャン*は、マルチプルのバックグラウンドワーカーを使用して、特定のクエリーに対してテーブルのスキャンを（つまり、並列に）実行することです。このプロセスにより、シーケンシャルスキャンなどの他の方法よりもパフォーマンスが向上します。

パラメータ

table

並列ヒントが適用されるテーブル。

parallel_degree | DEFAULT

parallel_degree は、パラレルスキャンに使用する希望のワーカー数を指定する正の整数です。指定されている場合、parallel_degree と設定パラメータ max_parallel_workers_per_gather のうち小さい方が、ワーカーの計画数として使用されます。max_parallel_workers_per_gather パラメータの詳細については、次の場所にある PostgreSQL コア文書の Resource Consumption の下にある Asynchronous Behavior を参照してください。

< <https://www.postgresql.org/docs/current/ 静的/runtime-config-resource.html> >

DEFAULT が指定されている場合、可能な最大並列度が使用されます。

parallel_degree と DEFAULT の両方が省略された場合、クエリーオプティマイザは並列度を決定します。この場合、オプティマイザが parallel_workers ストレージパラメータで設定されている場合、この値は並列度として使用されます。parallel_workers ストレージパラメータの詳細については、次の場所にある PostgreSQL コア文書の CREATE TABLE の下にある Storage Parameters を参照してください。

- < <https://www.postgresql.org/docs/current/ 静的/sql-createtable.html> >

状況に関係なく、並列度は設定パラメータ max_parallel_workers_per_gather の設定を超えることはありません。

例

以下の設定パラメータ設定が有効です。

```
SHOW max_worker_processes;
* max_worker_processes
```

(次のページに続く)

(前のページからの続き)

```

-----
      8
(1 row)

SHOW max_parallel_workers_per_gather;
      * max_parallel_workers_per_gather
-----
      2
(1 row)

```

次の例は、テーブル `pgbench_accounts` のデフォルトスキャンを示しています。クエリプランにシーケンシャルスキャンが表示されていることに注意してください。

```

SET trace_hints TO on;

EXPLAIN SELECT * FROM pgbench_accounts;
      *
      QUERY PLAN
-----
Seq Scan on pgbench_accounts (cost=0.00..53746.15 rows=2014215 width=97)
(1 row)

```

次の例では、`PARALLEL` ヒントを使用しています。クエリプランで、バックグラウンドワーカーを起動する Gather ノードは、2 つのワーカーが使用される予定であることを示しています。

!!! Note * `trace_hints` が `on` に設定されている場合、`pgbench_accounts` およびその他のヒント情報に対して `PARALLEL` が受け入れられたことを示す `INFO: [HINTS]` 行が表示されます。残りの例では、これらの行は通常同じ出力を示すため表示されません（つまり、`trace_hints` は `off` にリセットされています）。

```

EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;

INFO:  [HINTS] SeqScan of [pgbench_accounts] rejected due to PARALLEL hint.
INFO:  [HINTS] PARALLEL on [pgbench_accounts] accepted.
INFO:  [HINTS] Index Scan of [pgbench_accounts].[pgbench_accounts_pkey]
rejected due to PARALLEL hint.
      *
      QUERY PLAN
-----
Gather (cost=1000.00..244418.06 rows=2014215 width=97)
  * Workers Planned: 2
  * -> Parallel Seq Scan on pgbench_accounts (cost=0.00..41996.56 rows=839256,
->width=97)
(3 rows)

```

現在、`max_parallel_workers_per_gather` 設定が増加しています。

```
SET max_parallel_workers_per_gather TO 6;
```

(次のページに続く)

(前のページからの続き)

```
SHOW max_parallel_workers_per_gather;
* max_parallel_workers_per_gather
-----
6
(1 row)
```

pgbench_accounts で同じクエリーが再度発行され、PARALLEL ヒントに並列度が指定されていません。計画されたワーカーの数は、オプティマイザの決定に従って 4 に増加していることに注意してください。

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;
*
QUERY PLAN
-----

Gather (cost=1000.00..241061.04 rows=2014215 width=97)
* Workers Planned: 4
* -> Parallel Seq Scan on pgbench_accounts (cost=0.00..38639.54 rows=503554,
->width=97)
(3 rows)
```

現在、PARALLEL ヒントの並列度パラメータに 6 の値が指定されています。ワーカーの計画数は、この指定値として返されるようになりました。

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts 6) */ * FROM pgbench_accounts;
*
QUERY PLAN
-----

Gather (cost=1000.00..239382.52 rows=2014215 width=97)
* Workers Planned: 6
* -> Parallel Seq Scan on pgbench_accounts (cost=0.00..36961.03 rows=335702,
->width=97)
(3 rows)
```

並列度の DEFAULT 設定で同じクエリーが発行されるようになりました。結果は、労働者の最大許容数が計画されていることを示しています。

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts DEFAULT) */ * FROM
pgbench_accounts;
*
QUERY PLAN
-----

Gather (cost=1000.00..239382.52 rows=2014215 width=97)
* Workers Planned: 6
* -> Parallel Seq Scan on pgbench_accounts (cost=0.00..36961.03 rows=335702,
->width=97)
(3 rows)
```

テーブル `pgbench_accounts` が変更され、`parallel_workers` ストレージパラメータが 3 に設定されるようになりました。

!!! Note * `parallel_workers` パラメータを設定する `ALTER TABLE` コマンドのこのフォーマットは、オラクルデータベースと互換性がありません。

`parallel_workers` 設定は、`PSQL \d+` コマンドで表示されます。

```
ALTER TABLE pgbench_accounts SET (parallel_workers=3);

\d+ pgbench_accounts
      *
      Table "public.pgbench_accounts"
Column |      Type      | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
-  aid   | integer        | not null  | plain   |              |
bid     | integer        |          | plain   |              |
abalance| integer        |          | plain   |              |
filler  | character(84)  |          | extended|              |
Indexes:
      * "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
Options: fillfactor=100, parallel_workers=3
```

これで、並列度なしで `PARALLEL` ヒントが指定された場合、結果として計画されるワーカーの数は `parallel_workers` パラメータの値になります。

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;
      *
      QUERY PLAN
-----
Gather  (cost=1000.00..242522.97 rows=2014215 width=97)
  *   Workers Planned: 3
  *   -> Parallel Seq Scan on pgbench_accounts  (cost=0.00..40101.47 rows=649747
  ↳width=97)
(3 rows)
```

`PARALLEL` ヒントで並列度値または `DEFAULT` を指定すると、`parallel_workers` 設定がオーバーライドされます。

次の例は、`NO_PARALLEL` ヒントを示しています。 `trace_hints` が `on` に設定されている場合、`INFO: [HINTS]` メッセージは、`NO_PARALLEL` ヒントのためにパラレルスキャンが拒否されたことを示していることに注意してください。

```
EXPLAIN SELECT /*+ NO_PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;
INFO:  [HINTS] Parallel SeqScan of [pgbench_accounts] rejected due to
NO_PARALLEL hint.
      *
      QUERY PLAN
-----
Seq Scan on pgbench_accounts  (cost=0.00..53746.15 rows=2014215 width=97)
```

(次のページに続く)

(前のページからの続き)

(1 row)

5.8 Conflicting Hints

</div>

コマンドに2つ以上の競合するヒントが含まれている場合、サーバーは矛盾するヒントを無視します。次の表に、互いに矛盾するヒントを示します。

Hint	Conflicting Hint
<i>ALL_ROWS</i>	<i>FIRST_ROWS</i> - all formats
<i>FULL(table)</i>	<i>INDEX(table [index])</i> `  ` <i>PARALLEL(table [degree])</i>
<i>INDEX(table)</i>	<i>FULL(table)</i> `  ` <i>NO_INDEX(table)</i> `  ` <i>PARALLEL(table [degree])</i>
<i>INDEX(table index)</i>	<i>FULL(table)</i> `  ` <i>NO_INDEX(table index)</i> `  ` <i>PARALLEL(table [degree])</i>
<i>PARALLEL(table [degree])</i>	<i>FULL(table)</i> `  ` <i>INDEX(table)</i> `  ` <i>NO_PARALLEL(table)</i>
<i>USE_HASH(table)</i>	<i>NO_USE_HASH(table)</i>
<i>USE_MERGE(table)</i>	<i>NO_USE_MERGE(table)</i>
<i>USE_NL(table)</i>	<i>NO_USE_NL(table)</i>

第6章 dblink_ora

</div>

dblink_oraを使用すると、リモートOracleサーバーに任意のクエリを発行できます。dblink_oraは、Advanced Server 内からOracleシステムに保存された SELECT, INSERT, UPDATE または DELETE データを可能にする OCI ベースのデータベースリンクを提供します。

**** Oracleデータベースへの接続****

Oracleの接続を有効にするには、次のウェブサイトから無料で入手できる Oracle の OCI ドライバーをダウンロードします。< [http://www.oracle.com/technetwork/ データベース/ database-technologies / instant-client / overview / インデックス.html](http://www.oracle.com/technetwork/データベース/database-technologies/instant-client/overview/インデックス.html) >

Linux の場合、ダウンロードしたOracleインスタントクライアントに libclntsh.so ライブラリが含まれていない場合、ダウンロードしたバージョンを指す libclntsh.so という名前付けのシンボリックリンクを作成する必要があります。インスタントクライアントディレクトリに移動し、次のコマンドを実行します。

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

ここで、version は libclntsh.so ライブラリのバージョン番号です。例：

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

Oracleサーバーへのリンクを作成する前に、OCI ドライバの場所を Advanced Server に指示する必要があります。

Linux の LD_LIBRARY_PATH 環境変数（または Windows の PATH）をOracleクライアントインストールディレクトリの lib ディレクトリに設定します。

Windows の場合のみ、代わりに postgresql.conf ファイルの oracle_home 設定パラメータの値を設定できます。oracle_home 設定パラメータで指定された値は、Windows の PATH 環境変数をオーバーライドします。

Linux では LD_LIBRARY_PATH 環境環境変数（WindowsPATH 環境変数または oracle_home 設定パラメータ）は、Advanced Server をスタートするときに適切に設定する必要があります。

Linux サービススクリプトを使用して Advanced Server をスタートする場合は、スクリプトが pg_ctl ユーティリティを呼び出して Advanced Server をスタートするときに有効になるように、サービススクリプト内で LD_LIBRARY_PATH が設定されていることを確認してください。

**** Windows のみ：**** postgresql.conf ファイルで oracle_home 設定パラメータを設定するには、ファイルを編集し、次の行を追加します。

```
oracle_home = 'lib_directory'
```

`lib_directory` を `oci.dll` を含む Windows ディレクトリの名前に置き換えます。

`oracle_home` 設定パラメータを設定した後、変更を有効にするにはサーバーをリスタートする必要があります。Windows サービスコンソールからサーバーを再起動します。

6.1 dblink_ora Functions and Procedures

</div>

`dblink_ora` は、次の関数とプロシージャをサポートしています。

6.1.1 dblink_ora_connect()

</div>

`dblink_ora_connect()` ファンクションは、ユーザー指定の接続情報を使用してオラクルデータベースへの接続を確立します。このファンクションには2つの形式があります。最初のフォームのシグネチャは次のとおりです。

```
dblink_ora_connect(<conn_name>, <server_name>, <service_name>, <user_name>,
<password>, <port>, <asDBA>)
```

場所：

- `conn_name` は、リンクの名前を指定します。
- `server_name` は、ホストの名前を指定します。
- `service_name` は、サービスの名前を指定します。
- `user_name` は、サーバーへの接続に使用される名前を指定します。
- `password` は、ユーザ名前に関連付けられたパスワードを指定します。
- `port` はポート番号を指定します。

オラクルサーバーで `SYSDBA` 特権を要求する場合、`asDBA` は `True` です。このパラメータはオプション。省略した場合、デフォルト値は `FALSE` です。

`dblink_ora_connect()` の最初のフォームは、`TEXT` 値を返します。

`dblink_ora_connect()` ファンクションの2番目のフォームのシグネチャは次のとおりです。

```
dblink_ora_connect(<foreign_server_name>, <asDBA>)
```

場所：

- `foreign_server_name` は、外部サーバの名前を指定します。

オラクルサーバーで SYSDBA 特権を要求する場合、asDBA は True です。このパラメータはオプション。省略した場合、デフォルト値は FALSE です。

`dblink_ora_connect()` ファンクションの 2 番目のフォームでは、サーバーへの接続を確立するときに、事前定義された外部サーバの接続プロパティを使用できます。

`dblink_ora_connect()` ファンクションの 2 番目のフォームを呼び出す前に、`CREATE SERVER` コマンドを使用して、システムテーブルへのリンクの接続プロパティを保存します。`dblink_ora_connect()` ファンクションを呼び出すときは、`CREATE SERVER` コマンドで指定されたサーバー名前をリンクの名前に置き換えます。

`dblink_ora_connect()` の 2 番目のフォームは、TEXT 値を返します。

6.1.2 dblink_ora_status()

</ div>

`dblink_ora_status()` ファンクションは、データベースコネクションステータスを返します。シグネチャは次のとおりです。

```
dblink_ora_status(<conn_name>)
```

場所：

`conn_name` は、リンクの名前を指定します。

指定された接続がアクティブな場合、ファンクションは TEXT の値 OK を返します。

6.1.3 dblink_ora_disconnect()

</ div>

`dblink_ora_disconnect()` ファンクションはデータベースコネクションを閉じます。シグネチャは次のとおりです。

```
dblink_ora_disconnect(<conn_name>)
```

場所：

`conn_name` は、リンクの名前を指定します。

ファンクションは TEXT 値を返します。

6.1.4 dblink_ora_record()

</ div>

dblink_ora_record() ファンクションは、データベースから情報を取得します。シグネチャは次のとおりです。

```
dblink_ora_record(<conn_name>, <query_text>)
```

場所：

- conn_name は、リンクの名前を指定します。
- query_text は、オラクルサーバーで呼び出される SQL SELECT ステートメントのテキストを指定します。

ファンクションは SETOF レコードを返します。

6.1.5 dblink_ora_call()

</ div>

dblink_ora_call() ファンクションは、オラクルデータベースで非 SELECT ステートメントを実行し、結果セットを返します。シグネチャは次のとおりです。

```
dblink_ora_call(<conn_name>, <command>, <iterations>)
```

場所：

- conn_name は、リンクの名前を指定します。
- command は、オラクルサーバーで呼び出される SQL ステートメントのテキストを指定します。
- iterations は、ステートメントが実行される回数を指定します。

ファンクションは SETOF レコードを返します。

6.1.6 dblink_ora_exec()

</ div>

dblink_ora_exec() プロシージャは、リモートデータベースで DML または DDL ステートメントを実行します。シグネチャは次のとおりです。

```
dblink_ora_exec(<conn_name>, <command>)
```

場所：

- conn_name は、リンクの名前を指定します。

- **command** は、オラクルサーバーで呼び出される INSERT, UPDATE または DELETE SQL ステートメントのテキストを指定します。

ファンクションは VOID を返します。

6.1.7 dblink_ora_copy()

</div>

dblink_ora_copy() ファンクションは、オラクルテーブルを EDB テーブルにコピーします。
dblink_ora_copy() ファンクションは、コピーされた行の数を表す BIGINT 値を返します。シグネチャは次のとおりです。

```
dblink_ora_copy(<conn_name>, <command>, <schema_name>, <table_name>,  
<truncate>, <count>)
```

場所：

- **conn_name** は、リンクの名前を指定します。
- **command** は、オラクルサーバーで呼び出される SQL SELECT ステートメントのテキストを指定します。
- **schema_name** は、ターゲットスキーマの名前を指定します。
- **table_name** は、ターゲット表の名前を指定します。
- **truncate** は、サーバーがコピーの前にテーブルを TRUNCATE する必要があるかどうかを指定します。TRUE を指定して、サーバーがテーブルを TRUNCATE する必要があることを示します。**truncate** はオプション。省略した場合、値は FALSE です。
- **count** は、n レコードごとにステータス情報をレポートするようサーバーに指示します。n は指定された番号です。ファンクションの実行中に、Advanced Server はカウントの反復ごとに重大度 INFO の通知を出します。例、FeedbackCount が 10, **dblink_ora_copy()** の場合、10 レコードごとに通知が発生します。**count** はオプション。省略した場合、値は 0 です。

6.2 Calling dblink_ora Functions

</div>

次のコマンドは、**dblink_ora_connect()** ファンクションを使用して接続を確立します。

```
SELECT dblink_ora_connect('acctg', 'localhost', 'xe', 'hr', 'pwd', 1521);
```

この例では、ユーザ名前 **hr** とパスワード **pwd** を使用して、ポート 1521 (localhost 上) で実行されている **xe** 名前付けのサービスに接続します。接続名前 **acctg** を使用して、他の **dblink_ora** 関数を呼び出すときにこの接続を参照できます。

次のコマンドは、**edb_conn** 名前付けの接続で **dblink_ora_copy()** ファンクションを使用して、**ora_acctg** という名前のテーブル (オラクルサーバー上) から **empid** および **deptno** 列を **as_acctg** 名前付けの Advanced

Server のインスタンスの **public** スキーマにあるテーブルにコピーします。TRUNCATE オプションが適用され、3 のフィードバックカウントが指定されます。

```
edb=# SELECT dblink_ora_copy('edb_conn','select empid, deptno FROM  
ora_acctg', 'public', 'as_acctg', true, 3);
```

```
INFO: Row: 0
```

```
INFO: Row: 3
```

```
INFO: Row: 6
```

```
INFO: Row: 9
```

```
INFO: Row: 12
```

```
    * dblink_ora_copy
```

```
-----
```

```
12
```

```
(1 row)
```

次の SELECT ステートメントは、**dblink_ora_record()** ファンクションと **acctg** 接続を使用して、オラクルサーバーから情報を取得します。

```
SELECT * FROM dblink_ora_record( 'acctg', 'SELECT first_name from employees') AS  
→t1(id VARCHAR);
```

このコマンドは、**employees** テーブルの **first_name** 列のすべてのエントリを含むリストを取得します。

第7章 Open Client Library

</div>

Open Client Library は、オラクル Call Interface とアプリケーションの相互運用性を提供します。以前は「ロックイン」されていたアプリケーションは、EDB Postgres Advanced Server またはオラクルデータベースのいずれかで動作し、アプリケーションコードをほとんど変更しません。Open Client Library の EDB 実装は C で記述されています。

次の図は、Open Client Library とオラクル Call Interface アプリケーションスタックを比較しています。

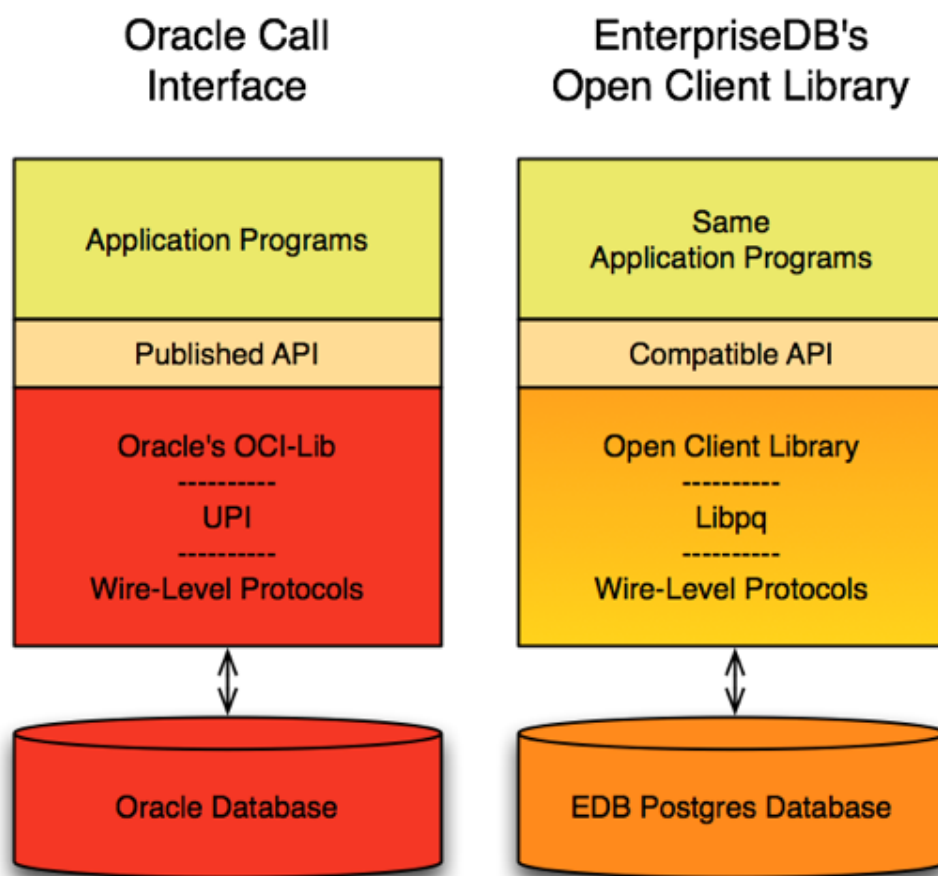


図 1: Open Client Library

Open Client Library とサポートされる機能の詳細な使用方法については、* EDB Postgres Advanced Server OCL Connector Guide *を参照してください。

https://www.enterprisedb.com/docs/ocl_connector/latest/

!!! Note * EDB は、オラクル Real Application Clusters (RAC) およびオラクル Exadata での Open Client Library の使用をサポートしていません。前述のオラクル製品は、この EDB 製品では評価も認定もされて

いません。

第8章 Oracle Catalog Views

</ div>

Oracle カタログ Views は、Oracle データディクショナリビューと互換性のある方法でデータベースオブジェクトに関する情報を提供します。サポートされているビューに関する情報は、次の場所にある「Oracle Developers Catalog Views ガイドのデータベース互換性ガイド」で入手できます。

<https://www.enterprisedb.com/docs>

第9章 Tools and Utilities

</ div>

互換性のあるツールとユーティリティプログラムにより、開発者は使い慣れた環境で Advanced Server を操作できます。Advanced Server がサポートするツールは次のとおりです。

EDB * Plus EDB * Loader EDB *ラップ - 動的ランタイム計装ツールアーキテクチャ (DRITA)

Advanced Server でサポートされている機能の詳細については、次の Web サイトで入手可能な『オラクル Developers Tools and Utilities Guide のデータベース互換性』*を参照してください。

<https://www.enterprisedb.com/docs>

第10章 ECPGPlus

</div>

EDB は ECPG (PostgreSQL プリコンパイラ) を拡張して ECPGPlus を作成しました。ECPGPlus を使用すると、C アプリケーションに埋め込み SQL コマンドを含めることができます。ECPGPlus を使用して、埋め込み SQL コマンドを含むアプリケーションをコンパイルすると、SQL コードは構文チェックされ、C に変換されます。

ECPGPlus は、Advanced Server データベースに接続されている場合、C プログラムで Pro * C 互換構文をサポートします。ECPGPlus は以下をサポートします：

オラクル Dynamic SQL –方法 4 (ODS-M4)。 - Pro * C 互換の匿名ブロック。オラクルデータベースと互換性のある CALL ステートメント。

ECPGPlus の Pro * C 互換性のパートとして、BEGIN DECLARE SECTION および END DECLARE SECTION 指示子を含める必要はありません。

ECPGPlus の使用の詳細については、次の EDB ウェブサイトから入手できる* EDB Postgres Advanced Server ECPG コネクタガイド*を参照してください。

<https://www.enterprisedb.com/docs>

第11章 System Catalog Tables

</ div>

システムカタログテーブルには、Advanced Server で使用可能なデータベースオブジェクトの定義が含まれています。システムテーブルのレイアウトは変更されるサブジェクトがあります。システムテーブルに格納されている情報に依存するアプリケーションを作成している場合、既存のカタログビューを使用するか、カタログビューを作成してシステムテーブルの変更からアプリケーションを分離することをお勧めします。

システムカタログテーブルの詳細については、次の場所にある 『オラクル Developers カタログ Views のデータベース互換性ガイド』を参照してください。

<https://www.enterprisedb.com/docs>