EnterpriseDB™

# Postgres Plus Advanced Server

오라클 호환성 개발자 가이드

OSS사업부

Postgres Plus Advanced Server 8.3

March 14, 2008

Oracle ®오라클 호환성 개발자 가이드
By EnterpriseDB Corporation


EnterpriseDB Corporation, 499 Thornall Street, Edison, New Jersey 08837-2210, USA
T +1 732 331 1300 F +1 732 331 1301 E info@enterprisedb.com www. enterprisedb.com

# 목차

# 제 1 장

## 소개

이 가이드에서는 Postgres Plus Advanced Server 의 오라클 호환성 사양에 대한 설명을 합니다. 오라클 호환성은 응용 프로그램의 소스코드를 거의 또는 전혀 변경하지 않고 응용 프로그램을 Postgres Plus Advanced Server 환경과 마찬가지로 오라클 환경에서 사용할 수 있음을 의미합니다.

Postgres Plus Advanced Server, PostgreSQL 또는 오라클 데이터베이스 응용 프로그램을 개발할 수 있는 다양한 기능을 가지고 있습니다. 이 설명서는 Oracle 호환성 기능에만 초점을 맞추고 있기 때문에 Postgres Plus Advanced Server 의 모든 기능에 대해 배우고 싶은 분은, "Postgres Plus Advanced Server Advanced Server documentation"를 참조하십시오.

Postgres Plus Advanced Server 에서 오라클 전환 응용 프로그램을 개발하려면 응용 프로그램의 설계에 특히 주의해야 합니다. 예를 들면, Oracle 호환 애플리케이션을 개발하는 경우에는 다음 사항이 필요합니다.

- 응용 프로그램 데이터베이스 테이블을 정의하는 것은 Oracle 호환 데이터 형식이어야 합니다
- Oracle 의 SQL 과 호환되는 SQL 문장이어야 합니다.
- SQL 문장이나 절차상의 논리에서는 Oracle 호환 시스템 및 내장 함수를 사용하지 않으면 안됩니다.
- 저장 처리 절차, 함수, 트리거, 패키지의 데이터베이스의 서버 측 응용 프로그램을 만들려면 Stored Procedure Language (SPL)를 사용하지 않으면 안 됩니다.
- 오라클 데이터 사전과 호환되는 시스템 카탈로그 뷰를 사용하지 않으면 안 됩니다.

Postgres Plus Advanced Server 에는 이 같은 기능이 있습니다.

또한 *Oracle Call Interface (OCI)를* 사용하여 개발된 응용 프로그램과 상호 연결하기 위하여, EnterpriseDB 는 Open Client Library (OCL)를 제공합니다.

이상은 EnterpriseDB 사양에 대한 자세한 내용을 설명합니다.

### 1.1 개정 내역

여기서는 Postgres Plus Advanced Server 8.2 에서 추가된 Oracle 호환 사양을 제공합니다.

- 매개변수 구성 edb_redwood_strings 하면 문자열 연결 시 null 의 변수 혹은 null 열을 null 값 대신 빈 문자열로 처리할 수 있게 되었습니다. edb_redwood_strings 에 대한 자세한 내용은 제 1.3.2 장을 참조하십시오.

- 매개변수 구성 edb_stmt_level_tx 를 통해 SQL 명령을 자동으로 삭제하는 롤백 영역을 트랜잭션 내의 모든 SQL 명령이 아닌 1 개의 SQL 명령에만 제한 할 수 있습니다. edb_stmt_level_tx 에 대한 자세한 내용은 제 1.3.3 장을 참조하십시오.
- 데이터베이스 연결, DELETE, INSERT, SELECT, UPDATE 명령이 지원되었습니다. 자세한 내용은 CREATE DATABASE LINK 명령과 DROP DATABASE LINK 명령을 참조하십시오.
- 지정된 실행 계획을 최적화 프로그램에 사용하는 최적화 팁, DELETE, SELECT, UPDATE 명령이 지원되었습니다. 최적화 힌트에 대한 자세한 내용은 3.4 장을 참조하십시오.
- AUTHID DEFINER, AUTHID CURRENT_USER 어구가 CREATE FUNCTION, CREATE PROCEDURE, CREATE PACKAGE 명령에 지원되었습니다. 이 프로그램을 참조하는 데이터베이스 개체에 대한 액세스 권한을 결정하기 위해 프로그램 소유자의 권한을 사용하는지, 프로그램을 실행하여 현재 사용자의 권한을 사용하는 방법을 지정합니다. 또 자격이 되지 않는 개체를 참조하는 데 사용하는 사용자의 검색 경로를 지정합니다. SPL 프로그램 보안에 대한 자세한 내용은 제 4.2.6 장을 참조하십시오.
- NOMINVALUE, NOMAXVALUE 어구가 CREATE SEQUENCE 명령에 지원되었습니다. 이것은 각 시퀀스의 기본 최소값 또는 최대값을 사용하도록 지정합니다.
- FORALL 명령문은 DELETE, INSERT, UPDATE 명령이 데이터베이스 서버에 여러 개의 값 집합을 전달할 수 있도록 합니다. 이것은 다른 가치에 대해 여러 번의 명령을 실행함으로써 발생하는 오버헤드를 줄일 수 있습니다. FORALL 문장에 대한 자세한 내용은 제 4.9.3 장을 참조하십시오.
- BULK COLLECT 어구를 사용하여 SELECT INTO, FETCH, DELETE RETURNING, INSERT RETURNING, UPDATE RETURNING 명령 결과 모집을 할 수 있습니다. 자세한 내용은 제 4.9.4 장을 참조하십시오.
- EDB*Plus 유틸리티 프로그램은 SQL*Plus 명령 라인 터미널 인터페이스를 제공합니다. 자세한 내용은 10.1 장을 참조하십시오.

### 1.2 본 설명서 작성 규칙

본 가이드에는 다양한 명령, 문장, 프로그램, 예제와 같은 의미로 사용하는 방법을 명확하게 하기 위해 여러 종류의 기술 규칙이 사용됩니다. 여기서는 이 규칙의 요지를 설명합니다.
여기서는 *용어는* 언어 키워드 사용자 제공 값이나 문자 등의 단어 또는 단어 군을 말합니다. 용어의 정확한 의미는 사용되는 상황에 따라 결정됩니다.

- *이탤릭체는* 새로운 용어가 처음 등장했을 때 사용됩니다.
- *고정 폭 글꼴*은 SQL 명령, 지문 중에서 특정 테이블 및 열 이름, 프로그래밍 언어 키워드처럼 그대로 지정 받아야 하는 용어를 사용합니다. 예) SELECT * FROM emp;
- *이탤릭체 고정 폭 글꼴은* 실제로는 사용자가 값을 할당 받아야 하는 용어를 사용합니다. 예) DELETE FROM *table_name;*

- 파이프 (|), 오른쪽 또는 왼쪽 또는 중 하나의 어휘를 선택하는 것을 의미합니다. 파이프는 대괄호 (옵션 선택) 나 괄호 (1 개를 선택) 가운데 2 개 이상의 옵션을 분리하는 데 사용됩니다.
- 대괄호 ([])는 둘러싸인 용어 중에서 1 개를 선택하거나 아무 것도 선택하지 않았는지 보여줍니다. 예를 들면, [a | b] 는 "a" 또는 "b" 중 1 개를 선택하거나 둘 다 선택하지 않을 수 있습니다.
- 괄호 ({})는 둘러싸인 용어 중에서 반드시 1 개를 선택 받아야 하는 것을 의미합니다. 예를 들면, {a | b}는 "a" 또는 "b" 중 1 개를 선택하지 않으면 안됩니다.
- ...이 용어가 반복되는 것을 의미합니다. 예를 들면, [a | b] ...는 "b a a b a" 로 이어질 가능성이 있다는 것을 보여줍니다.

## 1.3 오라클 호환 매개변수 구성

시작 부분에서 언급한대로, Postgres Plus Advanced Server 는 PostgreSQL 와 Oracle 애플리케이션을 개발하고 실행할 수 있습니다. 시스템이 PostgreSQL 이나 Oracle 준수도 달리게 많은 기능을 가지고 있습니다. 이 데이터베이스 클러스터 데이터 디렉토리에 있는 postgresql.conf 파일 매개변수 구성으로 제어됩니다. postgresql.conf 파일의 매개 변수를 변경하면 클러스터의 모든 데이터베이스의 동작이 변경됩니다. 이 매개 변수보다 나은 조정하는 방법은 데이터베이스 사용자, 그룹 수업으로 조정하는 것입니다. 매개 변수는 다음과 같습니다.

- edb_redwood_date - 시간 요소를 DATE 열을 포함할지 여부를 제어합니다. Oracle 호환을 위해서는 edb_redwood_date "true"로 설정합니다.
- edb_redwood_strings - 문자열의 결합을 위해, null 값을 빈 문자열로 취급합니다. Oracle 호환을 위해서는 edb_redwood_strings "true"로 설정합니다.
- edb_stmt_level_tx - SQL 명령을 자동으로 삭제하는 롤백 기능을 1 개의 SQL 문장만을 롤백으로 변경합니다. PostgreSQL 의 기본 동작이다 현재 트랜잭션의 모든 SQL 문장을 자동으로 롤백하는 것은 아닙니다. Oracle 호환을 위해서는 edb_stmt_level_tx "true"로 설정합니다. 그러나 절대로 필요한 경우에만 사용하도록 하십시오. 자세한 내용은 제 1.3.3 장을 참조하십시오.
- oracle_home - 오라클 설치 디렉토리에 Postgres Plus Advanced Server 의 포인트. 자세한 내용은 제 1.3.4 장을 참조하십시오.

### 1.3.1 edb_redwood_date

CREATE TABLE 명령 또는 ALTER TABLE 명령에서 열의 데이터 형식으로 DATE 를 지정하면 데이터베이스에 테이블 정의를 저장하는 동시에 데이터 형식이 TIMESTAMP (0)으로 변환됩니다. (단, 매개변수 구성 edb_redwood_date 가 "true"로 설정되어있는 경우) 따라서 열은 날짜와 함께 시간도 포함됩니다.
이것은 오라클 DATE 데이터 형식과 일치합니다.

edb_redwood_date "false"로 설정되어있는 경우, CREATE TABLE 명령 또는 ALTER TABLE 명령에서 열의 데이터 형식은 PostgreSQL 의 DATE 데이터 형식 그대로 데이터베이스에 저장됩니다. PostgreSQL 의 DATE 데이터 형식은 열 사이 시간은 포함하지 않고, 날짜만 저장합니다.

edb_redwood_date 설정에 관계없이, SPL 선언 부분에 변수의 데이터 형식, 절차와 함수의 매개 변수의 데이터 형식, 함수의 반환 데이터 형식과 같은 다른 형식의 데이터 형식으로 DATE 이 지정되지 때는 항상 내부에 TIMESTAMP (0)으로 변환됩니다. 따라서 시간을 처리할 수 있습니다. 날짜 / 시간 데이터 형식에 대한 자세한 내용은 제 3.2.4 장을 참조하십시오

### 1.3.2 edb_redwood_strings

Oracle 에서는 문자열이 null 변수와 null 열과 결합되면, 결과는 원래 텍스트로 변경되지 않습니다. 그러나 PostgreSQL 에서는 문자열이 null 변수와 null 열과 결합되면, 결과는 null 입니다. edb_redwood_strings 매개 변수가 "true"로 설정되어있는 경우 이러한 결합 결과는 Oracle 과 마찬가지입니다. edb_redwood_strings "false"로 설정되어있는 경우, PostgreSQL 동작은 변경되지 않습니다.

아래에는 차이 예시로 설명합니다.
다음 장에서 소개하는 샘플 응용 프로그램은 직원 테이블을 가지고 있습니다. 이 테이블은 대부분의 직원이 null 값인 comm 이라는 열을 가지고 있습니다.
edb_redwood_string "false"로 설정하면 다음과 같은 쿼리를 수행합니다. null 열을 비어 있지 않은 문자열과 결합하면 최종 결과는 null 입니다. 따라서 수수료를 가진 직원만이 문의 결과에 표시됩니다. 다른 모든 직원의 출력 열은 null 입니다

```
  SET edb_redwood_strings TO off;

  SELECT RPAD (ename, 10) | | ''| | TO_CHAR (sal, '99999 .99 ') | |' '| | TO_CHAR
(comm, '99999 .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)
```

다음은 edb_redwood_strings "true"로 설정하면 같은 쿼리를 수행합니다. 여기서 null 값이 빈 문자열로 취급됩니다. 빈 문자열이 비어 있지 않은 문자열과 결합하면 비어 있지 않은 문자열로 결과가 산출됩니다. 같은 질문을 실행하면 그 결과는 Oracle 의 결과와 일치합니다.

```
  SET edb_redwood_strings TO on;
```

```
  SELECT RPAD (ename, 10) | | ''| | TO_CHAR (sal, '99999 .99 ') | |' '| | TO_CHAR
(comm, '99999 .99') "EMPLOYEE COMPENSATION"FROM emp;

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

1.3.3 edb_stmt_level_tx

Oracle 에서는 SQL 명령을 실행할 때 오류가 발생하면 1 개의 명령으로 생긴 데이터베이스의 모든 변경 내용이 롤백됩니다. 이것을, *명령문레벨의 고립이라고 합니다.* 예를 들면, 1 개의 UPDATE 명령에 5 개의 행 업데이트에 성공했으나 6 번째 줄의 업데이트하는 동안 예외가 발생하면 이 명령이 실행된 6 개의 모든 행을 업데이트가 롤백 됩니다 . 커밋 또는 롤백 되지 않은 이 명령 이전 SQL 명령의 실행 결과는 대기 상태가 됩니다.

PostgreSQL 에서는 SQL 명령 실행 중에 예외가 발생하면 트랜잭션을 시작하고 그때까지는 데이터베이스에서 일어난 모든 변화가 롤백 됩니다.

또한, 트랜잭션도 중단 상태로 남아 있기 때문에, 다른 트랜잭션을 시작하기 전에 COMMIT 명령 또는 ROLLBACK 명령이 실행되지 않으면 안됩니다.

edb_stmt_level_tx "true"로 설정되어있는 경우, Oracle 과 같이 예외가 발생 했을 때, 이전 커밋 되지 않은 데이터베이스의 변경 사항은 자동으로 롤백 되지 않습니다. edb_stmt_le vel_tx "false"로 설정되어있는 경우 예외가 발생했을 경우는 커밋 되지 않은 데이터베이스의 변경 사항은 롤백 할 수 있습니다.

**주의 :** 현재 SPL 프로그램에서 항상 edb_stmt_level_tx "false"로 설정하는 것과 같은 동작을 합니다. SPL 프로그램에서 예외가 발생하면 마지막으로 예외가 발생한 블록 범위 내에서 일어난 변화에 반영되지 않은 것은 항상 롤백 됩니다. SPL 프로그램에서 edb_stmt_level_tx = "true"를 지원하는 것은 향후 릴리스입니다.

**주의 :** 이 매개 변수는 성능에 영향을 주기 때문에, 절대로 필요한 경우에만 edb_stmt_level_tx "true"로 설정하십시오.

PSQL 에서 실행되는 다음은 edb_stmt_level_tx "false"의 경우 두 번째 INSERT 명령이 취소되는 것과 동시에, 첫 번째 INSERT 명령을 롤백 되는 것을 의미합니다. PSQL 에서 ₩ set AUTOCOMMIT off 명령을 실행해야 할 주의하십시오. 그렇지 않으면 모든 SQL 문이 자동으로 반영되어 버리고 edb_stmt_level_tx 의 효과를 거둘 수 없습니다.

```
  \ 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)
```

다음은, edb_stmt_level_tx "true"로 설정합니다. 첫 번째 INSERT 명령에 오류가 발생하면 처음 INSERT 명령이 롤백 되지 않습니다. 이 때 첫 번째 INSERT 명령을 커밋 또는 롤백 할 수 있습니다.

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

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

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

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

  COMMIT;
```

COMMIT 명령 대신 ROLLBACK 명령을 실행할 수 있습니다. 이 경우 직원 ID 9001 삽입은 롤백 됩니다.

### 1.3.4 oracle_home

그 설정 매개변수 oracle_home는 파일 시스템내의 정정된 Oracle Home directory에 Postgres Plus Advanced Server 에 접근 할 수 있습니다. 이 매개변수는 오직 슈퍼유저에 의해 설정되어 질 수 있고, 온, 오프라인의 어느 레벨에서도 설정될 수 있습니다. 따라서 단일 EDB 서버에서 여러 Oracle 클라이언트를 설치와 오라클의 기존 및 최신 버전의 접근을 동시에 사용할 수 있도록 했습니다. 실행에서 만약 설정되었다면 oracle_home 설정 매개변수는 기본인 ORACLE_HOME 환경 변수에 오버라이드 될 것입니다.

## 1.4 본 설명서에 대한 예제

본 설명서에 지명되는 예제는 Postgres Plus Advanced Server 의 PSQL 프로그램을 사용하고 있습니다. PSQL 를 사용할 때 일반적으로 나타나는 프롬프트는 이 경우에서는 생략합니다. 설명하는 점이 매우 분명하게 표현하고 있습니다.

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

또한 다음 사항을 주의해야 합니다.

Postgres Plus Advanced Server 를 설치할 때 본 가이드에 지명되는 경우와 동일한 결과를 얻기 위하여, Oracle 호환 설정 및 기본 설정해야 합니다. 기본 Oracle 호환 구성은 PSQL 에서 다음 명령을 실행하여 확인할 수 있습니다. 다음 결과와 동일하다면 문제가 되지 않습니다.

```
SHOW edb_redwood_date;
edb_redwood_date
------------------
on
SHOW datestyle;
DateStyle
--------------
Redwood, DMY
SHOW edb_redwood_strings;
edb_redwood_strings
---------------------
 on
```

예제에서는 Postgres Plus Advanced Server 설치 시 생성, 로드 되는 샘플 테이블 dept, emp, jobhist 를 사용합니다. emp 테이블은 트리거와 함께 설치되지만 이 설명서에서 보여주는 결과와 동일한 결과를 얻기 위하여 트리거를 사용할 수 없도록 해야 합니다. enterprisedb 슈퍼에서 Postgres Plus Advanced Server 에 로그인하여 다음 명령으로 트리거를 사용하지 못하도록 합니다.

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

emp 테이블의 트리거는 다음 명령으로 다시 사용할 수 있습니다.

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

**제 2 장**

**SQL 자습서**

이 장에서는 관계형 데이터베이스 관리시스템을 처음 사용하는 분들을 위해 SQL 언어를 소개합니다. 테이블 만들기, 삽입 쿼리 업데이트 등 기본 사항은 보기를 들어 설명합니다.

뷰, 외래 키, 거래 같은 고급 개념도 설명합니다.

## 2.1 시작

*Postgres Plus Advanced Server 는* 관계형 데이터베이스 *관리 시스템 (RDBMS)입니다.* 이것은 연결된 데이터를 관리하는 시스템입니다. 관계는 본질적으로 *테이블을* 가리키는 수학 용어입니다. 테이블에 데이터를 저장한다는 생각은 오늘날 아주 일반적이며, 이해할 수 있는 사고 방식이지만 데이터베이스를 구성하는 다른 방법도 있습니다. Unix 와 같은 운영 체제의 파일 및 디렉토리 구성은 계층적 데이터베이스의 예라고 말할 수 있으며, 또한 최근에는 객체 지향 데이터베이스도 개발되어 있습니다.

각 테이블은 *행* 집합입니다. 행은 동일한 *열* 집합입니다. 각 항목에는 *데이터 형식을* 지정합니다. 각 줄 줄은 정해진 순서대로 나란히 있는 반면 SQL 은 테이블의 행 순서를 보증하지 않습니다. (비록 명시적으로 표시하기 위해 정렬할 수 있습니다)

테이블은 데이터베이스에서 그룹화됩니다. 단순한 EnterpriseDB 서버 인스턴스에서 처리한 그 데이터베이스 집합이 데이터베이스 *클러스터를* 구성합니다.

### 2.1.1 예제

이 매뉴얼에서는 데이터베이스의 개념을 기본부터 고급까지 설명하기 위하여 데이터베이스를 사용합니다.

#### 2.1.1.1 예제 데이터베이스 설치

Postgres Plus Advanced Server 을 설치한 경우 edb 라는 예제 데이터베이스를 자동으로 만들어집니다. 이 데이터베이스에는 사용하는 테이블이나 프로그램이 포함되어있습니다.

예제 데이터베이스의 테이블과 프로그램은 edb - sample.sql 스크립트를 실행하면 언제든지 다시 만들 수 있습니다. 스크립트는 EnterpriseDB 의 홈 디렉토리의 samples 디렉토리에 있습니다.

이 스크립트는 다음 작업을 수행합니다.

- 현재 연결된 데이터베이스에서 예제 테이블과 프로그램을 만듭니다
- PUBLIC 그룹 테이블에 모든 권한을 부여합니다

테이블과 프로그램은 검색 경로의 첫 번째 스키마에 만들어집니다. 현재 사용자는 스키마에서 테이블과 절차를 만들 능력을 가지고 있습니다. 다음 명령을 사용하여 검색 경로를 볼 수 있습니다.

```
SHOW SEARCH_PATH;
```

EnterpriseDB 의 PSQL 명령을 사용하여 검색 경로를 변경할 수 있습니다.

#### 2.1.1.2 예제 설명

데이터베이스는 조직의 직원에 대한 데이터입니다.

직원, 부서, 직원의 기록을 3 개의 레코드를 포함하고 있습니다.

각 직원은 직원 ID, 이름, 직업, 봉급, 관리자의 데이터를 가지고 있습니다. 월급 이외에 수수료를 벌어 직원도 있습니다. 직원과 관련된 모든 정보는 emp 테이블에 저장됩니다.

샘플 회사는 여러 곳에 부서가 있습니다. 그래서 부서의 주소를 기록하고 있습니다. 각 직원은 부서에 속해 있으며 각 부서는 부서 번호와 약어로 식별됩니다. 또한 각 부처는 1 개의 주소에 연결되어 있습니다. 부서에 속한 모든 정보는 dept 테이블에 저장됩니다.

회사는 직원이 종사하고 있던 직업 정보를 기록하고 있습니다. 오랫동안 회사에서 일하고 있는 직원도 있고, 승진, 승급하는 직원도 있고, 부서가 바뀐 직원도 있습니다. 직원들의 상황이 바뀐 경우, 회사는 이전 직책 마지막 날을 기록합니다. 그리고, 새로운 일을 시작 날짜, 제목, 부서, 급여, 배치 전환의 이유를 새로운 일을 기록으로 추가합니다. 직원 기록의 모든 정보는 jobhist 테이블에 저장됩니다.

다음은 예제 데이터베이스의 테이블에 대한 관계를 보여주는 다이어그램입니다.

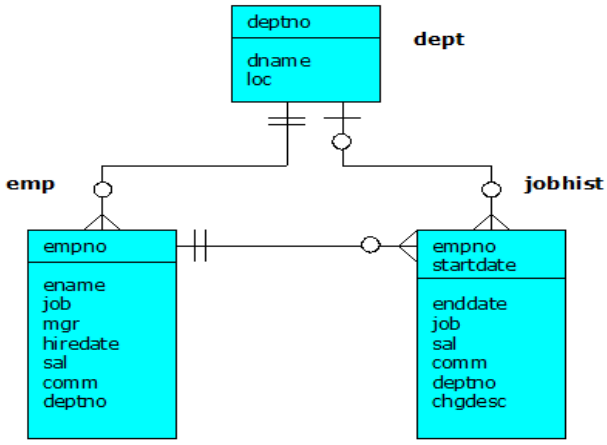

그림 1 데이터베이스 테이블

다음은 edb - sample.sql 스크립트입니다.

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

```
job VARCHAR2 (9)
mgr NUMBER (4),
hiredate DATE,
sal NUMBER (7,2) CONSTRAINT emp_sal_ck CHECK (sal> 0)
comm NUMBER (7,2),
deptno NUMBER (2) CONSTRAINT emp_ref_dept_fk
REFERENCES dept (deptno)
);
-
- Create the 'jobhist'table
-
CREATE TABLE jobhist (
empno NUMBER (4) NOT NULL,
startdate DATE NOT NULL,
enddate DATE,
job VARCHAR2 (9)
sal NUMBER (7,2),
comm NUMBER (7,2),
deptno NUMBER (2),
chgdesc VARCHAR2 (80),
CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate)
CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
REFERENCES emp (empno) ON DELETE CASCADE,
CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
REFERENCES dept (deptno) ON DELETE SET NULL,
CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
```

```
);
-
- Create the 'salesemp'view
-
CREATE OR REPLACE VIEW salesemp AS
SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
-
- Sequence to generate values for function 'new_empno'.
-
CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
-
- Issue PUBLIC grants
-
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
GRANT ALL ON next_empno TO PUBLIC;
-
- Load the 'dept'table
-
INSERT INTO dept VALUES (10, 'ACCOUNTING', 'NEW YORK');
INSERT INTO dept VALUES (20, 'RESEARCH', 'DALLAS');
INSERT INTO dept VALUES (30, 'SALES', 'CHICAGO');
INSERT INTO dept VALUES (40, 'OPERATIONS', 'BOSTON');
-
```

```
- Load the 'emp'table

-

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

INSERT INTO emp VALUES (7499, 'ALLEN', 'SALESMAN', 7698, '20 - FEB - 81 ',
1600,300,30);

INSERT INTO emp VALUES (7521, 'WARD', 'SALESMAN', 7698, '22 - FEB - 81 ',
1250,500,30);

INSERT INTO emp VALUES (7566, 'JONES', 'MANAGER', 7839, '02 - APR - 81 ',
2975, NULL, 20);

INSERT INTO emp VALUES (7654, 'MARTIN', 'SALESMAN', 7698, '28 - SEP - 81
', 1250,1400,30);

INSERT INTO emp VALUES (7698, 'BLAKE', 'MANAGER', 7839, '01 - MAY - 81 ',
2850, NULL, 30);

INSERT INTO emp VALUES (7782, 'CLARK', 'MANAGER', 7839, '09 - JUN - 81 ',
2450, NULL, 10);

INSERT INTO emp VALUES (7788, 'SCOTT', 'ANALYST', 7566, '19 - APR - 87 ',
3000, NULL, 20);

INSERT INTO emp VALUES (7839, 'KING', 'PRESIDENT', NULL, '17 - NOV - 81 ',
5000, NULL, 10);

INSERT INTO emp VALUES (7844, 'TURNER', 'SALESMAN', 7698, '08 - SEP - 81
', 1500,0,30);

INSERT INTO emp VALUES (7876, 'ADAMS', 'CLERK', 7788, '23 - MAY - 87 ',
1100, NULL, 20);

INSERT INTO emp VALUES (7900, 'JAMES', 'CLERK', 7698, '03 - DEC - 81 ',
950, NULL, 30);

INSERT INTO emp VALUES (7902, 'FORD', 'ANALYST', 7566, '03 - DEC - 81 ',
3000, NULL, 20);

INSERT INTO emp VALUES (7934, 'MILLER', 'CLERK', 7782, '23 - JAN - 82 ',
1300, NULL, 10);

-

- Load the 'jobhist'table

-

INSERT INTO jobhist VALUES (7369, '17 - DEC - 80 ', NULL,'CLERK ', 800,
NULL, 20,'New Hire ');
```

```
  INSERT INTO jobhist VALUES (7499, '20 - FEB - 81 ', NULL,'SALESMAN ',
1600,300,30,'New Hire ');

  INSERT INTO jobhist VALUES (7521, '22 - FEB - 81 ', NULL,'SALESMAN ',
1250,500,30,'New Hire ');

  INSERT INTO jobhist VALUES (7566, '02 - APR - 81 ', NULL,'MANAGER ', 2975,
NULL, 20,'New Hire ');

  INSERT INTO jobhist VALUES (7654, '28 - SEP - 81 ', NULL,'SALESMAN ',
1250,1400,30,'New Hire ');

  INSERT INTO jobhist VALUES (7698, '01 - MAY - 81 ', NULL,'MANAGER ', 2850,
NULL, 30,'New Hire ');

  INSERT INTO jobhist VALUES (7782, '09 - JUN - 81 ', NULL,'MANAGER ', 2450,
NULL, 10,'New Hire ');

  INSERT INTO jobhist VALUES (7788, '19 - APR - 87 ', '12 - APR - 88',
'CLERK', 1000, NULL, 20, 'New Hire');

  INSERT INTO jobhist VALUES (7788, '13 - APR - 88 ', '04 - MAY - 89',
'CLERK', 1040, NULL, 20, 'Raise');

  INSERT INTO jobhist VALUES (7788, '05 - MAY - 90 ', NULL,'ANALYST ', 3000,
NULL, 20,'Promoted to Analyst ');

  INSERT INTO jobhist VALUES (7839, '17 - NOV - 81 ', NULL,'PRESIDENT ',
5000, NULL, 10,'New Hire ');

  INSERT INTO jobhist VALUES (7844, '08 - SEP - 81 ', NULL,'SALESMAN ',
1500,0,30,'New Hire ');

  INSERT INTO jobhist VALUES (7876, '23 - MAY - 87 ', NULL,'CLERK ', 1100,
NULL, 20,'New Hire ');

  INSERT INTO jobhist VALUES (7900, '03 - DEC - 81 ', '14 - JAN - 83',
'CLERK', 950, NULL, 10, 'New Hire');

  INSERT INTO jobhist VALUES (7900, '15 - JAN - 83 ', NULL,'CLERK ', 950,
NULL, 30,'Changed to Dept 30 ');

  INSERT INTO jobhist VALUES (7902, '03 - DEC - 81 ', NULL,'ANALYST ', 3000,
NULL, 20,'New Hire ');

  INSERT INTO jobhist VALUES (7934, '23 - JAN - 82 ', NULL,'CLERK ', 1300,
NULL, 10,'New Hire ');

  -

  - Populate statistics table and view (pg_statistic / pg_stats)

  -
```

```
ANALYZE dept;

ANALYZE emp;

ANALYZE jobhist;

-

- Procedure that lists all employees 'numbers and names

- from the 'emp'table using a cursor.

-

CREATE OR REPLACE PROCEDURE list_emp

IS

v_empno NUMBER (4);

v_ename VARCHAR2 (10);

CURSOR emp_cur IS

SELECT empno, ename FROM emp ORDER BY empno;

BEGIN

OPEN emp_cur;

DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');

DBMS_OUTPUT.PUT_LINE ('----- -------');

LOOP

FETCH emp_cur INTO v_empno, v_ename;

EXIT WHEN emp_cur % NOTFOUND;

DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | v_ename);

END LOOP;

CLOSE emp_cur;

END;

/

-
```

```
- Procedure that selects an employee row given the employee
- number and displays certain columns.
-
CREATE OR REPLACE PROCEDURE select_emp (
p_empno IN NUMBER
)
IS
v_ename emp.ename % TYPE;
v_hiredate emp.hiredate % TYPE;
v_sal emp.sal % TYPE;
v_comm emp.comm % TYPE;
v_dname dept.dname % TYPE;
v_disp_date VARCHAR2 (10);
BEGIN
SELECT ename, hiredate, sal, NVL (comm, 0), dname
INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
FROM emp e, dept d
WHERE empno = p_empno
AND e.deptno = d.deptno;
v_disp_date : = TO_CHAR (v_hiredate, 'MM / DD / YYYY');
DBMS_OUTPUT.PUT_LINE ( 'Number :'| | p_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | v_ename);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | v_disp_date);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | v_sal);
DBMS_OUTPUT.PUT_LINE ( 'Commission :'| | v_comm);
DBMS_OUTPUT.PUT_LINE ( 'Department :'| | v_dname);
```

```
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Employee'| | p_empno | | 'not found');
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'The following is SQLERRM :');
DBMS_OUTPUT.PUT_LINE (SQLERRM);
DBMS_OUTPUT.PUT_LINE ( 'The following is SQLCODE :');
DBMS_OUTPUT.PUT_LINE (SQLCODE);
END;
/
-
- Procedure that queries the 'emp'table based on
- department number and employee number or name. Returns
- employee number and name as IN OUT parameters and job,
- hire date, and salary as OUT parameters.
-
CREATE OR REPLACE PROCEDURE emp_query (
p_deptno IN NUMBER,
p_empno IN OUT NUMBER,
p_ename IN OUT VARCHAR2,
p_job OUT VARCHAR2,
p_hiredate OUT DATE,
p_sal OUT NUMBER
)
IS
BEGIN
```

```
SELECT empno, ename, job, hiredate, sal

INTO p_empno, p_ename, p_job, p_hiredate, p_sal

FROM emp

WHERE deptno = p_deptno

AND (empno = p_empno

OR ename = UPPER (p_ename));

END;

/

-

- Procedure to call 'emp_query_caller'with IN and IN OUT

- parameters. Displays the results received from IN OUT and

- OUT parameters.

-

CREATE OR REPLACE PROCEDURE emp_query_caller

IS

v_deptno NUMBER (2);

v_empno NUMBER (4);

v_ename VARCHAR2 (10);

v_job VARCHAR2 (9);

v_hiredate DATE;

v_sal NUMBER;

BEGIN

v_deptno : = 30;

v_empno : = 0;

v_ename : = 'Martin';

emp_query (v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
```

```
DBMS_OUTPUT.PUT_LINE ( 'Department :'| | v_deptno);

DBMS_OUTPUT.PUT_LINE ( 'Employee No :'| | v_empno);

DBMS_OUTPUT.PUT_LINE ( 'Name :'| | v_ename);

DBMS_OUTPUT.PUT_LINE ( 'Job :'| | v_job);

DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | v_hiredate);

DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | v_sal);

EXCEPTION

WHEN TOO_MANY_ROWS THEN

DBMS_OUTPUT.PUT_LINE ( 'More than one employee was selected');

WHEN NO_DATA_FOUND THEN

DBMS_OUTPUT.PUT_LINE ( 'No employees were selected');

END;

/

-

- Function to compute yearly compensation based on semimonthly

- salary.

-

CREATE OR REPLACE FUNCTION emp_comp (

p_sal NUMBER,

p_comm NUMBER

) RETURN NUMBER

IS

BEGIN

RETURN (p_sal + NVL (p_comm, 0)) * 24;

END;

/
```

```
-
- Function that gets the next number from sequence, 'next_empno',
- and ensures it is not already in use as an employee number.
-
CREATE OR REPLACE FUNCTION new_empno RETURN NUMBER
IS
v_cnt INTEGER : = 1;
v_new_empno NUMBER;
BEGIN
WHILE v_cnt> 0 LOOP
SELECT next_empno.nextval INTO v_new_empno FROM dual;
SELECT COUNT (*) INTO v_cnt FROM emp WHERE empno = v_new_empno;
END LOOP;
RETURN v_new_empno;
END;
/
-
- EDB - SPL function that adds a new clerk to table 'emp'. This function
- uses package 'emp_admin'.
-
CREATE OR REPLACE FUNCTION hire_clerk (
p_ename VARCHAR2,
p_deptno NUMBER
) RETURN NUMBER
IS
v_empno NUMBER (4);
```

```
v_ename VARCHAR2 (10);
v_job VARCHAR2 (9);
v_mgr NUMBER (4);
v_hiredate DATE;
v_sal NUMBER (7,2);
v_comm NUMBER (7,2);
v_deptno NUMBER (2);
BEGIN
v_empno : = new_empno;
INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
TRUNC (SYSDATE), 950.00, NULL, p_deptno);
SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
FROM emp WHERE empno = v_empno;
DBMS_OUTPUT.PUT_LINE ( 'Department :'| | v_deptno);
DBMS_OUTPUT.PUT_LINE ( 'Employee No :'| | v_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | v_ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | v_job);
DBMS_OUTPUT.PUT_LINE ( 'Manager :'| | v_mgr);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | v_hiredate);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | v_sal);
DBMS_OUTPUT.PUT_LINE ( 'Commission :'| | v_comm);
RETURN v_empno;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'The following is SQLERRM :');
```

```
DBMS_OUTPUT.PUT_LINE (SQLERRM);
DBMS_OUTPUT.PUT_LINE ( 'The following is SQLCODE :');
DBMS_OUTPUT.PUT_LINE (SQLCODE);
RETURN -1;
END;
/
-
- PostgreSQL PL / pgSQL function that adds a new salesman
- to table 'emp'.
-
CREATE OR REPLACE FUNCTION hire_salesman (
p_ename VARCHAR,
p_sal NUMERIC,
p_comm NUMERIC
) RETURNS NUMERIC
AS $ $
DECLARE
v_empno NUMERIC (4);
v_ename VARCHAR (10);
v_job VARCHAR (9);
v_mgr NUMERIC (4);
v_hiredate DATE;
v_sal NUMERIC (7,2);
v_comm NUMERIC (7,2);
v_deptno NUMERIC (2);
BEGIN
```

```
v_empno : = new_empno ();
INSERT INTO emp VALUES (v_empno, p_ename, 'SALESMAN', 7698,
CURRENT_DATE, p_sal, p_comm, 30);
SELECT INTO
v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
empno, ename, job, mgr, hiredate, sal, comm, deptno
FROM emp WHERE empno = v_empno;
RAISE INFO 'Department : %', v_deptno;
RAISE INFO 'Employee No : %', v_empno;
RAISE INFO 'Name : %', v_ename;
RAISE INFO 'Job : %', v_job;
RAISE INFO 'Manager : %', v_mgr;
RAISE INFO 'Hire Date : %', v_hiredate;
RAISE INFO 'Salary : %', v_sal;
RAISE INFO 'Commission : %', v_comm;
RETURN v_empno;
EXCEPTION
WHEN OTHERS THEN
RAISE INFO 'The following is SQLERRM :';
RAISE INFO '%', SQLERRM;
RAISE INFO 'The following is SQLSTATE :';
RAISE INFO '%', SQLSTATE;
RETURN -1;
END;
$ $ LANGUAGE 'plpgsql';
/
```

```
-
- Rule to INSERT into view 'salesemp'
-
CREATE OR REPLACE RULE salesemp_i AS ON INSERT TO salesemp
DO INSTEAD
INSERT INTO emp VALUES (NEW.empno, NEW.ename, 'SALESMAN', 7698,
NEW.hiredate, NEW.sal, NEW.comm, 30);
-
- Rule to UPDATE view 'salesemp'
-
CREATE OR REPLACE RULE salesemp_u AS ON UPDATE TO salesemp
DO INSTEAD
UPDATE emp SET empno = NEW.empno,
ename = NEW.ename,
hiredate = NEW.hiredate,
sal = NEW.sal,
comm = NEW.comm
WHERE empno = OLD.empno;
-
- Rule to DELETE from view 'salesemp'
-
CREATE OR REPLACE RULE salesemp_d AS ON DELETE TO salesemp
DO INSTEAD
DELETE FROM emp WHERE empno = OLD.empno;
-
- After statement - level trigger that displays a message after
```

```
- an insert, update, or deletion to the 'emp'table. One message

- per SQL command is displayed.

-

CREATE OR REPLACE TRIGGER user_audit_trig

AFTER INSERT OR UPDATE OR DELETE ON emp

DECLARE

v_action VARCHAR2 (24);

BEGIN

IF INSERTING THEN

v_action : = 'added employee (s) on';

ELSIF UPDATING THEN

v_action : = 'updated employee (s) on';

ELSIF DELETING THEN

v_action : = 'deleted employee (s) on';

END IF;

DBMS_OUTPUT.PUT_LINE ( 'User'| | USER | | v_action | | TO_CHAR (SYSDATE,
'YYYY - MM - DD'));

END;

/

-

- Before row - level trigger that displays employee number and

- salary of an employee that is about to be added, updated,

- or deleted in the 'emp'table.

-

CREATE OR REPLACE TRIGGER emp_sal_trig

BEFORE DELETE OR INSERT OR UPDATE ON emp

FOR EACH ROW
```

```
DECLARE

sal_diff NUMBER;

BEGIN

IF INSERTING THEN

DBMS_OUTPUT.PUT_LINE ( 'Inserting employee'| | : NEW.empno);

DBMS_OUTPUT.PUT_LINE ( '.. New salary :'| | : NEW.sal);

END IF;

IF UPDATING THEN

sal_diff : = : NEW.sal - : OLD.sal;

DBMS_OUTPUT.PUT_LINE ( 'Updating employee'| | : OLD.empno);

DBMS_OUTPUT.PUT_LINE ( '.. Old salary :'| | : OLD.sal);

DBMS_OUTPUT.PUT_LINE ( '.. New salary :'| | : NEW.sal);

DBMS_OUTPUT.PUT_LINE ( '.. Raise :'| | sal_diff);

END IF;

IF DELETING THEN

DBMS_OUTPUT.PUT_LINE ( 'Deleting employee'| | : OLD.empno);

DBMS_OUTPUT.PUT_LINE ( '.. Old salary :'| | : OLD.sal);

END IF;

END;

/

-

- Package specification for the 'emp_admin'package.

-

CREATE OR REPLACE PACKAGE emp_admin

IS

FUNCTION get_dept_name (
```

```
p_deptno NUMBER
) RETURN VARCHAR2;
FUNCTION update_emp_sal (
p_empno NUMBER,
p_raise NUMBER
) RETURN NUMBER;
PROCEDURE hire_emp (
p_empno NUMBER,
p_ename VARCHAR2,
p_job VARCHAR2,
p_sal NUMBER,
p_hiredate DATE,
p_comm NUMBER,
p_mgr NUMBER,
p_deptno NUMBER
);
PROCEDURE fire_emp (
p_empno NUMBER
);
END emp_admin;
/
-
- Package body for the 'emp_admin'package.
-
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
```

```
-
- Function that queries the 'dept'table based on the department
- number and returns the corresponding department name.
-
FUNCTION get_dept_name (
p_deptno IN NUMBER
) RETURN VARCHAR2
IS
v_dname VARCHAR2 (14);
BEGIN
SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
RETURN v_dname;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Invalid department number'| | p_deptno);
RETURN '';
END;
-
- Function that updates an employee 's salary based on the
- employee number and salary increment / decrement passed
- as IN parameters. Upon successful completion the function
- returns the new updated salary.
-
FUNCTION update_emp_sal (
p_empno IN NUMBER,
p_raise IN NUMBER
```

```
) RETURN NUMBER

IS

v_sal NUMBER : = 0;

BEGIN

SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;

v_sal : = v_sal + p_raise;

UPDATE emp SET sal = v_sal WHERE empno = p_empno;

RETURN v_sal;

EXCEPTION

WHEN NO_DATA_FOUND THEN

DBMS_OUTPUT.PUT_LINE ( 'Employee'| | p_empno | | 'not found');

RETURN -1;

WHEN OTHERS THEN

DBMS_OUTPUT.PUT_LINE ( 'The following is SQLERRM :');

DBMS_OUTPUT.PUT_LINE (SQLERRM);

DBMS_OUTPUT.PUT_LINE ( 'The following is SQLCODE :');

DBMS_OUTPUT.PUT_LINE (SQLCODE);

RETURN -1;

END;

-

- Procedure that inserts a new employee record into the 'emp'table.

-

PROCEDURE hire_emp (

p_empno NUMBER,

p_ename VARCHAR2,

p_job VARCHAR2,
```

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

다음 항목보다는 SQL 명령의 기본을 설명합니다.

### 2.1.2 새 테이블 만들기

테이블과 테이블의 모든 열 이름과 형식을 지정하면 새 테이블을 만들 수 있습니다. 이번 예제에서는 테이블을 정의하는 데 필요한 최소한의 정보로 짧은 emp 테이블을 만듭니다.

```
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)
);
```

명령을 여러 줄에 PSQL 입력하실 수 있습니다. PSQL 은 세미콜론으로 끝날 때까지 그 명령은 계속하는 것으로 인식합니다.

SQL 명령은 공백 문자 (즉 공백, 탭, 줄 바꿈)를 자유롭게 사용할 수 있습니다. 즉, 위에 언급한 명령과는 다른 형태로 입력할 수 있다는 것을 의미합니다. 모든 내용을 1 줄에 입력하는 수도 있습니다. 2 개의 하이픈 ("--")은 코멘트를 시작합니다. 입력을 발견했을 때는 줄 끝까지 무시됩니다. SQL 은 키워드와 식별자에 대해 대소문자를 구별하지 않습니다. 그러나 식별자를 인용 부호로 둘러싸있을 경우 대소문자를 구분합니다.(위와는 다르게)

VARCHAR2 (10)는 10 문자의 텍스트를 저장할 수 있는 데이터 형식을 지정합니다. NUMBER (7,2)은 정확도가 7 스케일이 2 인 고정 소수점입니다. NUMBER (4)는 정확도가 4, 스케일 0 의 정수입니다.

Postgres Plus Advanced Server 는 표준 SQL 데이터 형식, INTEGER, SMALLINT, NUMBER, REAL, DOUBLE PRECISION, CHAR, VARCHAR2, DATE 과 TIMESTAMP 를 지원합니다. 이 데이터 형식 동의어도 지원합니다.

테이블이 더 이상 필요하지 않거나 다른 것을 새롭게 하고 싶다면 다음 명령을 사용하여 제거할 수 있습니다.

*DROP* TABLE tablename;

### 2.1.3 테이블에 행을 삽입

다음과 같이, INSERT 문을 사용하여 테이블에 행을 삽입합니다.

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

모든 데이터 형식 중 하나라고 하면 이해할 수 있는 입력 형식을 사용하고 있다는 것을 명심하십시오. 일반적으로 단순한 숫자가 아닌 상수는 이 예와 같이 작은 따옴표 ( ')로 입력해야 합니다. DATE 형식 승인은 실제로 매우 유연합니다. 그러나 이 연습에서는 애매함이 없는 형식에 집착하는 것입니다.

위의 구문은 열 순서를 기억할 필요가 있습니다. 다음 구문은 열 목록을 명시적으로 부여할 수 있습니다.

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

목록에 있는 줄은 원하는 순서대로 지정할 수 있습니다. 또 일부 열을 생략할 수 있습니다. 예를 들어 수수료를 모르면 다음과 같이 할 수 있습니다.

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

많은 개발자들은 절대적인 순서에 의존하는 것보다 열 목록을 명시적으로 지정하는 방법을 즐겨 사용합니다.

### 2.1.4 테이블의 쿼리

테이블에서 데이터를 뽑아 내기 위해 *테이블에* 쿼리를 합니다. 이를 위해 SQL SELECT 문이 사용됩니다. 이 문장은 선택 목록 (반환되는 열 목록 부분)과 테이블 목록 (데이터를 검색할

테이블의 목록 부분) 및 선택적 조건 (제한을 지정하는 부분)으로 나눌 수 있습니다. 예를 들면, emp 테이블의 모든 직원의 모든 열을 추출하려면 다음과 같은 쿼리를 합니다.

```
SELECT * FROM emp;
```

여기서 선택 목록에서 "*"은 "모든"과 "열"의 약어입니다. 이 쿼리의 결과는 다음과 같습니다.

```
 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)
```

선택 목록에서 원하는 형식을 사용하실 수 있습니다. 예를 들면 다음과 같습니다.

```
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 절을 사용하여 출력 열 다시 분류 부분에 주의하시기 바랍니다 (AS 어구는 생략할 수 있습니다.)

쿼리는 필요한 내용이 무엇인지 지정 WHERE 절을 추가하여 쿼리에 지정할 수 있습니다. WHERE 행에는 논리 (True 값) 식을 갖고, 이 계산식이 진정한 하는 행만 반환합니다. 자주 사용하는 논리 연산자 (AND, OR, NOT)을 사용할 수 있습니다.  예를 들어, 다음은 부서 번호가 20 에서 월급이 1000 달러가 넘는 직원 데이터를 추출합니다.

```
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)
```

쿼리의 결과를 정렬하여 반환하도록 지정할 수 있습니다.

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


ename | sal | deptno

--------+---------+--------

ADAMS | 1100.00 | 20

ALLEN | 1600.00 | 30

BLAKE | 2850.00 | 30

CLARK | 2450.00 | 10

FORD | 3000.00 | 20

JAMES | 950.00 | 30

JONES | 2975.00 | 20

KING | 5000.00 | 10

MARTIN | 1250.00 | 30

MILLER | 1300.00 | 10

SCOTT | 3000.00 | 20

SMITH | 800.00 | 20

TURNER | 1500.00 | 30

WARD | 1250.00 | 30

(14 rows)
```

쿼리 결과에서 중복 행을 제외하도록 지정할 수 있습니다.

```
SELECT DISTINCT job FROM emp;


job

-----------

ANALYST

CLERK

MANAGER

PRESIDENT

SALESMAN

(5 rows)
```

다음 절에서 하나의 쿼리에 여러 테이블에서 행을 획득하는 방법을 설명합니다.

### 2.1.5 테이블 간의 결합

여기까지의 쿼리들은 한 번에 1 개의 테이블에만 액세스할 수 있었습니다. 쿼리들은 한 번에 여러 테이블에 접근하는 것도 가능하고, 동시에 테이블의 여러 행에 작업을 수행할 경우, 같은 테이블에 액세스할 수 있습니다. 동시에 같은 테이블이나 여러 테이블의 여러 행에 액세스하는 것은 결합 쿼리입니다. 예를 들면, 관련 부서의 이름과 위치를 직원 데이터와 함께 표시할 경우입니다. 이를 위해서는 emp 테이블에 각 행의 deptno 열과, dept 테이블의 모든 행 deptno 열의를 비교하고 두 값이 일치하는 행의 조합을 선택하여야 합니다.  이것은 다음과 같은 쿼리를 통해 수행할 수 있습니다.

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


ename | sal | deptno | dname | loc

--------+---------+--------+------------+--------- -

MILLER | 1300.00 | 10 | ACCOUNTING | NEW YORK

CLARK | 2450.00 | 10 | ACCOUNTING | NEW YORK

KING | 5000.00 | 10 | ACCOUNTING | NEW YORK
```

```
SCOTT | 3000.00 | 20 | RESEARCH | DALLAS

JONES | 2975.00 | 20 | RESEARCH | DALLAS

SMITH | 800.00 | 20 | RESEARCH | DALLAS

ADAMS | 1100.00 | 20 | RESEARCH | DALLAS

FORD | 3000.00 | 20 | RESEARCH | DALLAS

WARD | 1250.00 | 30 | SALES | CHICAGO

TURNER | 1500.00 | 30 | SALES | CHICAGO

ALLEN | 1600.00 | 30 | SALES | CHICAGO

BLAKE | 2850.00 | 30 | SALES | CHICAGO

MARTIN | 1250.00 | 30 | SALES | CHICAGO

JAMES | 950.00 | 30 | SALES | CHICAGO

(14 rows)
```

이 결과에서 두 개의 결과를 알 수 있습니다.

- 부서 번호 40 에 대한 결과 행은 없습니다. emp 테이블에는 부서 번호 40 과 일치하는 항목이 아니라 결합 시 dept 테이블에서 일치하지 않는 부분은 무시되기 때문입니다. 이것이 어떻게 만들어지는지를 쉽게 알아 봅니다.

- 출력 열 목록 *을 사용하거나 다음과 같이 일부 테이블 이름을 지정하지 않는 것 보다, 명시적으로 테이블 이름을 사용하는 것이 바람직합니다.

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

열은 각각 다른 이름 (deptno 줄은 중복되므로 열 이름을 테이블 이름으로 한정해야 됩니다)이므로, 해석기는 자동으로 어떤 테이블의 열 또는 확인할 수 있습니다. 그러나 결합 쿼리는 열 이름을 테이블 이름으로 정규화하는 것이 좋다.

이러한 결합 쿼리는 다음과 같이 다른 형태로 나타내는 수도 있습니다.

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

이 문법은 실제 경우보다 일반적으로 사용되는 것은 아니지만 이후 주제의 이해를 돕기 위해 보여지고 있습니다.

이러한 결합 결과를 모두 보면 부서 번호 40 에 속한 직원은 없으며, 결과적으로 부서 번호 40 의 기록이 표시되지 않는다는 것을 알 수 있습니다. 여기에 일치하는 직원이 없는데도 우리는 부서 번호 40 의 기록을 얻을 수 있게 될 것입니다. 실행하고 싶은 쿼리는, dept 테이블을 검사하고, 각 행에 대해 emp 줄에 일치하는지 확인합니다. 일치하지 않는 행이 있으면, emp 테이블의 열 부분을 어떻게 "빈 값"을 대신하려고 합니다. 이러한 쿼리를 *외부 조인이라고 합니다* (지금까지의 join 은 내부 join 입니다.) 다음과 같은 명령입니다.

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

```
 ename | sal | deptno | dname | loc

--------+---------+--------+------------+--------- -

MILLER | 1300.00 | 10 | ACCOUNTING | NEW YORK

CLARK | 2450.00 | 10 | ACCOUNTING | NEW YORK

KING | 5000.00 | 10 | ACCOUNTING | NEW YORK

SCOTT | 3000.00 | 20 | RESEARCH | DALLAS

JONES | 2975.00 | 20 | RESEARCH | DALLAS

SMITH | 800.00 | 20 | RESEARCH | DALLAS

ADAMS | 1100.00 | 20 | RESEARCH | DALLAS

FORD | 3000.00 | 20 | RESEARCH | DALLAS

WARD | 1250.00 | 30 | SALES | CHICAGO

TURNER | 1500.00 | 30 | SALES | CHICAGO

ALLEN | 1600.00 | 30 | SALES | CHICAGO

BLAKE | 2850.00 | 30 | SALES | CHICAGO

MARTIN | 1250.00 | 30 | SALES | CHICAGO

JAMES | 950.00 | 30 | SALES | CHICAGO

| | 40 | OPERATIONS | BOSTON

(15 rows)
```

이 쿼리를 *왼쪽 외부 조인* 이라고 합니다. 결합 연산자의 왼쪽에 지정한 테이블의 각 행에 최소한 한 번 기록되고 반면에 오른쪽 테이블은 왼쪽 테이블 행과 일치하는 것만이 기록되기 때문입니다. 오른쪽의 테이블과 일치하지 않는 왼쪽 테이블 줄을 출력 시 오른쪽 테이블의 열은 비어 (null)로 대체합니다.

외부 조인을 표현하는 구문으로 WHERE 절에 결합 조건으로 외부 조인 연산자 "(+)"를 사용하는 방법도 있습니다. 외부 조인 연산자는 테이블의 열 이름 다음에 넣습니다. 일치하지 않는 행이 있는 경우 해당 테이블의 열 이름은 null 값으로 바뀝니다. 즉, dept 테이블 행에 대해 emp 테이블에 일치하는 행이 없으면 Postgres Plus Advanced Server 는 emp 열이 포함 선택 목록적으로 null 값을 반환합니다. 위 왼쪽 외부 조인 예제는 다음과 같이 다시 작성할 수 있습니다.

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

```
 ename | sal | deptno | dname | loc

--------+---------+--------+------------+--------- -

MILLER | 1300.00 | 10 | ACCOUNTING | NEW YORK

CLARK | 2450.00 | 10 | ACCOUNTING | NEW YORK

KING | 5000.00 | 10 | ACCOUNTING | NEW YORK

SCOTT | 3000.00 | 20 | RESEARCH | DALLAS

JONES | 2975.00 | 20 | RESEARCH | DALLAS

SMITH | 800.00 | 20 | RESEARCH | DALLAS

ADAMS | 1100.00 | 20 | RESEARCH | DALLAS

FORD | 3000.00 | 20 | RESEARCH | DALLAS

WARD | 1250.00 | 30 | SALES | CHICAGO

TURNER | 1500.00 | 30 | SALES | CHICAGO

ALLEN | 1600.00 | 30 | SALES | CHICAGO

BLAKE | 2850.00 | 30 | SALES | CHICAGO

MARTIN | 1250.00 | 30 | SALES | CHICAGO

JAMES | 950.00 | 30 | SALES | CHICAGO
```

```
| | 40 | OPERATIONS | BOSTON

(15 rows)
```

테이블을 자신에게 결합시킬 수 있습니다. 이것은 *자기 결합이라고 합니다.* 예를 들어, 각 직원의 이름과 함께 직원 관리자의 이름도 만회하는 것을 생각해 볼 수 있습니다. 만일 그렇다면, emp 행 mgr 열과 다른 모든 emp 행 empno 행을 비교해야 합니다.

```
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)
```

여기서는 선택 목록과 결합 조건 속에서 직원 정보를 표현하는 emp 테이블에 e1 라는 레이블이 붙어있습니다. 또한, 관리자의 직원 정보를 표현하는 emp 테이블에 e2 라는 레이블이 붙어있습니다. 일의 간단하게 하기 위하여 다른 쿼리에서도 이러한 별칭을 사용할 수 있습니다. 다음은 예제입니다.

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

```
 ename | mgr | deptno | dname | loc

--------+------+--------+------------+----------

 MILLER | 7782 | 10 | ACCOUNTING | NEW YORK

 CLARK | 7839 | 10 | ACCOUNTING | NEW YORK

 KING | | 10 | ACCOUNTING | NEW YORK

 SCOTT | 7566 | 20 | RESEARCH | DALLAS

 JONES | 7839 | 20 | RESEARCH | DALLAS

 SMITH | 7902 | 20 | RESEARCH | DALLAS

 ADAMS | 7788 | 20 | RESEARCH | DALLAS

 FORD | 7566 | 20 | RESEARCH | DALLAS

 WARD | 7698 | 30 | SALES | CHICAGO

 TURNER | 7698 | 30 | SALES | CHICAGO

 ALLEN | 7698 | 30 | SALES | CHICAGO

 BLAKE | 7839 | 30 | SALES | CHICAGO

 MARTIN | 7698 | 30 | SALES | CHICAGO

 JAMES | 7698 | 30 | SALES | CHICAGO

 (14 rows)
```

이러한 형태의 요약은 상당히 잘 이루어집니다.

### 2.1.6 집계 함수

다른 대부분의 관계형 데이터베이스 제품뿐만 아니라 Postgres Plus Advanced Server 은 집계 함수를 지원합니다. 집계 함수는 여러 행에서 1 개의 결과를 계산합니다. 예를 들면, 행 집합에 대해 COUNT (총계), SUM (총수), AVG (평균), MAX (최대), MIN (최소)와 같은 연산을 수행 집계가 있습니다.

예를 들어, 다음 쿼리에서 모든 직원의 봉급에서 가장 높은 월급과 가장 낮은 월급을 찾을 수 있습니다.

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


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

가장 임금이 높은 직원의 이름을 알고 싶다면 다음과 같은 쿼리를 실행할 수도 있습니다.

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


ERROR : aggregates not allowed in WHERE clause
```

그러나 MAX 집계 함수를 WHERE 절에 사용할 수 없기 때문에 이 명령이 작동하지 않습니다 (WHERE 절에 어떤 행을 집계 처리에 전달할 것인가를 결정하는 것이며, 따라서 집계 함수를 계산하기 전으로 평가되어야 하는 것은 당연하다.이 때문에 제한이 있습니다). 하지만 쿼리를 다시 작성하면 의도한 결과를 얻을 수 있습니다. 여기에는 다음과 같은 *서브 쿼리* 를 사용합니다.

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


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

서브쿼리는 외부의 쿼리에 별도로 결과를 얻는 독립적인 연산입니다.

또 GROUP BY 절과 결합한 통합은 아주 유용합니다. 예를 들어, 다음 쿼리에서 부서별로 가장 높은 급여를 결정할 수 있습니다.

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

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

여기에는 부서별로 1 행의 출력이 있습니다. 각각의 집계 결과는 각 부서에 맞는 테이블 행 전체에 대한 계산 결과입니다. 다음과 같이, HAVING 어구를 사용하여 그룹화된 행을 구별할 수 있습니다.

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

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

이 쿼리는 평균 월급이 2000 달러가 넘는 부서만을 출력합니다.

마지막으로, 다음 쿼리는 애널리스트의 평균 월급이 2000 달러가 넘는 부서에서 애널리스트의 가장 높은 급료만을 출력합니다.

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

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

WHERE 절과 HAVING 절에 미묘한 차이가 있습니다. WHERE 절은 그룹이나 집계를 계산하기 전에 입을 선택합니다. 한편 HAVING 절은 그룹과 집계를 계산하면, 그룹화된 행을 선택합니다.

앞의 예제에서는 애널리스트의 직원 전용입니다. 그 중에서, 애널리스트를 부서별로 그룹화하고, 애널리스트의 평균 월급이 2000 달러 이상 그룹만을 마지막 결과합니다. 그 결과, 부서 번호 20 그룹에만 적용되며 그 중 가장 높은 애널리스트의 월급은 3000 달러입니다.

### 2.1.7 업데이트

UPDATE 명령을 사용하여 기존 행의 열 값을 업데이트할 수 있습니다. 예를 들면, 다음 명령은 관리자 모두에 대해 10% 절감을 하기 전과 후를 보여줍니다.

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


ename | sal

-------+---------

JONES | 2975.00

BLAKE | 2850.00

CLARK | 2450.00

(3 rows)


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


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


ename | sal

-------+---------

JONES | 3272.50

BLAKE | 3135.00
```

```
CLARK | 2695.00

(3 rows)
```

### 2.1.8 삭제

DELETE 명령을 사용하여 테이블에서 행을 삭제할 수 있습니다. 예를 들면, 다음 명령은 부서 번호 20 의 모든 직원을 제거하기 전과 후를 보여줍니다.

```
SELECT ename, deptno FROM emp;


ename | deptno

--------+--------

SMITH | 20

ALLEN | 30

WARD | 30

JONES | 20

MARTIN | 30

BLAKE | 30

CLARK | 10

SCOTT | 20

KING | 10

TURNER | 30

ADAMS | 20

JAMES | 30

FORD | 20

MILLER | 10

(14 rows)
```

```
DELETE FROM emp WHERE deptno = 20;


SELECT ename, deptno FROM emp;

ename | deptno

--------+--------

ALLEN | 30

WARD | 30

MARTIN | 30

BLAKE | 30

CLARK | 10

KING | 10

TURNER | 30

JAMES | 30

MILLER | 10
```

아래와 같이 WHERE구 없이 DELETE명령문이 주어지는 것을 주의해 주십시오. 이 명령문은 주어진 그것을 떠나 완전히 비어있는 모든 행을 지울 것입니다. 그 시스템은 이것을 하기 전에 확정을 요구되지 않을 것입니다.

### 2.2 고급 제반 기능

이전 장에서는 Postgres Plus Advanced Server 에서 SQL 을 사용하여 데이터를 저장하거나 접속할 기본 사항을 설명했습니다. 이 장에서는 관리를 단순화하고, 데이터 손실 및 손상을 방지하는 SQL 의 고급 기능에 대해 설명합니다.

#### 2.2.1 뷰

우선 다음 SELECT 명령을 보십시오.

```
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)
```

이것을 반복되는 쿼리일 경우, 아래에서 보이는 *뷰*를 생성할 때 마다 전체 SELECT 명령을 다시 타이핑하는 것 없이 쿼리의 속기 방식을 사용할 수 있습니다.

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

employee_pay 뷰는 쿼리로 인해 일반 테이블처럼 사용할 수 있습니다.

```
 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)
```

뷰의 사용자의 자유로운 측면은 좋은 SQL 데이터베이스를 디자인하데 중요한 측면이다. 테이블 구조의 세부 정보를 캡슐화 하므로 애플리케이션 업데이트를 통해 테이블 구조가 바뀌었다 하더라도 일관된 인터페이스 - 페이스를 유지할 수 있습니다.

의견의 자유를 사용하여 SQL 데이터베이스를 만드는 좋은 디자인의 핵심 측면이다.

뷰는 실제 테이블에 사용할 수 있는 대부분의 경우에 사용할 수 있습니다. 뷰에 대한 보기를 만드는 것은 일반적입니다.

### 2.2.2 외래 키

모든 직원을 유요한 부서에 소속하기를 원합니다. 이것을 데이터 *참조 무결성* 유지 보수라고 합니다. 가장 간단한 데이터베이스는 dept 테이블에 일치하는 행이 있는지 여부를 먼저 확인하고, emp 테이블에 새 직원 레코드를 삽입하거나 거부하도록 구현해야 합니다. 이 방법에는

많은 문제가 있어 매우 불편합니다. Postgres Plus Advanced Server 에서는 이것을 좀더 쉽게 얻을 수 있습니다.

여기서는 제 2.1.2 장의 emp 테이블에 외래 키 제약 조건을 추가하도록 변경합니다. 수정된 emp 테이블은 다음과 같습니다.

```
CREATE TABLE emp (

empno NUMBER (4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,

ename VARCHAR2 (10),

job VARCHAR2 (9)

mgr NUMBER (4),

hiredate DATE,

sal NUMBER (7,2),

comm NUMBER (7,2),

deptno NUMBER (2) CONSTRAINT emp_ref_dept_fk

REFERENCES dept (deptno)

);
```

샘플 emp 테이블에 다음 INSERT 명령을 실행하면 외래 키 제약 조건 emp_ref_dept_fk 는 부서 번호 50 가 dept 테이블에 있는지 확인합니다. 그러나 없기 때문에 명령이 거부됩니다.

```
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"
```

외부 키의 기능으로 응용 프로그램을 훌륭하게 조율되었습니다. 외래 키를 올바르게 사용하면 틀림없이 데이터베이스 응용 프로그램의 질을 향상시킬 수 있으므로 프로그램의 향상에 몰두하고 있습니다.

### 2.2.3 ROWNUM 가짜 열

ROWNUM 은 가짜 열입니다. 쿼리에서의 순서에 따라 각 행에 고유 오름차순 숫자가 지정됩니다. 따라서 처음에 복구하는 줄은 ROWNUM 로 1 가 할당됩니다. 2 번째 줄은 2 에 지정되며 다음처럼 나타납니다.

이 기능은 다음과 같이 쿼리에서 복구 행 수를 제한하는 데 사용할 수 있습니다.

```
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 값은 결과를 정렬하기 전에 각 행에 할당됩니다. 따라서 ORDER BY 절이 지정되면 결과는 정렬됩니다. ROWNUM 값은 다음과 같이 반드시 오름차순으로 하지 않습니다.

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


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

다음은 jobhist 테이블의 각 행에 일련 번호를 추가하는 방법을 보여줍니다. 먼저 seqno 라는 새 열을 테이블에 추가합니다. 그리고, UPDATE 명령 seqno 에 ROWNUM 을 설정합니다.

```
ALTER TABLE jobhist ADD seqno NUMBER (3);

UPDATE jobhist SET seqno = ROWNUM;
```

다음 SELECT 명령 새로운 seqno 열의 값을 표시합니다.

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

```
seqno | empno | start | job

-------+-------+-----------+-----------

1 | 7369 | 17 - DEC - 80 | CLERK

2 | 7499 | 20 - FEB - 81 | SALESMAN

3 | 7521 | 22 - FEB - 81 | SALESMAN

4 | 7566 | 02 - APR - 81 | MANAGER

5 | 7654 | 28 - SEP - 81 | SALESMAN

6 | 7698 | 01 - MAY - 81 | MANAGER

7 | 7782 | 09 - JUN - 81 | MANAGER

8 | 7788 | 19 - APR - 87 | CLERK

9 | 7788 | 13 - APR - 88 | CLERK

10 | 7788 | 05 - MAY - 90 | ANALYST

11 | 7839 | 17 - NOV - 81 | PRESIDENT

12 | 7844 | 08 - SEP - 81 | SALESMAN

13 | 7876 | 23 - MAY - 87 | CLERK

14 | 7900 | 03 - DEC - 81 | CLERK

15 | 7900 | 15 - JAN - 83 | CLERK

16 | 7902 | 03 - DEC - 81 | ANALYST

17 | 7934 | 23 - JAN - 82 | CLERK

(17 rows)
```

### 2.2.4 동의어

*동의어는* SQL 문장에서 데이터베이스 개체를 다른 이름으로 참조하는 데 사용되는 ID 입니다. 동의어를 만들 수 있는 데이터베이스 개체 유형은 테이블, 뷰, 시퀀스, 다른 동의어입니다.

동의어에는 공용 동의어와 개인 동의어의 2 종류가 있습니다. *공용 동의어는* 데이터베이스에서 널리 사용할 수 있고 데이터베이스 클러스터의 모든 사용자가 볼 수 있습니다. 공용 동의어는 어떤 스키마에도 속하지 않습니다. 한편 개인 동의어는 특정 스키마에 속합니다. 현재 Postgres Plus Advanced Server 에서는 공용 동의어만을 지원합니다.

#### 2.2.4.1 공용 동의어 만들기

CREATE PUBLIC SYNONYM 명령 공용 동의어를 만들 수 있습니다. 공용 동의어는 기존의 동의어로 사용되지 않은 식별자를 지정해야 합니다. 다음은 명령의 예입니다.

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

이로써 DDL 과 DML 의 두 SQL 문장에서 personnel 동의어를 사용하여 enterprisedb 스키마의 emp 테이블을 볼 수 있습니다.

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



SELECT * FROM personnel;



empno | ename | job | mgr | hiredate | sal | comm | deptno

-------+----------+-----------+------+------------ --------+---------+-----
----+--------

7369 | SMITH | CLERK | 7902 | 17 - DEC - 80 00:00:00 | 800.00 | | 20

7499 | ALLEN | SALESMAN | 7698 | 20 - FEB - 81 00:00:00 | 1600.00 | 300.00
| 30

7521 | WARD | SALESMAN | 7698 | 22 - FEB - 81 00:00:00 | 1250.00 | 500.00 |
30

7566 | JONES | MANAGER | 7839 | 02 - APR - 81 00:00:00 | 2975.00 | | 20
```

```
7654 | MARTIN | SALESMAN | 7698 | 28 - SEP - 81 00:00:00 | 1250.00 |
1400.00 | 30

  7698 | BLAKE | MANAGER | 7839 | 01 - MAY - 81 00:00:00 | 2850.00 | | 30

  7782 | CLARK | MANAGER | 7839 | 09 - JUN - 81 00:00:00 | 2450.00 | | 10

  7788 | SCOTT | ANALYST | 7566 | 19 - APR - 87 00:00:00 | 3000.00 | | 20

  7839 | KING | PRESIDENT | | 17 - NOV - 81 00:00:00 | 5000.00 | | 10

  7844 | TURNER | SALESMAN | 7698 | 08 - SEP - 81 00:00:00 | 1500.00 | 0.00
| 30

  7876 | ADAMS | CLERK | 7788 | 23 - MAY - 87 00:00:00 | 1100.00 | | 20

  7900 | JAMES | CLERK | 7698 | 03 - DEC - 81 00:00:00 | 950.00 | | 30

  7902 | FORD | ANALYST | 7566 | 03 - DEC - 81 00:00:00 | 3000.00 | | 20

  7934 | MILLER | CLERK | 7782 | 23 - JAN - 82 00:00:00 | 1300.00 | | 10

  8142 | ANDERSON | CLERK | 7902 | 17 - DEC - 06 00:00:00 | 1300.00 | | 20

  (15 rows)
```

자세한 내용은 CREATE PUBLIC SYNONYM 명령을 참조하십시오.

#### 2.2.4.2 공용 동의어 삭제

DROP PUBLIC SYNONYM 명령 공용 동의어를 제거할 수 있습니다. 다음은 방금 만든 personnel 동의어를 삭제합니다.

```
DROP PUBLIC SYNONYM personnel;
```

자세한 내용은 DROP PUBLIC SYNONYM 명령을 참조하십시오.

#### 2.2.4.3 공용 동의어 네임 스페이스

공용 동의어는 동일한 데이터베이스의 다른 동의어와 같은 이름을 지정할 수 없습니다. 즉, 공용 동의어는 기존 스키마 테이블, 뷰 및 기타 데이터베이스 개체와 같은 이름을 사용할 수 있습니다.

따라서 다음 절에서 설명한대로, 검색 경로에 있는 다른 객체에 동일한 이름을 사용하는 경우, 예상치 못한 사건을 겪지 않도록 공용 동의어의 이름을 신중하게 결정해야 합니다.

#### 2.2.4.4 공용 동의어의 이름 확인 및 검색 경로

이름은 SQL 명령에 사용되는 개체를 정확하게 파악하기 위한 것입니다. 개체가 스키마 이름으로 확정되면, 개체는 스키마 이름으로 한정하는 것이 바람직합니다. 그러나 개체가 자격이 되지 않는 경우 객체의 위치를 확인하기 위해 일련의 작업을 수행합니다.

자격이 되지 않는 개체는 SQL 명령에 지정된 경우, 먼저 현재의 검색 경로에 사용자가 액세스할 수 있는 스키마를 확인합니다. 그래도 그 개체가 없을 경우에만 이름이 공용 동의어인지 데이터베이스에서 확인합니다. 그 것이 동의어인 경우 공용 동의어로 확인됩니다.

즉, SQL 명령에 사용을 위해 정의된 공용 동의어는 있지만 현재 검색 경로를 통해 사용자가 액세스할 수 있는 스키마에 같은 이름의 개체가 존재하는 경우 지정된 이름은 공개 동의어는 검색 경로의 개체로 이름 해결할 수 있습니다.

#### 2.2.4.5 공용 동의어 및 권한

모든 사용자가 공용 동의어를 만들 수 있습니다. 공용 동의어를 만드는 데 특별한 권한이 필요하지 않습니다. 또한, 모든 사용자가 SQL 명령에서 공용 동의어를 볼 수 있습니다. 그러나 SQL 명령을 실행하면, 동의어로 견장 붙은 데이터베이스 개체에 대해 현재 사용자의 권한을 확인합니다. 사용자가 개체에 적절한 권한을 가지고 있지 않은 경우 SQL 명령이 실패합니다.

### 2.2.5 계층 쿼리

*계층 쿼리는* 친자 관계를 형성하는 데이터를 기반으로 계층적으로 결과 행을 반환 쿼리입니다. 계층 구조는 전형적인 트리 구조로 표현됩니다. 트리는 상호 연결된 *노드로* 구성됩니다. 각 노드는 끝 노드 이거나 *자식* 노드와 연결합니다. 또한 각 노드는 부모가 없는 첫 번째 노드 *(루트 노드라고* 부른다)을 제외하고는 1 개의 *부모* 노드와 연결합니다. 각 트리는 반드시 1 개의 루트 노드를 갖습니다. 자식 노드가 없는 노드 (끝 노드)를 *리프 노드라고 합니다.* 트리는 항상 적어도 1 개의 리프 노드를 갖습니다. 예를 들면, 트리가 1 개의 노드로만 구성되는 경우 노드는 루트 노드이며, 리프 노드이다.

계층 쿼리는 결과 행은 여러 트리 노드를 표현합니다.

**주의 :** 1 개의 줄이 여러 트리에 존재하고 결과 사이에 여러 번 나오는 경우가 있습니다.

쿼리 계층 관계는 결과에 친자 관계를 형성하는 CONNECT BY 어구로 표현됩니다. SELECT 명령에 사용되는 CONNECT BY 어구 및 관련 어구 문맥은 다음과 같습니다.

*SELECT* select_list FROM table_expression [WHERE ...]

*[START* WITH start_expression]

CONNECT BY *(PRIOR* p arent_expr = *child_expr* |

*child_expr = PRIOR* parent_expr)

*[ORDER* SIBLINGS BY column1 [ASC | DESC]

[, *column2* [ASC | DESC] ...

[GROUP BY ...]

[HAVING ...]

*[other* ...]

*select_list* 결과 행을 여러 표현식입니다. *table_expression* 결과 행을 검색하는 다른 테이블이나 뷰를. *other 는* 다른 SELECT 명령에서 지원되는 어구입니다. 계층 쿼리에 관계 있는 어구, START WITH, CONNECT BY, ORDER SIBLINGS BY 는 다음의 항목에서 설명합니다.

**2.2.5.1 친자 관계의 정의**

결과 행 친자 관계, CONNECT BY 어구에 의해 결정됩니다. CONNECT BY 절은 Equal 연산자 (=)로 비교되는 2 개로 구성해야 합니다. 또한,이 2 개의 식 중 하나 전에 PRIOR 키워드를 지정해야 합니다.

모든 결과 행에 대해 다음과 같이 자식 노드를 결정합니다.

1. 결과 행에 대해 *parent_expr 을* 평가합니다.

*2. table_expression 에서* 파생된 다른 결과 행에 대해 *child_expr 을* 평가합니다.

*3. parent_expr = child_expr 의* 경우 *child_expr 에* 평가된 줄은 *parent_expr 에* 평가된 행 (부모 노드)의 자식 노드는 것입니다.

*4. table_expression 에서* 파생된 나머지 모든 결과 행에 대해 평가를 반복합니다. 3 단계의 요구에 응하는 모든 행이 지정된 부모 노드의 자식 노드입니다.

**주의 :** 행이 자식 노드를 결정하기 위한 평가는 WHERE 절이 *table_expression 에* 적용되기 전에 모든 결과 행에 대해 실행됩니다.

이제 위의 단계에 자식 노드로 결정한 각 노드를 부모 노드로, 이 평가를 차례로 반복하여 트리 구조를 완성하겠습니다. 더 이상 자식 노드가있는 노드가 없는 경우 등급이 종료됩니다. 자식 노드가 없는 노드 (끝 노드)가 리프노드입니다.

CONNECT BY 절을 포함하는 SELECT 명령은 일반적으로 START WITH 어구를 포함합니다. START WITH 절은 루트 노드가 될 행을 결정합니다. 즉, 위의 단계에서 처음으로 부모 노드가 될 행을 결정합니다. 이에 대해서는 다음 절에서 더 설명합니다.

#### 2.2.5.2 루트 노드 선택

START WITH 절은 *table_expression 에서* 얻어진 결과 줄에 루트 노드로 사용하는 행위를 결정합니다. *table_expression 에서* 파생되는 모든 결과 행을 즉 *start_expression 의* 평가가 "진정한"인 줄은 트리의 루트 노드가 될 수있습니다. 트리를 형성할 수있는 횟수는 루트 노드의 수와 같습니다. 결과적으로, START WITH 어구가 생략되면 *table_expression 에서* 파생되는 모든 결과 행을 자체적으로 트리의 루트노드입니다.

#### 2.2.5.3 예제 응용 프로그램에서 트리 만들기

샘플 응용 프로그램 emp 테이블을 생각해 볼 수 있습니다. emp 테이블 줄은 mgr 열 (관리자의 직원 ID 를 포함) 기준으로 계층 구조를 형성합니다. 각 직원은 많아 1 명의 관리자가 있습니다. KING 은이 회사 사장이기 때문에 관리자가 없습니다. 따라서 KING 의 mgr 열은 null 입니다. 또한 한 직원이 여러 직원의 관리자가 될 수 있습니다. 이 관계는 아래 그림과 같이 전형적인 트리 구조, 계층 조직을 형성합니다.

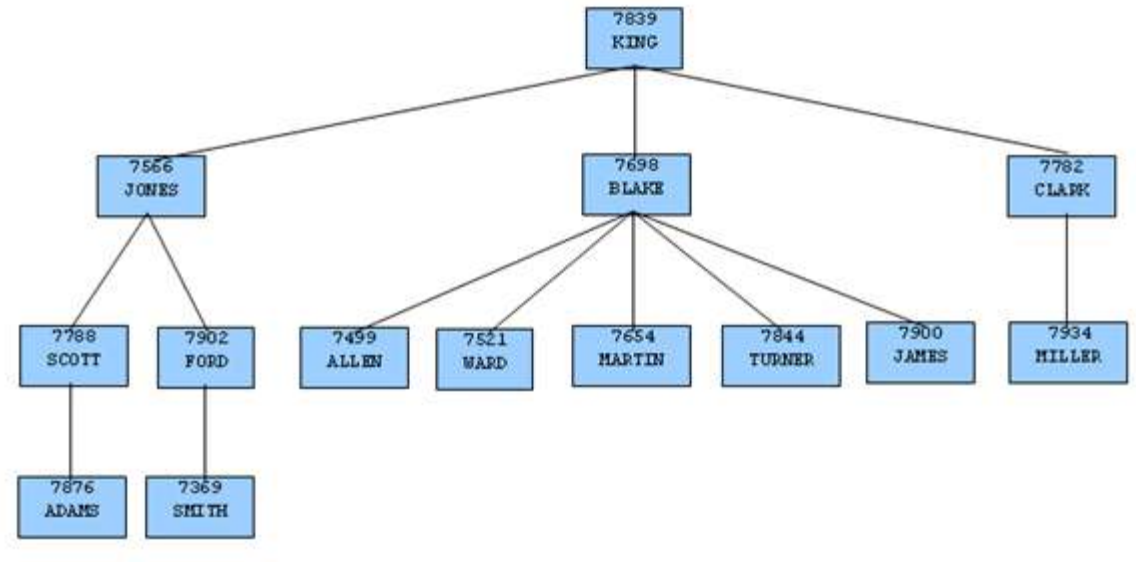


**그림 2 직원 계층 조직**

이 관계를 기반으로 하는 계층 쿼리를 실행하려면 SELECT 명령에 CONNECT BY PRIOR empno = mgr 절을 추가합니다. 예를 들면, 직원 ID 7839 사장 KING 을 생각하면 mgr 열이 7839 직원은 KING 의 직속 부하입니다. 해당 직원은 JONES, BLAKE, CLARK 입니다. (그들은 KING 자식

노드입니다.) 마찬가지로 직원 JONES 을 생각하면 mgr 열이 7566 과 다른 직원은 JONES 자식 노드입니다. 이 예에서는 SCOTT 과 FORD.

조직도 상위 KING 이고,이 트리에서 루트노드는 1 입니다. STAR T WITH mgr IS NULL 어구를 지정하기 위한 첫 번째 루트 노드로 KING 선택할 수 있습니다.

완전한 SELECT 명령은 다음과 같습니다.

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

쿼리 출력은 루트에서 리프까지 왼쪽에서 오른쪽으로 각 분기를 표시합니다. 다음은 이 쿼리의 결과입니다.

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

```
MILLER | 7934 | 7782

(14 rows)
```

#### 2.2.5.4 노드 수준

LEVEL 은 SELECT 명령에 사용할 수 있는 가짜 열입니다. 결과의 각 행에 대해 LEVEL 는 0 이 아니다. 노드 계층의 깊이를 나타내는 숫자를 반환합니다. 루트 노드 LEVEL 은 1 입니다. 루트 노드 아래의 하위 LEVEL 은 2 입니다. 다음처럼 나옵니다.

다음 쿼리는, LEVEL 가짜 열을 추가하여 위의 쿼리를 수정합니다. 또한 LEVEL 값을 통해 직원 이름이 들여쓰기 되어 각 줄의 계층의 깊이가 강조되고 있습니다.

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

 FROM emp START WITH mgr IS NULL

 CONNECT BY PRIOR empno = mgr;
```

다음은 쿼리의 결과입니다.

```
level | employee | empno | mgr

-------+-------------+-------+------

1 | KING | 7839 |

2 | JONES | 7566 | 7839

3 | SCOTT | 7788 | 7566

4 | ADAMS | 7876 | 7788

3 | FORD | 7902 | 7566

4 | SMITH | 7369 | 7902

2 | BLAKE | 7698 | 7839

3 | ALLEN | 7499 | 7698

3 | WARD | 7521 | 7698

3 | MARTIN | 7654 | 7698

3 | TURNER | 7844 | 7698
```

```
3 | JAMES | 7900 | 7698

2 | CLARK | 7782 | 7839

3 | MILLER | 7934 | 7782

(14 rows)
```

#### 2.2.5.5 형제 노드 정렬

일반적으로 부모가 있고 동일한 노드 수준에 있는 노드들을 *형제* 노드라고 합니다. 예를 들면, 위의 출력은 직원 ALLEN, WARD, MARTIN, TURNER, JAMES 이 형제 노드입니다. 이들은 레벨 3 에있어 부모 노드가 BLAKE 입니다. JONES, BLAKE, CLARK 도 형제 노드입니다. 이들은 2 단계에 있고, 부모 노드가 KING.

ORDER SIBLINGS BY 절을 사용하여 선택된 열의 값 형제 노드를 오름차순 또는 내림차순으로 정렬할 수 있습니다. 이것은 계층 쿼리에서만 사용할 수 있는 ORDER BY 절이 특별한 경우입니다.

ORDER SIBLINGS BY ename ASC 어구를 추가하여 위의 쿼리을 더 수정합니다.

```
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;
```

출력은 형제 노드 이름으로 오름차순으로 정렬되도록 수정합니다. 부모 노드 KING 의 밑의 형제 노드 BLAKE, CLARK, JONES 는 알파벳 순서로 나열되어있습니다. 부모 노드 BLAKE 의 밑의 형제 노드 ALLEN, JAMES, MARTIN, TURNER, WARD 도 알파벳 순서로 나열되어있습니다. 기타도 마찬가지입니다.

```
 level | employee | empno | mgr

 -------+-------------+-------+------

 1 | KING | 7839 |

 2 | BLAKE | 7698 | 7839
```

```
 3 | ALLEN | 7499 | 7698

 3 | JAMES | 7900 | 7698

 3 | MARTIN | 7654 | 7698

 3 | TURNER | 7844 | 7698

 3 | WARD | 7521 | 7698

 2 | CLARK | 7782 | 7839

 3 | MILLER | 7934 | 7782

 2 | JONES | 7566 | 7839

 3 | FORD | 7902 | 7566

 4 | SMITH | 7369 | 7902

 3 | SCOTT | 7788 | 7566

 4 | ADAMS | 7876 | 7788

 (14 rows)
```

마지막 예제에서는 WHERE 절을 추가하여 더 많은 3 개의 루트 노드를 지정합니다. 노드트리가 구성되면 WHERE 절에 의해 트리 노드가 강화되고, 결과가 나타납니다.

```
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;
```

이 쿼리의 결과는 3 개의 루트 노드 (Level 1)이 표시됩니다. BLAKE, CLARK, JONES 입니다. 또한 WHERE 절에 조건을 충족하지 않는 부분은 기록에서 삭제됩니다.

```
 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)
```

# 제 3 장
# SQL 언어

이 장에서는 Oracle 과 호환되는 Postgres Plus Advanced Server SQL 언어를 설명합니다. 이 장에서 설명하는 SQL 구문, 명령, 데이터 타입, 함수 등은 Postgres Plus Advanced Server 와 Oracle 모두에서 사용할 수 있습니다.

Oracle 과 호환되지 않는 " Postgres Plus Advanced Server SQL "의 다른 사양은 Postgres Plus documentation 에 설명되어 있습니다. 따라서 이 설명서에 있지 않은 특정 기능은 Postgres Plus documentation 에 있는 구문과 명령을 사용하여 설명될 수 있습니다.

이 장은 다음 항목으로 구성됩니다.

- Postgres Plus Advanced Server SQL 구문 및 언어 요소에 대한 일반적인 설명
- 데이터형
- SQL 명령 개요
- 내장 함수

## 3.1 SQL 구문

이 장에서는 SQL 의 일반적인 문법을 설명합니다. 이 단원의 내용은 데이터 정의 및 변경을 위해 SQL 명령을 적용하는 방법에 대해 자세히 설명하는 이후의 장을 이해하는데 기초가 됩니다.

### 3.1.1 어휘 구성

SQL 입력은 명령으로 이루어집니다. 명령은 토큰으로 구성되며 마지막은 세미콜론 ( ";")으로 끝납니다. 입력 스트림 종료 역시 명령을 끝내고 어떤 토큰을 사용할 수 있는지 특정 명령 구문에 따라 다릅니다.

토큰은 키워드, 식별자, 인용 부호로 묶인 식별자, 리터럴 (또는 상수), 또는 특수문자 기호입니다. 토큰은 공백 (스페이스, 탭, 줄바꿈)으로 구분되지만, 모호함이 없는 경우 (일반적으로 특수 문자가 다른 토큰 형식과 인접하여있는 경우) 필요 없습니다.

또한 입력된 SQL 에 주석이 있어도 상관없습니다. 코멘트는 토큰이 아니지만, 그 효과는 공백과 동일합니다.

예를 들면, 다음은 (문법적으로) 올바른 SQL 입력입니다.

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

이 경우는 1 줄에 1 개의 명령을 설명하는 3 개의 명령을 지속하고 있습니다 (반드시 1 개의 명령을 1 줄로 쓸 필요는 없습니다. 1 줄에 여러 명령을 입력하는 것도 가능하며, 1 개의 명령을 여러 줄에 작성할 수 있습니다).

SQL 구문은 어떤 토큰이 명령을 확인하고, 어느 것이 피연산자 또는 매개변수인지 식별하는 것에 관해서는 그다지 일관성 있지 않습니다. 처음 몇 토큰은 일반적으로 명령이름입니다. 따라서 위의 예제에서 "SELECT", "UPDATE", "INSERT" 명령에 대한 일반적인 설명입니다. 그러나, 예를 들어, UPDATE 명령은 SET 토큰이 특정 위치에 항상 기술되어야 하고, INSERT 명령의 특별한 경우, 명령을 완결하려면 VALUES 토큰이 필요합니다. 각 명령의 정확한 구문규칙은 3.3 장에서 설명되어 있습니다.

### 3.1.2 식별자와 키워드

위의 예제에 나오는 SELECT, UPDATE 또는 VALUES 같은 토큰은 키워드의 예제입니다. 키워드는 SQL 언어에서 고정된 의미가 있는 단어입니다. MY_TABLE 토큰 및 A 토큰은 식별자의 예입니다. 이들이 사용되는 명령은 테이블, 열, 다른 데이터베이스 객체의 이름을 식별합니다. 따라서 단순히 "이름"이라고도 합니다. 키워드와 식별자는 동일한 어휘 구조를 갖고 있으므로 언어를 알지 못하면 토큰이 식별자 또는 키워드인지를 잘 모를 수 있습니다.

SQL 식별자와 키워드는 문자 (a-z 또는 A-Z)로 시작해야 합니다. 식별자 또는 키워드 내에서 문자 다음으로 문자, 밑줄, 숫자 (0-9), 달러 기호 ($), 숫자 기호 (#)를 사용할 수 있습니다.

식별자 및 키워드 이름은 대소문자를 구분하지 않습니다.  따라서

```
UPDATE MY_TABLE SET A = 5;
```

는 다음 문장과 동일합니다.

```
uPDaTE my_TabLE SeT a = 5;
```

규약에서는 키워드를 대문자로, 이름을 소문자로 씁니다. 예를 들면 다음과 같습니다.

```
UPDATE my_table SET a = 5;
```

식별자는 부수적인 종류가 있습니다. 구분 식별자 또는 따옴표(인용부호)가 내장된 식별자입니다. 이 식별자의 문자는 큰따옴표 (")로 묶어 표시합니다. 구분된 식별자는 항상 식별자이고, 키워드가 아닙니다. 그래서 "select"는 "select"라는 이름의 열 또는 테이블을 조회하는데 사용할 수 있습니다. 반면에, 인용부호를 붙이지 않은 select 는 키워드로 이해되기 때문에 테이블 또는 열 이름이 예상되는 부분에서는 문법상 에러가 발생합니다. 인용부호를 붙인 식별자는 다음 예제와 같이 쓸 수 있습니다..

```
UPDATE "my_table"SET "a"= 5;
```

인용부호를 붙인 식별자는 문자 코드 0 을 제외하고, 모든 문자를 사용할 수 있습니다. (큰따옴표를 포함시킬 경우에는 인용 부호를 2 개 입력). 이것은 평소 사용할 수 없는 공백이나 앰퍼샌드(&)가 포함된 테이블 및 열 이름을 만들 수 있습니다. 이 경우에도 길이 제약이 적용됩니다.

인용부호가 붙지 않는 이름은 항상 소문자로 해석되지만 식별자를 인용해서 대소문자를 구분합니다. 예를 들면, 식별자 FOO, foo "foo"는 Postgres Plus Advanced Server 에서는 동일한 것으로 해석되지만 "Foo"와 "FOO"는 이들 3 개와 또한 서로 다른 것으로 해석됩니다. (Postgres Plus Advanced Server 가 인용부호가 붙지 않은 이름을 소문자로 해석하는 것은 Oracle 호환성 부분이 아닙니다. Oracle 에서 인용 부호가 붙지 않는 이름은 대문자로 해석하므로 Oracle 에서 foo 는 "FOO"와 동일하며, "foo"는 다릅니다. 만약 이식 가능한 애플리케이션을 쓰고 싶다면, 특정 이름에 항상 따옴표를 사용하거나 아예 인용부호를 붙이지 않는 것이 좋습니다).

### 3.1.3 상수

Postgres Plus Advanced Server 에는 문자열과 숫자라는 암묵적인 형의 상수가 있습니다. 상수는 명시적인 타입으로 지정할 수도 있고, 이 경우 시스템을 통해보다 정확한 표현과 효율적인 작업이 가능합니다. 이들의 다른 방법은 나중에 설명합니다.

#### 3.1.3.1 문자열

SQL 에서 문자열은 단일 인용부호( ')로 둘러싸 모든 일련의 문자입니다. 예를 들면, 'This is a string'입니다. 문자열에있는 단일 인용 부호 작성 방법은 연속적으로 단일 인용부호를 2 번 작성하는 것입니다. 예를 들면, 'Dianne' 'shorse'입니다. 큰따옴표( ")와 동일하지 않은 사항에 유의하십시오.

#### 3.1.3.2 숫자 상수

숫자 상수는 다음의 일반적인 형태로 받아들입니다.

```
digits
digits [digits] [e [+ -] digits]
[digits]. digits [e [+ -] digits]
digits e [+ -] digits
```

여기서 digits 는 1 개 또는 그 이상의 10 진수 숫자 (0 ~ 9)입니다. 소수점을 사용하려면 적어도 1 개의 숫자를 소수점 전이나 후에 두어야 합니다. 지수 상징 e 표를 붙이는 형식을 사용하는 경우에는 e 후에 적어도 1 개의 숫자를 두어야 합니다. 공백이나 다른 문자는 상수 속에 포함시킬 수 없습니다. + 혹은 - 기호를 앞에 붙이는 것은, 상수로 간주되지 않음을 주의하세요. 이 기호는 상수에 적용되는 연산자로 간주됩니다.

다음은 유효한 숫자 상수의 몇 가지 예입니다.

```
42
3.5
4.
.001
5e2
1.925e - 3
```

소수점도 지수도 포함되지 않은 숫자 상수의 경우 우선 값이 INTEGER 형 (32 비트)을 충족하면 INTEGER 형식으로 간주됩니다. 그렇지 않으면 BIGINT 형 (64 비트)에 적합하면 BIGINT 형식으로 간주됩니다. 둘 다 아니면, NUMBER 형식으로 간주됩니다. 상수를 소수점 또는 지수 또는 둘 모두를 포함하는 경우에는 항상 첫째로 NUMBER 형식으로 간주됩니다.

숫자 상수에 처음으로 부여되는 데이터형은 형식 확인 알고리즘의 시작점에 불과합니다. 대부분의 경우, 상수는 상황에 따라 자동으로 올바른 형식이 할당됩니다. 필요하면 이후의 장에서 설명된 것처럼, 캐스팅하여 숫자가 데이터형으로 해석되도록 시행할 수 있습니다.

### 3.1.3.3 다른 형식의 상수

임의의 형식의 상수는 다음 표기법을 사용하여 입력될 수 있습니다.

```
CAST ( 'string' AS type)
```

문자열 텍스트는 type 으로 불리는 형식의 입력 변환 루틴으로 전달됩니다. 결과는 지정된 형식의 상수입니다. 명시적인 타입 캐스팅은 상수가 어떤 형식이 되어야 할지에 대해 모호함이 없는 경우 (예를 들면 상수가 직접 테이블 열에 할당 되었을 경우), 생략해도 무방합니다. 이 경우 자동으로 형식이 강제됩니다.

CAST 는 임의의 식의 런타임 형식 변환을 지정하는 데에도 사용할 수 있습니다.

### 3.1.4 코멘트

코멘트는 이중 대시기호(--)로 시작하는 모든 문자의 행, 행의 말미까지 포함합니다. 예를 들면 다음과 같습니다.

```
-- This is a standard SQL comment
```

이외에도 C 언어 양식의 블록 코멘트를 사용할 수 있습니다.

```
/ * multiline comment
* block
* /
```

코멘트는 / *로 시작하며 * /로 끝납니다.

코멘트는 이후 구문분석 전에 입력 스트림에서 삭제되고, 공백으로 적절히 대체합니다.

## 3.2 자료형

다음 표는 범용의 내장된 데이터형을 나타냅니다.

**표 3-1 데이터형**

| 명칭 | 이명 | 설명 |
|---|---|---|
| BLOB | LONG RAW, RAW (n) | 이진 데이터 |
| BOOLEAN | | 논리값 (True / False) |
| CHAR [(n) | CHARACTER [(n) | n 문자의 고정 길이 문자열 |
| CLOB | LONG, LONG VARCHAR | 긴 문자열 |
| DATE | TIMESTAMP (0) | 날짜와 시간 |
| DOUBLE PRECISION | FLOAT,<br><br>FLOAT (25) - FLOAT (53) | 배정밀도 부동 소수점 |

| | | |
|---|---|---|
| INTEGER | INT, BINARY_INTEGER | 4 바이트 부호 있는 정수 |
| NUMBER | DEC, DECIMAL, NUMERIC | 선택할 수 있는 소수 자리의 정확한 수치 |
| NUMBER (p [, s) | DEC (p [, s),<br><br>DECIMAL (p [, s),<br><br>NUMERIC (p [, s) | 최대 정밀 숫자 P, 선택 가능한 스케일 s |
| REAL | FLOAT (1) - FLOAT (24) | 단정밀도 부동 소수점 |
| TIMESTAMP [(p) | | 날짜와 시간에 포함된 선택 가능한 초 소수점 이하의 정밀도 p |
| VARCHAR2 (n) | CH AR VARYING (n),CHARACTER VARYING (n),VARCHAR (n) | 가변 길이 문자열 (최대 길이 n 자) |

다음 장에서는 각 데이터형을 자세히 설명합니다.

### 3.2.1 숫자 데이터형

숫자 데이터형은 4 바이트 정수, 4 바이트 및 8 바이트 부동 소수점 및 고정된 정밀 소수점이 있습니다. 다음 의표에서 사용 가능한 형식을 열거합니다.

**3-2 숫자 데이터형**

| 모델명 | 저장 크기 | 설명 | 범위 |
|---|---|---|---|
| INTEGER | 4 바이트 | 일반적으로 사용하는 정수 | - 2,147,483,648 에서 +2,147,483,647 |
| NUMBER | 가변 길이 | 사용자 지정 정밀도, 정확 | 최대 1000 자리 |
| NUMBER (p [, s) | 가변 길이 | 최대 정밀 숫자 p, 선택 가능한 스케일 s | 최대 1000 자리 |
| REAL | 4 바이트 | 가변 정밀도, 부정확 | 6 자리 정밀도 |
| DOUBLE PRECISION | 8 바이트 | 가변 정밀도, 부정확 | 15 자리 정밀도 |

다음 절에서 데이터형에 대해 자세히 설명합니다.

### 3.2.1.1 정수 데이터형

INTEGER 형은 다양한 범위의 정수 즉, 소수점이 없는 수를 유지합니다. 허용 범위를 벗어난 값을 저장하려고 하면 에러가 발생합니다.

### 3.2.1.2 임의의 정확도를 가진 수

NUMBER 형은 최대 1000 자리의 정밀도로 값을 저장할 수 있고, 정확한 계산을 할 수 있습니다. 통화 금액 및 기타 정확도가 요구되는 수량을 저장할 때 특히 이 형을 추천합니다. 그러나, NUMBER 형은 다음 절에서 설명하는 부동 소수점 데이터 유형에 비해 아주 낮습니다.

다음 설명은 이와 같은 용어를 사용합니다. NUMBER 의 scale 은 소수점의 오른쪽 자릿수를 말합니다. NUMBER 의 정확도는 숫자 전체 자릿수입니다. 즉, 소수점의 양측 자리수의 합계입니다. 따라서 23.5141 라는 수치의 정확도는 6, scale 은 4 입니다. 정수는 0 의 scale 을 갖는다고 간주됩니다.

NUMBER 형의 scale 및 정밀도는 모두 설정할 수 있습니다. 열의 데이터형을 NUMBER 로 선언하려면 다음 구문을 사용합니다.

NUMBER (precision, scale)

정확도는 양수, 스케일은 0 또는 양수여야 합니다. 그밖에

NUMBER (precision)

는 스케일이 0 인 것을 선택합니다.

NUMBER

위와 같이 정확도 또는 scale 을 지정하지 않은 경우, 구현되는 한계 정확도까지 어떠한 정확도 또는 스케일의 값을 저장할 수 있는 열이 생성됩니다. 이러한 열은 어떤 특정한 스케일에 대해서 입력을 강요하는 것은 아니지만, 선언한 스케일을 가진 NUMBER 열이 입력 값의 스케일을 강요합니다. (표준 SQL 은 기본적으로 스케일 0 을 요구하고 있어 정수에 대한 정확도를 강제하고 있습니다. 최대 이식성을 위해, 항상 정확도와 스케일을 명시적으로 설정해야 합니다).

정확도나 값의 스케일이 선언된 정확도 또는 열의 스케일보다 큰 경우, 시스템은 값을 반올림합니다. 선언된 한계치를 충족시키기 위해 값이 반올림할 수 없는 경우, 에러가 발생합니다.

### 3.2.1.3 부동 소수점 데이터형

REAL 과 DOUBLE PRECISION 형은 부정확한 가변 정밀 숫자 데이터형입니다. 실제로, 이 데이터형은 사용하는 프로세서, 운영 체제 및 컴파일러로 지원되어, 일반적으로 (각각 단정밀도 및 배정밀도) 이진 부동 소수점 연산에서 IEEE 표준 754 를 구현한 것 입니다.

부정확은 내부형으로 정확하게 변환되지 않은 값, 그리고 근사치로 저장되는 값을 의미 합니다. 그래서 저장하는 값과 출력되는 값에 약간의 차이가 있습니다. 이러한 에러를 처리하고 계산을 통해 어떻게 전달 할 것인가, 수학 및 컴퓨터 과학 전반에 관한 것으로, 다음 사항을 제외하고 더 이상은 설명되지 않습니다.

(금액과 같은) 정확한 저장과 계산이 필요할 때는, 대신 NUMBER 데이터형을 사용하세요.

이 데이터형에서 무언가 중요한 건에 대해 복잡한 계산을 필요로 할 때, 특히 (무한대 또는 언더플로와 같은) 경계선에서 어떤 종류의 행동에 대해 신뢰를 설정해야 하는 경우 구현을 신중하게 검증해야 합니다.

2 개의 부동 소수점 값의 동일성 비교는 예상대로 또는 예상대로 되지 않을 때도 있습니다.

대부분의 플랫폼에서 REAL 은 최소 6 자리의 정밀도를 가지고 적어도 1E - 37 1E +37 범위의 값입니다. DOUBLE PRECISION 은 최소 15 자리의 정밀도를 가진 약 1E - 307 에서 1E +308 범위의 값입니다. 너무 크고 너무 작은 값은 에러의 원인이 됩니다. 입력 값의 정밀도가 높은 팔 경우 반올림을 할 수 있습니다. 제로에 한없이 가까운 값으로, 게다가 영과는 다른 것으로 간주되지 않는 수치는 언더플로 에러가 발생할 수 있습니다.

또한, Postgres Plus Advanced Server 는 부정확한 숫자를 규정하는 표준 SQL 의 FLOAT 와 FLOAT (p)를 지원합니다. 여기서 p 는 2 진수 자릿수에서 최소 허용하는 정밀도를 지정합니다. Postgres Plus Advanced Server 는 FLOAT (1)에서 FLOAT (24)까지 REAL 형을 지정한 것으로 취급하고, FLOAT (25)에서 FLOAT (53)까지 DOUBLE PRECISION 를 지정한 것으로 취급합니다. 허용 오차 p 이외의 값은 에러가 발생합니다. 정확도를 지정하지 않는 FLOAT 는 DOUBLE PRECISION 로 해석됩니다.

### 3.2.2 문자 형식

다음은 Postgres Plus Advanced Server 에서 사용할 수 있는 범용 문자 형식을 보여줍니다.

**표 3-3 문자 형식**

| 모델명 | 설명 |
|---|---|
| CHAR [(n) | 고정길이, 공백 매움 |
| CLOB | 1GB 까지 가변 길이 |

| | |
|---|---|
| VARCHAR2 (n) | 상한을 갖는 가변 길이 |

2 개 주요 문자 데이터형은 CHAR (n)와 VARCHAR2 (n)입니다. 여기서 n 은 양의 정수입니다. 이 데이터형은 2 개 모두 n 문자 길이의 문자열을 저장할 수 있습니다. CHAR 형으로 n 이 생략되면 기본값은 1 입니다. 보다 긴 문자열을 이러한 형의 열로 저장하려 할 때, 초과하는 문자가 모두 공백인 경우를 제외하고, 최대 길이로 잘립니다. 만일 선언한 길이보다 문자열이 짧은 경우, CHAR 값은 공백으로 채웁니다. VARCHAR2 값은 보다 짧은 문자열로 저장됩니다.

명시적으로 값을 VARCHAR2 (n) 또는 CHAR (n)으로 변환하는 경우, 지정된 길이를 초과하면 에러발생 없이 n 자로 잘립니다. (이것은 표준 SQL 의 사양입니다.)

CHAR 형의 값은 지정 길이 n 까지 공백으로 채우고, 그대로 저장 및 표시됩니다. 그러나 채워진 공백은 의미적으로 중요하지 않은 것으로 처리됩니다. 2 개의 CHAR 값을 비교할 때 마지막 공백은 무시됩니다. 또한 CHAR 값을 다른 문자열로 변환하기 위해서는 삭제됩니다. VARCHAR2 형의 값에서 마지막 공백은 의미적으로 중요한 요소이므로 주의해야 한다.

3 번째 문자 데이터형은 CLOB 입니다. 이것은 긴 문자열을 저장하는 데 사용됩니다. 길이제한을 지정하지 않는 것을 제외하고, CLOB 는 의미적으로 VARCHAR2 와 같습니다. 일반적으로, 최대 글자의 길이가 얼마가 될지 모를 경우, VARCHAR2 아니라 CLOB 을 사용합니다.

CLOB 형으로 저장할 수 있는 가장 긴 문자 크기는 약 1GB 입니다.

이러한 3 가지 종류의 데이터 용량 사양은 문자열이 127 바이트 보다 작을 때 원래 문자열에 1 을 더합니다. 문자열이 127 이상인 경우, 4 바이트를 원래의 문자열에 더한다. CHAR 일 경우에는, 채워 넣는데 저장할 곳이 필요하다. 긴 문자열은 시스템에서 자동적으로 축소됩니다. 따라서 디스크의 하드웨어 사양은 줄어들 수 도 있습니다. 긴 값은 또한 더 짧은 값에 대한 빠른 액세스를 방해하지 않도록 배경 테이블에 저장됩니다.

데이터베이스의 문자 집합은 텍스트 값을 저장하기 위해 사용하는 문자 집합을 결정합니다.

### 3.2.3 이진 데이터형

BLOB 데이터형은 이진 문자열을 저장할 수 있습니다.

**표 3-4 이진 문자열 데이터형**

| 모델명 | 저장 크기 | 설명 |
|---|---|---|

| | | |
|---|---|---|
| BLOB | 이진 문자열이 127 바이트보다 작은 경우, 실제 이진 문자열 길이에 1 바이트를 더한다. 127 바이트보다 큰 경우, 4 바이트를 더한다. | 가변 길이 이진 문자열 |

이진 문자열은 octet (바이트)의 연속입니다. 이진 문자열이 문자열과 다른 점은 다음의 2 가지입니다. 1 번째는 이진 문자열은 0 octet 과 다른 "출력할 수 없는" octet (보통 32 에서 126 까지의 범위를 벗어나는 바이트)을 저장할 수 있습니다. 2 번째는 이진 문자열을 계산하면 실제 바이트가 처리되는 반면 문자열 인코딩 및 프로세싱은 로케일 설정에 따른다는 것입니다.

### 3.2.4 날짜 / 시간 데이터형

다음의 날짜 / 시간 데이터형에 대한 설명은 테이블을 작성하거나 변경할 때 edb_redwood_date 설정 매개변수가 true 로 설정되어 있다고 가정합니다.

Postgres Plus Advanced Server 에서는 다음 표에 제시된 날짜 / 시간 데이터형을 지원 합니다.

**표 3-5 날짜 / 시간 데이터형**

| 모델명 | 저장 크기 | 설명 | 가장 멀리 과거 | 가장 멀리 미래 | 정확도 |
|---|---|---|---|---|---|
| DATE | 8 바이트 | 날짜와 시간 | 4713 BC | 5874897 AD | 1 초 |
| TIMESTAMP [(p) | 8 바이트 | 날짜와 시간 | 4713 BC | 5874897 AD | 1 μ 초 |

데이터 정의 언어 (DDL) 명령, CREATE TABLE 또는 ALTER TABLE 에서 열의 데이터형으로 DATE 를 사용하는 경우 시간 테이블 정의를 데이터베이스에 저장할 때 DATE 은 TIMESTAMP (0)으로 변환됩니다. 따라서 시간도 날짜와 함께 열에 저장됩니다.

SPL 내의 선언 부분에 변수의 데이터형으로 프로시저와 함수에 매개변수의 데이터형으로 또는 함수의 반환 데이터형으로, DATE 를 사용하는 경우 항상 TIMESTAMP (0)으로 변환됩니다. 따라서 시간을 처리할 수 있습니다.

TIMESTAMP 에서는 초 필드에 보유되는 값의 소수 자릿수를 지정하는 정밀도 값인 옵션 p 를 취할 수 있습니다. P 의 허용 범위는 0 부터 6 까지 입니다. 기본적으로 6 입니다.

TIMESTAMP 값을 배정밀도 부동 소수점 숫자로 저장하는 경우 (현재는 기본), 효과적인 정확도의 한계는 6 보다 더 작은지도 모릅니다. TIMESTAMP 값은 2000-01-01 자정을 기준으로 한 경과 시간 (초)으로 저장됩니다. 마이크로초의 정확도는 2000-01-01 에서 몇 년 이내의 날짜가 높지만, 정확도는 날짜가 (2000-01-01 부터) 멀어질수록 하락합니다. TIMESTAMP 값이 8

바이트 정수 (컴파일 시 옵션 지정)로 저장되는 경우에는 마이크로초 정확도는 모든 범위의 값을 사용할 수 있습니다. 그러나 8 바이트 TIMESTAMP 에서는 위에서 제시된 4713 BC 에서 294276 AD 까지의 날짜 범위보다 제한이 있습니다.

### 3.2.4.1 날짜 / 시간 입력

날짜와 시간 입력은 ISO 8601 SQL 호환 형식, Oracle 기본 서식 dd - MON – yy, 다른 날짜를 정확하게 표현하는 형식을 수용한다. 그러나 잘못 해석되는 것을 피하기 위해 TO_DATE 함수를 사용하는 것이 좋습니다. 제 3.5.6 장을 참조하십시오.

텍스트 문자열처럼 날짜와 시간 리터럴은 작은 따옴표로 묶어야 합니다. 다음 표준 SQL 구문 사용할 수 있습니다.

```
type 'value'
```

type 은 DATE 또는 TIMESTAMP 중 하나입니다. value 는 날짜 / 시간 문자열입니다.

#### 3.2.4.1.1 날짜

다음의 표는 날짜 입력이 가능한 서식의 일부를 보여줍니다. 모두 1999 년 1 월 8 일입니다.

**표 3-6 날짜 입력**

| 예 |
|---|
| January 8, 1999 |
| 1999-01-08 |
| 1999 - Jan – 08 |
| Jan - 08 – 1999 |
| 08 - Jan – 1999 |
| 08 - Jan – 99 |
| Jan - 08 – 99 |
| 19990108 |
| 990108 |

날짜 값은 DATE 또는 TIMESTAMP 형식의 열과 변수에 할당할 수 있습니다. 날짜 값과 시간 값이 추가되지 않으면, 시, 분, 초 필드에 0 이 설정됩니다.

3.2.4.1.2 시간

다음은 DATE 또는 TIMESTAMP 형식의 시간 예제를 보여줍니다.

**표 3-7 시간 입력**

| 예 | 설명 |
|---|---|
| 04:05:06.789 | ISO 8601 |
| 04:05:06 | ISO 8601 |
| 04:05 | ISO 8601 |
| 040506 | ISO 8601 |
| 04:05 AM | 04:05 와 같다. AM 은 값에 영향을 주지 않습니다. |
| 04:05 PM | 16:05 와 같다. 시간 입력은 12 이하여야 합니다. |

3.2.4.1.3 타임 스탬프

TIMESTAMP 형식에 유효한 입력은 날짜와 시간의 세트로 구성됩니다. 날짜 부분은 표 3-6 날짜 입력에 예제와 같은 형식입니다. 시간 부분은 표 3-7 시간 입력에 예제와 같은 형식입니다.

다음은 Oracle 의 기본 형식을 따르는 타임 스탬프의 예입니다.

```
08 - JAN - 99 04:05:06
```

다음은 ISO 8601 표준 형식을 따르는 타임 스탬프의 예입니다.

```
1999-01-08 04:05:06
```

### 3.2.4.2 날짜 / 시간 출력

날짜와 시간의 기본 출력 형식은 Redwood date style 로 불리는 Oracle 호환 서식 (dd - MON - yy)이나 ISO 8601 형식 (yyyy - mm - dd)입니다. 이 형식은 데이터베이스의 애플리케이션 인터페이스 따라 달라질 수 있습니다. SQL Interactive 같이 JDBC 를 사용하는 애플리케이션에서는 항상 ISO 8601 형식의 날짜가 제시됩니다. PSQL 와 같은 다른 애플리케이션에서는 Oracle 호환 형식의 날짜로 제시됩니다.

다음은 Oracle 호환과 ISO 8601 의 2 개의 출력 형식의 예를 보여줍니다.

**표 3-8 날짜 / 시간 출력 형식**

| 설명 | 예 |
|---|---|
| Oracle 호환 | 31 - DEC - 05 07:37:16 |
| ISO 8601 / 표준 SQL | 1997-12-17 07:37:16 |

### 3.2.4.3 내부 자료

Postgres Plus Advanced Server 는 모든 날짜와 시간 계산에 율리우스 일을 사용하고 있습니다. 이것은 1 년이 365.2425 일이라고 추정하고 기원전 4713 년부터 미래까지 모든 날짜를 정확하게 예측하거나 계산하는 우수한 특성을 가지고 있습니다.

## 3.2.5 논리 값 데이터형

Postgres Plus Advanced Server 는 표준 SQL 의 BOOLEAN 형식이 제공됩니다. BOOLEAN 은 "true(참)" 혹은 "false(거짓)"라는 2 가지 값 중 하나를 취할 수 있습니다. 제 3 의 상황인, "unknown"은 SQL 의 null 값으로 표시됩니다.

**표 3-9 논리 값 데이터형**

| 모델명 | 저장 크기 | 설명 |
|---|---|---|
| BOOLEAN | 1 바이트 | 논리값 (True / False) |

"true(참)"인 상황에 대한 효과적인 리터럴 값은 TRUE 입니다. "false(거짓)"상황에 대한 효과적인 리터럴 값은 FALSE 입니다.

**참고** : BOOLEAN 형은 SPL 프로그램에서 변수 선언에서만 사용할 수 있습니다. 테이블 열의 데이터 유형을 정의하기 위해서는 사용할 수 없습니다.

## 3.3 SQL 명령

이 장에서는 Postgres Plus Advanced Server 에서 지원하는 Oracle 호환 SQL 명령을 설명합니다. 이 장의 SQL 명령은 Oracle 데이터베이스와 Postgres Plus Advanced Server 데이터베이스에서 작동합니다.

다음 사항에 유의하십시오.

- Postgres Plus Advanced Server 는 여기에서 열거되지 않은 명령을 지원합니다. 그 명령은 Oracle 과 호환되지 않습니다. 또는 Oracle SQL 명령과 유사하거나 동일한 기능을 제공하지만 구문이 다릅니다.
- 이 장에서는 SQL 명령이 유효한 모든 구문, 옵션 및 기능을 필수적으로 설명하지 않습니다. Oracle 과 호환되지 않는 구문, 옵션 및 기능은 명령 설명에서 생략합니다.
- Postgres Plus documentation set 에서는 Oracle 과 호환되지 않는 명령의 사양도 설명하고 있습니다.

### 3.3.1 ALTER INDEX

**이름**

ALTER INDEX - 인덱스 정의를 변경한다

**개요**

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

**설명**

ALTER INDEX 는 기존의 인덱스의 정의를 변경합니다. RENAME 구문은 인덱스의 이름을 변경합니다. 저장된 데이터에는 영향을 주지 않습니다.

**매개변수**

name

수정할 기존 인덱스 이름 (스키마로 수식된 이름도 가능).

new_name

인덱스의 새로운 이름입니다.

**예**

기존의 인덱스의 이름을 바꿉니다.

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

**관련 항목**

CREATE INDEX, DROP INDEX

### 3.3.2 ALTER ROLE

**이름**

ALTER ROLE - 데이터베이스 역할을 변경한다

**개요**

ALTER ROLE name IDENTIFIED BY password

**설명**

ALTER ROLE 는 역할의 패스워드를 변경합니다. 슈퍼유저 또는 CREATEROLE 권한이 있는 사용자만이 이 명령을 사용할 수 있습니다. 수정할 역할에 SUPERUSER 속성이 있는 경우에는, 슈퍼유저만 이 명령을 사용할 수 있습니다. 역할에 LOGIN 속성이 없는 경우, 패스워드는 의미가 없습니다.

**매개변수**

name

패스워드를 변경하는 역할의 이름입니다.

password

역할의 새 패스워드입니다.

**주석**

구성원 자격 변경은 GRANT 와 REVOKE 를 사용하십시오.

**예**

역할의 패스워드를 변경합니다.

ALTER ROLE admins IDENTIFIED BY xyRP35z;

**관련 항목**

CREATE ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE

### 3.3.3 ALTER SEQUENCE

**이름**

ALTER SEQUENCE – 시퀀스 생성기의 정의를 변경한다

**개요**

*ALTER* SEQUENCE name *[INCREMENT* BY increment]

*[MINVALUE* minvalue] *[MAXVALUE* m axvalue]

*[CACHE* cache | NOCACHE] [CYCLE]

**설명**

ALTER SEQUENCE 는 기존 시퀀스 생성기의 매개변수를 변경합니다. ALTER SEQUENCE 로 지정되지 않은 매개변수는 이전 설정이 유지됩니다.

**매개변수**

*name*

변경될 시퀀스의 이름 (스키마로 수식된 이름도 가능).

*increment*

*INCREMENT* BY increment 어구는 선택 사항입니다. 양수 값은 오름차순 시퀀스, 음수 값은 내림차순 시퀀스를 만듭니다. 지정하지 않은 경우, 이전 값이 유지됩니다.

*minvalue*

*MINVALUE* minvalue 는 시퀀스 생성기가 생성하는 최소값을 결정합니다. 지정하지 않으면, 현재의 최소값을 유지합니다. NO MINVALUE 가 지정된 경우, 오름차순 때는 1 내림차순 때는 $-2^{63}$ -1 이 기본입니다. NO MINVALUE 키워드는 Oracle 과의 호환성이 없습니다.

*maxvalue*

*MAXVALUE* maxvalue 는 시퀀스 생성기가 생성하는 최대값을 결정합니다. 지정하지 않으면, 현재의 최대값이 유지됩니다. NO MAXVALUE 를 지정하면 오름차순 때는 $2^{63}$ -1 내림차순 때는 -1 이 기본입니다. NO MAXVALUE 키워드는 Oracle 과의 호환성이 없습니다.

*cache*

CACHE *cache* 어구를 사용하면 접속을 가속화하기 위해 시퀀스 번호를 미리 할당하고 메모리에 저장할 수 있습니다. 최소값은 1 입니다. (한번에 발생하는 값이 1 단지이기 때문에 캐시가 없는 상태가 됩니다. 즉, NOCACHE) 지정이 없는 경우, 이전 캐시 값이 유지됩니다.

CYCLE

CYCLE 옵션을 사용하면 시퀀스가 한계 (오름차순의 경우 *maxvalue* 내림차순의 경우 *minvalue)에* 이르렀을 때 그 순서를 회전시킬 수 있습니다. 한계에 이르렀을 때, 다음 생성되는 번호는 오름차순 경우 *minvalue* 내림차순의 경우 *maxvalue* 입니다. 지정이 없는 경우, 이전 설정이 유지됩니다. NO CYCLE 키워드를 지정하면, 시퀀스의 한계에 도달한 후에 회전되지 않습니다. NO CYCLE 키워드는 Oracle 과의 호환성이 없습니다.

**주석**

동일한 시퀀스로 번호를 취득하는 트랜잭션의 동시 차단을 방지하기 위해, ALTER SEQUENCE 를 롤백 하지 않습니다. ALTER SEQUENCE 변경 사항은 즉시 적용되며 취소할 수 없습니다.

ALTER SEQUENCE 는 백엔드의 NEXTVAL 결과에 대해서는 신속하게 효력을 발휘하지 않습니다. 이 백엔드는 미리 할당된 (캐시 된) 시퀀스 값을 가지고 있으며, 이 값이 모두 다 떨어진 후 변경된 시퀀스를 만드는 매개변수를 검색할 수 있습니다. 현재의 백엔드는 즉각 변경됩니다.

**예**

serial 이라는 시퀀스의 증가량과 캐시 값을 수정합니다.

ALTER SEQUENCE serial INCREMENT BY 2 CACHE 5;

**관련 항목**

CREATE SEQUENCE, DROP SEQUENCE

### 3.3.4 ALTER SESSION

**이름**

ALTER SESSION - 런타임 매개변수를 변경한다

**개요**

ALTER SESSION SET name = value

**설명**

ALTER SESSION 명령은 런타임 설정 매개변수를 변경합니다. ALTER SESSION 은 현재 세션에서 사용되는 값에만 영향을 줍니다. 이 매개변수의 일부는 Oracle 호환을 위해서만 사용되고 Postgres Plus Advanced Server 런타임에 는 아무런 영향을 미치지 않습니다. 다른 매개변수는 Postgres Plus Advanced Server 데이터베이스 서버에 대응하는 런타임 설정 매개변수를 변경합니다.

**매개변수**

name

설정 가능한 런타임 매개변수의 이름입니다. 사용할 수 있는 매개변수는 다음과 같습니다.

value

매개변수의 새로운 값입니다.

**설정 매개변수**

ALTER SESSION 명령을 사용하여, 다음의 설정 매개변수를 변경할 수 있습니다.

NLS_DATE_FORMAT (string)

날짜 및 시각 형식을 지정합니다. 이것은 애매한 날짜 입력 값에 대한 규칙이 됩니다. Postgres Plus Advanced Server 에서의 datestyle 런타임 설정 매개변수도 마찬가지입니다.

NLS_LANGUAGE (string)

표시되는 메시지의 언어를 설정합니다. Postgres Plus Advanced Server 에서의 lc_messages 매개변수도 마찬가지입니다.

NLS_LENGTH_SEMANTICS (string)

BYTE 또는 CHAR 중 하나를 선택할 수 있습니다. 기본값은 BYTE 입니다. 이 매개변수는 Oracle 호환을 위해서만 제공되며, Postgres Plus Advanced Server 에서는 아무런 영향을 미치지 않습니다.

OPTIMIZER_MODE (string)

쿼리 시 기본 Optimizer 모드를 설정합니다. ALL_ROWS, CHOOSE, FIRST_ROWS, FIRST_ROWS_10, FI RST_ROWS_100, FIRST_ROWS_1000 을 지정할 수 있습니다. 기본값은 CHOOSE 입니다. 자세한 내용은 3.4 장을 참조하십시오.

QUERY_REWRITE_ENABLED (string)

TRUE, FALSE, FORCE 를 지정할 수 있습니다. 기본값은 FALSE 입니다. 이 매개변수는 Oracle 호환을 위해서만 제공되며, Postgres Plus Advanced Server 에서는 아무런 영향을 미치지 않습니다.

QUERY_REWRITE_ INTEGRITY (string)

ENFORCED, TRUSTED, STALE_TOLERATED 을 지정할 수 있습니다. 기본값은 ENFORCED 입니다. 이 매개변수는 Oracle 호환을 위해서만 제공되며, Postgres Plus Advanced Server 에서는 아무런 영향을 미치지 않습니다.

**예**

UTF - 8 인코딩에서 영어 (US) 언어를 설정합니다. 이 예제에서 값, en_US.UTF – 8 은 Postgres Plus Advanced Server 에 지정된 형식입니다. 이 형식은 Oracle 호환이 없으므로 주의하세요.

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

날짜 형식을 지정합니다.

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

### 3.3.5 ALTER TABLE

**이름**

ALTER TABLE - 테이블 정의를 변경한다

**개요**

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

여기서 action 은 다음 중 하나입니다.

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

**설명**

ALTER TABLE 은 기존 테이블 정의를 변경합니다. 이 명령은 다음과 같은 서브폼(subform)이 있습니다.

ADD column type

이 구문을 사용하면 CREATE TABLE 과 같은 구문을 사용하여 새 열을 테이블에 추가할 수 있습니다.

DROP COLUMN

이 구문을 사용하면 테이블에서 열을 삭제할 수 있습니다. 삭제할 열을 포함하는 인덱스 및 테이블 제약도 자동으로 삭제됩니다.

ADD table_constraint

이 구문을 사용하면 CREATE TABLE 과 같은 구문을 사용하여 새로운 제약을 테이블에 추가할 수 있습니다.

DROP CONSTRAINT

이 구문을 사용하면 테이블에 지정된 제약을 제거할 수 있습니다. 현재, 테이블 제약은 유일한 이름을 요구하지 않습니다. 따라서 지정된 이름과 일치하는 여러 제약이 있을 수 있습니다. 이 경우 일치하는 모든 제약이 삭제됩니다.

RENAME

RENAME 구문을 사용하면 테이블 (또는 인덱스, 시퀀스 또는 뷰) 이름이나 테이블의 개별 열 이름을 변경할 수 있습니다. 저장된 데이터에는 영향을 미치지 않습니다.

ALTER TABLE 명령을 사용하려면 변경할 테이블을 소유하고 있어야 합니다.

**매개변수**

name

변경될 기존 테이블의 이름 (스키마로 수식된 이름도 가능)입니다.

column

신규 또는 기존 열의 이름입니다.

new_column

기존 열의 새로운 이름입니다.

new_name

테이블의 새 이름입니다.

type

새로운 열의 데이터형입니다.

table_constraint

테이블의 새로운 테이블 제약입니다.

constraint_name

삭제할 기존 제약의 이름입니다.

CASCADE

삭제된 제약에 의존하는 객체를 자동으로 삭제합니다.

**주석**

ADD COLUMN 으로 열을 추가했을 때, 테이블의 기존 행에 추가된 행은 모두 기본값 (DEFAULT 가 지정되지 않은 경우 null)으로 초기화됩니다.

null 이 아닌 기본값이 있는 열을 추가하려면 테이블 전체의 갱신이 필요합니다. 테이블이 거대한 경우, 이 작업에 많은 시간을 소모할 수 있습니다. 또 일시적으로 2 배의 디스크 공간이 필요합니다.

CHECK 또는 NOT NULL 제약 조건을 추가할 때는 기존 행이 제약을 준수하는지 확인하기 위해 테이블 검사가 필요합니다.

DROP COLUMN 구문은 열을 물리적으로 제거하지 않고 SQL 운영을 비가시화 시킵니다. 이 명령을 실행하면 테이블에 삽입하거나 갱신하면 삭제된 열에는 null 값이 포함되어 있습니다. 따라서 열 삭제는 신속하게 수행할 수 있습니다. 그러나 삭제된 열이 차지했던 공간이 아직 회수되지 않아 테이블 디스크 크기가 즉시 줄어들지 않습니다. 이 공간은 이후 기존 행을 갱신하는 시기에 수거합니다.

시스템 카탈로그 테이블의 내용은 어떠한 부분 변경이 허용되지 않습니다.

유효한 매개변수에 대한 자세한 설명은 CREATE TABLE 을 참조하십시오.

**예**

VARCHAR2 형의 열을 테이블에 추가합니다.

```
ALTER TABLE emp ADD address VARCHAR2 (30);
```

테이블에서 열을 삭제합니다.

```
ALTER TABLE emp DROP COLUMN address;
```

기존 열 이름을 변경합니다.

```
ALTER TABLE emp RENAME COLUMN address TO city;
```

기존 테이블의 이름을 변경합니다.

```
ALTER TABLE emp RENAME TO employee;
```

테이블에 check 제약을 부여합니다.

```
ALTER TABLE emp ADD CONSTRAINT sal_chk CHECK (sal> 500);
```

테이블에서 check 제약 조건을 삭제합니다.

```
ALTER TABLE emp DROP CONSTRAINT sal_chk;
```

**관련 항목**

CREATE TABLE, DROP TABLE

### 3.3.6 ALTER TABLESPACE

**이름**

ALTER TABLESPACE - 테이블스페이스 정의를 변경합니다.

**개요**

```
ALTER TABLESPACE name RENAME TO newname
```

**설명**

ALTER TABLESPACE 테이블스페이스의 정의를 변경합니다.

**매개변수**

name

기존 테이블스페이스의 이름입니다.

newname

새 테이블스페이스의 이름입니다. pg_로 시작하는 이름은 시스템 테이블스페이스를 위해 예약되어 사용할 수 없습니다.

**예**

empspace 테이블스페이스를 employee_space 라는 이름으로 변경합니다.

```
ALTER TABLESPACE empspace RENAME TO employee_space;
```

**관련 항목**

DROP TABLESPACE

### 3.3.7 ALTER USER

**이름**

ALTER USER – 데이터베이스 사용자 계정을 변경한다

**개요**

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

**설명**

ALTER USER 는 Postgres Plus Advanced Server 사용자 계정의 패스워드를 변경합니다. 데이터베이스 슈퍼유저 또는 CREATEROLE 권한이 있는 사용자 가이드 명령을 사용할 수 있습니다. 일반 사용자도 자신의 패스워드를 변경하기 위해 이 명령을 사용할 수 있습니다.

**매개변수**

name

패스워드를 변경할 사용자의 이름입니다.

password

이 계정에 대한 새 패스워드입니다.

**예**

사용자의 패스워드를 변경합니다.

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

**관련 항목**

CREATE USER, DROP USER

### 3.3.8 COMMENT

**이름**

COMMENT - 객체의 코멘트를 정의하거나 변경한다

**개요**

```
COMMENT ON
(
TABLE table_name |
COLUMN table_name. c olumn_name
) IS 'text'
```

**설명**

COMMENT 는 데이터베이스 객체에 대한 코멘트를 저장합니다. 코멘트를 변경하려면 동일한 객체에 대해 새로운 COMMENT 명령을 실행하세요. 각 객체에 저장할 수 있는 코멘트 문자열은 1 개뿐입니다. 코멘트를 제거하려면, 텍스트 부분에 빈 문자열 (중간에 공백 없이 단일 따옴표를 2 번 연속사용)를 작성하십시오. 객체가 삭제되었을 때, 코멘트는 자동으로 삭제됩니다.

**매개변수**

table_name

코멘트를 추가하는 테이블의 이름입니다. 테이블 이름은 스키마 수식이 가능합니다.

table_name. column_name

코멘트를 추가하는 table_name 테이블의 열 이름입니다. 테이블 이름은 스키마 수식이 가능합니다.

text

새로운 코멘트입니다.

**주석**

현재 코멘트는 보안 메커니즘이 존재하지 않습니다. 데이터베이스에 연결하는 사용자는 누구든지 데이터베이스에 있는 객체의 코멘트를 볼 수 있습니다 (단, 소유하지 않는 객체에 대한 코멘트를 변경할 수 있는 것은 슈퍼유저뿐입니다.) 따라서 코멘트는 보안에 중요한 정보를 제공하지 않습니다.

**예**

emp 테이블에 코멘트 처리합니다.

```
COMMENT ON TABLE emp IS 'Current employee information';
```

emp 테이블 empno 열에 코멘트 처리합니다.

```
COMMENT ON COLUMN emp.empno IS 'Employee identification number';
```

이와 같은 코멘트를 삭제합니다.

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

### 3.3.9 COMMIT

**이름**

COMMIT - 현재 트랜잭션을 커밋합니다

**개요**

```
COMMIT [WORK]
```

**설명**

COMMIT 은 현재 트랜잭션을 커밋합니다. 트랜잭션으로 생긴 모든 변경사항이 다른 사람에게 가시화 되고, 크래쉬가 발생해도 일관성이 보장됩니다.

**매개변수**

WORK

선택적 키워드입니다. 아무 효과가 없습니다.

**주석**

트랜잭션을 중단 하려면 ROLLBACK 을 사용하십시오.

트랜잭션 외부에서 COMMIT 을 발행하는 것은 특별한 문제가 발생하지 않습니다.

SPL 프로그램 안에 COMMIT 은 지원되지 않습니다.

**예**

현재 transaction 을 커밋하고, 모든 변경 사항을 유지합니다.

```
COMMIT;
```

**관련 항목**

ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT

### 3.3.10 CREATE DATABASE

**이름**

CREATE DATABASE - 새로운 데이터베이스를 생성합니다

**개요**

```
CREATE DATABASE name
```

**설명**

CREATE DATABASE 는 새로운 데이터베이스를 생성합니다.

데이터베이스를 생성하려면, 슈퍼유저 또는 특별한 CREATEDB 권한이 있는 사용자여야 합니다. 일반적으로 데이터베이스 생성자가 새 데이터베이스 소유자가 됩니다. CREATEDB 권한을 보유하는 비슈퍼유저는 그들이 소유하는 데이터베이스만 생성할 수 있습니다.

새 데이터베이스는 표준 시스템 데이터베이스 template1 을 복제하여 생성됩니다.

**매개변수**

name

생성할 데이터베이스의 이름입니다.

**주석**

CREATE DATABASE 는 트랜잭션 블록 내부에서 실행할 수 없습니다.

대부분의 경우 "could not initialize database directory"이라는 행이 포함된 에러는 데이터 디렉토리의 권한 부족, 디스크 공간 부족 등의 파일 시스템에 대한 문제와 관련된 것입니다.

**예**

새 데이터베이스를 생성합니다.

```
CREATE DATABASE employees;
```

### 3.3.11 CREATE DATABASE LINK

**이름**

CREATE DATABASE LINK - 새로운 데이터베이스 링크를 생성합니다

**개요**

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

**설명**

CREATE DATABASE LINK 는 새로운 데이터베이스 링크를 만듭니다. 데이터베이스 링크는 DELETE, INSERT, SELECT, UPDATE 명령 내에서 원격 데이터베이스의 테이블 또는 뷰를 참조할 수 있는 객체입니다. 데이터베이스 링크는 SQL 명령에서 참조하는 테이블이나 뷰의 이름 다음에 @dblink 를 추가함으로써 참조될 수 있습니다. dblink 는 데이터베이스 링크의 이름입니다.

데이터베이스 링크에는 public 또는 private 이 있습니다. public 데이터베이스 링크는 모든 사용자가 사용할 수 있습니다. private 데이터베이스 링크는 데이터베이스 링크 소유자만 사용할

수 있습니다. PUBLIC 옵션을 사용하면 public 데이터베이스 링크를 만듭니다. 이 옵션을 생략하면 private 데이터베이스 링크가 생성됩니다.

CREATE DATABASE LINK 명령을 실행하면, 데이터베이스 링크 이름과 지정된 속성이 Postgres Plus Advanced Server 의 pg_catalog.edb_dblink 라는 시스템 테이블에 저장됩니다. 데이터베이스 링크를 사용하는 경우에는 데이터베이스 링크를 정의하는 edb_dblink 엔트리가 있는 데이터베이스를 로컬 데이터베이스라고 합니다. 그리고 edb_dblink 엔트리에서 정의한 속성을 가진 대상 서버 및 데이터베이스를 원격 데이터베이스라고 합니다.

데이터베이스 링크에 대한 참조를 포함하는 SQL 명령은 로컬 데이터베이스에 연결할 때 실행되어야 합니다. @ dblink 를 포함하는 SQL 명령은 원격 데이터베이스에 대한 적절한 인증과 접속을 실행하여, 테이블 및 뷰에 액세스할 수 있도록 합니다.

**매개변수**

PUBLIC

모든 사용자가 사용할 수 있는 public 데이터베이스 링크를 만듭니다. 이 매개변수가 생략된 경우, private 데이터베이스 링크가 되어, 데이터베이스 링크 소유자만 사용할 수 있습니다.

name

생성하는 데이터베이스 링크의 이름입니다.

username

원격 데이터베이스에 접속할 때의 사용자 이름입니다.

password

username 의 패스워드입니다.

libpq

원격 Postgres Plus Advanced Server 데이터베이스의 접속을 지정합니다.

oci

원격 Oracle 데이터베이스의 접속을 지정합니다. 매개변수가 생략되면 기본값입니다.

hostname

원격 데이터베이스 서버의 이름이나 IP 주소입니다.

portnum

원격 데이터베이스 서버에 접속할 수 있는 포트 번호입니다.

database

원격 데이터베이스의 이름입니다.

**주석**

원격 Oracle 데이터베이스에 데이터베이스 링크를 참조하는 SQL 명령이 실행되는 경우, 서버는 디스크상의 Oracle 이 설치되어 있는 위치를 알 필요가 있습니다. Oracle 설치가 올바른 Home 디렉토리로 설정되어 Postgres Plus Advanced Server 가 시작하도록 하는 2 가지 방법이 있습니다.

- 환경 변수 ORACLE_HOME 이 올바른 디렉토리로 설정되어 있을 수 있습니다. 이것이 기본 Oracle 환경변수 설정입니다.
- postgresql.conf 의 환경설정 변수 oracle_home 은 Postgres Plus Advanced Server 를 파일 시스템내의 올바른 Oracle Home 디렉토리로 설정되도록 지시합니다. oracle_home 에 대한 정보는 1.3.4 장을 참조하세요.

**예**

다음 예제는 다음과 같이 가정합니다. Postgres Plus Advanced Server 샘플 애플리케이션의 emp 테이블의 복사본을 Oracle 데이터베이스에서 만듭니다. 두 번째 샘플 애플리케이션이 있는 Postgres Plus Advanced Server 클러스터는 포트 번호 5443 에서 접속을 수용합니다.

oralink 라는 public 데이터베이스 링크를 생성합니다. Oracle 데이터베이스의 이름은 xe, IP 주소는 127.0.0.1, 포트 번호는 1521 입니다. 또한 Oracle 데이터베이스에 접속하기 위한 사용자 이름은 edb, 패스워드는 password 입니다.

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password'USING '/ /
127.0.0.1/xe';
```

oralink 라는 데이터베이스 링크를 사용하여 Oracle 데이터베이스의 emp 테이블에서 SELECT 명령을 실행합니다.

```
SELECT * FROM emp @ oralink;
```

```
empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+--------+-----------+------+------------- ------+------+------+--------
7369 | SMITH | CLERK | 7902 | 17 - DEC - 80 00:00:00 | 800 | | 20
7499 | ALLEN | SALESMAN | 7698 | 20 - FEB - 81 00:00:00 | 1600 | 300 | 30
7521 | WARD | SALESMAN | 7698 | 22 - FEB - 81 00:00:00 | 1250 | 500 | 30
7566 | JONES | MANAGER | 7839 | 02 - APR - 81 00:00:00 | 2975 | | 20
7654 | MARTIN | SALESMAN | 7698 | 28 - SEP - 81 00:00:00 | 1250 | 1400 | 30
7698 | BLAKE | MANAGER | 7839 | 01 - MAY - 81 00:00:00 | 2850 | | 30
7782 | CLARK | MANAGER | 7839 | 09 - JUN - 81 00:00:00 | 2450 | | 10
7788 | SCOTT | ANALYST | 7566 | 19 - APR - 87 00:00:00 | 3000 | | 20
7839 | KING | PRESIDENT | | 17 - NOV - 81 00:00:00 | 5000 | | 10
7844 | TURNER | SALESMAN | 7698 | 08 - SEP - 81 00:00:00 | 1500 | 0 | 30
7876 | ADAMS | CLERK | 7788 | 23 - MAY - 87 00:00:00 | 1100 | | 20
7900 | JAMES | CLERK | 7698 | 03 - DEC - 81 00:00:00 | 950 | | 30
7902 | FORD | ANALYST | 7566 | 03 - DEC - 81 00:00:00 | 3000 | | 20
7934 | MILLER | CLERK | 7782 | 23 - JAN - 82 00:00:00 | 1300 | | 10
(14 rows)
```

edblink 라는 private 데이터베이스 링크를 만듭니다. Postgres Plus Advanced Server 데이터베이스 이름은 edb, 위치는 localhost, 링크할 수 있는 포트 번호는 5443 입니다. 또한 Postgres Plus Advanced Server 데이터베이스에 접속하기 위한 사용자 이름은 enterprisedb, 패스워드는 password 입니다.

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

로컬 edb_dblink 의 시스템 테이블에서 데이터베이스 링크 oralink 과 edblink 의 속성을 표시합니다.

```
SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;

lnkname | lnkuser | lnkconnstr
---------+--------------+------------------------ ------------
oralink | edb | / / 127.0.0.1/xe
edblink | enterprisedb | host = localhost port = 5443 dbname = edb
(2 rows)
```

Oracle 데이터베이스의 emp 테이블과 Postgres Plus Advanced Server 데이터베이스의 dept 테이블을 조인합니다.

```
SELECT d.deptno, d.dname, e.empno, e.ename, e.job, e.sal, e.comm FROM emp @ oralink e,
dept @ edblink d    WHERE e.deptno = d.deptno ORDER BY 1, 3;

deptno | dname | empno | ename | job | sal | comm
--------+------------+-------+--------+----------- +------+------
10 | ACCOUNTING | 7782 | CLARK | MANAGER | 2450 |
10 | ACCOUNTING | 7839 | KING | PRESIDENT | 5000 |
10 | ACCOUNTING | 7934 | MILLER | CLERK | 1300 |
20 | RESEARCH | 7369 | SMITH | CLERK | 800 |
20 | RESEARCH | 7566 | JONES | MANAGER | 2975 |
20 | RESEARCH | 7788 | SCOTT | ANALYST | 3000 |
20 | RESEARCH | 7876 | ADAMS | CLERK | 1100 |
20 | RESEARCH | 7902 | FORD | ANALYST | 3000 |
30 | SALES | 7499 | ALLEN | SALESMAN | 1600 | 300
30 | SALES | 7521 | WARD | SALESMAN | 1250 | 500
30 | SALES | 7654 | MARTIN | SALESMAN | 1250 | 1400
30 | SALES | 7698 | BLAKE | MANAGER | 2850 |
30 | SALES | 7844 | TURNER | SALESMAN | 1500 | 0
30 | SALES | 7900 | JAMES | CLERK | 950 |
(14 rows)
```

**관련 항목**

DROP DATABASE LINK

### 3.3.12 CREATE DIRECTORY

**이름**

CREATE DIRECTORY - 파일 시스템 디렉토리 경로로 별칭을 생성합니다.

**개요**

```
CREATE DIRECTORY name AS 'pathname'
```

**설명**

CREATE DIRECTORY 명령은 파일 시스템 디렉토리 경로로 별칭을 만듭니다. UTL_FILE 패키지 프로그램에 적절한 매개변수로 별칭이 지정된 경우에는 운영체제의 파일은 다른 이름에 해당하는 디렉토리에 생성되거나, 또는 그 디렉토리에 접근할 수 있습니다. UTL_FILE 패키지에 대한 자세한 내용은 7.4 장을 참조하십시오.

**매개변수**

name

디렉토리 경로 별칭입니다.

pathname

별칭과 관련된 디렉토리의 절대 경로입니다. CREATE DIRECTORY 명령은 운영체제의 디렉토리를 생성하지 않습니다. 실제 디렉토리는 적절한 운영체제의 명령을 사용하여 독립적으로 생성되어야 합니다.

**주석**

UTL_FILE 패키지가 디렉토리에 파일을 만들거나 디렉토리의 파일을 읽을 수 있다면, 운영체제의 사용자 ID enterprisedb 는 해당 디렉토리에 적절한 읽기 / 쓰기 권한이 있어야 합니다.

디렉토리 경로 별칭을 삭제하려면 pg_catalog.edb_dir 시스템 카탈로그 테이블에서 직접 삭제되어야 합니다. 이 작업은 슈퍼유저가 수행해야 합니다. edb_dir 는 Oracle 과 호환되지 않는 것에 주의하세요.

디렉토리 경로 별칭을 삭제해도 해당 파일 시스템의 실제 디렉토리는 제거되지 않습니다. 실제 디렉토리는 적절한 운영체제 명령으로 제거해야 합니다.

Linux 시스템에서 디렉토리 이름의 구분 기호는 슬래시 (/)입니다.

Windows 시스템에서 디렉토리 이름 구분 기호는 슬래시 (/) 또는 2 개의 백슬래시 (₩₩)로 지정할 수 있습니다.

**예**

Linux 의 디렉토리 / tmp / empdir 에 empdir 라는 별칭을 만듭니다.

```
CREATE DIRECTORY empdir AS '/ tmp / empdir';
```

Windows 의 디렉토리 C : ₩ TEMP ₩ EMPDIR 에 empdir 라는 별칭을 만듭니다.

```
CREATE DIRECTORY empdir AS 'C : ₩ TEMP / EMPDIR';
```

모든 디렉토리 경로 별칭을 표시합니다.

```
SELECT * FROM pg_catalog.edb_dir;

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

디렉토리 경로 별칭 empdir 를 삭제합니다.

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

### 3.3.13 CREATE FUNCTION

**이름**

CREATE FUNCTION - 새로운 함수를 정의한합니다

**개요**

```
CREATE [OR REPLACE] FUNCTION name
[(argname [IN | IN OUT | OUT] argtype [DEFAULT value]
[, ...])]
RETURN rettype
[AUTHID (DEFINER | CURRENT_USER)]
(IS | AS)
[declaration] [...]
BEGIN
      statement; [...]
[EXCEPTION
(WHEN exception [OR exception] [...] THEN
         statement; [...]) [...]
]
END [name]
```

**설명**

CREATE FUNCTION 은 새로운 함수를 정의합니다. CREATE OR REPLACE FUNCTION 은 새로운 함수를 만들거나 기존 정의를 대체합니다.

스키마 이름이 포함된 경우, 함수는 지정된 스키마에서 생성됩니다. 스키마 이름이 없으면, 함수는 현재 스키마에서 생성됩니다. 같은 스키마에서 동일한 인수 데이터형의 기존 함수의 이름은 새로운 함수의 이름으로 사용할 수 없습니다. 그러나 다른 인수 데이터형을 갖는 함수의 경우, 이름이 중복 되어도 (이것을 오버로딩이라 합니다). (함수 오버로딩은 Postgres Plus Advanced Server 기능입니다. 저장된 함수의 오버로딩은 Oracle 과 호환하지 않습니다.)

기존의 함수 정의를 갱신하려면, CREATE OR REPLACE FUNCTION 을 사용하십시오. 이 방법은 함수 이름과 인수 형을 변경할 수 없다는 것을 기억하십시오 (이렇게 하면 새로 다른 함수가 생성됩니다). 또한 CREATE OR REPLACE FUNCTION 는 기존 함수의 반환형을 변경할 수 없습니다. 반환형을 변경하려면, 해당 함수를 삭제하고 다시 만들어야 합니다.

함수를 만든 사용자가 해당 함수의 소유자입니다.

함수에 대한 자세한 정보는 4.2.4 장을 참조하십시오.

**매개변수**

name

생성하는 함수의 이름(스키마로 수식된 이름도 가능)입니다.

argname

인수의 이름입니다. 인수는 함수 본문 안에 이름에서 참조될 수 있습니다.

IN | IN OUT | OUT

인수 모드입니다. IN 은 입력 전용의 인수임을 선언합니다. 이것이 기본값입니다. IN OUT 는 인수가 값을 받고, 반환하는 것을 허용합니다. OUT 은 출력 전용의 인수임을 선언합니다.

argtype

함수의 인수 데이터형입니다. 기본 데이터형 또는 테이블 컬럼에 대한 참조를 사용할 수 있습니다. 모든 기본 데이터형에 길이를 지정하지 않습니다. 예를 들면, VARCHAR2 (10)이 아닌 VARCHAR2 로 지정합니다.

컬럼은 tablename. columnname % TYPE 을 작성하여 참조됩니다. 이것을 사용하면 테이블 정의가 변경 되더라도 함수가 영향을 받지 않도록 하는 데 유용합니다.

DEFAULT value

함수를 호출할 때 인수가 전달되지 않으면 입력 인수로 기본값이 사용됩니다. 인수의 모드가 IN OUT 또는 OUT 인 경우, DEFAULT 를 지정하지 않습니다.

rettype

함수에서 반환하는 데이터형입니다. argtype 처럼, rettype 에도 길이를 지정하지 않습니다.

DEFINER | CURRENT_USER

함수에서 참조하는 데이터베이스 객체에 대한 액세스 권한을 결정하기 위해 기능의 소유자 권한 (DEFINER)를 사용하는지, 함수를 실행하여 현재 사용자의 권한 (CURRENT_USER)를 사용하는지 지정합니다. 또한, DEFINER 에서는 자격이 되지 않는 데이터베이스 객체를 참조하기 위해 기능 소유자의 검색 경로를 사용합니다. 한편 CURRENT_USER 에서는 함수를 실행하여 현재 사용자의 검색 경로를 사용합니다. DEFINER 가 기본입니다.

declaration

변수, 형, REF CURSOR 선언입니다.

statement

SPL 프로그램 문장입니다. DECLARE - BEGIN - END 블록은 SPL 문장으로 간주된다는 점에 유의하시기 바랍니다. 따라서 함수 내부에 중첩된 블록이 있을 수 있습니다.

exception

NO_DATA_FOUND, OTHERS 와 같은 예외 상황의 이름입니다.

**주석**

Postgres Plus Advanced Server 는 함수 오버 로딩을 허용합니다. 즉, 인수형이 다른 경우에 여러 함수에 같은 이름을 사용할 수 있습니다.

**예**

emp_comp 함수는 2 개의 입력 인수와 1 개의 반환 값을 취합니다. 다음 SELECT 명령 함수를 사용하는 방법을 설명합니다.

```
CREATE OR REPLACE FUNCTION emp_comp (
p_sal NUMBER,
p_comm NUMBER
) RETURN NUMBER
IS
BEGIN
RETURN (p_sal + NVL (p_comm, 0)) * 24;
END;

SELECT ename "Name", sal "Salary"comm "Commission"emp_comp (sal, comm)
"Total Compensation"FROM emp;

Name | Salary | Commission | Total Compensation
--------+---------+------------+------------------ -
SMITH | 800.00 | | 19200.00
ALLEN | 1600.00 | 300.00 | 45600.00
WARD | 1250.00 | 500.00 | 42000.00
JONES | 2975.00 | | 71400.00
MARTIN | 1250.00 | 1400.00 | 63600.00
BLAKE | 2850.00 | | 68400.00
CLARK | 2450.00 | | 58800.00
SCOTT | 3000.00 | | 72000.00
KING | 5000.00 | | 120000.00
TURNER | 1500.00 | 0.00 | 36000.00
ADAMS | 1100.00 | | 26400.00
JAMES | 950.00 | | 22800.00
FORD | 3000.00 | | 72000.00
MILLER | 1300.00 | | 31200.00
(14 rows)
```

sal_range 함수는 월급이 지정된 범위에 있는 종업원의 수를 반환합니다. 다음 함수 호출은 첫 번째 2 개의 호출에 기본 인수를 사용합니다.

```
CREATE OR REPLACE FUNCTION sal_range (
p_sal_min NUMBER DEFAULT 0,
p_sal_max NUMBER DEFAULT 10000
```

```
) RETURN INTEGER
IS
v_count INTEGER;
BEGIN
SELECT COUNT (*) INTO v_count FROM emp
WHERE sal BETWEEN p_sal_min AND p_sal_max;
RETURN v_count;
END;

BEGIN
DBMS_OUTPUT.PUT_LINE ( 'Number of employees with a salary :'| |
sal_range);
DBMS_OUTPUT.PUT_LINE ( 'Number of employees with a salary of at least'
| | '$ 2000.00 :'| | sal_range (2000.00));
DBMS_OUTPUT.PUT_LINE ( 'Number of employees with a salary between'
| | '$ 2000.00 and $ 3000.00 :'| | sal_range (2000.00, 3000.00));

END;

Number of employees with a salary : 14
Number of employees with a salary of at least $ 2000.00 : 6
Number of employees with a salary between $ 2000.00 and $ 3000.00 : 5
```

**관련 항목**

DROP FUNCTION

### 3.3.14 CREATE INDEX

**이름**

CREATE INDEX - 새 인덱스를 정의한다

**개요**

```
CREATE [UNIQUE] INDEX name ON table
((column | (expression)))
[TABLESPACE tablespace]
```

**설명**

CREATE INDEX 는 지정한 테이블에서 name 인덱스를 만듭니다. 인덱스는 주로 데이터베이스의 성능 향상을 위해 사용됩니다 (그러나 인덱스를 잘못 사용하면 성능이 저하될 수 있습니다).

인덱스 키 필드는 열 이름, 또는 괄호로 둘러싸인 표현 식으로 지정됩니다. 여러 열에 대한 인덱스를 만들 때는 여러 필드를 지정할 수 있습니다.

인덱스 필드로 테이블 행의 1 개 이상 열 값을 계산하는 방식을 지정할 수 있습니다. 이 기능은 원본 데이터에 대한 변환 값을 기준으로 하는 데이터에 대한 빠른 액세스하는 수단으로 사용할 수 있습니다. 예를 들면, UPPER (col)의 계산에 따른 인덱스는 인덱스를 사용하기 위해 WHERE UPPER (col) = 'JIM'어구를 허용합니다.

Postgres Plus Advanced Server 는 B - tree 인덱스 메소드를 제공합니다. B - tree 인덱스 메소드는 Lehman - Yao high - concurrency B – trees 를 구현한 것입니다.

기본적으로 인덱스는 IS NULL 어구를 사용하지 않습니다.

인덱스의 정의에 사용되는 모든 함수와 연산자는 "불변"(immutable)해야 합니다. 즉, 결과는 입력 인수에 의존하고 (다른 테이블의 내용 또는 현재 시간과 같은) 외부의 영향을 받을 수 없습니다. 이 제약은 인덱스 작업이 충분히 정의되어 있는 것을 확인할 수 있습니다. 인덱스식에서 사용자 정의 함수를 사용하려면, 함수를 생성할 때 immutable 옵션을 붙이는 것을 잊지 마세요.

**매개변수**

UNIQUE

인덱스(이미 데이터가 있는 상황에서)를 만들 때, 테이블에 각각의 시간 데이터를 추가할 때 테이블의 값이 중복되지 않았는지 확인합니다. 데이터를 삽입하거나 갱신할 때는 중복 엔트리의 결과가 나오면서, 에러가 발생합니다.

name

생성되는 인덱스의 이름입니다. 스키마 이름을 여기에 포함시킬 수 없습니다. 인덱스는 항상 부모 테이블과 같은 스키마에서 생성됩니다.

table

인덱스 되는 생성 테이블의 이름(스키마로 수식된 이름도 가능)입니다.

column

테이블의 열 이름입니다.

expression

테이블에 1 개 이상의 열을 기본으로 하는 식입니다. 보통, 식은 구문에서 보여준 대로 괄호로 묶어야 합니다. 그러나 식이 함수 호출 형식으로 되어있는 경우에는 괄호를 생략할 수 있습니다.

tablespace

인덱스를 생성하는 테이블스페이스입니다. 지정되지 않은 경우 default_tablespace 을 사용합니다. default_tablespace 가 빈 문자열이면, 데이터베이스의 기본 테이블스페이스를 사용합니다.

**주석**

여러 열에 대한 인덱스로 지정할 수 있는 필드는 32 개입니다.

**예**

emp 테이블의 ename 열에 B - tree 인덱스를 만듭니다.

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

index_tblspc 테이블스페이스 내에 위와 같은 인덱스를 만듭니다.

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

**관련 항목**

ALTER INDEX, DROP INDEX

### 3.3.15 CREATE PACKAGE

**이름**

CREATE PACKAGE - 새로운 패키지 사양을 정의합니다

**개요**

```
CREATE [OR REPLACE] PACKAGE name
[AUTHID (DEFINER | CURRENT_USER)]
```

```
(IS | AS)
[declaration] [...]
[(PROCEDURE proc_name
[(argname [IN | IN OUT | OUT] argtype [DEFAULT value]
[, ...])];
|
FUNCTION func_name
[(argname [IN | IN OUT | OUT] argtype [DEFAULT value]
[, ...])]
RETURN rettype;
)
] [...]
END [name]
```

**설명**

CREATE PACKAGE 는 새로운 패키지 사양을 정의합니다. CREATE OR REPLACE PACKAGE 는 새로운 패키지 사양을 만들거나 또는 기존 사양을 대체합니다.

스키마 이름이 포함되어있는 경우, 패키지는 지정된 스키마에서 생성됩니다. 스키마 이름이 없는 경우, 패키지는 현재 스키마에서 생성됩니다. CREATE OR REPLACE PACKAGE 를 사용하여 기존 패키지 정의를 갱신하려는 의도가 있지 않는 한, 새로운 패키지의 이름은 동일한 스키마의 기존 패키지의 이름과 일치하지 않아야 합니다.

프로시저를 생성하는 사용자가 패키지의 소유자입니다.

패키지에 대한 자세한 내용은 6 장을 참조하십시오.

**매개변수**

name

생성하는 패키지의 이름 (스키마로 수식된 이름도 가능).

DEFINER | CURRENT_USER

패키지에서 참조하는 데이터베이스 객체에 대한 액세스 권한을 결정하기 위해 패키지의 소유자 권한 (DEFINER)을 사용할지, 패키지의 프로그램을 실행하여 현재 사용자의 권한 (CURRENT_USER)을 사용 것인가를 결정합니다. 또한, DEFINER 는 자격이 되지 않는 데이터베이스

객체를 참조하는 패키지 소유자의 검색 경로를 사용합니다. 한편 CURRENT_USER 에서는 패키지의 프로그램을 실행하여 현재 사용자의 검색 경로를 사용합니다. DEFINER 가 기본입니다.

declaration

public 변수, 형, 커서, REF CURSOR 선언입니다.

proc_name

public 프로시저의 이름입니다.

argname

인수의 이름입니다.

IN | IN OUT | OUT

인수 모드입니다.

argtype

프로그램의 인수 데이터형입니다.

DEFAULT value

입력 인수에 대한 기본값입니다.

func_name

public 함수의 이름입니다.

rettype

반환 값의 데이터형입니다.

**예**

empinfo 패키지 사양은 다음의 3 개의 public 요소가 포함되어있습니다. public 변수, public 프로시저, public 함수 입니다. 이 예제에 있는 패키지의 본문은 CREATE PACKAGE BODY 명령을 참조하십시오.

```
CREATE OR REPLACE PACKAGE empinfo
```

```
IS
emp_name VARCHAR2 (10);
PROCEDURE get_name (
p_empno NUMBER
);
FUNCTION display_counter
RETURN INTEGER;
END;
```

**관련 항목**

CREATE PACKAGE BODY, DROP PACKAGE

### 3.3.16 CREATE PACKAGE BODY

**이름**

CREATE PACKAGE BODY - 새로운 패키지 본문을 정의합니다.

**개요**

```
CREATE [OR REPLACE] PACKAGE BODY name
(IS | AS)
[declaration] [...]
[(PROCEDURE pro c_name
[(argname [IN | IN OUT | OUT] argtype [DEFAULT value]
[, ...])]
(IS | AS)
        program_body
END [proc_name];
|
FUNCTION func_name
[(argname [IN | IN OUT | OUT] argtype [DEFAULT value]
[, ...])]
RETURN rettype
(IS | AS)
        program_body
END [func_name];
)
] [...]
```

[BEGIN
        statement; [...]
END [name]

**설명**

CREATE PACKAGE BODY 은 새로운 패키지 본문을 정의합니다. CREATE OR REPLACE PACKAGE BODY 는 새로운 패키지 본문을 만들거나 기존 패키지의 본문 대체 중 하나를 수행합니다.

스키마 이름이 포함되어있는 경우, 패키지 본문은 지정된 스키마에서 생성됩니다. 스키마 이름이 없는 경우, 현재의 스키마에서 생성됩니다. 새로운 패키지 본문의 이름은 동일한 스키마의 기존 패키지 사양과 일치해야 합니다. 그러나 CREATE OR REPLACE PACKAGE BODY 을 사용하여 기존 패키지 본문의 정의를 변경하려는 의도가 있지 않는 한, 새 패키지 본문의 이름은 동일한 스키마의 기존 패키지 본문의 이름과 일치하지 않아야 합니다. 패키지 본문에 대한 자세한 내용은 제 6.1.2 장과 제 6.2.2 장을 참조하십시오.

**매개변수**

name

생성하는 패키지 본문의 이름 (스키마로 수식된 이름도 가능).

declaration

전용 변수, 형, 커서, REF CURSOR 선언입니다.

proc_name

public 프로시저 또는 private 프로시저의 이름입니다. proc_name 가 동일한 이름의 패키지 사양에 정의되어 있는 경우 public 프로시저입니다. 정의되지 않은 경우, private 프로시저입니다.

argname

인수의 이름입니다.

IN | IN OUT | OUT

인수 모드입니다.

argtype

프로그램의 인수 데이터형입니다.

DEFAULT value

입력 인수에 대한 기본값입니다.

program_body

함수 또는 프로시저의 본문을 구성하는 선언 및 SPL 문장입니다.

func_name

public 함수 혹은 private 함수의 이름입니다. func_name 이 동일한 이름의 패키지 사양에 정의되어있는 경우 public 함수입니다. 정의되지 않은 경우, private 함수입니다.

rettype

반환 값의 데이터형입니다.

statement

SPL 프로그램 문장입니다. 패키지 초기화 부분의 문장은 처음 한번만 실행됩니다.

**예**

다음은 empinfo 패키지의 본문입니다.

```
CREATE OR REPLACE PACKAGE BODY empinfo
IS
v_counter INTEGER;
PROCEDURE get_name (
p_empno NUMBER
)
IS
BEGIN
SELECT ename INTO emp_name FROM emp WHERE empno = p_empno;
v_counter : = v_counter + 1;
END;
FUNCTION display_counter
RETURN INTEGER
IS
```

```
BEGIN
RETURN v_counter;
END;
BEGIN
v_counter : = 0;
DBMS_OUTPUT.PUT_LINE ( 'Initialized counter');
END;
```

다음 예제는 empinfo 패키지 프로시저와 함수를 실행하여 public 변수를 표시합니다.

```
BEGIN
empinfo.get_name (7369);
DBMS_OUTPUT.PUT_LINE ( 'Employee Name :'| | empinfo.emp_name);
DBMS_OUTPUT.PUT_LINE ( 'Number of queries :'| | empinfo.display_counter);
END;

Initialized counter
Employee Name : SMITH
Number of queries : 1

BEGIN
empinfo.get_name (7900);
DBMS_OUTPUT.PUT_LINE ( 'Employee Name :'| | empinfo.emp_name);
DBMS_OUTPUT.PUT_LINE ( 'Number of queries :'| | empinfo.display_counter);
END;

Employee Name : JAMES
Number of queries : 2
```

**관련 항목**

CREATE PACKAGE, DROP PACKAGE

### 3.3.17 CREATE PROCEDURE

**이름**

CREATE PROCEDURE - 새로운 저장 프로시저를 정의한다

**개요**

```
CREATE [OR REPLACE] PROCEDURE name
[(argname [IN | IN OUT | OUT] arg type [DEFAULT value]
[, ...])]
[AUTHID (DEFINER | CURRENT_USER)]
(IS | AS)
[declaration] [...]
BEGIN
      statement; [...]
[EXCEPTION
(WHEN exception [OR exception] [...] THEN
         statement; [...]) [...]
]
END [name]
```

**설명**

CREATE PROCEDURE 는 새로운 저장 프로시저를 정의합니다. CREATE OR REPLACE PROCEDURE 는 새로운 프로시저를 만들거나 또는 기존 프로시저를 대체합니다.

스키마 이름이 포함된 경우, 프로시저는 지정된 스키마에서 생성됩니다. 스키마 이름이 없으면 프로시저는 현재 스키마에서 생성됩니다. CREATE OR REPLACE PROCEDURE 를 사용하여 기존 프로시저의 정의를 변경하려는 의도가 있지 않은 한, 새로운 프로시저의 이름은 동일한 스키마의 기존 프로시저의 이름과 일치하지 않아야 합니다. ,

프로시저를 만든 사용자가 프로시저의 소유자가 됩니다.

프로시저에 대한 자세한 내용은 제 4.2.3 장을 참조하십시오.

**매개변수**

name

생성하는 프로시저의 이름 (스키마로 수식된 이름도 가능).

argname

인수의 이름입니다. 인수는 프로시저 본문에 있는 이름에서 참조됩니다.

IN | IN OUT | OUT

인수모드입니다. IN 은 인수가 입력 전용의 인수임을 선언합니다. 이것이 기본값입니다. IN OUT 은 인수가 값을 반환하고, 취득할 수 있음을 허용합니다. OUT 은 출력 전용의 인수임을 지정합니다.

argtype

프로시저의 인수 데이터형입니다. 인수 데이터형은 기본 데이터형, 또는 기존의 컬럼이 될 수도 있습니다. 모든 기본 데이터형에 길이를 지정하지 않습니다. 예를 들면, VARCHAR2 (10)이 아닌 VARCHAR2 로 지정합니다.

컬럼은 tablename. columnname % TYPE 으로 참조됩니다. 이것을 사용하면 테이블 정의가 변경되더라도 프로시저가 영향을 받지 않도록 하는 데 유용할 수 있습니다.

DEFAULT value

프로시저를 호출할 때 인수가 전달되지 않으면 입력 인수로 기본값이 사용됩니다. 인수의 모드가 IN OUT 또는 OUT 의 경우 DEFAULT 가 지정되지 않습니다.

DEFINER | CURRENT_USER

프로시저에서 참조하는 데이터베이스 객체에 대한 액세스 권한을 결정하기 위해 프로시저 소유자의 권한 (DEFINER)를 사용하는지, 프로시저를 실행하여 현재 사용자의 권한 (CURRENT_USER)를 사용하는지 지정합니다. 또한, DEFINER 는 자격이 되지 않는 데이터베이스 객체를 참조하도록 프로시저 소유자의 검색 경로를 사용합니다. 한편 CURRENT_USER 에서는 프로시저를 실행하여 현재 사용자의 검색 경로를 사용합니다. DEFINER 가 기본입니다.

declaration

변수, 형, REF CURSOR 선언입니다.

statement

SPL 프로그램 문장입니다. DECLARE - BEGIN - END 블록은 SPL 문장으로 간주된다는 점에 유의하시기 바랍니다. 따라서 함수 본문에 중첩된 블록이 있을 수 있습니다.

exception

NO_DATA_FOUND, OTHERS 와 같은 예외 조건명 입니다.

**예**

다음 프로시저는 emp 테이블의 직원을 표시합니다.

```
CREATE OR REPLACE PROCEDURE list_emp
IS
v_empno NUMBER (4);
v_ename VARCHAR2 (10);
CURSOR emp_cur IS
SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
OPEN emp_cur;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH emp_cur INTO v_empno, v_ename;
EXIT WHEN emp_cur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | v_ename);
END LOOP;
CLOSE emp_cur;
END;

EXEC list_emp;

EMPNO ENAME
----- -------
7369 SMITH
7499 ALLEN
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7782 CLARK
7788 SCOTT
7839 KING
7844 TURNER
7876 ADAMS
7900 JAMES
7902 FORD
7934 MILLER
```

다음 프로시저는 직원 번호, 이름 및 직업을 반환하는 인수 모드의 IN OUT 과 OUT 을 사용합니다. 첫째로 지정된 직원 번호를 바탕으로 검색합니다. 찾을 수 없는 경우, 지정된 이름을 바탕으로 다시 찾습니다. 익명의 블록은 프로시저를 호출합니다.

```
CREATE OR REPLACE PROCEDURE emp_job (
p_empno IN OUT emp.empno % TYPE,
p_ename IN OUT emp.ename % TYPE,
p_job OUT emp.job % TYPE
)
IS
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
v_job emp.job % TYPE;
BEGIN
SELECT ename, job INTO v_ename, v_job FROM emp WHERE empno = p_empno;
p_ename : = v_ename;
p_job : = v_job;
DBMS_OUTPUT.PUT_LINE ( 'Found employee #'| | p_empno);
EXCEPTION
WHEN NO_DATA_FOUND THEN
BEGIN
SELECT empno, job INTO v_empno, v_job FROM emp
WHERE ename = p_ename;
p_empno : = v_empno;
p_job : = v_job;
DBMS_OUTPUT.PUT_LINE ( 'Found employee'| | p_ename);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Could not find an employee with'| |
'number'| | p_empno | | 'nor name,'| | p_ename);
p_empno : = NULL;
p_ename : = NULL;
p_job : = NULL;
END;
END;

DECLARE
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
```

```
v_job emp.job % TYPE;
BEGIN
v_empno : = 0;
v_ename : = 'CLARK';
emp_job (v_empno, v_ename, v_job);
DBMS_OUTPUT.PUT_LINE ( 'Employee No :'| | v_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | v_ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | v_job);
END;

Found employee CLARK
Employee No : 7782
Name : CLARK
Job : MANAGER
```

**관련 항목**

DROP PROCEDURE

### 3.3.18 CREATE PUBLIC SYNONYM

**이름**

CREATE PUBLIC SYNONYM - 새로운 public 동의어를 정의한다

**개요**

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

**설명**

CREATE PUBLIC SYNONYM 데이터베이스 객체에 대한 public 동의어를 정의합니다. 동의어는 객체를 참조하는데 사용할 수 있는 또 다른 이름입니다. public 동의어는 데이터베이스에서 널리 사용할 수 있습니다. 데이터베이스 클러스터의 모든 사용자가 참조할 수 있습니다.

CREATE OR REPLACE PUBLIC SYNONYM 도 마찬가지입니다. 하지만 같은 이름의 public 동의어가 있는 경우 대체됩니다.

일반적으로 데이터베이스 객체는 SQL 문장에서 적절히 참조되도록 스키마 이름으로 완전히 수식되는 경우에, 동의어가 유용합니다. 객체에 정의된 동의어는 수식되지 않은, 하나의 이름으로 간편하게 참조 합니다.

public 동의어에 대한 자세한 내용은 제 2.2.4 장을 참조하십시오.

**매개변수**

name

생성된 public 동의어의 이름입니다.

object

public 동의어가 생성하는 데이터베이스 객체의 이름 (스키마로 수식한 이름 가능). 데이터베이스 객체는 테이블, 뷰, 시퀀스, 다른 동의어가 될 수 있습니다.

**주석**

모든 사용자가 public 동의어를 만들 수 있습니다. 특별한 권한이 필요하지 않습니다.

public 동의어는 모든 사용자에 의해 모든 SQL 문장에서 참조될 수 있습니다. 그러나, 동의어가 참조하는 데이터베이스 객체에 적절한 권한이 있는 사용자의 경우에만, SQL 문 실행이 성공합니다.

public 동의어는 스키마의 구성원이 아니지만, database - wide 이름입니다.

public 동의어는 존재하지 않는 객체에 대해서도 만들 수 있습니다.

public 동의어에서 참조하는 데이터베이스 객체에 대한 액세스 여부는 현재 사용자의 권한으로 결정됩니다. 따라서 public 동의어 사용자는 참조 데이터베이스 객체에 적절한 권한을 갖고 있어야 합니다.

**예**

스키마 이름 enterprisedb 의 emp 테이블에 대한 public 동의어 personnel 을 생성합니다.

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

**관련 항목**

DROP PUBLIC SYNONYM

### 3.3.19 CREATE ROLE

**이름**

CREATE ROLE - 새로운 데이터베이스 역할을 정의한다

**개요**

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

**설명**

CREATE ROLE 은 Postgres Plus Advanced Server 데이터베이스 클러스터에 새 역할을 추가합니다. 역할은 자신의 데이터베이스 객체를 소유할 수 있고 데이터베이스 권한을 가지는 엔티티입니다. 역할은 용도에 따라 "사용자", "그룹", 혹은 둘 모두 생각할 수 있습니다. 새로 정의된 역할은 LOGIN 속성이 없습니다. 따라서 세션을 시작하는 데 사용될 수 없습니다. LOGIN 권한을 부여하려면 ALTER ROLE 명령을 사용하십시오. CREATE ROLE 명령을 사용하려면 CREATEROLE 권한이 있거나 해당 데이터베이스의 슈퍼유저이어야 합니다.

IDENTIFIED BY 절이 지정된 경우, CREATE ROLE 커멘드는 새로 생성되는 역할과 동일한 이름의 스키마를 생성합니다. 새로 만든 역할이 스키마의 소유자입니다.

역할은 데이터베이스 클러스터 수준에서 정의되므로 클러스터의 모든 데이터베이스에서 사용되는 것에 주의하십시오.

**매개변수**

name

새 역할의 이름입니다.

IDENTIFIED BY password

역할의 패스워드를 설정합니다. (패스워드는 LOGIN 속성을 가진 역할에만 의미가 있지만, 이 특성이 없는 역할을 정의할 수 있습니다.) 패스워드 인증을 실시할 예정이 없는 경우, 이 옵션을 생략할 수 있습니다.

**주석**

역할의 속성을
변경하려면 ALTER ROLE, 역할을 삭제하려면 DROP ROLE 을 사용하십시오. CREATE ROLE 에 지정된 모든 속성은 나중에 ALTER ROLE 명령으로 변경할 수 있습니다.

그룹으로 사용되고 있는 역할의 구성원 추가 및 삭제에 대한 권장
방법은 GRANT 와 REVOKE 를 사용하는 것입니다.

역할 이름과 패스워드는 최대 63 자까지입니다.

**예**

admins 라는 패스워드가 있는 역할(그리고 스키마)를 생성합니다.

```
CREATE ROLE admins IDENTIFIED BY Rt498zb;
```

**관련 항목**

ALTER ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE

### 3.3.20 CREATE SCHEMA

**이름**

CREATE SCHEMA - 새로운 스키마를 정의한다

**개요**

```
CREATE SCHEMA      AUTHORIZATION username      schema_element [...]
```

**설명**

CREATE SCHEMA 는 1 개 이상의 객체가 포함된 username 이 소유한 새로운 스키마를
생성합니다. 스키마와 객체는 1 개의 트랜잭션에서 생성되기 때문에 모든 객체가 생성되거나,
또는 스키마를 가진 모든 객체를 만드는 데 실패하는 것입니다. (Oracle 에서 새 스키마는
만들어지지 않습니다. username 과 스키마는 사전에 존재해야 합니다.)

스키마는 기본적으로 네임 스페이스입니다. 스키마는 명명된 객체 (테이블, 뷰 등)가 포함됩니다.
이러한 객체의 이름은 다른 스키마에 존재하는 다른 객체의 이름과 중복해도 상관없습니다.
명명된 객체는 스키마 이름을 접두어로 이름을 "한정"하거나 필요한 스키마를 포함하는 검색
경로를 설정하여 사용할 수 있습니다. 한정되지 않은 객체는 현재의 스키마 (CURRENT_SCHEMA

함수가 결정되는 검색 경로의 시작 부분)에서 생성됩니다.)(검색 경로의 개념과 CURRENT_SCHEMA 함수는 Oracle 과 호환하지 않습니다.)

CREATE SCHEMA 은 스키마에서 객체를 생성 하려면 하위 명령을 추가할 수 있습니다. 하위 명령은 기본적으로 스키마를 만든 후 발행되는 다른 명령과 동일하게 취급됩니다. 그러나 생성된 모든 객체는 지정한 사용자가 소유한다는 점에서 다릅니다.

**매개변수**

username

새로운 스키마를 소유한 사용자의 이름입니다. 스키마 이름도 username 과 같습니다. 슈퍼유저만이 자신 이외의 사용자가 소유하는 스키마를 생성할 수 있습니다. (오라클 호환성 주석: Postgres Plus Advanced Server 에서 역할, username 은 존재해야 합니다. 그러나 스키마는 존재하지 않습니다. Oracle 에서는 모두 존재해야 합니다.)

schema_element

스키마에서 생성된 객체를 정의하는 SQL 구문입니다. CREATE SCHEMA 내에서는 CREATE TABLE, CREATE VIEW, GRANT 를 어구로 사용할 수 있습니다. 다른 종류의 객체는 스키마를 생성한 후 별도의 명령을 사용하면 만들 수 있습니다.

**주석**

스키마를 만들려면, 실행하는 사용자가 현재 데이터베이스에서 CREATE 권한을 가져야 합니다. (물론, 슈퍼에게는 이러한 제약이 없습니다.)

Postgres Plus Advanced Server 는 여기에 나열된 형식 이외에도 CREATE SCHEMA 명령이 형식이 있습니다. 그러나, 그들은 Oracle 과 호환되지 않습니다.

**예**

```
CREATE SCHEMA AUTHORIZATION enterprisedb
CREATE TABLE empjobs (ename VARCHAR2 (10), job VARCHAR2 (9))
CREATE VIEW managers AS SELECT ename FROM empjobs WHERE job = 'MANAGER'
GRANT SELECT ON managers TO PUBLIC;
```

### 3.3.21 CREATE SEQUENCE

**이름**

CREATE SEQUENCE - 새로운 시퀀스 생성기를 정의합니다

**개요**

```
CREATE SEQUENCE name [INCREMENT BY increment]
[(NOMINVALUE | MINVALUE minvalue)]
[(NOMAXVALUE | MAXVALUE maxvalue)]
[START WITH start] [CACHE cache | NOCACHE] [CYCLE]
```

**설명**

CREATE SEQUENCE 는 새로운 시퀀스 번호 생성기를 만듭니다. 특히, 새로운 name 이라는 이름을 가진 1 개의 행을 지닌 특수한 테이블을 만들고 초기화합니다. 생성기는 이 명령을 실행하는 사용자가 소유 합니다.

스키마 이름이 부여된 경우, 시퀀스는 지정된 스키마에서 생성됩니다. 스키마 이름이 없으면 시퀀스는 현재 스키마에서 생성됩니다. 시퀀스 이름은 동일한 스키마의 다른 시퀀스, 테이블, 인덱스, 뷰의 이름과 달라야 합니다.

시퀀스를 만든 후, NEXTVAL, CURRVAL 함수를 사용하여 시퀀스를 조작합니다. 이 함수는 단원 3.5.8 에 기재되어 있습니다.

**매개변수**

name

생성하는 시퀀스의 이름 (스키마로 수식된 이름도 가능).

increment

INCREMENT BY increment 어구는 현재 시퀀스 값에서 새로운 시퀀스 값을 만들 때 값의 증가량을 설정합니다. 이 어구는 선택 사항입니다. 양수가 지정되면 오름차순 시퀀스, 음수가 지정되면 내림차순 시퀀스를 만듭니다. 지정하지 않은 경우 기본값은 1 입니다.

NOMINVALUE | MINVALUE minvalue

MINVALUE minvalue 절은 시퀀스가 생성하는 최소값을 지정합니다. 이 어구는 선택 사항입니다. 이 어구가 지정되지 않은 경우, 혹은 키워드인 NOMINVALUE 가 지정되면 기본값이 사용됩니다. 시퀀스의 기본 최소값은 오름차순 때는 1 내림차순 때는 -2 63 -1 입니다.

NOMAXVALUE | MAXVALUE maxvalue

MAXVALUE maxvalue 절은 시퀀스의 최대값을 결정합니다. 이 어구는 선택 사항입니다. 이 어구가 지정되지 않은 경우, 혹은 키워드인 NOMAXVALUE 가 지정되면 기본값이 사용됩니다. 시퀀스의 기본 최소값은 오름차순 때는 263 -1 내림차순 때는 -1 입니다.

start

START WITH start 어구를 사용하면 임의의 숫자에서 시퀀스 번호를 시작할 수 있습니다. 이 어구는 선택 사항입니다. 기본적으로, 시퀀스의 시작 값은 오름차순 경우 minvalue, 내림차순의 경우 maxvalue 입니다.

cache

CACHE cache 옵션은 미리 할당하고 메모리에 저장할 시퀀스 번호를 지정합니다. 이에 따라 액세스를 빠르게 할 수 있습니다. 최소값은 1 입니다 (한번에 1 개의 값만이 생성됩니다. 즉, NOCACHE) 이것이 기본입니다.

CYCLE

CYCLE 옵션을 사용하면 시퀀스가 한계 (오름차순의 경우 maxvalue, 내림차순의 경우 minvalue)에 이르렀을 때 그 시퀀스를 회전시킬 수 있습니다. 한계에 도달했을 때 다음 생성되는 번호는 오름차순의 경우 minvalue, 내림차순의 경우 maxvalue 입니다.

CYCLE 이 생략되면 (디폴트), 시퀀스가 한계에 도달한 후에 NEXTVAL 를 호출하면 에러가 발생합니다. NO CYCLE 키워드는 기본 설정을 검색하는 데 사용됩니다. 다만, 이 옵션은 Oracle 과 호환되지 않습니다.

**주석**

시퀀스는 big integer 연산을 기반으로 합니다. 따라서 8 바이트 정수 범위 (-9223372036854775808 과 9223372036854775807 사이)를 초과하는 것은 불가능합니다. 비교적 오래된 플랫폼은 8 바이트 정수를 지원하는 컴파일러가 없을 수도 있습니다. 이 경우 시퀀스는 보통 INTEGER 연산을 사용합니다. (이 경우에, 범위는, - 2147483648 에서 +2147483647).

cache 를 1 보다 큰 값으로 설정하고 시퀀스 객체를 여러 세션에서 동시에 사용하면 의외의 결과를 가져올 가능성이 있습니다. 각 세션은 시퀀스 객체에 엑세스하는 동안에 연속 시퀀스 값을 할당하고 캐시합니다. 그리고 캐시 하는 숫자에 따라 시퀀스 객체의 마지막 값을 증가시킵니다. 이 경우, 해당 세션 내에서 NEXTVAL 의 다음 cache -1 사용은 시퀀스 객체의 변경 없이 미리 할당한 시퀀스 값을 반환합니다. 세션에 할당되어 사용되지 않은 시퀀스 번호는 세션이 끝날 때 삭제되기 때문에 결과적으로 시퀀스에 "구멍"이 생깁니다.

더욱이, 여러 세션에서 별도의 시퀀스 값이 할당되는 것이 보장되지만, 모든 세션이 고려될 때, 시퀀스 범위 외의 값이 생성될 수 있습니다. 예를 들면, cache 가 10 인 경우를 생각해 볼 수 있습니다. 세션 A 는 1 에서 10 까지를 확보해 NEXTVAL = 1 을 반환합니다. 세션 B 에서는 세션 A 가 NEXTVAL = 2 를 생성하기 전에 11 에서 20 을 확보하고, NEXTVAL = 11 을 반환합니다. 따라서 cache 를 1 로 설정하면 NEXTVAL 값이 순서대로 생성되는 것에 문제가 없지만, cache 를 1 보다 큰 값으로 설정하면, NEXTVAL 값이 모두 다르게 되고, 순서대로 생성할 수 없습니다. 또한 last value 는 NEXTVAL 의 반환 여부에 관계없이 어느 세션을 통해서나 확보되는 마지막 값입니다.

**예**

101 으로 시작되는 serial 이라는 오름차순 시퀀스를 만듭니다.

```
CREATE SEQUENCE serial START WITH 101;
```

이 시퀀스에서 다음 번호를 선택합니다.

```
SELECT serial.NEXTVAL FROM DUAL;

 nextval
---------
     101
(1 row)
```

NOCACHE 옵션과 함께 supplier_seq 라는 시퀀스를 만듭니다.

```
CREATE SEQUENCE supplier_seq
MINVALUE 1
START WITH 1
INCREMENT BY 1
NOCACHE;
```

이 시퀀스에서 다음 번호를 선택합니다.

```
SELECT supplier_seq.NEXTVAL FROM DUAL;

 nextval
---------
       1
(1 row)
```

**관련 항목**

ALTER SEQUENCE, DROP SEQUENCE

### 3.3.22 CREATE TABLE

**이름**

CREATE TABLE - 새로운 테이블을 정의한다

**개요**

```
CREATE [GLOBAL TEMPORARY] TABLE table_name (
(column_name      data_type [DEFAULT default_expr]
[column_constraint [...]] | table_constraint) [...]
)
[ON COMMIT (PRESERVE ROWS | DELETE ROWS)]
[TABLESPACE tablespace]
```

column_constraint 이 있는 구문입니다.

```
[CONSTRAINT constraint_name]
(NOT NULL |
NULL |
UNIQUE [USING INDEX TABLESPACE tablespace] |
PRIMARY KEY [USING INDEX TABLESPACE tablespace] |
CHECK (expression) |
REFERENCES reftable [(refcolumn)
[ON DELETE act ion])
[DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED |
INITIALLY IMMEDIATE]
```

그리고 table_constraint 구문입니다.

```
[CONSTRAINT constraint_name]
(UNIQUE (column_name [...])
[USING INDEX TABLESPACE tablespace] |
PRIMARY KEY (column_name [...])
[USING INDEX TABLESPACE tablespace] |
CHECK (expression) |
```

FOREIGN KEY (column_name [...])
REFERENCES reftable [(refcolumn [...])]
[ON DELETE action])
[DEFERRABLE | NOT DEFERRABLE]
[INITIALLY DEFERRED | INITIALLY IMMEDIATE]

**설명**

CREATE TABLE 은 현재 데이터베이스에 빈 테이블을 만듭니다. 이 테이블은 이 명령을 실행한 사용자가 소유합니다.

스키마 이름이 주어진 경우 (예 : CREATE TABLE myschema. mytable ...), 테이블은 지정된 스키마에서 생성됩니다. 스키마 이름이 없으면 테이블은 현재 스키마에서 생성됩니다. 또한 특별한 스키마에 존재하는 임시 테이블을 만들 때 스키마 이름이 주어지지 않을 수 있습니다. 테이블 이름은 같은 스키마의 다른 테이블, 시퀀스, 인덱스, 뷰의 이름과 달라야 합니다.

또한 CREATE TABLE 은 테이블의 1 행에 상응하는 복합 데이터형을 자동적으로 생성합니다. 따라서 테이블은 같은 스키마의 기존 데이터형과 같은 이름을 가질 수 없습니다.

테이블은 1600 개 이상의 열을 가질 수 없습니다. (실제로는 튜플 길이 제약 때문에 보유할 수 있는 열 수는 더 적습니다.)

제약 어구는 제약(또는 테스트)을 지정합니다. 삽입, 갱신 작업을 충족시키기 위한 새로운 행 또는 갱신된 행의 제약을 지정합니다. 제약 어구는 선택 사항입니다. 제약은 테이블에서 각종 유효 값 집합을 정의할 때 유용한 SQL 객체입니다.

제약의 정의는 테이블 제약 및 열 제약이라는 2 가지 종류가 있습니다. 열 제약은 열 정의의 일부로 정의됩니다. 테이블 제약 정의는 특정 열과 연관되어 있지 않고 여러 열을 포함할 수 있습니다. 또한 모든 열 제약은 테이블 제약으로 작성할 수 있습니다. 열 제약은 1 개의 열에만 영향을 주는 제약 때문에, 간편한 작성법에 불과합니다.

**매개변수**

GLOBAL TEMPORARY

이 매개변수를 지정하면 테이블은 임시 테이블로 생성됩니다. 임시 테이블은 세션이 끝난 시점에 또는 현재 트랜잭션의 끝 (아래의 ON COMMIT 참조)에 자동으로 삭제됩니다. 임시 테이블이 있으면 이름이 같은 기존의 영구 테이블은 스키마로 수식된 이름으로 참조되지 않는 한, 현재 세션에서는 비가시화 됩니다. 또한 임시 테이블은 현재 세션 밖에서 비가시화 됩니다. (이 임시

테이블의 정보는 Oracle 과 호환되지 않습니다.) 임시 테이블에서 생성되는 인덱스도 모두 자동으로 일시적인 것입니다.

table_name

생성하는 테이블의 이름 (스키마로 수식된 이름도 가능).

column_name

새 테이블에서 만들어진 열의 이름입니다.

data_type

열 데이터형입니다. 여기에는 배열 지정자를 포함할 수 있습니다. Postgres Plus Advanced Server 에서 지원하는 데이터형 정보에 대한 자세한 내용은 제 3.2 장을 참조하십시오.

DEFAULT default_expr

DEFAULT 어구 내에 열 정의를 지정하면 해당 열에 기본 데이터 값이 할당됩니다. 값은 변수가 없는 식입니다. (서브쿼리 및 현재 테이블의 다른 열에 대한 교차 참조는 허용되지 않습니다). 기본식의 데이터형은 열의 데이터형과 일치해야 합니다.

기본식은 모든 삽입 작업에서 해당 열에 값을 지정하지 않은 경우에 사용됩니다. 열에 기본값이 없으면 기본값은 null 입니다.

CONSTRAINT constraint_name

선택적 열 제약 조건, 테이블 제약 조건의 이름입니다. 지정되지 않는 경우 시스템이 이름을 생성합니다.

NOT NULL

해당 열이 null 값을 갖지 않도록 지정합니다.

NULL

해당 열이 null 값을 가질 수를 있도록 지정합니다. 이것이 기본값입니다.

이 어구는 비 표준 SQL 데이터베이스와의 호환성을 위해서만 제공됩니다. 새로운 애플리케이션에서 이것을 사용하는 것은 권장하지 않습니다.

UNIQUE - 열 제약
UNIQUE (column_name [...]) - 테이블 제약

UNIQUE 제약 조건은 테이블의 1 개 이상의 열 또는 그룹이 고유한 값만 갖도록 지정합니다. 고유 테이블 제약 동작은 고유성 열 제약과 같지만 보다 여러 부분에 걸쳐 기능을 가지고 있습니다.

고유 제약 조건에서, null 값은 동일한 것으로 간주되지 않습니다..

각각의 고유한 테이블 제약은 테이블의 다른 고유 제약 조건이나 주 키 제약 조건에 의해 명명된 열 집합은 서로 다른 이름을 지닌 열의 집합을 지정해야 합니다 (같은 이름을 지정하면, 같은 제약이 2 회 열거됩니다).

PRIMARY KEY - 열 제약
PRIMARY KEY (column_name [...]) - 테이블 제약

주 키 제약 조건은 테이블의 1 열 또는 1 이상의 열이 고유한 값 (중복되지 않은 값), 비 null 값만 갖도록 지정합니다. 기술적으로, PRIMARY KEY 는 단순한 UNIQUE 와 NOT NULL 조합입니다. 그러나 주 키는 다른 테이블이 고유한 행 식별자로서 인식하기 때문에, 열 집합을 주 키로 지정하는 것은 스키마 설계에 관한 메타데이터 또한 제공하는 것을 의미합니다.

열 제약 또는 테이블 제약에 관계없이 1 개의 테이블에 사용할 수 있는 주 키는 1 개 뿐입니다.

주 키 제약은 동일한 테이블에 정의된 모든 고유 제약으로, 지정된 열 집합과는 다른 이름으로 열 집합을 지정해야 합니다.

CHECK (expression)

CHECK 절은 삽입 및 갱신 작업을 위한 새로운 행 또는 변경하는 행위를 충족시키는 Boolean 형의 결과를 반환하는 식입니다. 표현식이 true 또는 unknown 이면 성공입니다. 행 삽입, 갱신 작업의 결과, 식이 false 가 될 경우, 에러 예외가 생성되고, 삽입 및 갱신이 데이터베이스를 변경하지 않습니다. 열 제약으로 지정된 check 제약은 해당 열의 값만 참조할 수 있어야 합니다. 한편 테이블 제약에서 나오는 식은 여러 열을 참조할 수 수도 있습니다.

현재, CHECK 식은 서브쿼리와 현재 행의 열 이외의 값을 포함할 수 없습니다.

REFERENCES reftable [(refcolumn)] [ON DELETE action] - 열 제약
FOREIGN KEY (column [...]) REFERENCES reftable [(refcolumn [...])] [ON DELETE actio n] - 테이블 제약

이 어구는 외래 키 제약 조건을 지정합니다. 외래 키 제약 조건은 새 테이블에서 1 개 또는 1 개 이상의 열 집합이 참조된 테이블의 행의 열값과 일치하는 값이 있어야 함을 요구합니다. refcolumn 가 생략되는 경우, reftable 의 주 키를 사용합니다. 참조된 열은 참조된 테이블에서 고유 제약이나 주 키 제약을 가진 열이어야 합니다.

또한, 참조된 열 데이터가 변경된 경우에는 이 테이블의 열 데이터에 문제가 발생합니다. ON DELETE 절은 참조된 테이블의 행이 삭제되면, 실행할 동작을 지정합니다. 제약이 지연될 지라도 참조 동작은 지연될 수 없습니다. 각 어구에 대해 다음 작업을 지정할 수 있습니다.

CASCADE

삭제된 행을 참조하고 있는 부분은 모두 삭제합니다. 또한 참조하는 열의 값을 참조되는 열의 새로운 값으로 갱신합니다.

SET NULL

참조하는 열을 null 로 설정합니다.

참조된 열이 자주 변경되는 경우, 외래 키 열을 인덱스에 추가하고, 외래 키 열과 관련된 참조 동작을 보다 효율적으로 실행할 수 있도록 하는 것이 좋습니다.

DEFERRABLE
NOT DEFERRABLE

제약 조건을 지연시킬 수 있는지 여부를 제어합니다. 지연할 수 없는 제약은 각 명령 후 즉시 검사됩니다. 지연 가능한 제약 검사는( SET CONSTRAINTS 명령을 사용하여) 트랜잭션이 끝날 때까지 지연시킬 수 있습니다. NOT DEFERRABLE 이 기본입니다. 현재, 외래 키 제약만이 이 어구를 허용합니다. 다른 제약은 지연시킬 수 없습니다.

INITIALLY IMMEDIATE
INITIALLY DEFERRED

제약이 지연되는 경우, 이 어구는 제약 검사의 기본 시간을
지정합니다. 제약이 INITIALLY IMMEDIATE 의 경우, 각 문장을 실행한 후에 검사합니다. 이것이 기본입니다. 제약이 INITIALLY DEFERRED 인 경우, 트랜잭션이 끝날 때만 검사됩니다. 제약 검사 시간은 SET CONSTRAINTS 명령을 사용하여 변경할 수 있습니다.

ON COMMIT

ON COMMIT 을 사용하여 트랜잭션 블록의 종료 시점에서 임시 테이블의 동작을 제어할 수 있습니다. 다음 2 가지 옵션이 있습니다.

PRESERVE ROWS

트랜잭션의 종료 시점에서 특별한 동작이 실행되지 않습니다. 이것이 기본 동작입니다. (이것은 Oracle 과 호환되지 않습니다. Oracle 의 기본 동작은 DELETE ROWS 입니다.)

DELETE ROWS

임시 테이블의 모든 행은 각 트랜잭션 블록의 끝에서 삭제됩니다. 기본적으로 커밋 때마다 자동으로 TRUNCATE 가 실행됩니다.

TABLESPACE tablespace

tablespace 는 새 테이블이 만들어지는 테이블스페이스의 이름입니다. 지정되지 않은 경우, default_tablespace 를 사용합니다. default_tablespace 가 공문자열이면, 데이터베이스의 기본 테이블스페이스를 사용합니다.

USING INDEX TABLESPACE tablespace

이 어구를 사용하면, UNIQUE 또는 PRIMARY KEY 제약과 관련된 인덱스를 만들 테이블스페이스를 선택할 수 있습니다. 지정되지 않은 경우, default_tablespace 를 사용합니다. default_tablespace 이 공문자열인 경우, 데이터베이스의 기본 테이블스페이스를 사용합니다.

**주석**

Postgres Plus Advanced Server 는 자동으로 각 고유 제약과 주키 제약에 대한 인덱스를 만들고 그 고유성을 보장합니다. 따라서 주 키 열에 명시적인 인덱스를 생성할 필요가 없습니다 (자세한 내용은 CREATE INDEX 를 참조하십시오).

**예**

dept 테이블과 emp 테이블을 만듭니다.

```
CREATE TABLE dept (
deptno NUMBER (2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
dname VARCHAR2 (14),
loc VARCHAR2 (13)
);
CREATE TABLE emp (
empno NUMBER (4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
ename VARCHAR2 (10),
```

```
job VARCHAR2 (9)
mgr NUMBER (4),
hiredate DATE,
sal NUMBER (7,2),
comm NUMBER (7,2),
deptno NUMBER (2) CONSTRAINT emp_ref_dept_fk
REFERENCES dept (deptno)
);
```

dept 테이블에 고유 테이블 제약을 정의합니다. 고유 테이블 제약은 테이블의 1 개 또는 복수열을 정의할 수 있습니다.

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

검사 열 제약 조건을 정의합니다.

```
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)
);
```

검사 테이블 제약 조건을 정의합니다.

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

```
sal NUMBER (7,2),
comm NUMBER (7,2),
deptno NUMBER (2) CONSTRAINT emp_ref_dept_fk
REFERENCES dept (deptno),
CONSTRAINT new_emp_ck CHECK (ename IS NOT NULL AND empno> 7000)
);
```

jobhist 테이블에 주 키 테이블 제약을 정의합니다. 주 키 테이블 제약은 테이블의 1 개 또는 복수열을 정의할 수 있습니다.

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

다음은 job 열의 기본값으로 리터럴 상수를 지정합니다. 또 hiredate 열의 기본값은 행이 삽입된 날짜입니다.

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

diskvol1 테이블스페이스에 dept 테이블을 만듭니다.

```
CREATE TABLE dept (
```

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

**관련 항목**

ALTER TABLE, DROP TABLE

### 3.3.23 CREATE TABLE AS

**이름**

CREATE TABLE AS - 쿼리 결과에 따라 새로운 테이블을 정의합니다

**개요**

```
CREATE [GLOBAL TEMPORARY] TABLE table_name
[(column_name [...])]
[ON COMMIT (PRESERVE ROWS | DELETE ROWS)]
[TABLESPACE tablespace]
AS query
```

**설명**

CREATE TABLE AS 는 테이블을 만들고 SELECT 명령에 의해 산출된 데이터를 테이블에 저장합니다. 테이블의 열은 SELECT 출력 열과 연관된 이름과 데이터 형을 가지고 있습니다. (단, 새로운 열 이름을 명시한 목록을 전달하면 열 이름을 오버라이드 할 수 있습니다.)

CREATE TABLE AS 는 뷰를 생성하는 것과 비슷하지만 실제로는 전혀 다릅니다. CREATE TABLE AS 는 새로운 테이블을 만들고 새로운 테이블의 내용을 초기화하기 위해 한 번만 쿼리를 평가합니다. 그 이후 쿼리의 원본 테이블에 대한 변경 사항은 새 테이블에 반영되지 않습니다. 반대로, 뷰는 쿼리가 실행될 때마다, 정의된 SELECT 문을 다시 평가합니다.

**매개변수**

GLOBAL TEMPORARY

이것이 지정되면, 테이블은 임시 테이블로 생성됩니다. 자세한
내용은 CREATE TABLE 을 참조하십시오.

table_name

생성할 테이블의 이름 (스키마로 수식된 이름도 가능).

column_name

새 테이블의 열 이름입니다. 열 이름을 지정하지 않으면, 쿼리의 출력 열 이름을 사용합니다.

query

쿼리문입니다. (즉, SELECT 명령입니다.) 사용 가능한 구문 정보는 SELECT 를 참조하십시오.

**관련 항목**

CREATE TABLE , SELECT

### 3.3.24 CREATE TRIGGER

**이름**

CREATE TRIGGER - 새로운 트리거를 정의합니다

**개요**

```
CREATE [OR REPLACE] TRIGGER name
(BEFORE | AFTER)
(INSERT | UPDATE | DELETE)
[OR (INSERT | UPDATE | DELETE)] [...]
ON table
[FOR EACH ROW]
[DECLARE
declaration; [...]
BEGIN
statement; [...]
[EXCEPTION
(WHEN exception [OR exception] [...] THEN
```

```
        statement; [...]) [...]
]
END
```

**설명**

CREATE TRIGGER 는 새로운 트리거를 정의합니다. CREATE OR REPLACE TRIGGER 은 새로운 트리거를 만들거나 기존의 트리거를 대체합니다.

CREATE OR REPLACE TRIGGER 를 사용하여 기존의 트리거의 정의를 변경하려는 목적이 없는 한, 새로운 트리거 이름은 동일한 테이블에 정의된 기존 트리거의 이름과 일치하지 않아야 합니다.

트리거는 트리거링 이벤트가 정의된 테이블과 같은 스키마에서 생성됩니다.

트리거에 대한 자세한 정보는 5 장을 참조하십시오.

**매개변수**

name

생성하는 트리거의 이름입니다.

BEFORE | AFTER

트리거를 트리거링 이벤트 실행 전에 또는 후에 발행 할 것인지 결정합니다.

INSERT | UPDATE | DELETE

트리거링 이벤트를 정의합니다.

table

트리거링 이벤트가 발생하는 테이블의 이름입니다.

FOR EACH ROW

이 매개변수는 트리거 프로시저를 트리거링 이벤트에 의해 영향을 받는 각 행에 대해 1 회 발행하거나 또는 SQL 문장마다 1 회 발행할지 여부를 지정합니다. 지정된 경우, 트리거는 영향을 받는 각 행에 대해 1 회 발생합니다. (낮은 레벨의 트리거) 지정하지 않는 경우, 트리거는 SQL 문장마다 1 회 발행합니다.

declaration

변수, 형, R EF CURSOR 선언입니다.

statement

SPL 프로그램 문장입니다. DECLARE - BEGIN - END 블록은 SPL 문장으로 간주된다는 점에 유의하시기 바랍니다. 따라서 트리거 내부에 중첩된 블록이 있을 수 있습니다.

exception

NO_DATA_FOUND, OTHERS 등 예외 상황의 이름입니다.

**예**

다음은 SQL 문장 단위로 실행하는 트리거입니다. 트리거링 문장이 실행된 후에 트리거가 발행됩니다. (emp 테이블에 INSERT, UPDATE, DELETE 가 실행될 때마다 발행합니다.)

```
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;
```

다음은 낮은 레벨의 트리거입니다. emp 테이블의 각 행에 INSERT, UPDATE, DELETE 를 실행하기 전에 트리거가 발행됩니다.

```
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;
```

**관련 항목**

DROP TRIGGER

### 3.3.25 CREATE USER

**이름**

CREATE USER - 새로운 데이터베이스 사용자 계정을 정의합니다

**개요**

```
CREATE USER name IDENTIFIED BY password
```

**설명**

CREATE USER 는 Postgres Plus Advanced Server 데이터베이스 클러스터에 새로운 사용자를 추가합니다. CREATE USER 명령을 사용하려면 데이터베이스의 슈퍼유저여야 합니다.

CREATE USER 를 사용하면 새 사용자와 동일한 이름의 스키마를 만듭니다. 새로 만든 사용자가 스키마의 소유자입니다. 사용자가 자격이 되지 않는 이름을 가진 객체를 만들 때, 객체가 이 스키마에서 생성됩니다.

**매개변수**

name

사용자의 이름입니다.

password

사용자의 패스워드를 설정합니다. 패스워드는 ALTER USER 를 사용하여 나중에 변경할 수 있습니다.

**주석**

사용자 이름과 패스워드는 최대 63 자까지 입니다.

**예**

john 이라는 사용자를 만듭니다.

```
CREATE USER john IDENTIFIED BY abc;
```

**관련 항목**

ALTER USER, DROP USER

### 3.3.26 CREATE VIEW

**이름**

CREATE VIEW - 새로운 뷰를 정의합니다

**개요**

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

**설명**

CREATE VIEW 는 쿼리에 따른 뷰를 정의합니다. 뷰는 실체로 존재하는 것은 아닙니다. 대신, 쿼리에서 뷰를 참조 때마다 지정된 쿼리가 실행됩니다.

CREATE OR RE PLACE VIEW 도 비슷하지만, 같은 이름을 가진 뷰가 이미 존재하는 경우, 해당 뷰를 대체합니다.

스키마 이름이 주어진 경우 (예 : CREATE VIEW myschema.myview ...), 뷰는 지정된 스키마에서 생성됩니다. 스키마 이름이 주어지지 않은 경우, 뷰는 현재 스키마에서 생성됩니다. 뷰 이름은 동일한 스키마의 다른 뷰, 테이블, 시퀀스, 인덱스와 다른 이름이어야 합니다.

**매개변수**

name

생성하는 뷰의 이름 (스키마로 수식된 이름도 가능).

column_name

뷰의 열 이름으로 사용하는 이름의 목록입니다. 이 매개변수는 선택 사항입니다. 생략되는 경우, 쿼리에서 유래한 이름이 사용됩니다.

query

뷰의 열과 행을 제공하는 SELECT 문장입니다.

유효한 쿼리구문에 대한 자세한 정보는 SELECT 를 참조하십시오.

**주석**

현재, 뷰는 읽기 전용입니다. 시스템은 뷰에 삽입, 갱신, 삭제를 허용하지 않습니다. 그러나 뷰에 삽입 등의 다른 테이블을 처리하는데 갱신 룰을 만드는 것으로, 갱신할 수 있는 뷰와 같은 효과를 얻을 수 있습니다. 자세한 내용은 Postgres Plus documentation set 의 CREATE RULE 명령을 참조하십시오.

뷰가 참조하는 테이블에 액세스할 수 있는지 여부는 뷰 소유자의 권한으로 결정됩니다. 그러나 뷰에서 호출되는 함수는 뷰를 사용하여 쿼리에서 직접 호출되는 경우와 마찬가지로 취급됩니다. 따라서 뷰 사용자는 뷰에서 사용되는 모든 함수를 호출할 권한이 있어야 합니다.

**예**

deptno 30 의 모든 직원으로 구성된 뷰를 만듭니다.

```
CREATE VIEW dept_30 AS SELECT * FROM emp WHERE deptno = 30;
```

**관련 항목**

DROP VIEW

### 3.3.27 DELETE

**이름**

DELETE - 테이블에서 행을 삭제합니다

**개요**

```
DELETE [optimizer_hint] FROM tabl e [@ dblink]
[WHERE condition]
[RETURNING return_expression [...]
(INTO (record | variable [...])
| BULK COLLECT INTO collection [...])]
```

**설명**

DELETE 는 지정된 테이블에서 WHERE 절을 충족시키는 행을 제거합니다. WHERE 이 없으면, 지정한 테이블의 모든 행을 삭제할 수 있습니다. 따라서 결과는 유효하지만, 내용이 없는 빈 테이블이 남게 됩니다.

주석 : TRUNCATE 명령은 테이블에서 모든 행을 제거하는 보다 빠른 매커니즘을 제공합니다.

DELETE 명령이 SPL 프로그램에서 사용되는 경우에만 RETURNING INTO (record | variable [...]) 어구를 지정할 수 있습니다. 또한 DELETE 명령의 결과는 1 행 이상을 포함해서는 안됩니다. 결과가 2 줄 이상의 경우는 예외가 발생합니다. 결과가 비어 있으면, 대상 레코드 혹은 변수는 null 이 할당됩니다.

DELETE 명령이 SPL 프로그램에서 사용되는 경우에만 RETURNING BULK COLLECT INTO collection [...] 어구를 지정할 수 있습니다. BULK COLLECT INTO 어구를 대상으로 여러 collection 이 지정된

경우, 각 collection 은 단일의 스칼라 필드로 구성 되어야 합니다. 즉, collection 가 레코드 이여서는 안됩니다. DELETE 명령의 결과 행이 완전히 제거되지 않거나 여러 행을 삭제됩니다. 결과의 각 행에 대해 평가되는 return_expression 는 첫 번째 요소로 시작하는 collection 의 요소입니다. collection 에 있는 기존 행은 삭제됩니다. 결과가 비어 있으면, collection 은 비어있습니다.

삭제를 수행하려면 해당 테이블에 대한 DELETE 권한이 있어야 합니다. 또한, condition 에 대한 값을 읽기 위해 해당 값이 들어있는 모든 테이블에 대한 SELECT 권한이 있어야 합니다.

**매개변수**

optimizer_hint

최적화 프로그램의 실행 계획을 선택할 때 사용하는 주석에 있는 힌트입니다. 최적화 힌트에 대한 자세한 내용은 3.4 장을 참조하십시오.

table

기존 테이블 이름 (스키마로 수식된 이름도 가능).

dblink

원격 데이터베이스를 식별하는 데이터베이스 링크의 이름입니다. 데이터베이스 링크에 대한 자세한 내용은 CREATE DATABASE LINK 명령을 참조하십시오.

conditi on

삭제여부를 결정하는 BOOLEAN 값을 반환하는 식입니다.

return_expression

table 1 개 이상의 열이 포함된 식입니다. return_expression 에서 table 의 열 이름이 지정된 경우, return_expression 평가 후 열에 할당되는 값은 삭제된 행의 값입니다.

record

return_expression 으로 평가한 결과가 할당되는 레코드입니다. return_expression 의 첫 번째 결과가 record 에서 첫 번째 필드에 할당됩니다. 2 번째 결과가 2 번째 필드에 할당됩니다. 이하 동일합니다. record 의 필드 수는 식의 수와 동일해야 합니다. 또한 필드는 할당된 식과 호환성 있는 형이어야 합니다.

variable

return_expression 으로 평가한 결과가 할당되는 변수입니다. 1 개 이상의 return_expression 과 variable 이 지정되면, return_expression 의 첫 번째 결과가 첫 번째 variable 에 할당됩니다. 2 번째 결과가 2 번째 variable 에 할당됩니다. 이하 동일합니다. INTO 에 이어 지정된 변수의 수는 RETURNING 에 이어 지정 표현식의 수와 동일해야 합니다. 또한 변수는 할당된 식과 호환성 있는 형이어야 합니다.

collection

return_expression 으로 평가한 결과에서 생성된 요소의 collection 입니다. 하나의 collection 은 단일 필드의 collection 일 수도 있고, 또는 레코드형의 collection 일 수 있습니다. 또는 각 collection 이 단일 필드로 구성된 여러 collection 일지도 모릅니다. 반환된 식의 수는 모두 지정된 collection 의 필드 수와 동일해야 합니다. 또한 해당 return_expre ssion 과 collection 필드는 형태 호환성이 있어야 합니다.

**예**

Jobhist 테이블에서 empno 가 7900 직원을 모두 삭제합니다.

```
DELETE FROM jobhist WHERE empno = 7900;
```

jobhist 테이블을 비웁니다.

```
DELETE FROM jobhist;
```

**관련 항목**

TRUNCATE

### 3.3.28 DROP DATABASE LINK

**이름**

DROP DATABASE LINK - 데이터베이스 링크를 삭제합니다

**개요**

```
DROP [PUBLIC] DATABASE LINK name
```

**설명**

DROP DATABASE LINK 는 기존 데이터베이스 링크를 삭제합니다. 이 명령을 실행하려면, 슈퍼 사용자 또는 데이터베이스 링크의 소유자여야 합니다.

**매개변수**

name

삭제되는 데이터베이스 링크의 이름입니다.

PUBLIC

name 이 public 데이터베이스 링크에 있다는 것을 나타냅니다.

**예**

oralink 라는 public 데이터베이스 링크를 삭제합니다.

```
DROP PUBLIC DATABASE LINK oralink;
```

edblink 라는 private 데이터베이스 링크를 삭제합니다.

```
DROP DATABASE LINK edblink;
```

**관련 항목**

CREATE DATABASE LINK

### 3.3.29 DROP FUNCTION

**이름**

DROP FUNCTION - 함수를 삭제합니다

**개요**

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

**설명**

DROP FUNCTION 은 기존의 함수 정의를 삭제합니다. 이 명령을 수행하려면, 슈퍼유저 또는 함수의 소유자여야 합니다. 인수가 적어도 1 개 존재하는 경우, 함수의 인수형은 반드시 지정해야 합니다. (이 요구 사항은 Oracle 과 호환되지 않습니다. Postgres Plus Advanced Server 는 함수명의 오버로딩을 허용하므로 Postgres Plus Advanced Server 의 DROP FUNCTION 명령은 함수 서명을 요청합니다.)

**매개변수**

name

기존 함수의 이름(스키마로 수식된 이름도 가능)입니다.

type

함수의 인수 데이터형입니다.

**예**

다음의 명령은 emp_comp 함수를 삭제합니다.

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

**관련 항목**

CREATE FUNCTION

### 3.3.30 DROP INDEX

**이름**

DROP INDEX - 인덱스를 삭제합니다

**개요**

```
DROP INDEX name
```

**설명**

DROP INDEX 는 데이터베이스 시스템에서 기존 인덱스를 삭제합니다. 이 명령을 수행하려면 슈퍼유저 또는 해당 인덱스의 소유자여야 합니다. 해당 인덱스에 의존하는 객체가 있는 경우, 에러가 발생하고 인덱스는 삭제되지 않습니다.

**매개변수**

name

삭제할 인덱스의 이름 (스키마로 수식된 이름도 가능).

**예**

다음은 name_idx 인덱스를 삭제합니다.

```
DROP INDEX name_idx;
```

**관련 항목**

ALTER INDEX, CREATE INDEX

### 3.3.31 DROP PACKAGE

**이름**

DROP PACKAGE - 패키지를 제거한다

**개요**

```
DROP PACKAGE [BODY] name
```

**설명**

DROP PACKAGE 는 기존 패키지를 제거합니다. 이 명령을 수행하려면 슈퍼유저 또는 패키지의 소유자여야 합니다. BODY 가 지정된 경우, 패키지 본문만 삭제됩니다. 패키지 사양은 삭제되지 않습니다. BODY 가 생략되면 패키지 사양과 본문 모두 삭제됩니다.

**매개변수**

name

삭제할 패키지의 이름 (스키마로 수식된 이름도 가능).

**예**

이 명령은 emp_admin 패키지를 삭제합니다.

```
DROP PACKAGE emp_admin;
```

**관련 항목**

CREATE PACKAGE, CREATE PACKAGE BODY

### 3.3.32 DROP PROCEDURE

**이름**

DROP PROCEDURE - 프로시저를 삭제합니다

**개요**

```
DROP PROCEDURE name
```

**설명**

DROP PROCEDURE 는 기존 프로시저를 삭제합니다. 이 명령을 수행하려면 슈퍼유저 또는 그 프로시저의 소유자여야 합니다.

**매개변수**

name

기존 프로시저의 이름(스키마로 수식된 이름도 가능)입니다.

**예**

이 명령은 select_emp 프로시저를 삭제합니다.

```
DROP PROCEDURE select_emp;
```

**관련 항목**

CREATE PROCEDURE

### 3.3.33 DROP PUBLIC SYNONYM

**이름**

DR OP PUBLIC SYNONYM - public 동의어를 삭제합니다

**개요**

```
DROP PUBLIC SYNONYM name
```

**설명**

DROP PUBLIC SYNONYM 은 기존 public 동의어를 삭제합니다. 이 명령을 수행하려면 슈퍼유저 또는 public 동의어의 소유자여야 합니다.

public 동의어에 대한 자세한 내용은 제 2.2.4 장을 참조하십시오.

**매개변수**

name

제거할 public 동의어의 이름입니다.

**예**

이 명령은 personnel 이라는 public 동의어를 삭제합니다.

```
DROP PUBLIC SYNONYM personnel;
```

**관련 항목**

CREATE PUBLIC SYNONYM

### 3.3.34 DROP ROLE

**이름**

DROP ROLE - 데이터베이스 역할을 제거합니다

**개요**

DROP ROLE name [CASCADE]

**설명**

DROP ROLE 은 지정된 역할을 삭제합니다. 슈퍼유저 역할을 제거하려면, 자신도 슈퍼유저여야 합니다. 슈퍼유저 이외의 역할을 제거하려면 CREATEROLE 권한을 가져야 합니다.

임의의 데이터베이스 클러스터에서 참조하는 경우, 역할을 삭제할 수 없습니다. 삭제하려고 하면, 에러가 발생합니다. 역할을 삭제하기 전에, 해당 역할이 소유한 모든 객체를 삭제(또는 소유권 변경)해야 합니다. 또 그 역할에 부여된 권한도 거론해야 합니다.

하지만 역할에 포함된 역할 구성원 자격을 삭제할 필요는 없습니다. DROP ROLE 은 자동으로 다른 역할 내에 있는 대상 역할과 대상 역할에 있는 다른 역할의 구성원 자격을 취소합니다. 다른 역할이 삭제되지도 않고 달리 영향을 받지도 않습니다.

역할이 소유하는, 역할과 동일한 이름의 스키마에 역할이 소유한 객체 밖에 없는 경우, CASCADE 옵션을 지정할 수 있습니다. 이 경우 DROP ROLE name CASCADE 명령을 실행할 수 있는 것은 슈퍼유저뿐입니다. 이 명령을 사용하면 특정 역할 이외에, 역할과 동일한 이름의 스키마와 스키마에 있는 모든 객체를 삭제할 수 있습니다.

**매개변수**

name

삭제할 역할의 이름입니다.

CASCADE

이 매개변수가 지정된 경우, 역할이 소유하는, 역할과 동일한 이름의 스키마를 제거합니다. (스키마에 속해있는, 역할이 소유하는 객체가 모두 삭제합니다.) 지정할 수 있는 역할 또는 스키마에 다른 종속 관계가 없는 경우입니다.

**예**

역할을 삭제합니다.

```
DROP ROLE admins;
```

**관련 항목**

ALTER ROLE, CREATE ROLE, SET ROLE

### 3.3.35 DROP SEQUENCE

**이름**

DROP SEQUENCE - 시퀀스를 제거합니다

**개요**

```
DROP SEQUENCE    name [...]
```

**설명**

DROP SEQUENCE 는 시퀀스 번호 생성기를 삭제합니다. 이 명령을 수행하려면 슈퍼유저 또는 시퀀스의 소유자여야 합니다.

**매개변수**

name

시퀀스의 이름 (스키마로 수식된 이름도 가능).

**예**

Serial 이라는 시퀀스를 삭제합니다.

```
DROP SEQUENCE serial;
```

**관련 항목**

ALTER SEQUENCE, CREATE SEQUENCE

### 3.3.36 DROP TABLE

**이름**

DROP TABLE - 테이블을 삭제합니다

**개요**

DROP TABLE name

**설명**

DROP TABLE 은 데이터베이스에서 테이블을 삭제합니다. 테이블 소유자만이 테이블을 삭제할 수 있습니다. 테이블을 삭제하지 않고 테이블 행을 비우려면, DELETE 를 이용하세요.

DROP TABLE 은 삭제되는 테이블에 있는 인덱스, 룰, 트리거, 제약을 삭제합니다.

**매개변수**

name

삭제할 테이블의 이름 (스키마로 수식된 이름도 가능).

**예**

emp 테이블을 삭제합니다.

```
DROP TABLE emp;
```

**관련 항목**

ALTER TABLE, CREATE TABLE

## .3.37 DROP TABLESPACE

**이름**

DROP TABLESPACE - 테이블스페이스를 삭제합니다

**개요**

DROP TABLESPACE tablespacename

**설명**

DROP TABLESPACE 시스템에서 테이블스페이스를 삭제합니다.

테이블스페이스의 소유자 또는 슈퍼유저만이 테이블스페이스를 삭제할 수 있습니다. 테이블스페이스를 삭제하기 전에 모든 데이터베이스 객체가 비어 있어야 합니다. 현재 데이터베이스 내의 객체가 테이블스페이스를 사용하지 않았다 하더라도, 다른 데이터베이스의 객체가 테이블스페이스에 있을 수 있습니다.

**매개변수**

tablespacename

테이블스페이스의 이름입니다.

**예**

employee_space 테이블스페이스를 시스템에서 삭제합니다.

```
DROP TABLESPACE employee_space;
```

**관련 항목**

ALTER TABLESPACE

### 3.3.38 DROP TRIGGER

**이름**

DROP TRIGGER - 트리거를 삭제합니다

**개요**

```
DROP TRIGGER name
```

**설명**

DROP TRIGGER 는 트리거 관련 테이블에서 트리거를 삭제합니다. 이 명령을 수행하려면 슈퍼유저 또는 트리거가 정의된 테이블의 소유자여야 합니다.

**매개변수**

name

삭제되는 트리거의 이름입니다.

**예**

emp_sal_trig 트리거를 삭제합니다.

```
DROP TRIGGER emp_sal_trig;
```

**관련 항목**

CREATE TRIGGER

### 3.3.39 DROP USER

**이름**

DROP USER – 데이터베이스 사용자 계정을 삭제합니다

**개요**

```
DROP USER name [CASCADE]
```

**설명**

DROP USER 는 지정된 사용자를 삭제합니다. 슈퍼유저만이 슈퍼유저를 삭제 할 수 있습니다. 슈퍼유저 이외의 사용자를 삭제하려면, CREATEROLE 권한을 가져야 합니다.

데이터베이스 클러스터 중 하나에서 참조하는 경우 사용자를 삭제할 수 없습니다. 삭제하려고 하면 에러가 발생합니다. 사용자를 삭제하기 전에, 사용자가 소유한 모든 객체를 삭제 (또는 소유권 변경)해야 합니다. 또한 사용자에게 부여한 권한도 취소해야 합니다.

그러나 사용자를 포함한 역할 구성원 자격을 삭제할 필요는 없습니다. DROP USER 는 자동으로 다른 역할 내에 있는 대상 역할과 대상 역할에 있는 다른 역할의 구성원 자격을 취소합니다. 다른 역할이 삭제되지도 않고 달리 영향을 받지도 않습니다.

사용자가 소유한, 사용자와 동일한 이름의 스키마에 사용자가 소유한 객체 밖에 없는 경우 CASCADE 옵션을 지정할 수 있습니다. 이 경우, DROP USER name CASCADE 명령을 실행할 수 있는 것은 슈퍼유저뿐입니다. 이 명령을 사용하면 지정된 사용자 이외에, 사용자와 동일한 이름의 스키마, 스키마에 있는 모든 객체를 삭제할 수 있습니다.

**매개변수**

name

삭제할 사용자의 이름입니다.

CASCADE

이 매개변수를 지정하면, 사용자가 소유하는 사용자와 동일한 이름의 스키마를 삭제합니다. (스키마에 속하는 사용자가 소유하는 객체를 모두 삭제합니다.) 지정할 수 있는 사용자 또는 스키마에 다른 종속 관계가 없는 경우입니다.

**예**

객체를 소유하지 않고, 다른 객체에 대한 권한도 부여되지 않은 사용자 계정을 삭제합니다.

```
DROP USER john;
```

john 이라는 사용자 계정을 삭제합니다. 이 사용자는 자신이 소유한 john 이라는 스키마 외부 객체에 대한 권한이 부여되지 않으며, 외부 객체를 소유하고 있지 않습니다.

```
DROP USER john CASCADE;
```

**관련 항목**

ALTER USER, CREATE USER

### 3.3.40 DROP VIEW

**이름**

DROP VIEW - 뷰를 삭제합니다

**개요**

```
DROP VIEW name
```

**설명**

DROP VIEW 는 기존의 뷰를 삭제합니다. 이 명령을 수행하려면, 뷰의 소유자여야 합니다. 다른 객체가 지정된 뷰에 의존하는 경우 (뷰의 뷰와 같은 경우), 뷰가 삭제되지 않습니다.

**매개변수**

name

삭제할 뷰의 이름 (스키마로 수식된 이름도 가능).

**예**

다음은 dept_30 라는 뷰를 삭제합니다.

```
DROP VIEW dept_30;
```

**관련 항목**

CREATE VIEW

### 3.3.41 GRANT

**이름**

GRANT - 액세스 권한을 정의합니다

**개요**

```
GRANT ((SELECT | INSERT | UPDATE | DELETE | REFERENCES)
[,...] | ALL [PRIVILEGES])
ON tablename
TO (username | groupname | PUBLIC) [...]
[WITH GRANT OPTION]

GRANT (SELECT | ALL [PRIVILEGES])
ON sequencename
TO (username | groupname | PUBLIC) [...]
[WITH GRANT OPTION]

GRANT (EXECUTE | ALL [PRIVILEGES])
ON FUNCTION progname
([[argmode] [argname] argtype] [...])
TO (username | groupname | PUBLIC) [...]
[WITH GRANT OPTION]

GRANT (EXECUTE | ALL [PRIVILEGES])
```

```
ON PROCEDURE progname
[([[argmode] [argname] argtype] [...])]
TO (username | groupname | PUBLIC) [...]
[WITH GRANT OPTION]

GRANT (EXECUTE | ALL [PRIVILEGES])
ON PACKAGE packagename
TO (username | groupname | PUBLIC) [...]
[WITH GRANT OPTION]

GRANT role [...]
TO (username | groupname | PUBLIC) [...]
[WITH ADMIN OPTION]

GRANT (CONNECT | RESOURCE | DBA) [...]
TO (username | groupname      ) [...]
[WITH ADMIN OPTION]
```

**설명**

GRANT 명령은 기본적으로 2 가지 종류가 있습니다. 1 개는 데이터베이스 객체 (테이블, 뷰, 시퀀스, 프로그램)에 대한 권한 부여, 다른 1 개는 역할의 구성원 자격을 부여합니다. 이들은 많은 점에서 유사하지만, 설명은 별도로 설명될 만큼 차이가 있습니다.

Postgres Plus Advanced Server 에서는 사용자 및 그룹과 같은 개념은 역할이라는 단일 종류의 엔티티로 통합되었습니다. 이 본문에서, 사용자는 LOGIN 속성을 가진 역할입니다. 이 역할은 세션을 생성하고, 애플리케이션에 연결하는 데 사용됩니다. 그룹은 LOGIN 속성이 없는 역할입니다. 이곳은 세션 생성 및 애플리케이션에 연결하는 데 사용되지 않습니다.

역할은 다른 여러 역할의 구성원이 될 수 있습니다. 그런 의미에서, 그룹에 속하는 사용자의 전형적인 개념은 아직 유효합니다. 그러나, 현재는 사용자와 그룹이 일반화되어, 사용자는 사용자에 속할 수도, 그룹은 그룹에 속할 수도, 그룹이 사용자에 속할 수도 있습니다. 즉, 역할의 일반화된 다 계층 구조를 형성할 수 있습니다. 사용자 이름과 그룹 이름이 동일한 네임 스페이스에 공유됩니다. 따라서 GRANT 명령은 사용자와 그룹을 구별할 필요가 없습니다.

### 3.3.41.1 데이터베이스 객체에 대한 GRANT

GRANT 명령은 데이터베이스 객체에 대한 특정 권한을 역할에 부여하기도 합니다. 이미 권한이 다른 역할에 부여된 경우에도 추가로 부여됩니다.

PUBLIC 키워드는 앞으로 생성될 역할을 포함한 모든 역할에 대한 권한을 나타냅니다. PUBLIC 은 모든 역할을 포함한 암시적으로 정의된 그룹이라 생각할 수 있습니다. private 적인 역할은 모두 역할에 직접 부여된 권한, 역할이 현재 속한 역할에 부여된 권한, 그리고 PUBLIC 으로 부여된 권한을 합친 권한을 가지고 있습니다.

WITH GRANT OPTION 이 지정되면, 권한 수령자인 경우, 다른 사람에게 그 권한을 부여할 수 있습니다. 그랜트 옵션이 없는 경우, 받는 사람은 이것을 할 수 없습니다. 그랜트 옵션은 PUBLIC 에 부여할 수 없습니다.

객체의 소유자 (일반적으로 객체를 만든 사용자)는 기본적으로 모든 권한이 있기 때문에 소유자에게 권한을 허용할 필요는 없습니다 (단, 객체의 소유자가 안전을 위해 자신의 권한을 취소할 수 있습니다). 객체를 삭제하는 권리나 어떤 방법으로 객체 정의를 수정할 권한은 부여 가능한 권한이 아닙니다. 이러한 권한은 소유자의 고유한 것이며 권한을 주거나 취소하는 것은 불가능합니다. 소유자는 객체에 대한 그랜트 옵션을 암시적으로 보유하고 있습니다.

객체의 종류에 따라, 기본 권한으로, 처음부터 어떤 권한이 PUBLIC 으로 부여되는 경우가 있습니다. 기본적으로 테이블에 액세스할 수 없습니다. 또한, 함수, 프로시저, 패키지는 기본 권한으로 EXECUTE 권한이 부여됩니다. 물론 객체의 소유자는 이 권한을 취소할 수 있습니다 (최대의 안전을 위해, REVOKE 명령은 객체를 생성하는 같은 트랜잭션에서 발행합니다. 이렇게 하면 다른 사용자가 이 객체를 사용할 윈도우가 없습니다.)

구성 가능한 권한은 다음과 같습니다.

SELECT

지정한 테이블, 뷰, 시퀀스의 모든 열에 대한 SELECT 를 허용합니다. 시퀀스에 대해, 이 권한은 currval 함수의 사용을 허용합니다.

INSERT

지정한 테이블에 새 행의 INSERT 를 허용합니다.

UPDATE

지정한 테이블의 모든 열에 대한 UPDATE 를 허용합니다. SELECT ... FOR UPDATE 도 (SELECT 권한 외에) 이 권한을 필요로 합니다.

DELETE

지정한 테이블에서 행 DELETE 를 허용합니다.

REFERENCES

외래키 제약 조건을 만들려면, 참조하는 쪽과 참조되는 쪽 양쪽 테이블에 대해 이 권한을 갖고 있고 있어야 합니다.

EXECUTE

지정된 패키지 프로시저, 함수의 사용을 허용합니다. 패키지에 적용할 경우, 패키지에 포함된 모든 public 프로시저, public 함수, public 변수, 레코드, 커서 및 기타 public 객체와 객체의 사용을 허용합니다. EXECUTE 는 함수, 프로시저, 패키지에 적용할 수 있는 유일한 권한입니다.

Postgres Plus Advanced Server 에서 EXECUTE 사용 권한을 부여하는 구문은 완전히 Oracle 과 호환되지 않습니다. Postgres Plus Advanced Server 에서 FUNCTION, PROCEDURE, PACKAGE 중 하나를 프로그램 이름 앞에 붙여야 합니다. 한편, Oracle 에서는 이것을 생략해야 합니다. 또한 Postgres Plus Advanced Server 에서는 함수 이름 다음에 전체적인 함수 서명 (함수의 인수가 없으면 빈 괄호 포함)을 포함 해야 합니다. 프로시저에 있어서는 1 개 이상 인수를 가진 경우 전체 서명이 필요합니다. Oracle 에서는 함수와 프로시저 서명을 생략해야 합니다. 이것은 Oracle 에서는 모든 프로그램에서 동일한 네임 스페이스를 공유하기 때문입니다. 한편 Postgres Plus Advanced Server 에서 특정한 확장을 위해 프로그램 이름의 오버로딩이 가능하기 때문에, 함수, 프로시저, 패키지가 각각의 네임 스페이스를 갖습니다.

ALL PRIVILEGES

사용 가능한 모든 권한을 동시에 부여합니다.

기타 명령을 실행하는 데 필요한 권한은 해당 명령 참조 페이지에 기재되어 있습니다.

### 3.3.41.2 역할에 대한 GRANT

또 다른 GRANT 명령은 역할에 있는 구성원 자격을 1 개 이상의 역할에 부여합니다. 이에 따라 역할에 부여된 권한을 각 구성원에게 전달하므로, 역할에 있는 구성원 자격은 중요합니다.

WITH ADMIN OPTION 이 지정된 경우, 구성원은 역할의 구성원 자격을 다른 사람에게 부여할 수 있습니다. 또한 마찬가지로 역할에서 구성원 자격을 박탈하는 것도 가능합니다. admin option 없이, 일반 사용자는 다른 사람에 대한 권한 부여 및 박탈을 할 수 없습니다. 그러나 데이터베이스 슈퍼유저는 모든 역할의 구성원 자격을 누구나 부여하거나 박탈할 수 있습니다. CREATEROLE 권한을 가진 역할은 슈퍼유저가 아닌 역할의 구성원 자격 부여, 박탈이 가능합니다.

다음은 미리 정의된 3 가지 역할입니다.

CONNECT

CONNECT 역할을 부여하는 것은 LOGIN 권한을 부여하는 것과 동일합니다. 부여자는 CREATEROLE 권한이 있어야 합니다.

RESOURCE

RESOURCE 역할을 부여하는 것은 부여 받는 사람의 이름과 같은 이름의 스키마에 대해 CREATE 권한과 USAGE 권한을 부여하는 것과 동일합니다. 이 스키마는 명령을 실행하기 전에 존재해야 합니다. 부여자는 스키마에 대해 CREATE 권한과 USAGE 권한을 부여할 수 있는 권한이 있어야 합니다.

DBA

DBA 역할을 부여하는 것은 받는 사람을 슈퍼유저로 만드는 것과 동일합니다. 부여자는 슈퍼유저여야 합니다.

**주석**

권한을 취소하려면 REVOKE 명령을 사용합니다.

객체의 소유자도 아니고 객체에 대한 권한이 없는 사용자가 객체의 권한을 GRANT 하려고 해도 명령은 즉각 실패합니다. 어떤 권한을 갖고 있는 한, 명령은 진행되지만 부여할 수 있는 권한은 사용자가 가진 그랜트 옵션에 대한 권한입니다. 그랜트 옵션이 없는 경우, GRANT ALL PRIVILEGES 구문은 경고 메시지를 발행합니다. 한편 다른 구문은 명령에서 이름을 지정한 권한에 대한 그랜트 옵션이 없는 경우, 경고 메시지를 발행합니다 (대체로 여기까지의 설명은 객체의 소유자에 대해서도 마찬가지로 적용됩니다, 그러나 소유자는 항상 전체 그랜트 옵션을 보유하고 있는 것으로 취급하기 때문에 이런 상황은 결코 발생하지 않습니다).

데이터베이스 슈퍼유저만이 객체에 대한 권한 설정에 관계없이 모든 객체에 접근할 수 있다는 것에 주의해야 합니다. 슈퍼유저가 갖는 권한은 Unix 시스템의 root 권한과 비슷합니다. root 뿐만 아니라 절대적으로 필요한 경우가 아니면, 슈퍼유저로 수행하지 않는 것이 현명합니다.

슈퍼유저가 GRANT 와 REVOKE 발행을 선택하면, 명령은 해당 객체의 소유자가 발행한 것처럼 실행됩니다. 특히 이 같은 명령에 대한 권한은 객체의 소유자에 의해 부여된 것으로 나타납니다. (역할의 구성원 자격에 대해, 구성원 자격이 포함된 역할 자체가 부여된 것으로 나타납니다.)

GRANT 및 REVOKE 은 영향을 주는 객체의 소유자가 아닌 다른 역할을 통해 수행할 수 있지만, 객체를 소유하는 역할의 구성원이거나 해당 객체에 대해 WITH GRANT OPTION 권한을 가진 역할의 구성원인 경우, 이를 수행할 수 있습니다. 이 경우 해당 권한은 객체의 실제 소유자 역할이나 WITH GRANT OPTION 권한을 가진 역할에 의해 부여된 것으로 기록됩니다. 예를 들면, t1 테이블이 g1 역할에 의해 소유되고 u1 이 g1 역할의 구성원이 라고 가정합니다. 이 경우 u1 은 t1 의 권한을 u2 에게 부여할 수 있습니다. 그러나, 이러한 권한은 g1 에 의해 직접적으로 부여되는 것으로 나타납니다. 나중에 g1 역할의 다른 구성원이 권한을 취소할 수도 있습니다.

GRANT 를 수행한 역할이 역할의 여러 구성원 자격 경로를 통해 간접적으로 필요한 권한을 가진 경우, 어떤 역할이 권한을 부여하는 역할로 기록되는지의 내용은 지정되지 않습니다. 이럴 경우 SET ROLE 을 사용하여 GRANT 를 수행하고자 하는 역할이 되는 것이 좋습니다.

현재로서는 Postgres Plus Advanced Server 는 테이블의 각 열에 대해 권한을 부여하거나 취소 할 수 없습니다. 이를 해결하려면 대상으로 하는 열이 있는 뷰를 만든 다음, 이 뷰에 대한 권한을 부여합니다.

**예**

emp 테이블에 데이터를 추가할 수 있는 권한을 모든 사용자에게 부여합니다.

```
GRANT INSERT ON emp TO PUBLIC;
```

salesemp 뷰에서 사용 가능한 모든 권한을 mary 사용자에게 부여합니다.

```
GRANT ALL PRIVILEGES ON salesemp TO mary;
```

위 명령을 슈퍼유저와 emp 소유자가 실행하면 모든 권한이 부여되지만, 다른 사용자가 실행하면 해당 사용자가 가지고 있는 그랜트 옵션이 있는 권한만 부여되는 것을 주의하십시오.

admins 역할의 구성원 자격을 joe 사용자에게 부여합니다.

```
GRANT admins TO joe;
```

CONNECT 권한을 joe 사용자에게 부여합니다.

```
GRANT CONNECT TO joe;
```

**관련 항목**

REVOKE, SET ROLE

### 3.3.42 INSERT

**이름**

IN SERT - 테이블에 새 행을 생성합니다

**개요**

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

**설명**

INSERT 는 테이블에 새 행을 삽입시킵니다. 단일 행을 삽입할 뿐 아니라 쿼리의 결과로 한번에 1 행 이상의 행을 삽입할 수 있습니다.

데이터를 삽입하는 줄은 어떤 순서로 나열 되어있을 수도 있습니다. 대상 리스트에 제시되지 않은 각각의 열은 기본값을 사용하여 삽입 됩니다. 열에는 지정된 기본값이나 null 이 들어있습니다.

각 열의 식이 올바른 데이터 형이 아닌 경우에는 자동으로 형 변환을 수행합니다.

INSERT 명령이 SPL 프로그램에서 VALUES 어구와 함께 사용되는 경우에만 RETURNING INTO (record | variable [...]) 어구를 지정할 수 있습니다.

INSERT 명령이 SPL 프로그램에서 사용되는 경우에만 RETURNING BULK COLLECT INTO collection [...] 어구를 지정할 수 있습니다. BULK COLLECT INTO 어구를 대상으로 여러 collection 이 지정된 경우 각 collection 은 단순한 스칼라 필드로 구성되어야 합니다. 즉, collection 레코드는 안됩니다. 삽입된 각 행에 대해 평가된 return_expression 은 첫 번째 요소로 시작하는 collection 의 요소입니다. collection 에 있는 기존 행은 삭제됩니다. 결과가 비어 있으면 collection 은 비어있습니다.

테이블에 행을 추가하려면 해당 테이블에 INSERT 권한이 있어야 합니다. 쿼리를 사용하여 조회 결과를 근거로 행을 삽입하는 경우, 쿼리에서 사용되는 모든 테이블에 대해 SELECT 권한이 있어야 합니다.

**매개변수**

table

기존 테이블의 이름 (스키마로 수식된 이름도 가능).

dblink

원격 데이터베이스를 식별하는 데이터베이스 링크의 이름입니다. 데이터베이스 링크에 대한 자세한 내용은 CREATE DATABASE LINK 명령을 참조하십시오.

column

table 에 있는 열 이름입니다.

expression

해당 column 에 할당하는 식 또는 값을 지정합니다.

DEFAULT

해당 column 에 기본값을 설정합니다.

query

삽입하는 행을 제공하는 질문 (SELECT 문)입니다. 구문에 대한 설명은 SELECT 를 참조하십시오.

return_expression

table 의 1 개 이상의 열이 포함 식입니다. return_expression 에서 table 의 열 이름이 지정된 경우, return_expression 평가 후 열에 할당되는 값은 다음과 같이 결정됩니다.

return_expression 에서 지정한 열이 INSERT 명령에서 값을 할당하면, return_expression 평가 중, 할당된 값이 사용됩니다.

return_expression 에서 지정한 열이 INSERT 명령에서 값이 할당되지 않고, 테이블의 열 정의에서도 열의 기본값이 설정되지 않은 경우 return_expression 평가 중에, null 이 사용됩니다.

return_expression 에서 지정한 열이 INSERT 명령에서 값이 할당되지 않고, 테이블의 열 정의에서 열의 기본값이 지정된 경우 return_e xpression 평가 중에 기본값이 사용됩니다.

record

return_exp ression 으로 평가한 결과가 할당되는 레코드입니다. return_expression 의 첫 번째 결과가 record 에서 첫 번째 필드에 할당됩니다. 2 번째 결과가 2 번째 필드에 할당됩니다. 이하 동일합니다. record 의 필드의 수는 식의 수와 동일해야 합니다. 또한 필드는 할당하는 식의 형식(type)과 호환되어야 합니다.

variable

return_expression 으로 평가한 결과가 할당되는 변수입니다. 1 개 이상의 return_expression 과 variable 이 지정되면, return_expression 의 첫 번째 결과가 첫 번째 variable 에 할당됩니다. 2 번째 결과가 2 번째 variable 에 할당됩니다. 이하 동일합니다. INTO 에 이어 제공되는 변수의 수는 RETURNING 에 이어지는 표현 식의 수와 동일해야 합니다. 또한 변수는 할당하는 식의 형식과 호환되어야 합니다.

collection

return_expression 으로 평가한 결과에서 생성되는 요소의 집합(collection)입니다. 하나의 집합, 즉 단일 필드 집합 수도 있고, 레코드 형식의 집합일 수도 있습니다. 또는 각 집합이 단일 필드로 구성된 여러 집합일지도 모릅니다. 반환된 식의 수는 모두 지정된 집합의 필드 수와 동일해야 합니다. 또한 해당 return_expression 과 collection 필드는 형식과 호환되어야 합니다.

**예**

emp 테이블에 1 행을 삽입합니다.

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

이 예에서는 comm 열이 생략되었습니다. 따라서 comm 열에는 기본값 null 이 할당됩니다.

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

이 예에서는 hiredate 열과 comm 열에 값을 지정하는 것이 아니라 DEFAULT 어구를 사용하고 있습니다.

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

이 경우, deptnames 테이블을 만들고, dept 테이블의 dname 열에서 선택한 결과를 삽입합니다.

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

### 3.3.43 LOCK

**이름**

LOCK - 테이블에 락을 겁니다

**개요**

```
LOCK TABLE name [...] IN lockmode MODE [NOWAIT]
```

lockmode 는 다음 중 하나입니다.

```
ROW SHARE | ROW EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE
```

**설명**

LOCK TABLE 테이블 수준의 락을 가져옵니다. 필요하면 충돌하는 락을 해제할 때까지 기다립니다. NOWAIT 이 지정되면, LOCK TABLE 이 필요로 하는 락을 얻으려고 대기하지 않습니다. 이 경우 즉시 락을 인식하지 못한다면, 명령을 중지하고 에러를 출력합니다. 락은 한번 취득하면 현재 트랜잭션이 완료될 때까지 유지됩니다 (UNLOCK TABLE 같은 명령은 없습니다. 락은 항상 트랜잭션을 종료할 때 해지됩니다).

테이블을 참조하는 명령을 위해 자동으로 락을 거는 경우, Postgres Plus Advanced Server 는 항상 사용 가능한 가장 약한 락 모드를 사용합니다. LOCK TABLE 은 보다 엄격한 락이 필요한 경우를 위해 제공됩니다. 예를 들면, 격리 수준의 읽기가 커밋 된 상황에서 트랜잭션을 수행하는 애플리케이션이 있고, 트랜잭션 내내 테이블의 데이터를 안정적으로 안정시킬 필요가 있는 경우를 생각해 볼 수 있습니다. 이 경우 쿼리를 실행하기 전에 테이블 전체에 SHARE 락 모드를 사용합니다. 이렇게 하면 데이터가 동시에 변경되는 것을 방지하고 이후 테이블의 읽기를 안정시킬 수 있습니다. 왜냐하면 SHARE 락 모드는 쓰기 측이 얻을 ROW EXCLUSIVE 락과 충돌하기 때문에, LOCK TABLE name IN SHARE MODE 문은 ROW EXCLUSIVE 모드 락을 유지하는 트랜잭션이 커밋 되거나 롤백 되기를 기다립니다. 따라서 한번 락을 확보하면, 커밋 되지 않은

상태의 기록은 존재하지 않는 것입니다. 또한 락을 해제하기 전에 다른 애플리케이션은 쓰기(write)를 시작할 수 없습니다.

직렬화 격리 수준에서 실행되는 트랜잭션에서 같은 효과를 얻으려면, 모든 데이터의 변경된 문장을 실행하기 전에 LOCK TABLE 문을 실행해야 합니다. 직렬화 트랜잭션 측 데이터의 뷰는 먼저 데이터 변경된 문장이 시작되면 고정됩니다. 나중에 트랜잭션에서 LOCK TABLE 을 실행하면 동시 쓰기를 방지할 수 있지만, 트랜잭션 읽기 데이터 값이 커밋 된 최신 값인 것은 보장하지 않습니다.

이러한 트랜잭션에서 테이블 데이터를 변경하려면, SHARE 모드가 아닌 SHARE ROW EXCLUSIVE 락 모드를 사용해야 합니다.

이에 따라 한번에 한 트랜잭션만이 실행 될 수 있습니다. SHARE ROW EXCLUSIVE 를 사용하지 않으면 교착 상태가 발생할 수 있습니다. 2 개의 트랜잭션이 모두, SHARE 모드를 보유하고 있으면서 실제 변경하는 데 필요한 ROW EXCLUSIVE 모드를 확인할 수 없는 상태가 될 가능성이 있기 때문입니다 (트랜잭션 자신이 소유하고 있는 락 사이의 충돌이 없기 때문에, 트랜잭션이 SHARE 형태를 유지하고 있는 동안 ROW EXCLUSIVE 를 획득 할 수 있지만 다른 트랜잭션이 SHARE 모드를 유지하고 있는 경우에는 ROW EXCLUSIVE 을 획득할 수 없습니다). 교착 상태를 해결하려면, 모든 트랜잭션이 반드시 동일 객체에 대해 동일한 순서로 락을 획득하는 것이 좋습니다. 또한 1 개의 객체에 대해 다중의 락 모드를 호출하는 경우, 트랜잭션은 항상 가장 엄격한 모드를 먼저 취득해야 합니다.

**매개변수**

name

락을 걸기 위한 기존 테이블의 이름 (스키마로 수식된 이름도 가능).

LOCK TABLE a, b; 명령은 LOCK TABLE a; LOCK TABLE b 와 동일합니다. 테이블은 하나씩 LOCK TABLE 명령에 지정된 순서로 락이 걸립니다.

Lockmode

락 모드는 충돌하는 것에 락을 걸도록 지정합니다.

락 모드를 지정하지 않으면, 가장 제약이 강한 ACCESS EXCLUSIVE 모드를 사용합니다. (ACCESS EXCLUSIVE 는 Oracle 과 호환되지 않습니다. Postgres Plus Advanced Server 에서, 이 모드는 다른 트랜잭션이 어떤 방법으로도 락이 걸린 테이블에 액세스할 수 없게 합니다.)

NOWAIT

LOCK TABLE 이 충돌하는 락이 해제되는 것을 대기하지 않도록 지정합니다. 지정된 락을 즉시 획득할 수 없으면 트랜잭션은 중단됩니다.

**주석**

모든 형태의 LOCK 은 UPDATE 및/또는 DELETE 권한이 있어야 합니다.

LOCK TABLE 은 트랜잭션 블록 내부에서만 사용할 수 있습니다. 따라서 트랜잭션이 완료되면 락도 삭제됩니다. 트랜잭션 블록 외부에서 LOCK TABLE 명령을 실행하면 명령만을 포함한 트랜잭션이 형성되기 때문에, 이 락은 취득 후 즉시 삭제됩니다.

LOCK TABLE 은 테이블 수준의 락만을 처리합니다. 따라서 ROW 를 포함하는 모드 이름은 적절치 않습니다. 이 모드 이름은 보통 락이 걸린 테이블에서 행 수준의 락을 얻을 수 있다고 생각 될 것입니다. 또한, ROW EXCLUSIVE 모드는 공유할 수 있는 테이블
락입니다. LOCK TABLE 에 대해서는 전체 락 모드가 같은 의미를 가지고 있다는 것을 명심하십시오**.** 다른 것은 어떤 모드에 관하여 어떤 락 모드와 충돌 한다는 규칙뿐입니다.

### 3.3.44 REVOKE

**이름**

REVOKE - 액세스 권한을 취소한다

**개요**

```
REVOKE ((SELECT | INSERT | UPDATE | DELETE | REFERENCES)
[,...] | ALL [PRIVILEGES])
ON tablename
FROM (username | groupname | PUBLIC) [...]
[CASCADE | RESTRICT]

REVOKE (SELECT | ALL [PRIVILEGES])
ON sequencename
FROM (username | groupname | PUBLIC) [...]
[CASCADE | RESTRICT]

REVOKE (EXECUTE | ALL [PRIVILEGES])
ON FUNCTION progname
```

([[argmode] [argname] argtype] [...])
FROM (username | groupname | PUBLIC) [...]
[CASCADE | RESTRICT]

REVOKE (EXECUTE | ALL [PRIVILEGES])
ON PROCEDURE progname
[([[argmode] [argname] argtype] [...])]
FROM (username | groupname | PUBLIC) [...]
[CASCADE | RESTRICT]

REVOKE (EXECUTE | ALL [PRIVILEGES])
ON PACKAGE packagename
FROM (username | groupname | PUBLIC) [...]
[CASCADE | RESTRICT]

REVOKE role [...] FROM (username | groupname | PUBLIC)
[...]
[CASCADE | RESTRICT]

REVOKE (CONNECT | RESOURCE | DBA) [...]
FROM (username | groupname) [...]

**설명**

REVOKE 명령은 1 개 이상의 역할에 대해 이전에 부여한 권한을 취소합니다. PUBLIC 키워드는 암시적으로 정의된 모든 역할로 구성된 그룹입니다.

권한 종류의 의미에 대해서는 GRANT 명령에 대한 설명을 참조하십시오.

전체 역할은 그 역할에 직접 부여된 권한, 현재 속해있는 역할에 부여된 권한, PUBLIC 에 부여된 권한의 3 가지 능력을 가지고 있다는 것을 명심하십시오. 따라서, 예를 들면, PUBLIC 에서 SELECT 권한을 취소하는 것은 전체 역할이 객체에 대한 SELECT 권한을 잃는 것을 의미합니다. 권한이 직접적으로 허용되는 역할 또는 다른 역할을 통해 허용되는 역할은 SELECT 권한이 유지됩니다.

권한이 그랜트 옵션과 함께 부여되어있는 경우, 권한뿐만 아니라 그랜트 옵션도 취소됩니다.

그랜트 옵션과 권한이 있는 사용자가 그 권한을 다른 사용자에게 부여하면, 부여 받은 사용자가 보유하고 있는 권한은 의존 권한이라고 합니다. 첫 번째 사용자가 보유한 권한 또는 그랜트 옵션이 취소된 때, 그 권한에 의존 권한이 있으면, 그리고 CASCADE 가 지정되면 의존 권한도

취소됩니다. 지정되지 않으면 권한 취소가 실패합니다. 이 재귀적인 권한 취소는 연이은 사용자를 통해 부여된 권한 속에서도 REVOKE 를 실행한 사용자에게 추적 가능한 범위 내에서만 적용됩니다. 따라서, 의존 권한이 있는 사용자가 다른 사용자를 통해 권한을 부여 받는 다면, REVOKE 을 실행한 후 그 권한을 보유하고 있을 가능성이 있습니다.

**참고:** CASCADE 는 Oracle 호환 옵션이 없습니다. Oracle 은 기본적으로 의존 권한을 취소 cascades 합니다. 그러나 Postgres Plus Advanced Server 는 분명히 CASCADE 를 지정해야 합니다. 지정하지 않으면 REVOKE 명령은 실패합니다.

역할의 구성원 자격을 취소하는 경우처럼 행동하지만 GRANT OPTION 아니라 ADMIN OPTION 가 호출됩니다.

**주석**

취소할 수 있는 사용자가 직접 부여한 권한입니다. 예를 들어, 사용자 A 가 그랜트 옵션을 사용하여 사용자 B 에게 권한을 주고, 그런 다음 사용자 B 가 사용자 C 에게 그 권한을 주었다고 하면 사용자 A 는 사용자 C 의 권한을 직접 취소할 수 없습니다. 대신, 사용자 A 가 사용자 B 에게서 그랜트 옵션을 박탈할 수 있고, 반대로 CASCADE 옵션을 이용함으로 인해 사용자 C 에게 권한을 박탈 당할 수 있습니다. 다른 상황을 생각해 봅시다. A 와 B 모두가 동일한 권한을 C 에게 준 경우, A 는 A 에 부여한 권한을 취소할 수 있지만, B 에 부여한 권한을 취소할 수 없습니다. 따라서 C 는 실질적으로 그 권한을 계속 가지고 있는 것입니다.

객체의 소유자가 아니면, 이 객체의 권한에 대해 REVOKE 를 실행하면 사용자가 객체에 대한 어떠한 권한도 가지고 있지 않는 경우, 즉시 명령이 실패합니다. 어떤 권한이 있는 경우 명령 처리가 계속되지만, 취소할 수 있는 권한은 사용자에게 그랜트 옵션이 있는 권한입니다. REVOKE ALL PRIVILEGES 구문을 전혀 권한이 없는 상태에서 실행하면 경고 메시지를 발행합니다. 다른 구문의 경우 그 명령에 명명된 권한에 대한 그랜트 옵션이 없는 상태에서 실행하면 경고를 메시지를 발행합니다. (대체로 여기까지의 설명은 객체의 소유자에 대해서도 마찬가지로 적용됩니다, 그러나 소유자는 항상 전체 그랜트 옵션을 보유하고 있는 것으로 취급하기 때문에 이런 상황은 결코 발생하지 않습니다)

슈퍼유저가 GRANT 와 REVOKE 발행을 선택하면, 명령은 해당 객체의 소유자가 발행한 것처럼 실행됩니다. 근본적으로는 모든 권한은 객체의 소유자에서부터 전달되는 것이므로 (단, 그랜트 옵션의 연쇄를 통해 간접적으로 전달되는 경우도 있지만), 슈퍼 사용자는 모든 권한을 취소할 수 있습니다. 그러나 이 경우에는 위에서 언급한 CASCADE 를 사용해야 합니다.

REVOKE 는 영향을 주는 객체의 소유자가 아닌 다른 역할을 통해 수행할 수 있지만 객체를 소유하는 역할의 구성원이거나 해당 객체에 대해 WITH GRANT OPTION 권한을 가진 역할의 구성원이어야 합니다. 이 경우 해당 객체의 실제 소유자 역할이나 WITH GRANT OPTION 권한을 가진 역할에 의해 발행된 것 같이, 이 명령은 실행됩니다. 예를 들면, t1 테이블이 g1 역할에 의해

소유되고 u1 이 g1 역할의 구성원이 있다고 가정합니다. 이 경우 u1 이 g1 가 부여한 것으로 기록되어 있는 t1 권한을 취소할 수 있습니다. 여기에는 u1 이 부여한 권한과 g1 역할의 다른 멤버에 의해 부여된 권한이 포함됩니다.

REVOKE 를 수행한 역할이 역할의 여러 구성원 자격 경로를 통해 간접적으로 필요한 권한을 가진 경우, 어떤 역할이 권한을 부여하는 역할로 기록되는지의 내용은 지정되지 않습니다. 이럴 경우 SET ROLE 을 사용하여 REVOKE 을 수행하고자 하는 특정 역할이 되는 것이 좋습니다. 이와 같이 하지 않으면 의도하지 않은 역할에 의해 권한을 박탈하는 것이 되고, 취소 자체가 실패하게 될 수 있습니다.

**예**

public 에게 준 emp 테이블에 대한 삽입 권한을 취소합니다.

```
REVOKE INSERT ON emp FROM PUBLIC;
```

salesemp 뷰에서, mary 사용자에게 준 모든 권한을 취소합니다.

```
REVOKE ALL PRIVILEGES ON salesemp FROM mary;
```

이것은 "자신이 준 모든 권한을 취소한다"는 것을 의미한다는 것에 주의하세요.

사용자 joe 에서 역할 admins 의 구성원 자격을 취소합니다.

```
REVOKE admins FROM joe;
```

사용자 joe 에서 CONNECT 권한을 취소합니다.

```
REVOKE CONNECT FROM joe;
```

**관련 항목**

GRANT, SET ROLE

### 3.3.45 ROLLBACK

**이름**

ROLL BACK - 현재 트랜잭션을 중단합니다

**개요**

ROLLBACK [WORK]

**설명**

ROLLBACK 은 현재 트랜잭션을 롤백하고 해당 트랜잭션에서 갱신된 것 전체를 폐기합니다.

**매개변수**

WORK

선택 가능한 키워드입니다. 아무런 효과가 없습니다.

**주석**

트랜잭션을 성공적으로 완료하기 위해서는 COMMIT 을 사용하십시오.

트랜잭션 외부에서 ROLLBACK 을 실행해도 문제가 되지 않습니다.

ROLLBACK 은 SPL 프로그램에서 지원되지 않습니다.

**예**

모든 변경 사항을 중단합니다.

```
ROLLBACK;
```

**관련 항목**

COMMIT, ROLLBACK TO SAVEPOINT, SAVEPOINT

### 3.3.46 ROLLBACK TO SAVEPOINT

**이름**

ROLLBACK TO SAVEP OINT - 세이브포인트로 롤백 한다.

**개요**

ROLLBACK [WORK] TO [SAVEPOINT] savepoint_name

**설명**

세이브포인트 설정 후 실행된 명령을 모두 롤백 합니다. 세이브포인트는 유효한 상태로 남아 있기 때문에, 필요에 따라 나중에 다시 돌아갈 수 있습니다.

ROLLBACK TO SAVEPOINT 는 명명된 세이브포인트 이후에 설정된 모든 세이브포인트를 암시적으로 삭제합니다.

**매개변수**

savepoint_name

롤백 할 세이브포인트.

**주석**

설정되지 않은 세이브포인트의 이름을 지정하면 에러가 발생합니다.

ROLLBACK TO SAVEPOINT 는 SPL 프로그램에서 지원되지 않습니다.

**예**

depts 설정 후 실행한 명령의 효력을 취소합니다.

```
₩ set AUTOCOMMIT off
INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');
SAVEPOINT depts;
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
ROLLBACK TO SAVEPOINT depts;
```

**관련 항목**

COMMIT, ROLLBACK, SAVEPOINT

### 3.3.47 SAVEPOINT

**이름**

SAVEPOINT - 현재 트랜잭션에서 새로운 세이브포인트를 정의합니다

**개요**

SAVEPOINT savepoint_name

**설명**

SAVEPOINT 는 현재 트랜잭션에서 새로운 세이브포인트를 설정합니다.

세이브포인트는 트랜잭션에 붙이는 특별한 표시입니다. 세이브포인트를 설정하면, 그 이후에 실행된 명령을 모두 롤백 하여 세이브포인트를 지정한 시점의 트랜잭션의 상태로 되돌릴 수 있습니다.

**매개변수**

savep oint_name

새로운 세이브포인트에 부여할 이름입니다.

**주석**

세이브포인트로 롤백 하려면 ROLLBACK TO SAVEPOINT 를 이용하세요.

세이브포인트는 트랜잭션 블록 내부에서만 설정할 수 있습니다. 1 개의 트랜잭션 중에는 여러 세이브포인트를 설정할 수 있습니다.

이미 존재하는 세이브포인트와 같은 이름의 세이브포인트가 설정된 경우에는 오래된 세이브포인트도 유지됩니다. 하지만 롤백 때는 새로운 방식의 세이브포인트를 사용합니다.

**예**

세이브포인트를 설정하고 실행하는 모든 명령의 효력을 취소합니다.

```
₩ set AUTOCOMMIT off
INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');
SAVEPOINT depts;
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
```

```
SAVEPOINT emps;
INSERT INTO jobhist VALUES (9001, '17 - SEP - 07 ', NULL,'CLERK ', 800, NULL, 50,'New Hire ');
INSERT INTO jobhist VALUES (9002, '20 - SEP - 07 ', NULL,'CLERK ', 700, NULL, 50,'New Hire ');
ROLLBACK TO depts;
COMMIT;
```

이 트랜잭션은 dept 테이블에 행의 삽입은 커밋 되지만, emp 테이블과 jobhist 테이블에의 삽입은 롤백됩니다.

**관련 항목**

COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT

### 3.3.48 SELECT

**이름**

SELECT – 테이블이나 뷰에서 행을 반환합니다

**개요**

```
SELECT [optimizer_hint] [ALL | DISTINCT]
* | expression [AS output_name] [...]
FROM from_item [...]
[WHERE condition]
[[START WITH start_expression]
CONNECT BY (PRIOR parent_expr = child_expr |
            c hild_expr = PRIOR parent_expr)
[ORDER SIBLINGS BY expression [ASC | DESC] [...]]]
[GROUP BY expression [...] [LEVEL]
[HAVING condition [...]
[(UNION [ALL] | INTERSECT | MINUS) select]
[ORDER BY expression [ASC | DESC] [...]
[FOR UPDATE]
```

여기서 from_item 는 다음 중 하나입니다.

```
table_name [@ dblink] [alias]
```

(select) alias
from_item [NATURAL] join_type from_item
[ON join_condition | USING (join_column [...])]

**설명**

SELECT 는 1 개 이상의 테이블에서 행을 반환합니다. SELECT 의 일반적인 과정은 다음과 같습니다.

1. FROM 리스트에 있는 모든 요소를 계산합니다 (FROM 리스트에서의 각 요소는 실제 테이블 또는 가상 테이블입니다). FROM 역할에 여러 요소가 지정되면, 그들은 교차 조인됩니다 (아래의 FROM 절 참조).

2. WHERE 절이 지정된 경우, 조건을 충족하지 않는 모든 행은 출력에서 삭제합니다 (아래의 WHERE 절을 참조하십시오).

3. GROUP BY 절이 지정되면, 1 개 이상의 값이 조건에 맞는 각 행에 대해 그룹으로 나눠 출력됩니다. HAVING 절이 있는 경우, 지정한 조건을 충족하지 않는 그룹은 삭제됩니다 (아래의 GROUP BY 절과 HAVING 절을 참조하십시오).

4. UNION, INTERSECT, MINUS 연산자를 사용하면, 여러 SELECT 문의 출력을 1 개의 결과 집합으로 조인할 수 있습니다. UNION 연산자는 두 결과 집합에 있는 행과 한쪽의 결과 집합에 있는 행을 모두 반환합니다. INTERSECT 연산자는 두 결과 집합에 있는 행을 반환합니다. MINUS 연산자는 첫 번째 결과 집합이고, 2 번째 결과 집합에 없는 행을 반환합니다. 이러한 3 가지 경우 모두, 중복되는 행은 삭제됩니다. UNION 연산자의 경우, ALL 이 지정되면 중복되는 행은 삭제되지 않습니다 (아래 UNION 어구, INTERSECT 어구, MINUS 절을 참조하십시오).

5. 실제로는 선택한 각 행에 대해 SELECT 출력 방식을 사용하여 계산한 결과의 행이 표시됩니다 (아래의 SELECT 역할을 참조하십시오).

6. CONNECT BY 절이 지정된 경우, 계층 관계가 있는 데이터를 선택합니다. 그러한 자료는 행 사이에 친자 관계를 가지고 있습니다 (아래의 CONNECT BY 절을 참조하십시오).

7. ORDER BY 절이 지정된 경우, 반환된 행은 지정된 순서로 정렬됩니다. ORDER BY 가 지정되지 않으면, 시스템이 조사 과정에서 찾은 가장 빠른 순서대로 행을 반환합니다 (아래의 ORDER BY 절을 참조하십시오).

8. DISTINCT 는 결과에서 중복 행을 삭제합니다. ALL(기본값)에서 중복 행을 포함하는 모든 후보 행을 반환합니다 (자세한 내용은 아래 DISTINCT 절을 참조하십시오).

9. FOR UPDATE 절을 사용하면, SELECT 문은 동시에 발생하는 갱신에 대비해 선택된 행에 락을 겁니다. (아래의 FOR UPDATE 절을 참조하십시오).

테이블에서 값을 읽으려면 SELECT 권한이 있어야 합니다. FOR UPDATE 를 사용하려면 또한 UPDATE 권한이 있어야 합니다.

**매개변수**

optimizer_hint

최적화 프로그램이 실행 계획을 선택할 때 사용하는 주석에 둔 힌트입니다. 최적화 힌트에 대한 자세한 내용은 3.4 장을 참조하십시오.

기타 매개변수는 다음 장에서 설명합니다.

### 3.3.48.1 FROM 어구

FROM 절에는 SELECT 의 대상이 되는 원본 테이블을 1 개 이상 지정합니다. 여러 원본을 지정하면 결과는 전체 원본의 카티션 곱(교차 조인)입니다. 일반적으로 제약 조건을 지정하고 카티션 곱의 일부를 반환하도록 결과 행을 제약합니다.

FROM 절에 다음과 같은 요소를 포함할 수 있습니다.

table_name [@ dblink]

기존 테이블 또는 뷰 이름 (스키마로 수식된 이름도 가능). dblink 는 원격 데이터베이스를 식별하는 데이터베이스 링크 이름입니다. 데이터베이스 링크에 대한 자세한 내용은 CREATE DATABASE LINK 명령을 참조하십시오.

alias

별칭을 포함하는 FROM 항목의 대체 이름입니다. 별칭은 간결하게 하기 위해, 혹은 자기 조인 (동일한 테이블을 여러 번 검사하는 조인)의 애매함을 없애기 위해 사용됩니다. 별칭이 지정된 경우, 해당 별칭을 통해 실제 테이블 이름 또는 함수이름을 완전히 숨깁니다. 예를 들면, FROM foo AS f 로 지정되면, 이후 SELECT 문장에서는 이 FROM 항목을 foo 가 아닌 f 로 참조해야 합니다.

select

FROM 절에는 하위 SELECT 를 사용할 수 있습니다. 단일의 SELECT 명령을 실행하는 동안, 하위 SELECT 의 결과는 임시 테이블에 있는 것처럼 작동합니다. 하위 SELECT 는 괄호로 둘러싸여 있어야 합니다. 또한 반드시 별칭을 주지 않으면 안됩니다.

join_type

다음 중 하나입니다.

[INNNER] JOIN
LEFT [OUTER] JOIN
RIGHT [OUTER] JOIN
FULL [OUTER] JOIN
CROSS JOIN

INNER 및 OUTER 조인 형식에서는 조인 조건,
즉 NATURAL, ON join_condition, USING (join_column [...]) 중 1 개만을 지정해야 합니다. 각각의 의미는 아래를 참조하세요. CROSS JOIN 은 이 어구를 작성하지 않아도 상관없습니다.

JOIN 절은 2 개의 FROM 항목을 결합합니다. 중첩된 순서를 결정하는 데 필요하다면 괄호를 사용하십시오. 괄호 없이, JOIN 은 왼쪽에서 오른쪽으로 순환합니다. 어떤 경우에도 JOIN 은 콤마로 분리된 FROM 항목보다 강한 속박을 갖습니다.

CROSS JOIN 과 INNER JOIN 은 카티션 곱 1 개를 생성합니다. 이것은 FROM 의 맨 위에 2 개의 항목을 조인한 결과와 동일합니다. 그러나 조인 조건(만약 있다면)에 의해 제약을 걸 수 있습니다. CROSS JOIN 은 INNER JOIN ON (TRUE)과 동일하며, 조건에 의해 삭제되는 행은 없습니다. 이 조인 형식은 기술상의 편의를 위해서만 사용할 수 있습니다. 이에 따라 FROM 와 WHERE 으로 실행하지 않으면 아무것도 실행하지 않습니다.

LEFT OUTER JOIN 은 조건에 맞는 카티션 곱의 모든 행 (즉, 그 조인 조건을 만족하는 모든 조합) 이외 왼쪽 테이블에서 오른쪽 테이블에 조인 조건에 응하는 행이 없는 행의 복사본을 반환합니다. 이 왼쪽 테이블의 행을 조인한 결과 테이블의 폭을 확장하기 위해 오른쪽 테이블의 열에 null 값이 삽입됩니다. 행이 일치하는 가를 결정할 때는 JOIN 어구 자신의 조건만을 고려하는 것을 주의하세요. 다른 외부 조인 조건은 나중에 적용됩니다.

역으로, RIGHT OUTER JOIN 은 모든 조인 행 이외에 왼쪽 테이블의 행과 일치하는 행이 없는 오른쪽 행 (왼쪽은 null 로 확장됩니다)을 반환합니다. 좌우 테이블에 입력 값을 바꿀 수 있는 경우, LEFT OUTER JOIN 로 변환할 수 있으므로 RIGHT OUTER JOIN 는 기술상의 편의를 위해 제공된 것에 불과합니다.

FULL OUTER JOIN 은 모든 조인 행 이외에 일치하지 않은 왼쪽 테이블의 행(오른쪽은 null 로 확장), 일치하지 않은 오른쪽 테이블의 행 (왼쪽은 null 로 확장)을 모두 반환합니다.

ON join_condition

join_condition 은 조인에서 어떤 행이 일치할 지의 여부, BOOLEAN 형식의 값을 반환하는 식 (WHERE 절과 유사합니다)입니다.

USING (join_column [...])

USING (a, b, ...) 절은 ON left_table.a = right_table.a AND left_table.b = right_table.b .... 의 약어입니다. USING 은 해당하는 열의 각 두 쪽 모두가 아니라 한 쪽만을 조인 출력에 포함되는 것을 의미합니다.

NATURAL

NATURAL 은 2 개의 테이블에서 같은 이름을 가진 행을 모두 선택한 USING 리스트의 약어입니다.

### 3.3.48.2 WHERE 어구

WHERE 어구는 일반적으로 다음과 같은 형식이 됩니다 (이 어구는 생략 가능).

WHERE condition

condition 은 평가 결과로 BOOLEAN 형식을 반환하는 모든 표현식입니다. 이 조건에 부합하지 않는 행은 출력에서 삭제됩니다. 모든 변수에 실제 행의 값을 대입하고, 식이 True 를 반환하면 그 부분은 조건을 충족시키는 것으로 간주됩니다.

#### 3.3.48.3 GROUP BY 구문

임의의 GROUP BY 절은 보통 다음과 같은 형식입니다.

*GROUP* BY expression [...]

GROUP BY 는 그룹화된 식을 위해 같은 값을 가지는 선택된 모든 행을 하나의 행으로 압축합니다. *expression 은* 입력 열 명이나 이름, 출력 열의 서수(SELECT 목록 항목), 입력 열 값 형식의 임의의 식으로 표현될 수 있습니다. 애매한 경우, GROUP BY 이름은 출력 열 이름 보다는 입력 열 이름으로 해석됩니다.

집계 함수를 사용하면, 각 그룹의 모든 행을 대상 계산 되어 지고, 그 결과로 각 그룹의 값이 생성됩니다(한편, GROUP BY 이 없으면 집계 함수는 선택된 모든 행을 대상으로 계산을 실시하고, 하나의 값을 생성합니다). GROUP BY 가 있으면, 집계 함수를 제외하고, 그룹화되지 않은 열을 참조하는 SELECT 목록은 사용할 수 없습니다. 그룹화되지 않은 열을 반환하는 값은 여러 값이 될 가능성이 있기 때문입니다.

#### 3.3.48.4 HAVING 구문

선택적인 HAVING 구문은 일반적으로 다음과 같은 형식입니다.

*HAVING* condition

*condition 은* WHERE 절에 지정하는 것과 동일합니다.

HAVING 은 조건을 만족하지 않는 그룹 행을 제거합니다. HAVING 과 WHERE 은 다음 사항이 크게 다릅니다. WHERE 은 GROUP BY 를 적용하기 전에 각 행을 필터링 하는 반면, HAVING 은 GROUP BY 를 적용한 후에 생성된 그룹 행에 대해 필터링을 합니다. 집계 함수 내에서 참조되는 경우를 제외하고, 조건에서 참조된 각 열은 분명히 그룹화된 열을 참조해야만 합니다.

**3.3.48.5 SELECT 목록**

SELECT 목록 (SELECT 와 FROM 사이에 있는 키워드)는 SELECT 문의 출력 행을 형성하는 표현식을 지정합니다. 이 표현식을 통해 FROM 절의 처리 후, 참조되는 열을 볼 수 있습니다. (일반적으로 이동합니다). AS *output_name 절을* 사용하면, 출력 열에 다른 이름을 지정할 수 있습니다. 이 이름은 주로 표시하기 위한 열의 라벨로 사용됩니다. 또한, ORDER BY 절과 GROUP BY 절의 열 값을 참조할 때도 이 이름을 사용할 수 있습니다. 그러나 WHERE 및 HAVING 절에서는 사용할 수 없습니다. 이들은 표현식을 작성해야만 합니다.

표현식을 대신해, *은 선택된 행의 열을 빠르게 출력 리스트에 입력할 수 있도록 합니다.

**3.3.48.6 UNION 절**

UNION 절은 일반적으로 다음과 같은 형식입니다.

*select_statement UNION* [ALL] *select_statement*

*select_statement 는* ORDER BY 나 FOR UPDATE 절이 없는 모든 SELECT 문장입니다 (ORDER BY 는 괄호로 둘러싼 하위 표현식을 부여할 수 있습니다. 괄호가 없는 경우에는 이 절은 우측 입력 표현식이 아닌, UNION 결과에 적용됩니다).

UINON 연산자는 SELECT 문을 포함하는 반환 행의 결합을 계산합니다. 최소한 결과 집합 하나가 있을 때, 두 결과의 결합된 행이 설정됩니다. UINON 연산자를 직접적으로 나타내는 두 가지 SELECT 문은 같은 수의 열을 산출해야 하며, 대응하는 열은 데이터 형식이 호환되어야만 합니다.

UNION 연산자는 SELECT 문을 포함하는 행을 리턴하는 결합 집합을 산출합니다.

ALL 옵션이 지정되지 않는 한, UNION 의 결과는 중복되는 행을 가지지 않습니다. ALL 은 중복되는 행을 제거하지 않습니다.

괄호로 지정되지 않는다면, 같은 SELECT 문장에서 복수의 UNION 연산자는 왼쪽에서 오른쪽으로 실행됩니다.

현재, FOR UPDATE 는 UNION 결과나 UNION 의 입력 결과를 위해 지정할 수 없습니다.

**3.3.48.7 INTERSECT 구문**

INTERSECT 절은 일반적으로 다음과 같은 형식입니다.

*select_st atement INTERSECT select_statement*

*select_statement 는* ORDER BY 나 FOR UPDATE 절이 없는 모든 SELECT 문장입니다.

INTERSECT 연산자는 SELECT 문장이 포함된 리턴 행의 교집합을 산출합니다. 두 결과 집합에 나타나는 경우, 두 결과의 교집합의 행이 설정됩니다.

INTERSECT 의 결과는 중복된 행을 포함하지 않습니다.

괄호가 지정되지 않는 한, 같은 SELECT 문장에서 복수의 INTERSECT 연산자는 왼쪽에서 오른쪽으로 실행됩니다. INTERSECT 는 UNION 보다 더 엄중하게 구속합니다. 즉, A UNION B INTERSECT C 는 A UNION (B INTERSECT C)로 해석됩니다.

**3.3.48.8 MINUS 구문**

MINUS 절은 일반적으로 다음과 같은 형식입니다.

*select_statement MINUS select_statement*

*select_statement 는* ORDER BY 나 FOR UPDATE 절이 없는 모든 SELECT 문장입니다.

MINUS 은 왼쪽 SELECT 문장의 결과는 존재하지 오른쪽 SELECT 문의 결과에 없는 행 집합을 생성합니다.

MINUS 결과에 중복 행을 포함하지 않습니다.

같은 SELECT 문에서 여러 MINUS 연산자를 사용하는 경우, 괄호를 지정하지 않는 한 왼쪽에서 오른쪽으로 실행됩니다. MINUS 는 UNION 과 동일한 수준으로 구속합니다.

**3.3.48.9 CONNECT BY 구문**

CONNECT BY 절은 계층 쿼리를 실행할 때, 부모-자식 관계를 결정합니다. 일반적으로 다음과 같은 형식입니다.

CONNECT BY *(PRIOR* parent_expr = *child_expr* |
*child_expr* = *PRIOR* parent_ex pr)

*parent_expr 는* 부모의 후보 행으로 실행됩니다. FROM 구문에 반환되는 행에서 *parent_expr* = *child_expr 0|"*"true"인 경우, 이 행은 부모의 자식 행으로 간주됩니다.

다음 절은 선택해서 CONNECT BY 절과 함께 지정될 수 있습니다.

*START* WITH start_expression

FROM 구문에 반환되는 행을 *start_expression 으로* 실행한 결과가 "true"일 경우, 이는 계층의 루트 노드가 됩니다.

*ORDER* SIBLINGS BY expression [ASC | DESC] [...]

결과 집합의 *expression 으로* 계층의 동기 행이 정돈됩니다.

(계층 쿼리에 대한 자세한 내용은 2.2.5 절을 참조하십시오.)

**3.3.48.10 ORDER BY 구문**

선택할 수 있는 ORDER BY 절은 일반적으로 다음과 같은 형식입니다.

*ORDER* BY expression [ASC | DESC] [...]

*expression 은* 출력 열(SELECT 목록 항목)의 이름이나 서수, 또는 입력 열 값에서 형성되는 임의의 수식이 될 수 있습니다.

ORDER BY 절은 사용하면 결과 행을 지정된 식에 따라 정렬할 수 있습니다. 가장 왼쪽의 표현식에 따라, 2 개의 행이 동등하다고 판단되면, 다음 표현식과 비교합니다. 그 결과도 동등한 경우, 다음과 같은 방식으로 계속 진행됩니다. 지정된 모든 식이 동등할 경우, 실행에 의존하는 순서에 따라 반환됩니다.

서수는 결과 열의 위치 (왼쪽에서 오른쪽으로)를 참조합니다. 이 기능은 유일한 이름이 없는 열의 기본 순서를 정의할 수 있도록 합니다. AS 절을 사용하면, 결과 열에 이름을 할당할 수 있기 때문에, 이는 꼭 필요하지는 않습니다.

또한, ORDER BY 절에는 SELECT 결과 목록에 나타나지 않는 열이 포함하여, 모든 공식을 사용할 수 있습니다. 따라서 다음과 같은 문장을 사용할 수 있습니다.

```
SELECT ename FROM emp ORDER BY empno;
```

이 기능의 한계는 UNION, INTERSECT 나 MINUS 구문 결과를 적용하는 ORDER BY 가 표현식이 아닌, 출력 열 이름이나 숫자로 지정되어야만 한다는 것입니다.

ORDER BY 표현식 결과 열 이름을 입력 열 이름에 모두 일치하는 단순한 이름이 주어진 경우, ORDER BY 는 결과 열 이름으로 취급합니다. 이는 동일한 상황에서 GROUP BY 의 선택과 반대됩니다. 이러한 불일치는 SQL 표준과 호환성을 유지하기 때문에 발생합니다.

ORDER BY 구문의 표현식 후에, 선택적으로 키워드 ASC (오름차순)나 DESC (내림차순)을 추가하실 수 있습니다. 지정돼있지 않는 경우, 디폴트로 ASC 를 가정합니다.

null 값은 다른 어떤 값보다 높은 값으로 정렬됩니다. 즉, 오름 정렬 순서에서 null 값은 마지막에 정렬되며, 내림차순에서는 null 값이 처음으로 정렬됩니다.

문자열 데이터는 데이터베이스 클러스터를 초기화하는 동안, 결정된 로케일 지정 대조 순서에 따라 정렬됩니다.

#### 3.3.48.11 DISTINCT 구문

DISTINCT 가 지정되면, 중복되는 모든 부분은 결과 집합으로부터 삭제됩니다 (각 그룹의 중복으로부터 한 행만 유지됩니다). ALL 은 반대로 모든 행이 유지되며, 기본으로 설정됩니다.

#### 3.3.48.12 FOR UPDATE 구문

FOR UPDATE 절은 다음과 같은 형식입니다.

```
FOR UPDATE
```

FOR UPDATE 를 사용하면, SELECT 문은 검색된 행이 업데이트에 대해 잠기게 됩니다. 이에 따라 현재 트랜잭션이 완료될 때까지, 이 줄은 다른 트랜잭션에 의해 변경되거나 삭제되지 않습니다. 즉, 현재의 트랜잭션이 종료될 때까지, 이 행은 다른 트랜잭션 UPDATE, DELETE 나 SELECT FOR UPDATE 의 실행을 차단합니다. 또한, 다른 트랜잭션으로부터 UPDATE, DELETE, SELECT FOR UPDATE 에 의해 선택된 행이 이미 잠겨있을 경우, SELECT FOR UPDATE 는 다른 트랜잭션이 종료되기를 기다리며, 그 다음 락을 실행하고, 업데이트된 행을 반환합니다(행이 삭제되면 반환하지 않습니다).

FOR UPDATE 는 반환된 행이 테이블의 모든 행에 해당하는지 명확하게 확인할 수 없는 경우에는 사용할 수 없습니다. 예를 들면, 집합으로 사용할 수 없습니다.

**예제**

dept 테이블을 emp 테이블과 조인합니다.

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

deptno | dname | empno | ename | mgr | hiredate
--------+------------+-------+--------+------+---- ----------------
10 | ACCOUNTING | 7934 | MILLER | 7782 | 23 - JAN - 82 00:00:00
10 | ACCOUNTING | 7782 | CLARK | 7839 | 09 - JUN - 81 00:00:00
10 | ACCOUNTING | 7839 | KING | | 17 - NOV - 81 00:00:00
20 | RESEARCH | 7788 | SCOTT | 7566 | 19 - APR - 87 00:00:00
20 | RESEARCH | 7566 | JONES | 7839 | 02 - APR - 81 00:00:00
20 | RESEARCH | 7369 | SMITH | 7902 | 17 - DEC - 80 00:00:00
20 | RESEARCH | 7876 | ADAMS | 7788 | 23 - MAY - 87 00:00:00
20 | RESEARCH | 7902 | FORD | 7566 | 03 - DEC - 81 00:00:00
30 | SALES | 7521 | WARD | 7698 | 22 - FEB - 81 00:00:00
30 | SALES | 7844 | TURNER | 7698 | 08 - SEP - 81 00:00:00
30 | SALES | 7499 | ALLEN | 7698 | 20 - FEB - 81 00:00:00
30 | SALES | 7698 | BLAKE | 7839 | 01 - MAY - 81 00:00:00
30 | SALES | 7654 | MARTIN | 7698 | 28 - SEP - 81 00:00:00
30 | SALES | 7900 | JAMES | 7698 | 03 - DEC - 81 00:00:00
(14 rows)
```

전체 직원의 sal 과 부서 번호를 이용한 그룹 결과의 열을 합계를 구하기 위해서 다음과 같이 합니다.

```
SELECT deptno, SUM (sal) AS total
FROM emp
GROUP BY deptno;

deptno | total
--------+----------
10 | 8750.00
20 | 10875.00
30 | 9400.00
(3 rows)
```

전체 직원의 sal 과 부서 번호를 이용한 결과 그룹 열의 합계를 구하고, 10000 보다 총계가 적은 그룹을 보여줍니다.

```
SELECT deptno, SUM (sal) AS total
FROM emp
GROUP BY deptno
```

```
HAVING SUM (sal) <10000;

deptno | total
--------+---------
10 | 8750.00
30 | 9400.00
(2 rows)
```

다음의 두 가지 예제는 두 번째 열(dname)의 내용에 따른 개개의 결과를 정렬하는 동일한 방법입니다.

```
SELECT * FROM dept ORDER BY dname;

deptno | dname | loc
--------+------------+----------
10 | ACCOUNTING | NEW YORK
40 | OPERATIONS | BOSTON
20 | RESEARCH | DALLAS
30 | SALES | CHICAGO
(4 rows)

SELECT * FROM dept ORDER BY 2;

deptno | dname | loc
--------+------------+----------
10 | ACCOUNTING | NEW YORK
40 | OPERATIONS | BOSTON
20 | RESEARCH | DALLAS
30 | SALES | CHICAGO
(4 rows)
```

### 3.3.49 SET CONSTRAINTS

#### 이름

SET CONSTRAINTS -- 현재 트랜잭션을 위한 제한 검사 모드 설정

#### 개요

SET CONSTRAINTS (ALL | *name* [...]) (DEFERRED | IMMEDIATE)

**설명**

SET CONSTRAINTS 는 현재 트랜잭션에서 제한 검사 방법을 설정합니다. IMMEDIATE 제한은 각 문장의 실행이 끝날 때마다 확인합니다. DEFERRED 제한은 트랜잭션이 처리되기 전에 검사되지 않습니다. 각각의 제한은 IMMEDIATE 또는 DEFE RRED 중 하나의 형태를 가지고 있습니다.

생성에 따라, 제한은 DEFERRABLE INITIALLY DEFERRED, DEFERRABLE INITIALLY IMMEDIATE, NOT DEFERRABLE 3 개 중 하나의 성격을 갖게 됩니다. 3 번째 클래스는 항상 IMMEDIATE 형태이고 SET CONSTRAINTS 커맨드의 영향을 받지 않습니다. 처음 두 클래스에서는 트랜잭션을 지정된 형태로 시작하지만, 트랜잭션에서 SET CONSTRAINTS 를 사용하면 행동을 수정할 수 있습니다.

제한 이름 목록을 가진 SET CONSTRAINTS 을 변경하는 것은 이러한 제한 모드입니다 (이들은 모두 연기할 수 있습니다). 지정된 이름과 일치하는 여러 제한 사항이 있을 경우, 모든 제한 조건에 영향을 받습니다. SET CONSTRAINTS ALL 지연 수 있는 모든 제한 형태를 변경합니다.

SET CONSTRAINTS 를 사용하여 제한 형태를 DEFERRED 에서 IMMEDIATE 로 변경할 때, 새로운 형태는 소급적으로 적용됩니다. 즉, 트랜잭션의 종료 시점에서 임의의 데이터 변경이 검사되지 않았을 경우, SET CONSTRAINTS 커맨드가 실행되는 동안 대신해서 검사합니다. 만약 제한을 위반했을 경우, SET CONSTRAINTS 는 실패합니다(그리고 제한 형태는 변경되지 않습니다). 따라서 SET CONSTRAINTS 를 이용하면, 트랜잭션의 특정한 시점에서 강제로 제한 검사를 수행할 수 있습니다.

현재, 외부 키 제한 조건만이 설정에 따라 영향을 받습니다. 검사 제한 및 유일성 제약은 항상 효과적으로 지연되지는 않습니다.

**참고**

이 커맨드만 변경하는 것은 현재 트랜잭션 내의 제한 동작입니다. 따라서 트랜잭션 블록의 외부에서 이 커맨드를 실행해도 아무런 효과가 없습니다.

### 3.3.50 SET ROLE

**이름**

SET ROLE - 현재 세션에서 현재 사용자 ID 를 설정할

**개요**

SET ROLE { *rolename* | NONE }

**설명**

이 커맨드는 현재 SQL 세션에서 현재 사용자 식별자를 *rolename 로* 설정합니다. SET ROLE 후에, SQL 커맨드에 대한 권한 검사는 임명된 롤에 정상적으로 로그인한 경우와 마찬가지로 수행됩니다.

지정된 *rolename 는* 현재 세션 사용자가 구성원으로 속한 롤이어야 합니다. (세션 사용자가 슈퍼유저일 경우, 어떤 역할이라도 선택될 수 있습니다.)

NONE 은 현재 사용자의 식별자를 현재 세션 사용자의 식별자로 재설정합니다. 이 형식은 모든 사용자에 의해 실행될 수 있습니다.

**참고**

이 커맨드를 사용하여 권한을 추가하거나 제한하는 것이 가능합니다. 세션 사용자의 역할이 INHERITS 속성을 가지는 경우, 자동으로 SET ROLE 에 설정된 모든 역할의 권한을 가지게 됩니다. 이 경우, SET ROLE 은 실제 세션 사용자에 직접 할당된 권한 세션 사용자가 속한 역할에 할당된 권한 중 지정된 목록에서 사용할 수 있는 권한을 남겨두고 다른 사람을 모두 삭제합니다. 다른 한편으로, 세션 사용자의 역할이 NOINHERITS 속성을 가진 경우, SET ROLE 는 세션 사용자에 직접 할당되는 모든 권한을 모두 제거하고 지정된 역할에서 사용할 수 있는 권한을 획득합니다.

특히, 슈퍼유저가 슈퍼유저가 아닌 사용자로 SET ROLE 을 선택하면, 슈퍼유저 권한을 잃게 됩니다.

**예제**

사용자 mary 를 admins 역할의 사용자 ID 를 설정합니다.

```
SET ROLE admins;
```

사용자 mary 를 원래 사용자 ID 로 변경합니다.

```
SET ROLE NONE;
```

**관련 항목**

ALTER ROLE, CREATE ROLE, DROP ROLE, GRANT, REVOKE

### 3.3.51 SET TRANSACTION

**이름**

SET TRANSACTION -- 현재 트랜잭션의 특성을 설정

**개요**

*SET* TRANSACTION tran saction_mode

여기서 *transaction_mode 는* 다음 중 하나입니다.

ISOLATION LEVEL (SERIALIZABLE | READ COMMITTED)
READ WRITE | READ ONLY

**설명**

SET TRANSACTION 커맨드는 현재 트랜잭션의 특성을 설정합니다. 이것은 다음 트랜잭션에 영향을 미치지 않습니다.

사용 가능한 트랜잭션 특징은 트랜잭션의 격리 수준과 트랜잭션의 액세스 모드 (읽기 / 쓰기 또는 읽기 전용 모드)입니다.

트랜잭션의 격리 수준은 동시에 실행되는 다른 트랜잭션이 존재할 경우, 트랜잭션이 볼 수 있는 데이터를 결정하는 것입니다.

READ COMMITTED

문장은 이것이 시작되기 전에 처리된 행만을 볼 수 있습니다. 이것이 디폴트로 설정되어 있습니다.

SERIALIZABLE

현재 트랜잭션의 모든 문장은 트랜잭션에서 첫 번째 쿼리나 데이터 변경 문장이 실행되기 전에 처리된 행만을 볼 수 있습니다.

트랜잭션 격리 수준은 트랜잭션의 첫 번째 쿼리나 데이터 변경 문장 (SELECT, INSERT, DELETE, UPDATE 나 FETCH)가 실행된 이후에는 변경할 수 없습니다.

트랜잭션 접근 모드는 해당 트랜잭션의 읽기 / 쓰기 또는 읽기 전용 여부를 결정합니다. 기본값은 읽기 / 쓰기입니다. 읽기 전용 트랜잭션은 다음의 SQL 커맨드를 거부합니다 : INSERT, UPDATE, DELETE. 또한, CREATE, ALTER, DROP 커맨드는 모든 SQL 커맨드 : COMMENT, GRANT, REVOKE, TRUNCATE 를 실행 할 수 없습니다. 그리고 EXECUTE 커맨드는 위의 커맨드를 포함할 경우, 커맨드를 실행할 수 없습니다. 이는 디스크 입력을 예방하지 않기 때문에 때문에 읽기 전용을 높은 수준에서 실현하는 개념입니다.

### 3.3.52 TRUNCATE

**이름**

TRUNCATE – 비어 있는 테이블

**개요**

*TRUNCATE* TABLE name

**설명**

TRUNCATE 는 테이블로부터 모든 행이 빠르게 삭제됩니다. 이는 조건 없는 DELETE 를 사용하는 것과 같은 효과를 가지지만, 실제로 테이블을 검사하지 않기 때문에 더 빠릅니다. 이 커맨드는 큰 테이블을 대상으로 하는 경우에 가장 유용합니다.

**매개 변수**

*name*

빈 테이블의 이름 (선택적으로 스키마 권한도 가능).

**참고**

테이블이 다른 테이블의 외래 키 참조하는 경우, TRUNCATE 를 사용할 수 없습니다. 이런 경우에는 검사 유효성은 테이블 스캔을 요구하며, 전체적인 관점에서 실행되지 않습니다.

TRUNCATE 은 테이블에 존재하는 사용자 정의 ON DELETE 트리거를 일절 사용하지 않습니다.

**예제**

bigtable 테이블을 비웁니다.

```
TRUNCATE TABLE bigtable;
```

**관련 항목**

DELETE

### 3.3.53 UPDATE

**이름**

UPDATE - 테이블의 행을 업데이트합니다

**개요**

```
UPDATE [optimizer_hint] table [@ dblink]
SET column = (e xpression | DEFAULT) [...]
[WHERE condition]
[RETURNING return_expression [...]
(INTO (record | variable [...])
| BULK COLLECT INTO collection [...])]
```

**설명**

UPDATE 은 조건을 충족시키는 모든 행의 지정된 열 값을 변경합니다. SET 구문에는 변경되는 열만을 지정해야 합니다. SET 구문에서 명시적으로 지정되지 않은 열의 값은 변경되지 않습니다.

RETURNING INTO *(record* | *variable* [...]) 어구는 SPL 프로그램 내에 지정되어야 합니다. 또한 UPDATE 커맨드의 결과 집합은 1 행 다음에 반환되면 안됩니다. 그 밖에는 예외를 던집니다. 결과 집합이 비어 있으면 대상 레코드 내용이거나 변수가 null 로 설정됩니다.

UPDA TE 커맨드가 SPL 프로그램에서 사용되는 경우에만 *RETURNING* BULK COLLECT IN TO collection [...] 구문을 지정할 수 있습니다. BULK COLLECT INTO 구문을 대상으로 여러 *collection 이* 지정된 경우, 각 *collection 은* 단순한 스칼라 필드로 구성되어야 합니다. 즉, *collection 는* 레코드일 수 없습니다. UPDATE 커맨드의 결과는 아무것도 포함하지 않거나 하나 이상의 행을 포함할 수 있습니다. *return_expression 는 collection 의* 요소로서 첫 번째 행을 시작으로 결과 집합의 각 행을 위해 실행됩니다. *collection 에 있는* 기존 행은 삭제됩니다. 결과가 비어 있으면 *collection 은* 비게 됩니다.

테이블을 변경하려면 UPDATE 권한을 가져야 합니다. 또한 expression 이나 *condition 에서* 가져올 테이블에 대한 SELECT 권한도 필요합니다.

**매개 변수**

*optimizer_hint*

실행 계획의 선택을 위한 옵티마이저에 관해 주석이 내장된 힌트입니다. 옵티마이저 힌트에 대한 자세한 내용은 3.4 절을 참조하십시오.

*table*

업데이트되는 테이블의 이름 (스키마 이름도 가능).

*dblink*

원격 데이터베이스를 식별하는 데이터베이스 연결 이름입니다. 데이터베이스 링크에 대한 자세한 내용은 CREATE DATABASE LINK 커맨드를 참조하십시오.

*column*

*table 에있는* 열 이름입니다.

*expression*

열에 할당하는 표현식입니다. 이 식은 테이블의 다른 열과 이전 값을 사용할 수 있습니다.

DEFAULT

열에 기본값을 설정합니다 (기본 표현식을 지정하지 않으면 null 입니다).

*condition*

BOOLEAN 형식을 반환하는 식입니다. 이 식이 true 를 반환하는 행만 업데이트됩니다.

*return_expression*

테이블로부터 1 개 이상의 열을 포함하는 식입니다. *return_expression* 에서 테이블의 열 이름이 지정된 경우, *return_expression* 실행 후 열에 할당되는 값은 다음과 같이 결정됩니다.

*return_expression 에서* 지정한 열에 UPDATE 커맨드의 값을 할당하면, *ret urn_expression* 실행 중에 값이 사용됩니다.

*return_expression 에서* 지정한 열에 UPDATE 커맨드의 값이 할당되지 않은 경우, *return_expression* 실행 중에 대상 행의 현재 열 값을 사용합니다.

*record*

*return_expression 으로* 실행한 결과가 할당되는 레코드입니다. *return_expression 의* 첫 번째 결과가 *record 에서* 첫 번째 필드에 할당되고, 2 번째 결과가 2 번째 필드에 할당됩니다. 이하 동일합니다. *record 의* 필드 수는 식의 숫자와 동일해야 합니다. 또한 필드는 할당된 식의 형식에 호환되어야 합니다.

*variable*

*return_expression 으로* 실행한 결과가 할당되는 변수입니다. 1 개 이상 *return_expression 과 variable 가* 지정되면, *return_expression 의* 첫 번째 결과가 첫 번째 *variable 에* 할당됩니다. 2 번째 결과가 2 번째 *variable 에* 할당됩니다. 이하 동일합니다. INTO 키워드 다음에 지정되는 변수의 수는 RETURNING 키워드 뒤에 표현식의 수와 동일해야 합니다. 또한 변수는 할당된 식과 형식이 호환되어야 합니다.

*collection*

*return_expression 으로* 실행한 결과에서 만들어지는 요소의 집합입니다. 하나의 컬렉션 즉, 단일 필드 집합 수도 있고, 레코드 형식의 집합일 수도 있습니다. 또는 각 집단이 단일 필드로 구성된 여러 집단일 수도 있습니다. 반환되는 표현식 수는 모두 지정된 집합 필드 수의 순서와 동일해야 합니다. 그리고 대응하는 *return_expression 과 collection* 필드는 형식이 호환되어야만 합니다.

**예제**

dept 테이블에서 부서의 20 를 위한 AUSTIN 로 장소를 변경합니다.

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

emp 테이블의 job SALESMAN 를 가진 직원에 대해, 10%의 월급을 업데이트 하고, 수수료를 500 을 증가시킵니다.

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

### 3.4 옵티마이저

DELETE, SELECT 및 UPDATE 커맨드가 실행되면 Postgres Plus Advanced Server 데이터베이스 서버는 반환된 마지막 행 집합을 커맨드 결과 집합으로 생성하는 프로세스 제어가 돌아갑니다. 이러한 결과 집합을 생성하는 방법은 *쿼리 플래너 (query planner)*, 또는 *쿼리 옵티마이저*의 일입니다. 지정된 커맨드에 따라, 1 개 이상의 실행 후보가 존재합니다. 이를 *쿼리 계획(query plans)*이라고 합니다. 플래너는 결과 집합을 만들기 위한 가능한 방법을 검토합니다. 실제로 커맨드를 실행하는 데 사용되는 프로그램의 선택은 다양한 요구 사항에 따라 달라집니다.

- 데이터 검색을 위한 각종 처리 비용을 할당 (Postgresql.conf 파일 플래너 코스트 상수 참조)
- 플래너 메소드 인자 설정 (postgresql.conf 파일 플래너 메소드 설정 영역 참조)
- ANALYZE 커맨드는 테이블 데이터에 집계된 열 통계 정보 (ANALYZE 커맨드 및 열 통계 정보에 대한 자세한 내용은 *Postgres Plus* 설명서 참조)

대체로 다양한 실행 계획 중에서, 쿼리 플래너는 실제 실행에서 가장 작은 비용을 선택합니다.

그러나 주어진 DELETE, SELECT 및 UPDATE 커맨드에 옵티마이저 힌트를 사용하여 전체 또는 일부의 선택에 영향을 줄 수 있습니다. *옵티마이저 힌트는* DELETE, SELECT, 그리고 UPDATE 키워드를 따르는 코멘트와 비슷한 형식으로 플래너가 결과 집합을 생성하기 위해 특정한 접근을 이용할 수 있을 지를 알려줍니다.

**개요**

(DELETE | SELECT | UPDATE) / * + *(h int [comment])* [...] * /
*statement_body*

(DELETE | SELECT | UPDATE) - + *(h int [comment])* [...]
*statement_body*

옵티마이저 힌트는 위와 같이 2 가지 다른 형식으로 제공됩니다. 두 가지 형식에서 플러스 표시(+)는 / *나 -- 주석을 여는 기호는 힌트로서 사용하기 위해 공백이 없어야 합니다.

첫 번째 형식은 힌트와 옵션 코멘트가 여러 라인일 수 있습니다. 2 번째 형식은 모든 힌트와 옵션 코멘트가 하나의 라인이어야 합니다. 문장의 나머지는 새로운 라인의 시작해야만 합니다.

**설명**

옵티마이저 사용에 대해 다음과 같은 점에 주의해야 합니다.

- 데이터베이스 서버는 가능한 항상 지정된 힌트를 사용하려고 합니다.
- 플래너 메소드 인자가 특정한 계획 형식을 사용하지 못하도록 설정하면, 플래너가 다른 가능한 선택권이 전혀 없을 경우를 제외하고, 힌트에 지정되어도 이 계획은 사용되지 않습니다. 플래너 메소드 인자의 예제는 enable_indexscan, enable_seqscan, enable_hashjoin, enable_mergejoin 및 enable_nestloop 입니다. 이들은 모두 Boolean 인자입니다.
- 힌트는 주석으로 포함되어 있음을 주의하십시요. 결과적으로, 힌트가 오타이거나 인자가 힌트로서 뷰, 테이블, 열 이름이 오타일 경우, SQL 커맨드에 존재하지 않는 경우는 에러에 대한 통지가 발생하지 않습니다. 구문 오류가 없으면 모든 힌트는 간단하게 무시됩니다.
- SQL 커맨드에 테이블이나 뷰 이름을 위해 별칭을 사용하는 경우, 원래 이름이 아닌 힌트로서 사용되어야 합니다. 예를 들어, SELECT / * + FULL (acct) * / * FROM accounts acct ..., acct, 는 테이블 이름이 아닌 accounts 의 별칭으로 FULL 힌트에 지정되어야 합니다.
- 힌트가 정확하게 형성되었나 확인하고 싶거나, 플래너가 이 힌트를 사용하고 있는지 여부를 확인하기 위해서는 EXPLAIN 커맨드를 쓰십시오. *Postgres Plus 의* EXPLAIN 커맨드를 설명하는 문서를 참조하십시오.

- 일반적으로 옵티마이저 힌트는 실행 어플리케이션에서 사용되어야 하는 것은 아닙니다. 특히, 테이블 데이터는 어플리케이션 주기에 따라 변화합니다. 좀 더 동적인 열을 확실히 함으로써 ANALYZEd 가 수시로 이루어 지고, 열의 통계 정보는 값의 변화를 업데이트 할 것입니다. 또한 플래너는 주어진 커맨드를 실행하기 위한 최소한의 계획 비용을 생성하기 위해 이 정보를 사용합니다. 옵티마이저 힌트의 사용은 이러한 프로세스의 목적과 겹쳐지지 않으며 테이블 데이터가 어떻게 변화하는지를 간주하는 동일한 계획입니다.

**인자**

*hint*

Optimizer 힌트 지시.

*comment*

추가적인 정보 문자열. 어떤 문자가 주석에 포함되는가에 대해 제한이 있다는 것을 주의하십시오. 일반적으로 *comment 는* 알파벳, 숫자, 밑줄, 달러 기호, 번호 기호, 공백입니다. 이들은 식별자 구문을 준수해야 합니다. 식별자에 대한 자세한 정보는 3.1.2 절을 참조하십시오. 형식에 주석이 있지 않으면, 다음 힌트는 무시되어 질 것입니다.

*statement_body*

DELETE, SELECT, 혹은 UPDATE 커맨드의 나머지 부분

다음 절에서는 다양한 옵티마이저 힌트 지정에 대해 자세히 설명합니다.

### 3.4.1 기본 최적화 모드

Postgres Plus Advanced Server 데이터베이스 클러스터의 디폴트 설정은 많은 최적화 모드에서 선택될 수 있습니다. 이 설정은 옵티마이저 힌트 내의 DELETE, SELECT, UPDATE 커맨드 뿐 아니라, ALTER SESSION 커맨드를 사용하여 세션을 변경할 수 있습니다. 이러한 디폴트 모드를 제어하는 설정 매개 변수를 OPTIMIZER_MODE 이라고 합니다. 다음 표는 가능한 값을 나타냅니다.

**표 3-10 기본 최적화 모드**

| 힌트 | 설명 |
|---|---|
| ALL_ROWS | 모든 결과 집합을 검색하도록 최적화합니다 |
| CHOOSE | 결과 집합으로부터 검색된 가정 행 수를 기본으로 기본 최적화를 실행하지 않습니다. 이것이 기본값이다. |

| | |
|---|---|
| FIRST_ROWS | 결과 집합에서 첫 번째 행만을 검색하도록 최적화합니다. |
| FIRST_ROWS_10 | 결과 집합에서 처음 10 행을 검색하도록 최적화합니다. |
| FIRST_ROWS_100 | 결과 집합에서 처음 100 행을 검색하도록 최적화합니다. |
| FIRST_ROWS_100 0 | 결과 집합에서 처음 1000 행을 검색하도록 최적화합니다. |
| FIRST_ROWS *(n)* | 결과 집합에서 처음 n 줄을 검색하도록 최적화합니다. 이 형식은 ALTER SESSION SET OPTIMIZER_MODE 커맨드 개체로는 사용되지 않습니다. SQL 커맨드의 힌트 형태로만 사용됩니다. |

이러한 최적화 모드는 클라이언트가 발행한 SQL 커맨드가 결과 집합에서 첫 번째 "n"행만을 보고 싶으며, 나머지는 버린다는 가정을 기반으로 합니다. 쿼리에 할당되는 리소스는 다음과 같이 조정됩니다.

**예제**

현재 세션은 결과 집합의 처음 10 행을 검색하도록 최적화합니다.

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

OPTIMIZER_MODE 매개 변수의 현재 값은 SHOW 커맨드로 볼 수 있습니다. 이 커맨드는 유틸리티 의존 커맨드라는 것을 주의하십시오. PSQL 에서 SHOW 커맨드는 다음과 같습니다.

```
SHOW OPTIMIZER_MODE;


optimizer_mode
----------------
first_r ows_10
(1 row)
```

Oracle 과 호환되는 SHOW 커맨드는 다음과 같은 형식입니다.

```
SHOW PARAMETER OPTIMIZER_MODE;


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

다음은 SELECT 커맨드에 대한 힌트를 사용하여 최적화 모드를 보여줍니다.

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

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

### 3.4.2 액세스 방법 힌트

다음 힌트는 결과 집합을 생성하기 위해 옵티마이저가 어떻게 관계에 접근하는가에 영향을 줍니다.

**표 3-11 액세스 방법 힌트**

| 힌트 | 설명 |
|---|---|
| FULL *(table)* | *table 에* 대해 완전한 순차적인 검사 실시. |
| INDEX *(table [index]* [...]) | 관계에 접근하기 위하여 테이블에 인덱스를 사용한다. |
| NO_INDEX *(table [index]* [...]) | 관계에 접근하기 위하여 테이블에 인덱스를 사용하지 않는다. |

또한, Table 3-10 에서 보여주는 ALL_ROWS, FIRST_ROWS, FIRST_ROWS *(n)을* 사용할 수 있습니다.

**예제**

간단한 어플리케이션은 옵티마이저 힌트 효과를 설명하기 위한 충분한 자료가 없기 때문에, 이 절에서는 나머지 예에서는 Postgres Plus Advanced Server dbserver ₩ bin 하위 디렉터리에 위치한 pgbench 어플리케이션으로 생성되는 은행 데이터베이스를 사용합니다.

다음 단계는 accounts, branches, tellers, history, 테이블로 구성된 bank 라는 이름의 데이터베이스를 만듭니다. - s 5 옵션은 5 개 지점 생성에 따른 5 개의 크기 조정 요소를 지정합니다. 각각 100,000 계좌를 가지고, account 테이블에 총 500,000 행을 결과로 갖습니다. 또한, branches 테이블에 5 개의 행이 있습니다. 10 명의 텔러는 tellers 테이블로 각 지점마다 총 50 개의 행을 결과로 할당 받습니다.

만약 Linux 환경이라면, 다음 SET PATH 커맨드 대신, 아래의 export 커맨드를 사용하십시오.

```
export PATH = / opt/Postgres Plus Advanced Server/8.3/dbserver/bin : $ PATH
```

다음은 Windows 에서의 실행 예입니다.

```
SET PATH = C : ₩ Postgres Plus Advanced Server ₩ 8.3 ₩ dbserver ₩ bin; % PATH %

createdb - U Postgres Plus Advanced Server bank
CREATE DATABASE

pgbench - i - s 5 - U Postgres Plus Advanced Server - d bank

creating tables ...
10000 tuples done.
20000 tuples done.
30000 tuples done.
.
.
.
470000 tuples done.
480000 tuples done.
490000 tuples done.
500000 tuples done.
set primary key ...
vacuum ... done.
```

클라이언트 당 10 개의 트랜잭션은 8 명의 클라이언트를 위해 80 개의 트랜잭션으로 처리됩니다. 이는 history 테이블에 80 행으로 작성됩니다.

```
pgbench - U Postgres Plus Advanced Server - d bank - c 8 - t 10
```

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

테이블의 정의는 다음과 같습니다.

```
₩ d accounts

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

₩ d branches

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

₩ d tellers

Table "public.tellers"
Column | Type | Modifiers
----------+---------------+-----------
```

```
tid | integer | not null
bid | integer |
tbalance | integer |
filler | character (84) |
Indexes :
"tellers_pkey"PRIMARY KEY, btree (tid)

₩ d history

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

EXPLAIN 커맨드는 쿼리 플래너에서 선택한 프로그램을 표시합니다. 다음 예제에서, aid 는 주키입니다. 따라서, accounts_pkey 인덱스를 이용한 빠른 검색을 사용합니다.

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

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

FULL 힌트가 아래와 같이 인덱스를 사용하는 것을 대신해서, 완전 순차 스캔을 강요하는 경우는 다음과 같습니다.

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

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

NO_INDEX 힌트는 또한 순차 스캔을 아래와 같이 강요합니다.

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

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

위의 예제에서 보여준 EXPLAIN 커맨드를 사용 외에, 플래너에 의해 힌트가 사용되는지 여부에 대한 세한 정보는 client_min_messages 과 trace_hint 설정 매개 변수를 다음과 같이 설정함으로써 얻을 수 있습니다.

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

NO_INDEX 힌트를 가지는 SELECT 커맨드는 전에 기술한 설정 매개 변수를 지정할 때, 발생하는 추가 정보를 설명하기 위해 반복합니다.

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

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

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

만약 힌트가 무시되면, INFO: [HINTS] 라인이 표시되지 않습니다. 이것은 구문 오류나 힌트의 오타가 있음을 나타냅니다. 다음 예제는 인덱스 이름에 오타가 있습니다.

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

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

(2 rows)

### 3.4.3 조인 관계 힌트

2 개의 테이블을 조인하면, 조인을 실행하기 위해 가능한 3 개의 계획이 있습니다.

- *중첩 루프 조인* - 오른쪽 테이블은 왼쪽 테이블에서 모든 행이 한번 스캔 됩니다.
- *병합 정렬 조인* - 각 테이블을 조인 시작 전에, 조인 속성에 따라 정렬합니다. 두 테이블은 병렬로 스캔되고, 일치하는 행은 조인 행을 형성하기 위해 결합됩니다.
- 해시 조인 - 적당한 테이블이 스캔되고, 조인 속성은 해시 테이블로 조인 특성을 해시 키로 사용하여 로드됩니다. 그런 다음, 왼쪽 테이블은 스캔되고, 조인 속성은 우측 테이블로부터 일치하는 열을 찾기 위해 해시 키로서 사용됩니다.

다음 표는 플래너에 영향을 주기 위해 사용할 수 있는 옵티마이저 힌트를 조인 계획의 한 형식으로 사용하기 위해 열거합니다.

**표 3-12 조인 힌트**

| 힌트 | 설명 |
|---|---|
| USE_HASH *(table [...])* | 테이블 조인 특성에서 만든 해시 테이블에 의한 해시 조인을 사용합니다. |
| NO_USE_HASH *(table [...])* | 테이블 조인 특성에서 만든 해시 조인을 사용하지 않습니다. |
| USE_MERGE *(table [...])* | 테이블에 병합 정렬 조인을 사용합니다. |
| NO_USE_MERGE *(table [...])* | 테이블에 병합 정렬 조인을 사용하지 않습니다. |
| USE_NL *(table [...])* | 테이블에 중첩 루프 조인을 사용합니다. |
| NO_USE_NL *(table [...])* | 테이블에 중첩 루프 조인을 사용하지 않습니다. |

**예제**

다음 예제에서는 조인은 branches 과 accounts 테이블에서 실행됩니다. 쿼리 플랜은 branches 테이블의 조인 특성에서 해시 테이블을 생성하고, 해시 조인이 사용되는 것을 의미합니다.

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

QUERY PLAN
---------------------------------------------- -----------------------
Hash Join (cost = 1.11 .. 20092.70 rows = 500488 width = 12)
Hash Cond : (a.bid = b.bid)
```

```
-> Seq Scan on accounts a (cost = 0.00 .. 13209.88 rows = 500488 width = 12)
-> Hash (cost = 1.05 .. 1.05 rows = 5 width = 4)
-> Seq Scan on branches b (cost = 0.00 .. 1.05 rows = 5 width = 4)
(5 rows)
```

USE_HASH (a) 힌트를 사용하면 플래너가 branches 테이블이 아닌, accounts 조인 속성에서 해시 테이블을 만들도록 강요할 수 있습니다. USE_HASH 힌트는 accounts 테이블 별칭 a 를 사용하고 있다는 것을 명심하십시오.

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


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

다음으로, NO_USE_HASH (ab) 힌트는 플래너가 해시 테이블이 아닌 접근을 사용하도록 강요합니다. 결과적으로 중첩 루프입니다.

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


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

마지막으로 USE_MERGE 힌트는 플래너가 병렬 조인을 사용하도록 강요합니다.

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


QUERY PLAN
```

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

이 세 테이블의 조인 예에서, 플래너는 첫째, branches 와 history 테이블에서 해시 조인을 실행합니다. 그 다음 마지막으로, accounts 테이블의 accounts_pkey 인덱스를 가지는 결과의 중첩 루프 조인을 실행합니다.

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

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

이 플랜은 병합 정렬 조인과 해시 조인 결합해서 사용하도록 변경할 수 있습니다.

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

```
QUERY PLAN
---------------------------------------------- ----------------------------------------
Merge Join (cost = 23480.11 .. 23485.60 rows = 26 width = 20)
Merge Cond : (h.bid = b.bid)
-> Sort (cost = 23479.00 .. 23481.55 rows = 1020 width = 20)
Sort Key : h.bid
```

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

### 3.4.4 글로벌 힌트

여기까지, 힌트는 SQL 커맨드에 참조된 테이블을 직접적으로 참조했습니다. 뷰가 SQL 커맨드를 참조할 때, 뷰에 나타나는 테이블에 힌트를 적용할 수 있습니다. 힌트는 뷰에 나타나지 않고, 뷰에서 참조하는 SQL 커맨드에서 표현됩니다.

**서식**

*hint (view. table)*

**인수**

*hint*

표 3-11 또는 표 3-12 힌트.

*view*

*table 을* 포함하는 뷰 이름입니다.

*table*

힌트가 적용되는 테이블.

**예제**

3.4.3 절의 마지막 예제에서, history, branches, accounts 의 3 개 테이블 조인으로부터 tx 라는 뷰가 생성됩니다.

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

쿼리 플랜은 이 뷰로부터 선택됨으로써 실행되며, 다음과 같습니다.

```
EXPLAIN SELECT * FROM tx;

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

3.4.3 절의 마지막 조인에 적용되는 같은 힌트는 아래의 뷰에서도 적용될 수 있습니다.

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

QUERY PLAN

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

```
-> Sort (cost = 1.11 .. 1.12 rows = 5 width = 4)
Sort Key : b.bid
-> Seq Scan on branches b (cost = 0.00 .. 1.05 rows = 5 width = 4)
(12 rows)
```

저장된 뷰 내 테이블에 대한 힌트를 적용하고, 힌트는 다음 예제에서 설명하는 것처럼 하위 쿼리의 테이블에 적용될 수 있습니다. 간단한 어플리케이션의 emp 테이블 쿼리에서는 emp 테이블을 별칭 b 로 정의하고, emp 테이블의 하위쿼리와 조인하고, 직원과 관리자 목록을 보여줍니다.

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

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

이 계획은 다음의 쿼리 플래너에 의해 선택됩니다.

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

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

```
-> Sort (cost = 1.41 .. 1.44 rows = 14 width = 13)
Sort Key : emp.empno
-> Seq Scan on emp (cost = 0.00 .. 1.14 rows = 14 width = 13)
(8 rows)
```

힌트는 하위 쿼리 내의 emp 테이블에 테이블 스캔을 대신해서, emp_pk, 인덱스 스캔을 실행하기 위해 적용될 수 있습니다. 쿼리 플랜의 차이를 주의하십시오.

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


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

### 3.4.5 충돌 힌트

힌트를 다루는 마지막 절에서는 2 개 이상의 서로 충돌하는 SQL 커맨드에 대한 정보를 제공합니다. 이런 경우, 서로 충돌하는 힌트는 무시됩니다. 다음은 서로 상반되는 힌트의 목록입니다.

**표 3-13 충돌 힌트**

| 힌트 | 충돌 힌트 |
|---|---|
| ALL_ROWS | FIRST_ROWS - 전체 포맷 |
| FULL *(table)* | INDEX *(table [i ndex)* |
| INDEX *(table)* | FULL *(table)*<br><br>NO_INDEX *(table)* |
| INDEX *(table index)* | FULL *(table)*<br><br>NO_INDEX *(table index)* |
| USE_HASH *(table)* | NO_USE_HASH *(table)* |

| | |
|---|---|
| USE_MERGE *(table)* | NO_USE_MERGE *(table)* |
| USE_NL *(table)* | NO_USE_NL *(table)* |

### 3.5.1 논리 연산자

일반적인 논리 연산자를 사용할 수 있습니다. : AND, OR, NOT

SQL 은 3 값의 Boolean 연산 논리를 사용하고, 이 NULL 값은 "알 수 없음"을 의미합니다. 다음 진리표를 참조하십시오.

**Table 3-14 AND/OR 진리표**

| a | b | a AND b | a OR b |
|---|---|---|---|
| True | True | True | True |
| True | False | False | True |
| True | Null | Null | True |
| False | False | False | False |
| False | Null | False | Null |
| Null | Null | Null | Null |

**Table 3-15 NOT 진리표**

| a | NOT a |
|---|---|
| True | False |
| False | True |
| Null | Null |

AND 연산자 OR 연산자를 교환할 수 있습니다. 즉, 결과에 영향을 주지 않고 좌우 피연산자를 교환할 수 있습니다.

### 3.5.2 비교 연산자

아래 표의 일반적인 비교 연산자를 사용할 수 있습니다.

**Table 3-16 비교 연산자**

| 연산자 | 설명 |
|---|---|
| < | 더 작은 |
| > | 더 큰 |
| <= | 작거나 동일한 |
| >= | 크거나 동일한 |
| = | 동일 |
| <> | 동일하지 않은 |
| != | 동일하지 않은 |

비교 연산자는 그 의미가 통하면 모든 데이터 형식에서 사용할 수 있습니다. 모든 비교 연산자는 BOOLEAN 형태의 값을 반환하는 바이너리 연산자입니다. 1 < 2 < 3 와 같은 표현은 유효하지 않습니다.( 3 과 Boolean 값을 비교하는 < 연산자가 없기 때문입니다.)

비교 연산자 이외에, 특별한 BETWEEN 구문을 사용할 수 있습니다.

*a* BETWEEN *x* AND *y*

는 동일합니다.

*a* >= *x* AND *a* <= *y*

비슷하게

*a* NOT BETWEEN *x* AND *y*

는 동일합니다.

*a* < *x* OR *a* > *y*

첫 번째 방식을 두 번째 방식으로 내부적으로 갱신하는데 필요한 CPU 주기의 각각의 형식에는 차이가 없습니다.

값이 NULL 인지 아닌지 확인하려면 다음 구문을 사용합니다

*expression* IS NULL
*expression* IS NOT NULL

NULL 은 "equal to" NULL 와 같지 않기 때문에, *expression* = NULL 를 사용하지 마세요.(null 값은 알려지지 않은 값을 보여주고, 두 개의 알려지지 않은 값과의 동일여부를 알 수 없습니다.) 이 행동은 SQL 표준을 따릅니다.

어떤 애플리케이션은 *expression* 가 null 값을 평가하는 경우, *expression* = NULL 가 참을 반환할 것으로 예상할 수 있습니다. SQL 표준에 따르도록 애플리케이션이 변경되는 것을 추천합니다.

### 3.5.3 수학 함수와 연산자

수학 연산자는 여러 Postgres Plus Advanced Server 형태에서 제공됩니다. 가능한 모든 연산에 대한 일반적인 수학 조건이 없는 형식 (예를 들면, 날짜 / 시간 데이터 형식)은 후행 항목에서 실제 동작을 설명됩니다.

다음 표는 사용 가능한 수학 연산자를 보여줍니다.

**Table 3-17 수학 연산자**

| 연산자 | 설명 | 예 | 결과 |
|---|---|---|---|
| + | 가산 | 2 + 3 | 5 |
| - | 감산 | 2 – 3 | -1 |
| * | 곱셈 | 2 * 3 | 6 |
| / | 나눗셈 (정수 나누기 나머지 생략) | 4 / 2 | 2 |

다음 표는 사용 가능한 수학 함수를 보여줍니다. 이 중 여러 함수들은 다 인자 형태와 함께 여러 형태로 제공됩니다. 별다른 주의가 없으면, 함수의 주어진 형태는 인자와 같은 데이터 형태를 반환합니다. DOUBLE PRECISION 데이터를 사용하는 함수는 주로 호스트시스템의 C 라이브러리의 가장 상위에서 구현됩니다. 경계 부분의 정확성과 행동은 호스트 시스템에 의존하고 바뀝니다.

**Table 3-18 수학 함수**

| 함수 | 반환 형태 | 설명 | 예 | 결과 |
|---|---|---|---|---|
| ABS(*x*) | *X*와 동일 | 절대값 | ABS(-17.4) | 17.4 |
| CEIL(DOUBLE PRECISION or NUMBER) | 입력 형태와 동일 | 인자보다 작지 않은 최소 정수 | CEIL(-42.8) | -42 |
| EXP(DOUBLE PRECISION or | 입력 형태와 동일 | 지수 | EXP(1.0) | 2.7182818284590452 |

| | | | | |
|---|---|---|---|---|
| NUMBER) | | | | |
| FLOOR(DOUBLE PRECISION or NUMBER) | 입력 형태와 동일 | 인자보다 크지 않은 최대 정수 | FLOOR(-42.8) | 43 |
| LN(DOUBLE PRECISION or NUMBER) | 입력 형태와 동일 | 자연 로그 | LN(2.0) | 0.6931471805599453 |
| LOG(*b* NUMBER, *x* NUMBER) | NUMBER | *b* 기반 로그 | LOG(2.0, 64.0) | 6.0000000000000000 |
| MOD(*y*, *x*) | 인자 형태와 동일 | *y*/*x* 의 나머지 | MOD(9, 4) | 1 |
| NVL(*x*, *y*) | 인자 형태와 동일; 두 인자는 동일한 데이터 형태 | *x* 가 null 인 경우, NVL 는 y 를 반환 | NVL(9, 0) | 9 |
| POWER(*a* DOUBLE PRECISION, *b* DOUBLE PRECISION) | DOUBLE PRECISION | *a* 의 *b* 승 | POWER(9.0, 3.0) | 729.0000000000000000 |
| POWER(*a* NUMBER, *b* NUMBER) | NUMBER | *a* 의 *b* 승 | POWER(9.0, 3.0) | 729.0000000000000000 |
| ROUND(DOUBLE PRECISION or NUMBER) | 입력 형태와 동일 | 반올림 | ROUND(42.4) | 42 |
| ROUND(*v* NUMBER, *s* INTEGER) | NUMBER | *s* 자리에서 반올림 | ROUND(42.4382, 2) | 42.44 |
| SIGN(DOUBLE PRECISION or NUMBER) | 입력 형태와 동일 | 인자 기호(-1, 0, +1) | SIGN(-8.4) | -1 |
| SQRT(DOUBLE PRECISION or NUMBER) | 입력 형태와 동일 | 제곱근 | SQRT(2.0) | 1.414213562373095 |
| TRUNC(DOUBLE PRECISION or NUMBER) | 입력 형태와 동일 | 생략 | TRUNC(42.8) | 42 |

| | | | | |
|---|---|---|---|---|
| TRUNC(*v* NUMBER, *s* INTEGER) | NUMBER | s 자리에서 생략 | TRUNC(42.4382, 2) | 42.43 |
| WIDTH_BUCKET(*op* NUMBER, *b1* NUMBER, *b2* NUMBER, *count* INTEGER) | INTEGER | 피연산자가 할당한 *b1* 에서 *b2* 까지, 버킷 수 *count* 등등, 히스토그램의 버킷을 반환합니다. | WIDTH_BUCKET(5.35, 0.024, 10.06, 5) | 3 |

다음 표는 사용 가능한 삼각 함수를 보여줍니다. 삼각 함수는 인자를 취하고, DOUBLE PRECISION 형태의 값을 반환합니다.

**Table 3-19 삼각함수**

| 함수 | 설명 |
|---|---|
| ACOS(*x*) | 코사인 반비례 |
| ASIN(*x*) | 사인 반비례 |
| ATAN(*x*) | 탄젠트 반비례 |
| ATAN2(*x*, *y*) | x/y 의 탄젠트 반비례 |
| COS(*x*) | 코사인 |
| SIN(*x*) | 사인 |
| TAN(*x*) | 탄젠트 |

### 3.5.4 문자열 함수와 연산자

여기에서는 문자열 값을 검색하고 변경하는 함수와 연산자를 설명합니다. 이 컨텍스트의 문자열은 CHAR, VARCHAR2, CLOB 의 모든 형태의 값을 포함합니다. 별다른 주의가 없는 경우, 아래에 나열된 모든 함수는 이런 모든 형태를 작동시킵니다. CHAR 형태의 사용시, 자동 삽입되는 잠재적 결과에 주의해야 합니다. 일반적으로, 여기에 설명된 함수들은 문자열이 없는 형태의 데이터에도, 처음 문자열 표현으로 변환함으로써 작동합니다.

**Table 3-20 SQL 문자열 함수와 연산자**

| 함수 | 반환형태 | 설명 | 예 | 결과 |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| *string* || *string* | CLOB | 문자열 결합 | 'Enterprise' || 'DB' | EnterpriseDB |
| CONCAT(*string*, *string*) | CLOB | 문자열 결합 | 'a' || 'b' | ab |
| INSTR(*string*, *set*, [ *start* [, *occurrence* ] ]) | INTEGER | *String* 의 문자 집합*의* 위치를 찾고, *string* 문자열의 *start* 위치에서 시작하고, 첫 번째, 두 번째, 세 번째, 등 연속으로 집합의 발생 찾기 | INSTR('PETER PIPER PICKED UP A PACK OF PICKED PEPPERS','PI',1,3) | 33 |
| LOWER(*string*) | CLOB | *String* 을 소문자로 변환 | LOWER('TOM') | tom |
| SUBSTR(*string*, *start* [, *count* ]) | CLOB | *Start* 부터 시작하는 서브문자열을 추출하고, *count* 문자에 보낸다. *Count* 가 지정되지 않은 경우, 문자열은 시작부터 끝까지 생략됩니다. | SUBSTR('This is a test',6,2) | is |
| TRIM([ LEADING \| TRAILING \| BOTH ] [ *characters* ] FROM *string*) | CLOB | 문자만을 (기본값에 의한 공백) 포함하는 가장 긴 문자열을 문자열의 시작과 끝, 양쪽 끝에서 제거한다. | TRIM(BOTH 'x' FROM 'xTomxx') | Tom |
| LTRIM(*string* [, *set*]) | CLOB | *Set* 에 지정된 모든 문자를 주어진 *string* 의 왼쪽에서 제거한다. *set* 가 지정되지 않은 경우, 빈 공백은 기본값으로 사용됩니다. | LTRIM('abcdefghi', 'abc') | defghi |
| RTRIM(*string* [, *set*]) | CLOB | *Set* 에 지정된 모든 문자를 주어진 *string* 의 왼쪽에서 제거한다. *set* 가 지정되지 않은 경우, 빈 공백은 기본값으로 사용됩니다. | RTRIM('abcdefghi', 'ghi') | abcdef |
| UPPER(*string*) | CLOB | *string* 을 대문자로 전환 | UPPER('tom') | TOM |

추가적 문자열 변경 함수가 가능하며 다음의 표에 있습니다. 표 3-20 에 나열된 SQL 표준 문자열 함수의 구현을 위해 내부적으로 사용됩니다.

**Table 3-21 다른 문자열 함수**

| 함수 | 반환형태 | 설명 | 예 | 결과 |
|---|---|---|---|---|
| ASCII(*string*) | INTEGER | 인자의 첫 번째 바이트의 ASCII 코드 | ASCII('x') | 120 |
| CHR(INTEGER) | CLOB | 문자와 함께 주어진 ASCII 코드 | CHR(65) | A |
| DECODE(*expr*, *expr1a*, *expr1b* [, *expr2a*, *expr2b* ]... [, *default* ]) | Same as argument types of *expr1b*, *expr2b*,..., *default* | *expr1a*, *expr2a*, 등과 *Expr* 의 첫 번째 일치를 찾는다. 일치를 발견하면, expr1b, expr2b 등의 매개변수 쌍을 반환한다. 일치하는 것이 없으면 *default* 을 반환한다. 일치하지 않고, *default* 까지 지정되지 않으면 *null* 을 반환한다. | DECODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found') | Three |
| INITCAP(*string*) | CLOB | 각 단어의 첫 번째 글자를 대문자로, 나머지를 소문자로 변환한다. 여기서 단어는 숫자가 아닌 문자로 구분된 영문자와 숫자로 구성된 문자 행을 말한다. | INITCAP('hi THOMAS') | Hi Thomas |
| LPAD(*string*, *length* INTEGER [, *fill* ]) | CLOB | 문자 *fill* (기본값의 공간)앞에 추가하여 *string* 을 *length* 크기로 합니다. *string* 이 이미 *length* 의 길이를 초과하는 경우는 | LPAD('hi', 5, 'xy') | xyxhi |

| | | | | |
|---|---|---|---|---|
| | | 생략된다(오른쪽). | | |
| NVL(*expr1*, *expr2*) | 인자 형식과 동일; 두 인자는 동일한 형식 | *expr1* 가 null 이 아닌 경우에는 *expr1* 를, 그렇지 않으면 *expr2* 를 반환한다. | NVL(null, 'abc') | abc |
| REPLACE(*string*, *search_string* [, *replace_string* ] | CLOB | 문자열의 값을 다른 값과 교체한다. *replace_string* 값을 지정하지 않으면, *search_string* 값은 제거된다. | REPLACE( 'GEORGE', 'GE', 'EG') | EGOREG |
| RPAD(*string*, *length* INTEGER [, *fill* ]) | CLOB | 문자 *fill* (기본값의 공간)앞에 추가하여 *string* 을 *length* 크기로 합니다. *String* 가 이미 *length* 의 길이를 초과하는 경우는 생략된다. | RPAD('hi', 5, 'xy') | hixyx |
| TRANSLATE(*string*, *from*, *to*) | CLOB | *from* 집합의 문자와 일치하는 *string* 의 문자는 *to* 집합의 문자와 교체됩니다. | TRANSLATE('12345', '14', 'ax') | a23x5 |

### 3.5.5 LIKE 연산자 사용의 패턴 일치

Postgres Plus Advanced Server 는 전통적인 SQL LIKE 연산자의 사용과 일치하는 패턴을 제공합니다. LIKE 연산자 구문은 다음과 같습니다.

*string* LIKE *pattern* [ ESCAPE *escape-character* ]
*string* NOT LIKE *pattern* [ ESCAPE *escape-character* ]

모든 *pattern* 은 문자열 집합을 결정합니다. LIKE 표현은 *string* 이 *pattern* 에 의해 문자열 집합에 포함될 경우 참을 반환합니다. 예상하듯이, NOT LIKE 표현은 LIKE 이 참을 반환하거나 그 반대일 경우, 거짓을 반환합니다. 동일한 표현은 NOT 입니다. (*string* LIKE *pattern*).

*pattern* 이 백분율기호 또는 밑줄을 포함하지 않는 경우, 패턴은 오직 문자열만을 표시합니다. LIKE 의 경우, 동일한 연산자 작동을 합니다. *pattern* 의 underscore 밑줄(_)은 모든 문자를 의미하고, 백분율기호(%)는 0 또는 그 이상의 문자의 문자열과의 일치를 의미합니다.

예:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'     false
```

LIKE 패턴 일치는 항상 전체 문자열을 대상으로 합니다. 문자열 내의 패턴을 일치시키기 위해, 패턴은 백분율기호로 시작하고 끝나야 합니다.

다른 문자와 일치하지 않고, 리터럴 밑줄이나 백분율 기호와 일치시키기 위해, *pattern* 의 각 문자는 이스케이프 문자에 의해 선행되어야 합니다. ESCAPE 구문으로 다른 것을 선택할 수 있지만, 기본값 이스케이프 문자는 백슬래시 되어야 합니다.

백슬래시는 문자열 리터럴에 특별한 의미가 있음을 주의하세요. 따라서 백슬래시를 포함하는 패턴을 계속해서 작성하기 위해서는, SQL 명령문에 두 개의 백슬래시를 작성해야 합니다. 그러므로, 사실상 리터럴 백슬래시와 일치하는 패턴을 작성하는 것은 명령문에 네 개의 백슬래시를 작성하는 것을 의미합니다. ESCAPE 와 함께, 다른 이스케이프 문자를 선택함으로써 피할 수 있습니다. 백슬래시는 더 이상 LIKE 에 있어서 특별한 것이 아닙니다.(그러나, 여전히 문자열 리터럴 파서에 특별하므로, 두 개의 백슬래시가 여전히 필요합니다.)

ESCAPE ''을 작성하는 것으로 이스케이프 문자를 선택하지 않는 것이 가능합니다. 효과적으로 이스케이프 기법이 작동하지 않게 됩니다. 패턴에서의 밑줄과 백분율 기호의 특별한 의미를 취소할 수 없게 합니다.

### 3.5.7 날짜/시간 함수와 연산자

Table 3-28 는 다음의 서브섹션에 나타난 세부사항과 함께, 날짜/시간 값 프로세스가 가능한 함수를 보여줍니다. Table 3-27 는 기본 산술 연산자(+, -)의 행동을 설명합니다. 함수의 형식화를 위해, 3.5.6 섹션을 참조합니다. 3.2.4 섹션으로부터 날짜/시간 데이터 형식의 배경 정보에 익숙해야 합니다.

**Table 3-27 날짜/시간 연산자**

| 연산자 | 예 | 결과 |
|---|---|---|
| + | DATE '2001-09-28' + 7 | 05-OCT-01 00:00:00 |
| + | TIMESTAMP '2001-09-28 13:30:00' + 3 | 01-OCT-01 13:30:00 |
| - | DATE '2001-10-01' - 7 | 24-SEP-01 00:00:00 |
| - | TIMESTAMP '2001-09-28 13:30:00' - 3 | 25-SEP-01 13:30:00 |
| - | TIMESTAMP '2001-09-29 03:00:00' - TIMESTAMP '2001-09-27 12:00:00' | @ 1 day 15 hours |

Table 3-28 의 날짜/시간 함수에서 DATE 와 TIMESTAMP 데이터 형식의 사용은 교환 가능합니다.

**Table 3-28 날짜/시간 함수**

| 함수 | 반환형식 | 설명 | 예 | 결과 |
|---|---|---|---|---|
| ADD_MONTHS(DATE, NUMBER) | DATE | 날짜에 월을 더한다; 3.5.7.1 참조 | ADD_MONTHS('28-FEB-97', 3.8) | 31-MAY-97 00:00:00 |
| CURRENT_DATE | DATE | 현재 날짜; 3.5.7.7 참조 | CURRENT_DATE | 04-JUL-07 |
| EXTRACT(*field* FROM TIMESTAMP) | DOUBLE PRECISION | 서브필드 추출; 3.5.7.2 참조 | EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40') | 20 |
| LAST_DAY(DATE) | DATE | 주어진 날짜에 의해 달의 마지막 날을 반환한다. 주어진 날짜가 시간부분을 포함하면, 결과는 변경이 없이 이월됩니다. | LAST_DAY('14-APR-98') | 30-APR-98 00:00:00 |
| LOCALTIMESTAMP [ (*precision*) ] | TIMESTAMP | 현재 날짜와 시간 (현재 트랜잭션의 시작); 3.5.7.7 참조 | LOCALTIMESTAMP | 04-JUL-07 15:33:23.484 |
| MONTHS_BETWEEN(DATE, DATE) | NUMBER | 두 날짜 사이 달의 수; 3.5.7.3 참조 | MONTHS_BETWEEN('28-FEB-07', '30-NOV-06') | 3 |
| NEXT_DAY(DATE, *dayofweek*) | DATE | 주어진 날짜를 뒤따르는 *dayofweek*와 일치하는 날짜 3.5.7.4 참조 | NEXT_DAY('16-APR-07','FRI') | 20-APR-07 00:00:00 |
| ROUND(DATE [, *format* ]) | DATE | *format*에 따라 반올림된 날짜; 3.5.7.5 참조 | ROUND(TO_DATE('29-MAY-05'), 'MON') | 01-JUN-05 00:00:00 |
| SYSDATE | DATE | 현재 날짜와 시간 | SYSDATE | 06-AUG-07 10:06:27 |
| TRUNC(DATE [, *format* ]) | DATE | *format*에 따라 생략; 3.5.7.6 참조 | TRUNC(TO_DATE('29-MAY-05'), 'MON') | 01-MAY-05 00:00:00 |

### 3.5.7.1 ADD_MONTHS

ADD_MONTHS 함수는 주어진 날짜에 월의 지정된 수를 가산합니다. (또는 두 번째 매개변수가 부일 경우 감산합니다.) 월의 마지막 날을 제외하고, 그 달의 날짜 결과는 주어진 날짜와 동일합니다. 이 경우에는 항상 그 달의 결과 날은 월의 마지막 날입니다.

월 매개변수의 수의 일부분은 계산이 수행되기 전에 생략됩니다.
주어진 날짜가 시간 부분을 포함하면, 변경되지 않은 결과를 유지합니다.
다음은 ADD_MONTHS 함수의 예입니다.

```
SELECT ADD_MONTHS('13-JUN-07',4) FROM DUAL;
```

```
    add_months
--------------------
 13-OCT-07 00:00:00
(1 row)

SELECT ADD_MONTHS('31-DEC-06',2) FROM DUAL;

    add_months
--------------------
 28-FEB-07 00:00:00
(1 row)

SELECT ADD_MONTHS('31-MAY-04',-3) FROM DUAL;

    add_months
--------------------
 29-FEB-04 00:00:00
(1 row)
```

#### 3.5.7.2 EXTRACT

EXTRACT 함수는 날짜/시간 값으로부터 년이나 시간과 같은 서브필드를 추출합니다. EXTRACT 함수는 DOUBLE PRECISION 형의 값을 반환합니다. 다음은 유효 필드명 입니다.

YEAR

년 필드

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
     2001
(1 row)
```

MONTH

년 내 월의 수(1 - 12)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
        2
(1 row)
```

DAY

(월의) 날짜 필드 (1 - 31)

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

```
 date_part
-----------
       16
(1 row)
```

HOUR

시간 필드(0 - 23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
       20
(1 row)
```

MINUTE

분 필드(0 - 59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
       38
(1 row)
```

SECOND

분수 부분을 포함한, 초 필드 (0 - 59)

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
       40
(1 row)
```

#### 3.5.7.3 MONTHS_BETWEEN

MONTHS_BETWEEN 함수는 두 날짜 사이의 달 수를 반환합니다. 첫 번째 데이터가 두 번째 데이터보다 큰 경우는 정, 첫 번째 데이터가 더 두 번째 데이터보다 적은 경우는 부로 결과는 숫자 값입니다.

결과는 두 날짜 매개변수의 날이 동일하거나, 두 날짜 매개변수가 각 달의 마지막 날인 경우 항상 모든 달의 수 입니다.

다음은 MONTHS_BETWEEN 함수의 예입니다.

```
SELECT MONTHS_BETWEEN('15-DEC-06','15-OCT-06') FROM DUAL;
```

```
 months_between
----------------
              2
(1 row)

SELECT MONTHS_BETWEEN('15-OCT-06','15-DEC-06') FROM DUAL;

 months_between
----------------
             -2
(1 row)

SELECT MONTHS_BETWEEN('31-JUL-00','01-JUL-00') FROM DUAL;

 months_between
----------------
    0.967741935
(1 row)

SELECT MONTHS_BETWEEN('01-JAN-07','01-JAN-06') FROM DUAL;

 months_between
----------------
             12
(1 row)
```

#### 3.5.7.4 NEXT_DAY

NEXT_DAY 함수는 주어진 날짜보다 더 큰 지정된 평일의 첫 번째 발생을 반환합니다. 예를 들어, SAT 와 같이, 최소한 평일의 첫 세 글자가 지정되어야 합니다. 주어진 날짜는 시간 부분을 포함하고, 변하지 않는 결과를 유지합니다.

다음은 NEXT_DAY 함수의 예입니다.

```
SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'SUNDAY') FROM DUAL;

     next_day
---------------------
 19-AUG-07 00:00:00
(1 row)

SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'MON') FROM DUAL;

     next_day
---------------------
 20-AUG-07 00:00:00
(1 row)
```

#### 3.5.7.5 ROUND

ROUND 함수는 지정된 템플릿 양식에 따라 날짜를 반환합니다. 템플릿 양식이 생략된 경우, 날짜는 가장 가까운 날에 반올림 됩니다. 다음의 표는 ROUND 함수의 템플릿 양식을 보여줍니다.

**Table 3-29 함수의 템플릿 날짜 양식**

| 양식 | 설명 |
|---|---|
| CC, SCC | *cc*01 년 1 월 1 일을 반환합니다. *Cc* 는 마지막 두 숫자가 <= 50 이거나, 첫 번째 두 숫자가 1 많은 마지막 두 숫자가 > 50 이면, 주어진 년도의 처음 두 숫자를 나타냅니다. (서기 년) |
| SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y | *yyyy* 년 1 월 1 일을 반환합니다. Yyyy 는 6 월 30 일 이전, 7 월 1 일 이후, 최근 년으로 반올림됩니다. |
| IYYY, IYY, IY, I | ISO 년의 시작을 반올림합니다. 이것은 월과 일이 6 월 30 일 이전, 또는 7 월 1 일 이후, 반올림에 의해 결정됩니다. |
| Q | 월과 일이 두 번째 분기의 15 일이나 그 이전, 또는 두 번째 분기의 16 일이거나 그 이후, 반올림에 따라 분기의 첫 날을 반환합니다. |
| MONTH, MON, MM, RM | 날이 15 일이나 그 이전이면, 지정된 달의 첫 날을, 16 일이나 그 이후이면, 그 다음 달의 첫 날을 반환합니다. |
| WW | 해당 년도의 첫날과 같은, 한 주의 동일한 최근 날짜를 반올림합니다. |
| IW | ISO 년의 첫날과 같은, 한 주의 요일과 동일한 최근 날짜를 반올림합니다. |
| W | 해당 월의 첫날과 같은, 한 주의 동일한 최근 날짜를 반올림합니다. |
| DDD, DD, J | 최근 날짜의 시작을 반올림합니다. 11:59:59 AM 또는 더 이른 경우 동일한 날짜의 시작을 반올림합니다. 12:00:00 PM 또는 더 늦은 경우 그 다음 날의 시작을 반올림합니다. |
| DAY, DY, D | 최근 일요일을 반올림합니다. |
| HH, HH12, HH24 | 최근 시를 반올림합니다. |
| MI | 최근 분을 반올림합니다. |

ROUND 함수 사용법의 예입니다.

다음은 최근 100 년에서 반올림하는 예입니다.

```
SELECT TO_CHAR(ROUND(TO_DATE('1950','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM
DUAL;

   Century
-------------
 01-JAN-1901
(1 row)
```

```
SELECT TO_CHAR(ROUND(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM
DUAL;

   Century
-------------
 01-JAN-2001
(1 row)
```

최근 년의 반올림 예입니다.

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY')
"Year" FROM DUAL;

    Year
-------------
 01-JAN-1999
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY')
"Year" FROM DUAL;

    Year
-------------
 01-JAN-2000
(1 row)
```

다음은 가장 최근 ISO 년 반올림의 예입니다. 첫 번째 예는 2004 년에 반올림 합니다. 2004 의 ISO 년은 2003 년 12 월 29 일에 시작합니다. 두 번째 예는 2005 년에 반올림합니다. 같은 년도의 1 월 3 일에 ISO 년을 시작합니다.

(ISO 년은 신년의 월요일에서 일요일까지 중 최소 4 일을 포함하는, 7 일 단위의 첫 번째 월요일부터 시작합니다. 따라서, 전년도의 12 월 ISO 년부터 시작할 수 있습니다.)

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY')
"ISO Year" FROM DUAL;

  ISO Year
-------------
 29-DEC-2003
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY')
"ISO Year" FROM DUAL;

  ISO Year
-------------
 03-JAN-2005
(1 row)
```

다음은 최근 분기의 반올림 예입니다.

```
SELECT ROUND(TO_DATE('15-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

     Quarter
```

```
--------------------
 01-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

     Quarter
--------------------
 01-APR-07 00:00:00
(1 row)
```

다음은 최근 달의 반올림 예입니다.

```
SELECT ROUND(TO_DATE('15-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

      Month
--------------------
 01-DEC-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

      Month
--------------------
 01-JAN-08 00:00:00
(1 row)
```

다음은 최근 ISO 주에 반올림합니다. 첫 번째 경우는 2007 년 첫 날은 월요일입니다. 1 월 18 일에 가까운 일요일은 1 월 15 일입니다. 두 번째 예에서는 1 월 19 일은 1 월 22 일에 가까운 월요일입니다.

```
SELECT ROUND(TO_DATE('18-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

       Week
--------------------
 15-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

       Week
--------------------
 22-JAN-07 00:00:00
(1 row)
```

다음은 최근 ISO 주에 반올림합니다. ISO 주 월요일부터 시작됩니다. 첫 번째 경우는 2004 년 1 월 1 일은 2003 년 12 월 29 일에 가장 가까운 월요일입니다. 두 번째 예에서는 2004 년 1 월 2 일은 2004 년 1 월 4 일에 가장 가까운 월요일입니다,

```
SELECT ROUND(TO_DATE('01-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

     ISO Week
--------------------
 29-DEC-03 00:00:00
```

```
(1 row)

SELECT ROUND(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

     ISO Week
--------------------
 05-JAN-04 00:00:00
(1 row)
```

다음 예는 월의 첫 번째 날처럼 같은 날에 시작하는 것으로 간주되는 한 주의 가장 가까운 주를 반올림합니다.

```
SELECT ROUND(TO_DATE('05-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

       Week
--------------------
 08-MAR-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('04-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

       Week
--------------------
 01-MAR-07 00:00:00
(1 row)
```

다음 예는 최근 날의 반올림입니다.

```
SELECT ROUND(TO_DATE('04-AUG-07 11:59:59 AM','DD-MON-YY HH:MI:SS AM'),'J') "Day"
FROM DUAL;

       Day
--------------------
 04-AUG-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J') "Day"
FROM DUAL;

       Day
--------------------
 05-AUG-07 00:00:00
(1 row)
```

다음 예는 주의 최근 날 시작의 반올림입니다.

```
SELECT ROUND(TO_DATE('08-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

   Day of Week
--------------------
 05-AUG-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;
```

```
   Day of Week
--------------------
 12-AUG-07 00:00:00
(1 row)
```

다음 예는 최근 시간의 반올림입니다.

```
SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:29','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY
HH24:MI:SS') "Hour" FROM DUAL;

       Hour
--------------------
 09-AUG-07 08:00:00
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY
HH24:MI:SS') "Hour" FROM DUAL;

       Hour
--------------------
 09-AUG-07 09:00:00
(1 row)
```

다음 예는 최근 분의 반올림입니다.

```
SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:29','DD-MON-YY HH:MI:SS'),'MI'),'DD-
MON-YY HH24:MI:SS') "Minute" FROM DUAL;

      Minute
--------------------
 09-AUG-07 08:30:00
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY HH:MI:SS'),'MI'),'DD-
MON-YY HH24:MI:SS') "Minute" FROM DUAL;

      Minute
--------------------
 09-AUG-07 08:31:00
(1 row)
```

#### 3.5.7.6 TRUNC

TRUNC 함수는 지정된 템플릿 양식을 따른 날짜를 반환합니다. 템플릿 양식이 생략된 경우, 날짜는 최근 날은 생략됩니다. 다음 표는 TRUNC 함수의 템플릿 양식을 보여줍니다.

**Table 3-30 TRUNC 함수의 템플릿 날짜 양식**

| 양식 | 설명 |
|---|---|
| CC, SCC | *cc*01 년 1 월 1 일을 반환하고, 주어진 년도의 첫 두 숫자는 *cc* 입니다. |
| SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y | *yyyy* 년 1 월 1 일을 반환하고, *yyyy* 는 주어진 년 입니다. |

| | |
|---|---|
| IYYY, IYY, IY, I | 주어진 날짜를 포함하는 ISO 년의 시작 날짜를 반환합니다. |
| Q | 주어진 날짜를 포함하는 분기의 첫날을 반환합니다. |
| MONTH, MON, MM, RM | 지정된 달의 첫날을 반환합니다. |
| WW | 그 년도의 첫날과 같은, 한 주의 주어진 날짜와 동일하거나, 이전의 가장 늦은 날짜를 반환합니다. |
| IW | 주어진 날짜를 포함하는 ISO 주의 시작을 반환합니다. |
| W | 그 달의 첫날과 같은, 한 주의 주어진 날짜와 동일하거나, 이전의 가장 늦은 날짜를 반환합니다. |
| DDD, DD, J | 주어진 날짜의 날의 시작을 반환합니다. |
| DAY, DY, D | 주어진 날짜를 포함하는 한 주(일요일)의 시작을 반환합니다. |
| HH, HH12, HH24 | 시의 시작을 반환합니다. |
| MI | 분의 시작을 반환합니다. |

다음은 TRUNC 함수 사용법의 예입니다.

다음 예는 100 년 단위로 생략됩니다.

```
SELECT TO_CHAR(TRUNC(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM
DUAL;

   Century
-------------
 01-JAN-1901
(1 row)
```

다음 예는 년이 생략됩니다.

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY')
"Year" FROM DUAL;

    Year
-------------
 01-JAN-1999
(1 row)
```

다음 예는 ISO 년의 시작이 생략됩니다.

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY')
"ISO Year" FROM DUAL;

  ISO Year
-------------
 29-DEC-2003
(1 row)
```

다음 예는 분기의 시작 날짜가 생략됩니다.

```
SELECT TRUNC(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;
```

```
     Quarter
---------------------
 01-JAN-07 00:00:00
(1 row)
```

다음 예는 월의 시작이 생략됩니다.

```
SELECT TRUNC(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

      Month
---------------------
 01-DEC-07 00:00:00
(1 row)
```

다음 예는 년의 첫 번째 날에 의해 결정되는 한 주의 시작을 생략합니다. 2007 년의 첫 번째 날은 월요일이고, 1 월 19 일 이전 월요일은 1 월 15 일입니다.

```
SELECT TRUNC(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

       Week
---------------------
 15-JAN-07 00:00:00
(1 row)
```

다음 예는 ISO 주의 시작을 생략합니다. ISO 주는 월요일로 시작합니다. 2004 년 1 월 2 일은 2003 년 12 월 29 일 월요일에 시작한 ISO 주입니다.

```
SELECT TRUNC(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

     ISO Week
---------------------
 29-DEC-03 00:00:00
(1 row)
```

다음 예는, 월의 첫 번째 날로써, 같은 날 시작한 것으로 간주되는 한 주의 시작은 생략합니다.

```
SELECT TRUNC(TO_DATE('21-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

       Week
---------------------
 15-MAR-07 00:00:00
(1 row)
```

다음 예는 날의 시작을 생략합니다.

```
SELECT TRUNC(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J') "Day"
FROM DUAL;

       Day
---------------------
 04-AUG-07 00:00:00
(1 row)
```

다음 예는 한 주의 시작(일요일)을 생략합니다.

```
SELECT TRUNC(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

   Day of Week
--------------------
 05-AUG-07 00:00:00
(1 row)
```

다음 예는 시간의 시작을 생략합니다.

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY
HH24:MI:SS') "Hour" FROM DUAL;

       Hour
--------------------
 09-AUG-07 08:00:00
(1 row)
```

다음 예는 분을 생략합니다.

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY HH:MI:SS'),'MI'),'DD-
MON-YY HH24:MI:SS') "Minute" FROM DUAL;

      Minute
--------------------
 09-AUG-07 08:30:00
(1 row)
```

#### 3.5.7.7 CURRENT DATE/TIME

Postgres Plus Advanced Server 는 현재 날짜와 시간과 관련된 값을 반환하는 함수의 수를 제공합니다. 이러한 함수는 현재 트랜잭션의 시작 시간을 기준으로 값을 반환합니다.

- CURRENT_DATE
- LOCALTIMESTAMP
- LOCALTIMESTAMP(*precision*)
- SYSDATE

LOCALTIMESTAMP 는 정밀 매개변수를 선택적으로 제공할 수 있고, 두 번째 필드의 반올림된 여러 부분적 숫자의 결과를 초래합니다.

```
SELECT CURRENT_DATE FROM DUAL;

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

SELECT LOCALTIMESTAMP FROM DUAL;
```

```
      timestamp
------------------------
 06-AUG-07 16:11:35.973
(1 row)

SELECT LOCALTIMESTAMP(2) FROM DUAL;

      timestamp
-----------------------
 06-AUG-07 16:11:44.58
(1 row)

SELECT SYSDATE FROM DUAL;

    timestamp
--------------------
 06-AUG-07 16:11:48
(1 row)
```

이러한 함수는 현재 트랜잭션의 시작 시간을 반환하기 때문에, 그 값들은 트랜잭션 동안에 변화하지 않습니다. 다음과 같은 특성을 고려합니다. '현재' 시간의 일관성 있는 표현이 가능한 단일 트랜잭션을 허용하는 의도를 가지며, 동일 트렌잭션 내의 다수의 변경은 동일 타임스템프를 생성합니다.  다른 데이터베이스 시스템은 이러한 값을 수시로 증가시킵니다.

### 3.5.8 시퀀스 조작 함수

여기서는 시퀀스 객체 연산을 위한 Postgres Plus Advanced Server's 의 함수를 설명합니다. 시퀀스 객체(시퀀스 생성자 또는 시퀀스라고 불리는)는 CREATE SEQUENCE 명령으로 생성된 특별한 한 개 행 테이블입니다. 시퀀스 객체는 보통 테이블의 행의 유일한 식별자를 생성하기 위해 사용됩니다. 아래에 나열된 시퀀스 함수는 시퀀스 객체로부터 연속되는 시퀀스 값을 획득하는 멀티유저-보안 방법을 단순히 제공합니다.

*sequence*.NEXTVAL
*sequence*.CURRVAL

*Sequence* 는 CREATE SEQUENCE 명령의 시퀀스에 할당된 식별자입니다. 다음은 이러한 함수의 사용법을 설명합니다.

NEXTVAL

시퀀스 객체는 그 다음 값을 추진하고, 그 값을 반환합니다. 자동적으로 행해집니다: 다수 세션이 NEXTVAL 을 동시에 생성할 경우에만, 안전하게 뚜렷한 각 시퀀스 값을 받을 수 있습니다.

CURRVAL

현재 세션에서 이 시퀀스의 가장 최근에 NEXTVAL 에 의해 획득한 값을 돌려줍니다. (이 세션에서 시퀀스의 NEXTVAL 가 불린 적이 없는 경우, 에러가 보고됩니다.) 세션-로컬 값을 반환하기 때문에, 현재 세션에서 그래왔듯이, 다른 세션이 NEXTVAL 의 실행여부에 대해서 예상 가능한 답을 주는 것에 주의하세요.

시퀀스 객체는 기본값 매개변수와 함께 생성되는 경우, NEXTVAL 는 1 로 시작하는 연속적인 값을 반환합니다. 기타 동작으로 CREATE SEQUENCE 명령에서 특수한 매개변수를 사용하여 얻을 수 있습니다.

**중요사항**: 동시 트랜잭션의 블록을 막기 위해, 같은 시퀀스로부터 수를 검색합니다. NEXTVAL 연산은 롤백하지 않습니다. 즉, 한번 값이 나오면, 트랜잭션이 나중에 NEXTVAL 을 중단한다고 해도, 사용된 것으로 간주합니다. 이것은 중단된 트랜잭션이 할당된 값의 시퀀스에 사용된 '구멍'을 남기는 것을 의미합니다.

### 3.5.9 조건문 표현식

여기에서는 Postgres Plus Advanced Server 에서 사용 가능한 SQL-준수 구문에 대해 설명합니다.

#### 3.5.9.1 CASE

SQL CASE 식은 다른 언어에서의 if/else 명령문과 유사한 일반 조건문 표현식입니다,

```
CASE WHEN condition THEN result
   [ WHEN ... ]
   [ ELSE result ]
END
```

CASE 구는 식이 유효하면 어디서나 사용할 수 있습니다. *condition* 은 boolean 형식의 결과를 반환하는 식입니다. 만약 결과가 참이라면 CASE 표현식은 *result* 입니다. 만일 후속 WHEN 구의 결과가 거짓인 경우, 동일한 방법으로 점검합니다. WHEN 의 *condition* 이 참이면 아니면, CASE 표현식은 ELSE 구의 *result* 입니다. ELSE 구가 어떤 조건과 일치하지 않으면, 결과는 NULL 입니다.

다음은 예입니다.:

```
SELECT * FROM test;

 a
---
 1
 2
 3
(3 rows)
```

```
SELECT a,
   CASE WHEN a=1 THEN 'one'
        WHEN a=2 THEN 'two'
        ELSE 'other'
   END
FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

모든 *result* 표현식의 데이터 형식은 하나의 출력 형식에 바꿀 수 있습니다.

다음의 '단순한' CASE 표현식은 일반적 형식의 특화된 변형입니다.

```
CASE expression
    WHEN value THEN result
  [ WHEN ... ]
  [ ELSE result ]
END
```

*Expression* 은, 동등한 하나를 찾을 때까지, WHEN 구에 지정된 모든 값을 계산하고 비교합니다. 동등한 것이 없으면, ELSE 구(또는 null 값)의 *result* 가 반환됩니다.

단순한 CASE 구문을 사용해 위의 예가 작성될 수 있습니다.

```
SELECT a,
   CASE a WHEN 1 THEN 'one'
          WHEN 2 THEN 'two'
          ELSE 'other'
   END
FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

CASE 표현식은 결과를 결정하기 위해 필요하지 않은 서브표현식을 평가하지 않습니다. 예를 들어, 0 으로 나누기 오류를 방지하기 위한 방법입니다.

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

#### 3.5.9.2 COALESCE

COALESCE 함수는 null 이 아닌 첫 번째 인자를 반환합니다. Null 은 인자가 null 인 경우에만 반환됩니다.

COALESCE(*value* [, *value2* ] ... )

데이터를 표시에 검색될 때, null 값 대신 기본값이 사용됩니다. 예를 들면:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

CASE 식 처럼, COALESCE 는 결과를 결정하는데 필요하지 않은 인자는 평가하지 않습니다. 즉, 첫 번째 null 이 아닌 인자의 오른쪽 인자는 평가되지 않습니다. SQL 표준 함수는 to NVL 과 IFNULL 의 비슷한 기능을 제공하고, 다른 일부 데이터 베이스 시스템에서 사용됩니다..

#### 3.5.9.3 NULLIF

NULLIF 함수는 *value1* 와 *value2* 가 동일한 경우, null 값을 반환합니다. 그렇지 않으면, *value1* 을 반환합니다.

NULLIF(*value1*, *value2*)

주어진 위의 COALESCE 예의 역연산 수행에 사용됩니다.

```
SELECT NULLIF(value1, '(none)') ...
```

*value1* 이 (none) 이면 *null* 을 반환하고, 그렇지 않으면 *value1* 을 반환합니다.

#### 3.5.9.4 GREATEST and LEAST

GREATEST 와 LEAST 함수는 표현식의 수 목록으로부터 가장 크거나 가장 작은 값을 선택합니다.

GREATEST(*value* [, *value2* ] ... )
LEAST(*value* [, *value2* ] ... )

표현식은 결과의 형식이 되는 보통의 데이터 형식을 변환할 수 있어야 합니다. 목록 내의 Null 값은 무시됩니다. 결과는 모든 식이 null 로 평가될 경우에만 null 이 됩니다.

GREATEST 와 LEAST 는 SQL 표준에는 없지만, 일반적인 확장인 것에 주의하세요.

### 3.5.10 집계함수

집계 함수는 여러 입력 값으로부터 단일 결과를 계산합니다. 다음 표는 내장된 집계 함수를 보여줍니다.

**Table 3-31 일반 목적의 집계 함수**

| 함수 | 인자 데이터 형식 | 반환 형식 | 설명 |
|---|---|---|---|
| AVG(*expression*) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | 정수 인자는 NUMBER, 부동 소수점 인자는 DOUBLE PRECISION, 나머지는 인자 데이터 형식과 동일 | 모든 입력 값의 평균<br>(산술평균) |
| COUNT(*) | | BIGINT | 입력 행의 수 |
| COUNT(*expression*) | Any | BIGINT | 식이 null 이 아닌 값을 가지는 입력 행의 수 |
| MAX(*expression*) | 숫자, 문자열, 또는 날짜/시간 형식 | 인자형식과 동일 | 모든 입력 값 간의 식의 최대값 |
| MIN(*expression*) | 숫자, 문자열, 또는 날짜/시간 형식 | 인자형식과 동일 | 모든 입력 값 간의 식의 최소값 |
| SUM(*expression*) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | SMALLINT 또는 INTEGER 형식의 인자는 BIGINT, BIGINT 형식의 인자는 NUMBER, 부동 소수점 인자는 DOUBLE PRECISION, 나머지는 인자는 데이터 형식과 동일 | 모든 입력 값 간의 식의 합계 |

이 함수는 COUNT 함수를 제외하고, 1 행을 선택하지 않은 경우 NULL 값을 반환하는 것을 주의하십시오. 특히, SUM 의 행은 0 이 아닌, null 을 반환하지 않습니다. 필요한 경우 null 을 0 으로 대체하기 위해 COALESCE 함수를 사용할 수 있습니다.

통계 분석 작업에 자주 사용되는 집계 함수가 다음 표에 나와있습니다. (주로 일반적으로 사용되는 집계의 목록의 혼란을 피하기 위해 분리됩니다.) 설명 부분에서 *N*은, 모든 입력식이 null 이 아닌 입력 행의 개수를 나타냅니다. 모든 경우에서, 예를 들면 *N*이 0 인 경우와 같이, 계산이 무의미한 경우에 null 이 반환됩니다.

**Table 3-32 통계에 대한 집계함수**

| 함수 | 인자형식 | 반환형식 | 설명 |
|---|---|---|---|
| CORR(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | 상관계수 |
| COVAR_POP(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | 모집단 공분산 |
| COVAR_SAMP(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | 표본 공분산 |
| REGR_AVGX(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | 독립 변수의 평균 (sum(*X*) / *N*) |
| REGR_AVGY(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | 종속 변수의 평균 (sum(*Y*) / *N*) |
| REGR_COUNT(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | null 이 아닌 두 식의 입력 행의 수 |
| REGR_INTERCEPT(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | *(X, Y)*의 조합에 의해 결정된 선형 방정식에 대한 최소 두 곱셈 y 역수 |
| REGR_R2(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | 상관계수의 제곱 |
| REGR_SLOPE(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | *(X, Y)*의 조합에 의해 결정된 최소 두 곱셈에 맞는 선형 방정식의 기울기 |
| REGR_SXX(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | Sum ($X^2$) – sum ($X$)$^2$ / *N* (독립 변수의 "제곱의 합") |
| REGR_SXY(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | Sum (*X*Y*) – sum (*X*) * sum (*Y*) / *N* (독립변수 x 종속변수의 "곱의 합") |
| REGR_SYY(*Y*, *X*) | DOUBLE PRECISION | DOUBLE PRECISION | Sum ($Y^2$) – sum ($Y$)$^2$ / *N* (종속 변수의 "제곱의 합") |
| STDDEV(*expression*) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | 부동 소수점 형식의 인자는 DOUBLE PRECISION, 그렇지 않으면 NUMBER | STDDEV_SAMP 의 역사적 명칭 |
| STDDEV_POP(*expression*) | INTEGER, REAL, | 부동 소수점 형식의 | 입력 값의 모집단 |

| | | | |
|---|---|---|---|
| | DOUBLE PRECISION, NUMBER | 인자는 DOUBLE PRECISION, 그렇지 않으면 NUMBER | 표준분산 |
| STDDEV_SAMP(*expression*) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | 부동 소수점 형식의 인자는 DOUBLE PRECISION, 그렇지 않으면 NUMBER | 입력 값의 표본 표준분산 |
| VARIANCE(*expression*) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | 부동 소수점 형식의 인자는 DOUBLE PRECISION, 그렇지 않으면 NUMBER | VAR_SAMP 의 역사적 명칭 |
| VAR_POP(*expression*) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | 부동 소수점 형식의 인자는 DOUBLE PRECISION, 그렇지 않으면 NUMBER | 입력 값의 모집단분산(모집단 표준편차의 제곱) |
| VAR_SAMP(*expression*) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | 부동 소수점 형식의 인자는 DOUBLE PRECISION, 그렇지 않으면 NUMBER | 입력 값의 표본분산 (표본 표준편차의 제곱) |

### 3.5.11 서브쿼리 식

여기서는 SQL 기반의 Postgres Plus Advanced Server 에서 사용 가능한 서브쿼리 표현에 대해 설명합니다. 이 섹션에서 설명하는 모든 식은 Boolean(참/거짓) 결과를 돌려줍니다.

#### 3.5.11.1 EXISTS

EXISTS 의 인자는 임의의 SELECT 명령문이나 서브쿼리입니다. 서브쿼리는 어떤 행의 반환여부를 결정하기 위해 평가됩니다. 적어도 한 행을 반환하면, EXISTS 의 결과는 '참', 서브쿼리가 아무 행도 돌려주지 않으면, EXISTS 의 결과는 '거짓'입니다.

EXISTS(*subquery*)

서브쿼리는 쿼리 주변으로부터의 변수를 참조할 수 있습니다. 서브쿼리의 평가에서 상수 역할을 합니다.

서브쿼리는 본래 완료하는 모든 방법이 아닌, 적어도 한 행이 반환여부를 결정하기에 충분할 때까지 실행됩니다. 실제로 부작용이 생길지 여부를 예상하기는 어려우므로, (시퀀스 함수 호출과 같은) 부작용이 있는 하위 쿼리를 작성하는 것은 권장하지 않습니다.

결과는 행의 내용이 아니라, 어떤 행을 반환하는지에 의존하기 때문에, 일반적으로 서브쿼리의 출력 리스트는 관계없습니다. 보통의 코딩 전환은 EXISTS(SELECT 1 WHERE ...)형식의 모든 EXISTS 테스트를 작성하기 위한 것입니다. 그렇지만, 이러한 규칙에 INTERSECT 를 사용하는 서브쿼리와 같은 예외가 있습니다.

이 단순한 예는 deptno 에 내부결합과 같습니다만, 다수의 emp 행과 정합하는 것이 있더라도, 각 dept 행의 출력 행을 생성합니다.

```
SELECT dname FROM dept WHERE EXISTS (SELECT 1 FROM emp WHERE emp.deptno =
dept.deptno);

   dname
------------
 ACCOUNTING
 RESEARCH
 SALES
(3 rows)
```

#### 3.5.11.2 IN

우측 편은 한 열을 확실히 반환하는, 괄호 처리된 서브쿼리입니다. 좌측 식은 서브쿼리 결과의 각 행에 비교되고 평가됩니다. 동일한 서브쿼리 행이 발견되면, IN 의 결과는 '참'입니다. 동일한 행이 발견되지 않으면(아무 행도 반환하지 않는 서브쿼리가 특별한 경우를 포함), 결과는 '거짓'입니다.

*expression* IN (*subquery*)

좌측 식이 null 을 산출할 경우나, 동일한 우측 값이 없는, 적어도 하나의 우측 행이 null 을 산출하는 경우, IN 구성의 결과는 거짓이 아니라, null 이 됩니다. null 값의 Boolean 조합 SQL 의 기본 규칙은 동일합니다.

EXISTS 와 같이, 완전하게 평가되는 서브쿼리를 가정하는 것을 권장하지 않습니다.

#### 3.5.11.3 NOT IN

우측 편은 한 열을 확실히 반환하는, 괄호 처리된 서브쿼리입니다. 좌측 식은 서브쿼리 결과의 각 행에 비교되고 평가됩니다. 동일하지 않은 서브쿼리 행이 발견되면, NOT IN 의 결과는 '참'입니다. 동일한 행이 발견되면(아무 행도 반환하지 않는 서브쿼리가 특별한 경우를 포함), 결과는 '거짓'입니다.

*expression* NOT IN (*subquery*)

좌측 식이 null 을 산출할 경우나, 동일한 우측 값이 없는, 적어도 하나의 우측 행이 null 을 산출하는 경우, NOT IN 구성의 결과는 참이 아니라, null 이 됩니다. null 값의 Boolean 조합 SQL 의 기본 규칙은 동일합니다.

EXISTS 와 같이, 완전하게 평가되는 서브쿼리를 가정하는 것을 권장하지 않습니다.

#### 3.5.11.4 ANY/SOME

우측은 괄호로 둘러싸인 서브쿼리에, 확실하게 한 열을 반환해야 합니다. 좌측 식은 평가되고 주어진 연산자를 사용해, Boolean 결과를 산출하는 각 서브쿼리의 결과 행과 비교됩니다. 결과가 참인 경우, ANY 결과는 "참"입니다. 참인 결과가 발견되지 않는 경우, (아무 행도 반환하지 않는 서브쿼리가 특별한 경우를 포함) 결과는 '거짓'입니다.

*expression operator* ANY (*subquery*)
*expression operator* SOME (*subquery*)

SOME 은 ANY 와 동의어입니다. IN 은 = ANY 와 동일합니다.

성공하지 않거나, 우측 행이 연산자의 결과로 null 을 산출하는 경우, ANY 구성의 결과는 거짓이 아닌, null 이 되는 것을 주의해 주세요. Null 값의 Boolean 조합은 SQL 의 기본 규칙과 동일합니다.

EXISTS 처럼, 서브쿼리가 완전하게 평가되는 것을 가정하는 것을 권장하지 않습니다.

#### 3.5.11.5 ALL

우측은 괄호로 둘러싸인 서브쿼리에, 확실하게 한 열을 반환해야 합니다. 좌측 식은 평가되고 주어진 연산자를 사용해, Boolean 결과를 산출하는 각 서브쿼리의 결과 행과 비교됩니다. 결과가 참인 경우, ALL 결과는 "참"입니다. 참인 결과가 발견되지 않는 경우. (아무 행도 반환하지 않는 서브쿼리가 특별한 경우를 포함) 결과는 '거짓'입니다. 비교한 것이 행에 실패를 반환하지 않고, 적어도 한 행에 null 을 반환하는 경우, 결과는 Null 입니다.

*expression operator* ALL (*subquery*)

NOT IN 은 <> ALL 과 동일합니다.

EXISTS 처럼, 서브쿼리가 완전하게 평가되는 것을 가정하는 것을 권장하지 않습니다.

## 제 4 장 저장 프로시저 언어

이 장에서는 저장 프로시저 언어 - SPL 에 대해 설명합니다. SPL 은 Postgres Plus Advanced Server 에서 custom 절차, 함수, 트리거, 패키지 작성을 위한 생산성 높은 프로그램 언어입니다.

SPL 은 다음과 같은 특징이 있습니다.

- SQL 언어를 보완하기 위한 전체적인 절차상의 프로그래밍 기능을 추가합니다.
- 데이터베이스를 위한 저장 프로시저, 함수, 트리거, 패키지를 만드는 유일한 공통 언어입니다.
- pgAdmin Ⅲ 와 통합되어 완벽한 개발 및 테스트 환경을 제공합니다.
- 코드 재사용을 촉진합니다.
- 쉽게 사용할 수 있습니다.

이 장에서는 우선 SPL 프로그램의 기본 요소에 대해 설명합니다. SPL 프로그램 구조에 대한 개요를 살펴보고, 프로시저와 함수를 작성하는 방법에 대해 설명합니다. 트리거는 SPL 을 사용하는 동안은 내용이 많이 다르므로, 별도로 설명하겠습니다. 제 5 장 트리거에 대한 정보를 살펴보고, 제 6 장에서는 패키지에 대해 설명합니다

이 장 후반 영역에서는 SPL 언어에 대한 자세한 내용을 설명하고 동시에 어플리케이션의 예를 살펴봅니다.

### 4.1 SPL 의 기본 요소

이 장에서는 SPL 프로그램의 기본 요소에 대해 설명합니다.

#### 4.1.1 문자 집합

SPL 프로그램은 입력할 때, 다음의 문자 집합을 사용합니다.

- 대문자 A ~ Z 와 소문자 a 부터 z
- 숫자 0 ~ 9
- 기호 ( ) + - * / < > = ! ~ ^ ; : . ' @ % , " # $ & _ | ( ) ? [ ]
- 공백, 문자 탭, 스페이스, CR (carriage returns)

식별자, 표현식, 문장, 제어 구조 등은 이러한 문자들을 사용해 입력되는 SPL 언어를 구성합니다.

**참고: SPL 프로그램에서 처리된 데이터는 데이터베이스 인코딩에서 지원되는 문자 집합에 의해 결정됩니다.**

### 4.1.2 대소문자 구분

SPL 프로그램에서 사용되는 키워드와 사용자 정의 식별자는 대소문자를 구분하지 않습니다. 예를 들어, DBMS_OUTPUT.PUT_LINE ( 'Hello World') ; 문은 dbms_output.put_line ( 'Hello World'); 또는 Dbms_O utput.Put_Line ( 'Hello World'); 나 DBMS_output.Put_line ( 'Hello World');와 동일한 것으로 간주됩니다.

하지만 문자와 문자 상수는 Postgres Plus Advanced Server Advanced Server 데이터베이스에서 가져온 자료와 외부 소스로부터 입력된 데이터와 같이 대소문자를 구별해야 합니다. DBMS_OUTPUT.PUT_LINE ( 'Hello World!') ; 문장은 다음과 같이 출력됩니다.

```
Hello World!
```

또한 DBMS_OUTPUT.PUT_LINE ( 'HE LLO WORLD!'); 문장은 다음과 같이 출력됩니다.

```
HELLO WORLD!
```

### 4.1.3 식별자

*Identifiers* 는 변수, 커서, 프로그램 인자 등 SPL 프로그램의 다양한 요소를 식별하는 데 사용되는 사용자 정의 이름입니다.

식별자 구문 규칙은 SQL 언어의 식별자와 동일합니다. SQL 언어의 식별자에 관해서는 3.1.2 절을 참조하십시오.

식별자는 SPL 키워드나 SQL 언어 키워드와 같아서는 안됩니다. 다음은 유효한 식별자의 예입니다.

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

### 4.1.4 수식어(Qualifiers)

수식어는 자격있는 오브젝트 항목의 소유자나 내용을 기술하는 이름입니다. 자격있는 오브젝트는 자격있는 이름과 뒤이어 공백이 없는 점, 공백 없는 오브젝트 이름으로 지정됩니다. 이 구문은 점 문법(*dot notation* )으로 불립니다.

다음은 수식된 오브젝트의 구문을 나타냅니다.

*수식어.  수식어.] ... 객체*

수식어는 오브젝트의 소유자 이름이며 오브젝트는 수식어에 속하는 항목의 이름입니다.  이전 수식어가 다음 수식어와 오브젝트로 구별되는 항목을 가지는 연쇄되는 수식어를 가질 수 있습니다.

대부분의 식별자는 수식될 수 있습니다. 따라서 식별자가 무엇을 나타내고 내용의 용도는 무엇인가에 따라 수식됩니다.

수식어 예제는 다음과 같습니다.

- 속해있는 스키마에 의해 수식된 프로시저 및 함수이름-예 *schema_name. procedure_name (...)*
- 속해있는 스키마에 의해 수식된 트리거 이름 – 예 *schema_name. trigger_name*
- 속해있는 테이블에 의해 수식된 열 이름 – 예 emp.empno
- 속해있는 스키마에 의해 수식된 테이블 이름 – 예 public.emp
- 테이블과 스키마에 의해 수식된 열 이름 – 예 public.emp.empn o

일반적인 규칙으로, SPL 문장 구문으로 이름을 사용하면 수식된 이름으로 간주됩니다.

일반적으로 수식된 이름은 그 이름의 관련성이 모호한 경우에만 사용합니다.

예를 들어, 동일한 이름을 가진 2 개의 프로시저가 2 개의 다른 스키마에 속해있으며, 하나의 프로그램에서 호출될 경우나 같은 프로그램 내에서 테이블 열과 SPL 변수가 같은 이름으로 사용되는 경우입니다.

가능하다면, 수식된 이름을 사용하지 않도록 권합니다. 이 장에서는 다음과 같은 규정이 이름의 중복을 피하기 위해 사용되고 있습니다.

- SPL 프로그램의 선언 부분에 지정된 모든 변수는 v_로 시작됩니다. 예 – v_empno
- 프로시저나 함수 정의에 선언된 모든 형식 인자는 P_로 시작됩니다. – 예 P_empno
- 열 이름과 테이블 이름은 특별한 접두사를 가지지 않습니다. – 예 column empno in table emp

### 4.1.5 상수

상수 또는 문자열은 고정 값이고, SPL 프로그램에서 사용되는 다양한 형식의 표현합니다.  - 예 숫자, 문자열, 날짜 등. 상수는 다음과 같은 형식으로 분류됩니다.

- 숫자 (정수와 실수) – 숫자 상수에 관한 더 자세한 정보는 3.1.3.2 절 참조

- 문자와 문자 - 문자와 문자열 상수에 관한 더 자세한 정보는 3.1.3.1 절 참조
- 날짜/ 시간 - 날짜 / 시간 데이터 형식과 상수에 관한 더 자세한 정보는 3.2.4 절을 참조

### 4.2 SPL 프로그램

SPL 은 절차상 블록 구조의 언어입니다. SPL 에서 생성되는 프로그램은 프로시저, 함수, 트리거, 패키지의 4 가지 유형이 있습니다.

프로시저와 함수에 대해서는 이 장의 뒷부분에서 더 자세히 설명합니다. 트리거는 5 장에서, 패키지에 관해서는 제 6 장에서 설명합니다.

#### 4.2.1 SPL 블록 구조

프로그램이 프로시저, 함수나 트리거인 것과 관계 없이 SPL 프로그램은 동일한 블록 구조를 가지고 있습니다. 블록은 3 개의 영역으로 구성됩니다. 옵션 선언 영역, 필수 실행 영역, 그리고 옵션 예외 영역입니다. 적어도, 1 개의 블록은 BEGIN 과 END 키워드 내에 하나, 혹은 하나 이상의 SPL 문장으로 구성된 1 개의 실행 영역을 가집니다.

실행 영역과 예외 영역 이내의 문장을 사용하는 변수, 커서, 타입을 정의하는 옵션의 선언 영역이 존재합니다. 선언은 실행 영역 BEGIN 키워드 이전에 작성합니다. 블록이 사용되는 문맥에 따라 선언 영역은 키워드 DECLARE 로 시작합니다.

마지막으로, BEGIN - END 블록 안에 옵션 예외 영역이 있습니다. 예외 영역은 키워드 EXCEPTION 으로 시작하고, 해당 블록의 마지막까지 계속됩니다. 블록에서 문장에 의해 예외가 던져지면, 프로그램 제어는 예외 영역으로 이동합니다. 예외 영역의 내용과 예외에 따라서 처리되거나 되지 않을 수 있습니다.

아래는 일반적인 블록의 구조를 나타냅니다.

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

*Declarations 는* 블록에서 하나 또는 그 이상의 변수, 커서, 타입을 선언합니다. 각각의 선언문은 세미콜론으로 종료되어야 합니다. 키워드 DECLARE 사용은 블록이 나온 정황에 따라 달라집니다.

*statements* 는 하나 또는 그 이상의 SPL 문장입니다. 각 문장은 세미콜론으로 종료됩니다. END 키워드가 표시된 블록의 끝도 세미콜론으로 종료되어야 합니다.

예외 영역이 존재한다면, EXCEPTION 키워드로 시작됩니다. *exception_condition 은* 하나 또는 그 이상의 예외 유형을 테스트하는 조건을 설명합니다 만약 발생한 예외가 *exception_condition* 중 하나의 조건을 충족하면 WHEN *exception_conditio n* 항목을 따르는 문장이 실행됩니다. 구문을 동반하는 하나 또는 그 이상의 WHEN *exception_condition* 항목이 존재합니다.

다음은 가장 간단한 NULL 문장을 실행 영역으로 구성된 블록의 예입니다.

```
BEGIN
NULL;
END;
```

다음 블록은 선언 및 실행 영역을 가진 블록입니다.

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

이 경우, 3 개의 숫자 변수가 NUMBER 로 선언됩니다. 실행 영역에서는 2 개의 변수에 값을 할당하고, 하나의 변수는 다른 변수로 나눠집니다. 3 번째 변수에 그 결과를 저장하고 결과를 표시합니다. 이 블록이 실행되면 결과는 다음과 같습니다.

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

다음 블록은 3 가지 영역으로 구성됩니다. - 선언, 실행, 예외 영역

```
DECLARE
v_numerator NUMBER (2);
v_denominator NUMBER (2);
v_result NUMBER (5,2);
BEGIN
v_numerator : = 75;
v_denominator : = 0;
v_result : = v_numerator / v_denominator;
DBMS_OUTPUT.PUT_LINE (v_numerator | | 'divided by'| | v_denominator | |
'is'| | v_result);
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'An exception occurred');
END;
```

다음 출력은 0 으로 나눴을 때 발생되는 예외 영역 문장의 실행 결과를 보여줍니다.

```
An exception occurred
```

### 4.2.2 익명 블록

이전 절에서 블록의 기본 구조를 설명했습니다. 블록은 간단하게 Postgres Plus Advanced Server 에서 실행될 수 있습니다. 이런 블록을 익명 블록 *(anonymous block)이라고 합니다.* 익명 블록은 이름이 없으며, 데이터베이스에 저장되지 않습니다. 블록은 한 번 실행되고 애플리케이션의 버퍼로부터 삭제됩니다. 블록의 코드가 다시 애플리케이션에 나타날 때까지 반복할 수 없습니다.

익명 블록은 테스트와 같은 즉시 실행하거나 한 번 실행하는 프로그램에 유용합니다.

하지만 보통 같은 블록코드를 여러 번 실행할 수 있습니다. 코드를 여러 번 입력하지 않고 반복 블록을 실행하기 위해서는, 익명 블록을 프로시저와 함수로 전환합니다. 다음에는 데이터베이스에 저장할 수 있으며, 다른 프로시저나 함수 또는, 어플리케이션에서 반복적으로 실행 가능한 프로시저 및 함수를 작성하는 방법을 설명합니다.

### 4.2.3 프로시저 개요

프로시저는 독립적인 SPL 프로그램 문장으로서 호출 가능한 SPL 프로그램입니다. 프로시저가 호출되면, 프로시저는 호출한 곳으로부터 입력 인자 형식의 값을 선택적으로 받을 수 있습니다. 그리고 출력 인자 형식의 값을 선택적으로 리턴할 수 있습니다.

#### 4.2.3.1 프로시저 작성

CREATE PROCEDURE 커맨드는 데이터베이스에 저장된 프로시저를 정의하고 이름을 지정합니다.

```
CREATE [OR REPLACE] PROCEDURE name [(parameters)
[AUTHID (DEFINER | CURRENT_USER)]
  (IS | AS)
     [      declarations      ]
  BEGIN
        statements
  END [      name      ];
```

*name 은* 프로시저 식별자입니다. 만약 [OR REPLACE ]이 지정되어있고, 프로시저와 같은 이름이 이미 스키마에 있다면, 새로운 프로시저는 기존 프로시저를 대체합니다. 만약 [OR REPLACE] 이 지정되지 않는 경우, 프로시저와 같은 이름이 이미 동일한 스키마에 존재하여도 기존 프로시저를 새로운 프로시저로 대체하지 않습니다. *parameters* 인자목록입니다. 만약 AUTHID 항목이

생략되었거나 DEFINER 가 기술되었다면, 프로시저 소유자의 검색 경로와 권한은 데이터베이스 오브젝트에 대한 액세스 권한과 권한이 없는 데이터베이스 오브젝트 참조를 결정하는데 이용됩니다. 만약 CURRENT_USER 가 기술되어 있으면, 프로시저를 실행하는 현재 유저의 권리와 검색 경로는 데이터베이스 오브젝트에 액세스할 수 있는 권한과 자격이 없는 오브젝트 참조를 결정하도록 사용됩니다.

*declarations* 변수, 커서, 타입을 정의합니다. *statements 는* SPL 프로그램 문장입니다. BEGIN - END 블록은 EXCEPTION 영역을 포함할 수 있습니다.

다음의 간단한 프로시저는 인자를 필요로 하지 않습니다.

```
CREATE OR REPLACE PROCEDURE simple_procedure
IS
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'That' 's all folks!');
END simple_procedure;
```

Postgres Plus Advanced Server 에 프로시저 코드를 입력하여 프로시저를 데이터베이스에 저장합니다.

프로시저 생성에 관한 정보는 CREATE PROCEDURE 커맨드를 참조하십시오.

#### 4.2.3.2 프로시저 호출

프로시저는 다른 SPL 프로그램으로부터 인자나 세미콜론 다음의 프로시저 이름을 간단히 기술하여 호출될 수 있습니다.

*name [(parameters);*

*name 은* 프로시저의 식별자입니다. *para meters 는* 실제 인자 목록입니다.

**참고 : 건네진 실제 인자가 없을 경우, 프로시저는 열고 닫는 괄호 없이 호출되어야 합니다.**

다음은 익명 블록에서의 프로시저 호출입니다.

```
BEGIN
simple_procedure;
END;

That 's all folks!
```

**참고 : 애플리케이션은 프로시저 호출에 각각 특유의 방법이 있습니다. Java 애플리케이션에서 애플리케이션 프로그래밍 인터페이스, JDBC,가 사용됩니다.**

#### 4.2.3.3 프로시저 삭제

프로시저는 DROP P ROCEDURE 커맨드를 사용해서 데이터베이스에서 삭제할 수 있습니다.

*DROP* PROCEDURE name;

*Name 은* 삭제 프로시저 이름입니다.

이전에 작성된 프로시저는 이 예제에서 삭제됩니다.

```
DROP PROCEDURE simple_procedure;
```

더 자세한 내용은 DROP PROCEDURE 커맨드를 참조하십시오.

### 4.2.4 기능 개요

함수는 식으로 호출되는 SPL 프로그램입니다. 호출하면 함수가 있는 식을 대신해서 값이 리턴됩니다. 함수는 입력 인자 형식으로 호출하는 프로그램으로부터 값을 선택적으로 받습니다. 또한, 함수가 자체 값을 반환한다는 사실은 다른 값을 출력 인자 형태로 호출한 측에게 반환한다는 것을 일컫습니다. 하지만 함수에서 출력 인자의 이용은 추천되는 프로그래밍 방법이 아닙니다.

#### 4.2.4.1 함수 작성

CREATE FUNCTION 커맨드는 데이터베이스에 저장되는 함수를 정의하고 이름을 지정합니다.

```
CREATE [OR REPLACE] FUNCTION name [(parameters)
RETURN data_type
[AUTHID (DEFINER | CURRENT_USER)]
(IS | AS)
[     declarations     ]
BEGIN
      statements
END [     name     ];
```

*Name 은* 함수의 식별자입니다. 만약 [ OR REPLACE ]이 지정되어 있고, 스키마에 이미 같은 이름의 함수가 존재하면, 기존 함수를 새로운 함수로 교체합니다. 만약 [ OR REPLACE ]이 지정되지 않은 경우, 같은 스키마에 존재하는 동일한 이름의 함수를 새로운 함수로 대체하지 않습니다. *Parameters 은* 공식적인 인자 목록입니다. *data_type 은* 함수로 반환한 값의 데이터 형식입니다. 만약 AUTHID 항목이 없거나 DEFINER 이 지정되어 있으면, 함수 소유자의 검색 경로와 권한은 데이터베이스 오브젝트에 대한 액세스 권한과 권한이 없는 데이터베이스 오브젝트 참조를 결정하도록 각각 사용됩니다. 만약 CURRENT_USER 이 지정되어 있으면, 함수를 실행하는

현재 사용자의 권한과 검색 경로가 데이터베이스 오브젝트에 대한 액세스 권한과 권한이 없는 데이터베이스 오브젝트 참조를 결정하는데 각각 사용됩니다. *Declarations* 는 변수, 커서, 타입 정의입니다. *statements 는* SPL 프로그램 문장입니다. BEGIN - END 블록에는 EXCEPTION 영역이 포함되어 있습니다.

다음은 인자를 가지지 않는 간단한 함수의 예입니다.

```
CREATE OR REPLACE FUNCTION simple_function
RETURN VARCHAR2
IS
BEGIN
RETURN 'That' 's All Folks!';
END simple_function;
```

다음은 2 개의 입력 인자를 갖는 다른 함수입니다. 인자에 대해서는 다음 절에서 자세히 설명합니다.

```
CREATE OR REPLACE FUNCTION emp_comp (
p_sal NUMBER,
p_comm NUMBER
) RETURN NUMBER
IS
BEGIN
RETURN (p_sal + NVL (p_comm, 0)) * 24;
END emp_comp;
```

더 자세한 내용은 CREATE FUNCTION 커맨드를 참조하십시오,

#### 4.2.4.2 함수 호출

함수는 SPL 문장 어디서나 작성할 수 있습니다. 함수는 만약 괄호로 둘러싼 인자가 있다면, 뒤에 이름을 기술함으로써 간단히 호출됩니다.

*name [(parameters)*

*name 은* 함수 이름입니다. *Parameters 는* 실제 인자 목록입니다.

**참고 :** 만약 건네진 인자가 없을 경우, 함수는 빈 인자 목록에서 호출되거나 전체적으로 열고 닫는 괄호가 생략될 수 있습니다.

다음은 다른 SPL 프로그램에서 함수가 호출되는 방법을 보여주고 있습니다.

```
BEGIN
DBMS_OUTPUT.PUT_LINE (simple_function);
END;

That 's All Folks!
```

다음 함수는 일반적으로 SQL 구문 안에 사용됩니다.

```
SELECT empno "EMPNO"ename "ENAME", sal "SAL"comm "COMM"
emp_comp (sal, comm) "YEARLY COMPENSATION"FROM emp;

EMPNO | ENAME | SAL | COMM | YEARLY COMPENSATION
------+-------+-------+-------+------------ --------
7369 | SMITH | 800.00 | | 19200.00
7499 | ALLEN | 1600.00 | 300.00 | 45600.00
7521 | WARD | 1250.00 | 500.00 | 42000.00
7566 | JONES | 2975.00 | | 71400.00
7654 | MARTIN | 1250.00 | 1400.00 | 63600.00
7698 | BLAKE | 2850.00 | | 68400.00
7782 | CLARK | 2450.00 | | 58800.00
7788 | SCOTT | 3000.00 | | 72000.00
7839 | KING | 5000.00 | | 120000.00
7844 | TURNER | 1500.00 | 0.00 | 36000.00
7876 | ADAMS | 1100.00 | | 26400.00
7900 | JAMES | 950.00 | | 22800.00
7902 | FORD | 3000.00 | | 72000.00
7934 | MILLER | 1300.00 | | 31200.00
(14 rows)
```

#### 4.2.4.3 함수 삭제

DROP FUNCTION 커맨드는 함수를 데이터베이스로부터 삭제할 수 있습니다.

*DROP* FUNCTION name;

*name* 은 삭제하는 함수 이름입니다.

이 예제에서는 전에 생성된 함수를 삭제합니다,

```
DROP FUNCTION simple_function;
```

더 자세한 내용은 DROP FUNCTION 커맨드를 참조하십시오.

### 4.2.5 프로시저와 함수 인자

프로시저와 함수 사용에서 중요한 부분은 호출 프로그램에서 프로시저나 함수에 데이터를 전달하고, 프로시저나 함수로부터 데이터를 받는 기능입니다. 이것은 *parameters 를* 이용하여 이루어집니다.

인자는 프로시저나 함수 정의에 선언된 이름을 따르는 괄호 안에 표시됩니다. 프로시저나 함수에 선언된 인자는 *formal parameters (형식 인자)라고 합니다.* 프로시저와 함수가 호출되면, 호출 프로그램은 호출된 프로그램이 이용한 실제 데이터와 처리 결과를 받는 변수를 전달합니다. 프로시저와 함수를 호출됐을 때, 호출한 프로그램에서 제공되는 데이터와 변수는 *actual parameters (실제 인자)로서 참조되어집니다.*

다음은 형식 인자의 일반 형식을 정의한 것입니다.

*(name* [ IN | OUT | IN OUT ] *data_type)*

*Name 은* 형식 인자에 지정된 식별자입니다. IN 은 프로시저와 함수가 받는 입력 데이터를 위한 인자를 정의합니다. IN 인자는 기본값으로 초기화됩니다. OUT 은 프로시저와 함수의 반환 데이터를 위한 인자를 정의합니다. IN, OUT 이 지정되어 있으면, 인자는 입력 및 출력에 이용됩니다. 만일 IN, OUT, IN OUT 전체가 생략되면, 인자는 디폴트로 IN 이 정의된 것처럼 행동합니다. 인자가 IN, OUT 또는 IN OUT 중 하나이면 인자 *mode* 로서 참조됩니다. *data_type* 은 인자 데이터 형식을 정의합니다.

다음은 인자를 가진 프로시저의 예입니다.

```
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;
```

이 예에서는 p_deptno 은 IN 형식 인자이고, p_empno 과 p_ename 은 IN OUT 형식 인자, p_job, p_hiredate 와 p_sal 는 OUT 형식 인자입니다.

**참고:** 이전의 예에서는, VARCHAR2 인자에 최대 길이가 지정되지 않았습니다. 그리고 NUMBER 인자는 정밀도와 스케일이 지정되지 않았습니다. 인자 선언에서 길이, 정밀도, 스케일 및 기타 제약 조건을 지정하는 것은 오류를 일으켰습니다. 이러한 제한은 프로시저와 함수가 호출될 때, 실제 인자에서 자동으로 상속됩니다.

emp_query 프로시저는 실제 인자와 조화하여 다른 프로그램에서 호출될 수 있습니다. 다음은 emp_query 를 호출하는 다른 SPL 프로그램의 예입니다.

```
DECLARE
v_deptno NUMBER (2);
v_empno NUMBER (4);
v_ename VARCHAR2 (10);
v_job VARCHAR2 (9);
```

```
v_hiredate DATE;
v_sal NUMBER;
BEGIN
v_deptno : = 30;
v_empno : = 7900;
v_ename : = '';
emp_query (v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
DBMS_OUTPUT.PUT_LINE ( 'Department :'| | v_deptno);
DBMS_OUTPUT.PUT_LINE ( 'Employee No :'| | v_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | v_ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | v_job);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | v_hiredate);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | v_sal);
END;
```

이 경우에, v_deptno, v_empno, v_ename, v_job, v_hiredate 및 v_sal 는 실제 인자입니다.

위의 예에서 출력은 다음과 같습니다.

```
Department : 30
Employee No : 7900
Name : JAMES
Job : CLERK
Hire Date : 03 - DEC - 81
Salary : 950
```

#### 4.2.5.1 인자 형태

전에 언급했듯이, 인자는 IN, OUT, IN OUT 3 가지 모드 중 1 개를 가집니다. 다음의 형식 인자 특성은 그 형태에 따라 달라집니다.

- 프로시저 또는 함수를 호출할 때의 초기값
- 호출된 프로시저와 함수가 형식인자를 변경할 수 있는가
- 어떻게 실제 인자가 호출 프로그램에서 호출 대상 프로그램에 인도되는가
- 호출 대상 프로그램에서 처리할 수 없는 예외가 발생했을 때, 형식 인자 값에 무슨 일이 일어나는가

다음 표는 형태에 따른 인자의 행동을 요약합니다.

**표 4-1 인자 형태**

| 모드 속성 | IN | IN OUT | OUT |
|---|---|---|---|
| 초기화되는 형식 인자 포맷 : | 실제 인자 값 | 실제 인자 값 | 실제 인자 값 |
| 형식 인자를 호출한 프로그램으로 변경할 수 있는가? | 아니오 | 예 | 예 |
| 실제 인자 내용 : (일반적으로 호출한 프로그램 종료 후) | 호출 이전의 원래 실제 인자 값 | 형식 인자의 최종 값 | 형식 인자의 최종 값 |

| | | | |
|---|---|---|---|
| 실제 인자 내용 : (호출된 프로그램에서 예외 처리 후) | 호출 이전의 원래 실제 인자 값 | 형식 인자의 최종 값 | 호출 이전의 원래 실제 인자 값 |
| 실제 인자 내용 : (호출된 프로그램에서 예외가 처리되지 않은 후) | 호출 이전의 원래 실제 인자 값 | 호출 이전의 원래 실제 인자 값 | 호출 이전의 원래 실제 인자 값 |

테이블에 표시된 것처럼, IN 형식 인자는 기본값으로 초기화되지 않는 한, 호출되는 실제 인자로 초기화됩니다. IN 인자는 호출한 프로그램에서 참조할 수 있지만, 호출된 프로그램은 IN 인자에 새로운 값을 할당하지는 않을 것입니다. 호출 프로그램으로 제어가 돌아가면, 실제 인자는 항상 호출하기 전에 지정된 값을 유지하고 있습니다.

OUT 형식 인자는 호출할 때 실제 인자로 초기화됩니다. 호출된 프로그램은 그 값을 참조하여 형식 인자에 새 값을 할당합니다. 만약 호출되는 프로그램이 예외 없이 완료되면, 실제 인자는 형식 인자에 마지막으로 설정된 값입니다. 만약 처리되는 예제가 발생하면, 실제 인자 값은 형식 인자의 마지막 값이 됩니다. 처리할 수 없는 예제가 발생한다면, 실제 인자의 값은 호출 전의 상태로 남아있습니다.

IN 인자처럼, IN OUT 형식 인자는 호출될 때 실제 인자로 초기화됩니다. OUT 인자와 같이, IN OUT 형식 인자는 호출된 프로그램에 의해 변경 가능하며, 만약 호출된 프로그램이 예외 없이 종료됐다면, 형식 인자의 최근 값은 호출 프로그램의 실제 인자로 전달됩니다. 처리되는 예제가 발생되면, 실제 인자의 값은 형식 인자에 할당된 마지막 값입니다. 처리되지 않는 예제가 발생한 경우, 실제 인자에는 호출 이전의 값이 그대로 남아있습니다.

#### 4.2.6 프로그램의 보안

사용자가 SPL 프로그램을 실행하거나 사용자에게 SPL 프로그램을 실행할 수 있도록 한 데이터베이스 오브젝트 액세스에 대한 보안은 다음과 같이 제어됩니다.

- 프로그램을 실행시키는 권한
- 프로그램이 접근하려는 데이터베이스 오브넥트(다른 SPL 프로그램 포함)에 부여된 권한
- 프로그램이 소유자의 권한으로 정의되거나 호출 권한으로 정의되는가

다음 절에서는 이러한 사실을 설명합니다.

##### 4.2.6.1 EXECUTE 권한

SPL 프로그램(함수, 프로시저 또는 패키지)은 다음 중 어느 하나가 true 일 경우에만 실행을 시작할 수 있습니다.

- 현재 사용자가 관리자이다. 또는
- 현재 사용자가에게 해당 SPL 프로그램 EXECUTE 권한이 부여되었다, 또는
- 현재 사용자가 EXECUTE 권한을 가진 그룹의 일원으로 EXECUTE 권한을 상속받았다, 또는
- EXECUTE 권한이 PUBLIC 그룹에게 부여되었다.

Postgres Plus Advanced Server 는 SPL 프로그램이 생성되면 자동적으로 EXECUTE 사용 권한이 PUBLIC 그룹에 할당됩니다. 따라서 모든 사용자가 즉시 프로그램을 실행할 수 있습니다.

기본 권한은 REVOKE EXECUTE 커맨드를 사용하여 해제할 수 있습니다. 자세한 내용은 REVOKE 커맨드를 참조하십시오. 예를 들면 다음과 같습니다.

```
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
```

프로그램에서 확실한 EXECUTE 권한은 개별 사용자 또는 그룹에 부여할 수 있습니다.

```
GRANT EXECUTE ON PROCEDURE list_emp TO john;
```

지금, 사용자 john 은 list_emp 프로그램을 실행할 수 있으나, 다른 사용자는 이 영역의 처음 조건 목록에 만족하지 않기 때문에 프로그램을 실행할 수 없습니다.

한번 프로그램이 실행되면, 다음 보안 처리는 프로그램이 다음과 같은 데이터베이스 오브젝트에 대한 작업을 수행할 수 있는 권한이 있는지를 확인하는 것입니다.

- 테이블이나 뷰를 데이터 읽기 및 갱신
- 테이블, 뷰, 인덱스, 시퀀스와 같은 데이터베이스 오브젝트의 생성, 갱신, 삭제
- 시퀀스로부터 현재 값과 다음 값을 취득
- 다른 프로그램 (함수, 프로시저 및 패키지)를 호출

.

각각의 활동은 데이터베이스 오브젝트의 권한으로 사용자를 허용하거나 허용하지 않음으로써 보호될 수 있습니다.

**참고:** 데이터베이스는 동일한 형식으로 이름이 같은 오브젝트를 하나 이상 가질 수 있는 것에 유의하십시오. 그러나 데이터베이스에서 이 오브젝트는 다른 스키마에 속합니다. 이런 경우, SPL 프로그램에 의해 참조되는 오브젝트는 무엇일까요? 이것이 다음 절의 주제입니다.

#### 4.2.6.2 데이터베이스 오브젝트 이름 결정

SPL 프로그램에서 데이터베이스 오브젝트는 자격 있는 이름이나 자격이 없는 이름에 의해서 참조될 수 있습니다. 자격 있는 이름은 *schema. name* 형식으로 *schema 는 name* 식별자를 가진 데이터베이스 오브젝트 스키마의 이름입니다. 자격이 없는 이름은 *"schema."*부분을 가지지

않습니다. 자격 있는 이름으로 참조가 이루어지면, 지정된 스키마가 존재하는지 아닌지를 가리키는 데이터베이스 오브젝트에 대해 명확해집니다.

그러나 자격 없는 이름의 오브젝트 위치를 검색하는 것은 최근 사용자의 검색 경로를 필요로 합니다. 사용자가 세션의 현재 사용자가 되면, 디폴트 검색 경로는 항상 관련된 사용자에게 할당됩니다. 검색 경로는 스키마 목록을 가지고 있으며, 이는 자격 없는 데이터베이스 오브젝트에 대한 참조 위치를 왼쪽에서 오른쪽으로 검색한 결과 목록입니다. 만약 검색 경로에서 어떠한 스키마도 발견되지 않으면, 해당 오브젝트가 존재하지 않는 것으로 판단됩니다. 기본 검색 경로는 PSQL 의 SHOW search_path 커맨드를 사용하여 나타냅니다.

```
SHOW search_path;

search_path
----------------------
$ user, public, sys, dbo
(1 row)
```

위의 검색 경로에서 $ user, 위의 현재 세션 사용자가 enterprisedb 인 경우, 현재 사용자를 참조하는 일반적인 플레이스 폴더입니다. 자격이 없는 데이터베이스 오브젝트는 다음 스키마에서 차례로 검색됩니다. 순서는 enterprisedb 다음에 public 다음에 sys 마지막으로 dbo 입니다.

검색 경로에 자격 없는 이름이 발견되면, 현재 사용자가 특정한 오브젝트에서 요청을 수행할 수 있는 적절한 권한이 있는지 결정됩니다.

**참고:** 검색 경로의 개념은 Oracle 과 호환되지 않습니다. 자격 없는 이름을 참조하면, Oracle 은 단순히 현재 사용자의 스키마 데이터베이스 오브젝트 이름을 검색하면 됩니다. Oracle 에서 사용자와 스키마는 동일한 항목이지만, Postgres Plus Advanced Server 에서는 사용자와 스키마는 2 개의 독립적인 오브젝트입니다. 이 사실은 중요합니다.

#### 4.2.6.3 데이터베이스 오브젝트 권한

한번 SPL 프로그램이 실행을 시작하면, 참조하는 오브젝트에 대한 현재 사용자의 사용 권한을 검사하여 전체 데이터베이스 오브젝트에 대한 액세스가 실행됩니다. 데이터베이스 오브젝트에 대해 GRANT 와 REVOKE 커맨드를 사용하여 각각 권한을 부여하거나 해제할 수 있습니다. 만일 현재 사용자가 데이터베이스 오브젝트에 권한 없이 접속하면 프로그램은 예외를 발생시킵니다. 예외 처리에 대한 자세한 내용은 4.5.5 절을 참조하십시오.

마지막 주제는 정확히 현재 사용자가 누구인가를 설명합니다.

#### 4.2.6.4 소유자 권한 vs. 호출자 권한

SPL 프로그램이 실행을 시작하려고 할 때, 어떤 사용자가 이 과정과 관련되는지가 결정됩니다. 사용자가 *current user* (현재 사용자)로 간주됩니다. 자격이 없는 오브젝트 참조를 해결하기 위해 현재 사용자의 검색 경로를 결정합니다. 현재 사용자의 데이터베이스 오브젝트 권한은 허락된 프로그램을 참조하는 데이터베이스 오브젝트에 액세스할 수 있는지 여부를 결정하는데 사용됩니다.

SPL 프로그램은 소유자 또는 호출자의 권한으로 생성되는지에 따라 현재 사용자의 선택이 영향을 받습니다. AUTHID 항목은 그 선택을 결정합니다. AUTHID DEFINER 항목의 계정은 소유자 권한을 프로그램에 제공합니다. 만약 AUTHID 항목이 생략되어도 이것은 기본 값입니다. AUTHID CURRENT _USER 항목을 사용하면 호출자에게 권한이 부여됩니다. 이 둘의 차이점을 아래에 요약합니다.

- 만약 프로그램이 소유자 권한을 가지고 있다면, 프로그램이 시작하는 순간 프로그램의 소유자가 현재 사용자가 됩니다. 프로그램 소유자의 검색 경로가 자격이 없는 오브젝트 참조를 해결하는데 사용됩니다. 그리고 프로그램 소유자의 데이터베이스 오브젝트 권한은 참조하는 오브젝트를 액세스할 수 있는지 여부를 결정하기 위해 사용됩니다. 어떤 사용자가 실제로 프로그램을 호출하는 지는 관련이 없습니다.

- 만약 프로그램이 호출자 권한을 가지는 경우, 프로그램이 호출될 때의 현재 사용자가 프로그램 실행되는 동안 현재 사용자로 남습니다. (그러나 호출된 서브프로그램은 예외입니다. - 아래 설명 참조) 호출자 권한 프로그램이 시작되면, 현재 사용자는 일반적으로 세션이 시작됩니다 (예를 들면, 데이터베이스 접속 생성). 그러나 세션이 시작한 후에도 SET ROLE 커맨드를 사용하여 현재 사용자를 변경할 수 있습니다. 호출자 권한 프로그램에서는, 프로그램의 실제 소유자가 누구인지는 전혀 관계가 없습니다.

이상의 정의는 다음과 같이 말할 수 있습니다.

- 만약 소유자 권한 프로그램이 소유자 권한 프로그램을 호출하면, 현재 사용자가 호출한 프로그램이 실행되는 동안 호출 프로그램의 사용자에서 호출되는 프로그램의 소유자로 전환됩니다

- 만약 소유자 권한 프로그램이 호출자 권한 프로그램을 호출하면, 호출 프로그램 소유자가 호출 및 호출 대상 프로그램 양측의 현재 사용자로 남게 됩니다.

- 만약 호출자 권한 프로그램이 호출자 권한 프로그램을 호출하면, 호출 프로그램의 현재 사용자는 호출될 프로그램을 실행하는 동안 현재 사용자로 남게 됩니다

- 만약 호출자 권한 프로그램이 소유자 권한 프로그램을 호출하면, 현재 사용자가 호출한 프로그램이 실행되는 동안 소유자 권한 프로그램 소유자로 전환됩니다.

위에서 언급한대로, 호출된 프로그램이 점차 다른 프로그램을 호출할 경우, 같은 원칙이 적용됩니다.

보안과 관련된 절은 간단한 어플리케이션을 사용하는 예제로 마무리 합니다.

#### 4.2.6.5 보안 예제

다음 예제에서는 새 데이터베이스를 두 명의 사용자와 함께 생성됩니다. - hr_mgr 는 스키마 hr_mgr 안에 모든 샘플 어플리케이션의 사본을 보유한 사용자; 그리고 sales_mgr 는 스키마 sales_mgr 을 보유하고 영업에서 일하는 직원만을 포함하는 emp 테이블의 복사본만을 보유하고 있습니다.

프로시저 list_emp, hire_clerk 함수, emp_admin 패키지는 이 예제에서 사용됩니다. 샘플 애플리케이션을 설치할 때 주어진 모든 기본 권한은 삭제하고 예제에 좀더 안전한 환경에서 다시 명시적인 권한을 부여합니다.

list_emp 과 hire_clerk 프로그램은 기본 소유자 권한에서 호출자 권한으로 변경됩니다. sales_mgr 가 이 프로그램을 실행했을 때를 설명하면 다음과 같습니다. 프로그램은 sales_mgr 의 검색 경로와 권한을 이름 확인과 인증 검사를 위해 사용하기 때문에, sales_mgr 스키마 내의 emp 테이블이 실행됩니다.

emp_admin 패키지의 get_dept_name 및 hire_emp 프로그램은 sales_mgr 를 통해 실행됩니다. 이 경우 hr_mgr 스키마 내 dept 테이블과 emp 테이블은 소유자 권한을 가진 emp_admin 패키지의 소유자인 hr_mgr 로 접근됩니다.

**1 단계 - 데이터베이스와 사용자 만들기**

사용자 enterprisedb, 데이터베이스 hr 생성 :

```
CREATE DATABASE hr;
```

hr 데이터베이스로 전환하고 사용자 생성 :

```
₩ c hr enterprisedb
CREATE USER hr_mgr IDENTIFIED BY password;
CREATE USER sales_mgr IDENTIFIED BY password;
```

**2 단계 - 예제 어플리케이션 만들기**

모든 샘플 애플리케이션을 작성합니다. 스키마는 hr_mgr 의 소유자는 hr_mgr 입니다.

```
W c - hr_mgr
W i C : / Postgres Plus Advanced Server/8.3/samples/edb-sample.sql

BEGIN
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE VIEW
CREATE SEQUENCE
.
.
.
CREATE PACKAGE
CREATE PACKAGE BODY
COMMIT
```

**3 단계 - 스키마 sales_mgr 에 emp 테이블을 작성**

스키마 sales_mgr 에 sales_mgr 가 소유한 emp 테이블의 하위 집합을 만듭니다.

```
W c - hr_mgr
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
W c - sales_mgr
CREATE TABLE emp AS SELECT * FROM hr_mgr.emp WHERE job = 'SALESMAN';
```

위의 예에서, GRANT USAG E ON SCHEMA 커맨드는 스키마 hr_mgr 에 emp 테이블의 복사본을 만들려면 sales_mgr 에 권한을 부여해야 합니다. Oracle 은 사용자와 전혀 다른 스키마라는 개념이 없기 때문에 이 단계는 Postgres Plus Advanced Server 이 필요하며, Oracle 과 호환성이 없습니다.

**4 단계 - 기본 권한 삭제**

다음에 최소한의 필요한 권한을 설정하기 위해 모든 권한을 제거합니다.

```
W c - hr_mgr
REVOKE USAGE ON SCHEMA hr_mgr FROM sales_mgr;
REVOKE ALL ON dept FROM PUBLIC;
REVOKE ALL ON emp FROM PUBLIC;
REVOKE ALL ON next_empno FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION new_empno () FROM PUBLIC;
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION hire_clerk (VARCHAR2, NUMBER) FROM PUBLIC;
REVOKE EXECUTE ON PACKAGE emp_admin FROM PUBLIC;
```

**Step 5 - list_emp 를 호출자 권한으로 변경**

사용자 hr_mgr 로서 접속하는 동안, list_emp 프로그램에 AUTHID CURRENT_USER 항목을 추가하고 이를 Postgres Plus Advanced Server 에 다시 저장합니다. 이 단계를 수행할 때는

hr_mgr 로 로그인해야 합니다. 그렇지 않으면 수정된 프로그램이 hr_mgr 스키마가 아닌 public 스키마에 저장됩니다.

```
CREATE OR REPLACE PROCEDURE list_emp
AUTHID CURRENT_USER
IS
v_empno NUMBER (4);
v_ename VARCHAR2 (10);
CURSOR emp_cur IS
SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
OPEN emp_cur;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH emp_cur INTO v_empno, v_ename;
EXIT WHEN emp_cur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | v_ename);
END LOOP;
CLOSE emp_cur;
END;
```

**6 단계 - hire_clerk 를 호출자 권한으로 변경하고 new_empno 를 권한이 주어진 호출**

hr_mgr 사용자로 접속하는 동안, hire_clerk 프로그램에 AUTHID CURRENT_USER 항목을 추가합니다.

또한, BEGIN 문장 다음에 new_empno 에 hr_mgr.new_empno 를 참조하는 자격을 부여합니다. 이것은 hire_clerk 함수가 hr_mgr 스키마 내 new_empno 함수를 호출할 수 있도록 하기 위한 것입니다. 호출은 완전한 수식 이름으로 변경되어야 합니다. 한편 hire_clerk 은 현재 호출자 권한 프로그램이기 때문에, new_empno 에 대한 자격 없는 호출은 실제로 이 프로그램이 있는 hr_mgr 스키마에 지정된 것이 아닌, hire_clerk 를 호출하는 원래의 검색 경로 스키마의 new_empno 을 찾습니다.

이 프로그램을 다시 저장할 때 hr_mgr 로 로그인 되었는지 확인하십시오. 그렇지 않으면 변경된 프로그램은 hr_mgr 스키마가 아닌 public 스키마에 저장될지 모릅니다.

```
CREATE OR REPLACE FUNCTION hire_clerk (
p_ename VARCHAR2,
p_deptno NUMBER
) RETURN NUMBER
AUTHID CURRENT_USER
IS
v_empno NUMBER (4);
v_ename VARCHAR2 (10);
v_job VARCHAR2 (9);
v_mgr NUMBER (4);
v_hiredate DATE;
v_sal NUMBER (7,2);
v_comm NUMBER (7,2);
v_deptno NUMBER (2);
```

```
BEGIN
v_empno : = hr_mgr.new_empno;
INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
TRUNC (SYSDATE), 950.00, NULL, p_deptno);
SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
FROM emp WHERE empno = v_empno;
DBMS_OUTPUT.PUT_LINE ( 'Department :'| | v_deptno);
DBMS_OUTPUT.PUT_LINE ( 'Employee No :'| | v_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | v_ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | v_job);
DBMS_OUTPUT.PUT_LINE ( 'Manager :'| | v_mgr);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | v_hiredate);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | v_sal);
DBMS_OUTPUT.PUT_LINE ( 'Commission :'| | v_comm);
RETURN v_empno;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'The following is SQLERRM :');
DBMS_OUTPUT.PUT_LINE (SQLERRM);
DBMS_OUTPUT.PUT_LINE ( 'The following is SQLCODE :');
DBMS_OUTPUT.PUT_LINE (SQLCODE);
RETURN -1;
END;
```

### 7 단계 - 필요한 권한 부여

hr_mgr 사용자가 접속하는 동안, sales_mgr 이 list_emp 프로시저, hire_clerk 함수와 emp_admin 패키지를 실행할 수 있는 권한을 부여합니다. 데이터 오브젝트 sales_mgr 은 sales_mgr 스키마 내 emp 테이블에만 액세스할 수 있는 권한을 가지고 있는 것을 주의하십시요. sales_mgr 은 hr_mgr 스키마의 모든 테이블에 접근할 권한이 없습니다.

```
GRANT EXECUTE ON PROCEDURE list_emp TO sales_mgr;
GRANT EXECUTE ON FUNCTION hire_clerk (VARCHAR2, NUMBER) TO sales_mgr;
GRANT EXECUTE ON FUNCTION new_empno () TO sales_mgr;
GRANT EXECUTE ON PACKAGE emp_admin TO sales_mgr;
```

### 8 단계 - list_emp 과 hire_clerk 프로그램 실행

sales_mgr 사용자로 접속하고, 다음의 익명 블록을 실행합니다.

```
W c - sales_mgr
DECLARE
v_empno NUMBER (4);
BEGIN
hr_mgr.list_emp;
DBMS_OUTPUT.PUT_LINE ('*** Adding new employee ***');
v_empno : = hr_mgr.hire_clerk ( 'JONES', 40);
DBMS_OUTPUT.PUT_LINE ('*** After new employee added ***');
hr_mgr.list_emp;
END;
```

```
EMPNO ENAME
----- -------
7499 ALLEN
7521 WARD
7654 MARTIN
7844 TURNER
*** Adding new employee ***
Department : 40
Employee No : 8000
Name : JONES
Job : CLERK
Manager : 7782
Hire Date : 08 - NOV - 07 00:00:00
Salary : 950.00
*** After new employee added ***
EMPNO ENAME
----- -------
7499 ALLEN
7521 WARD
7654 MARTIN
7844 TURNER
8000 JONES
```

익명 차단 프로그램으로 접근한 테이블과 시퀀스는 아래의 그림으로 나타냈습니다. 회색 타원형은 sales_mgr 과 hr_mgr 스키마입니다. 프로그램 실행 시 현재 사용자는 굵은 적색 글씨로 괄호 안에 표현됩니다.

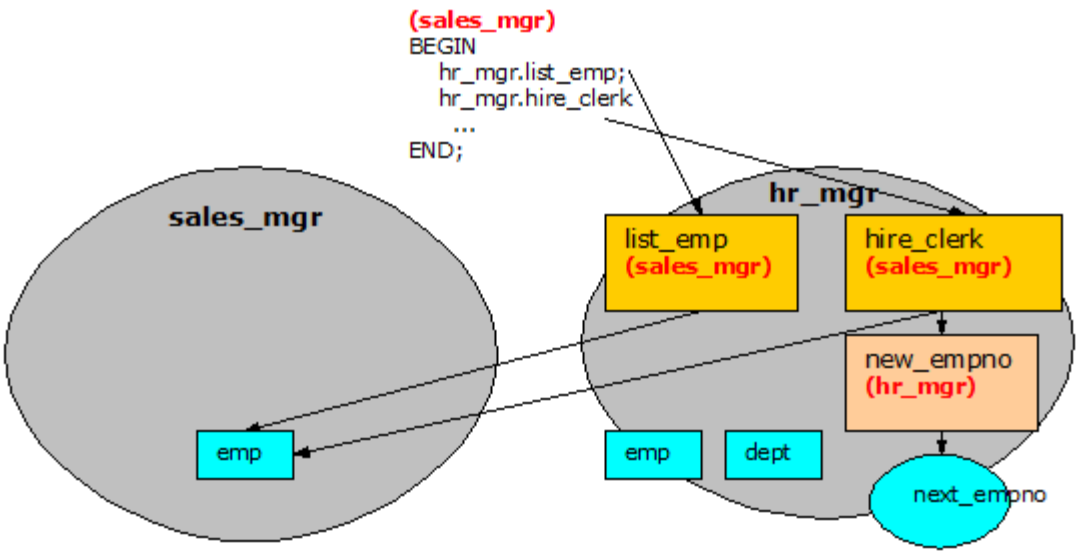


**그림 3 - 호출자 권한 프로그램**

sales_mgr 의 emp 테이블로부터의 선택은 테이블에서 이루어진 업데이트를 표시합니다.

```
SELECT empno, ename, hiredate, sal, deptno, hr_mgr.emp_admin.get_dept_name (deptno) FROM
sales_mgr.emp;
```

```
empno | ename | hiredate | sal | deptno | get_dept_name
  -------+--------+-------------------+---------+-- ------+---------------
  7499 | ALLEN | 20 - FEB - 81 00:00:00 | 1600.00 | 30 | SALES
  7521 | WARD | 22 - FEB - 81 00:00:00 | 1250.00 | 30 | SALES
  7654 | MARTIN | 28 - SEP - 81 00:00:00 | 1250.00 | 30 | SALES
  7844 | TURNER | 08 - SEP - 81 00:00:00 | 1500.00 | 30 | SALES
  8000 | JONES | 08 - NOV - 07 00:00:00 | 950.00 | 40 | OPERATIONS
  (5 rows)
```

다음 그림은 sales_mgr 스키마 내의 emp 테이블을 참조하는 SELECT 커맨드를 보여줍니다. emp_admin 패키지의 get_dept_name 함수가 참조하는 dept 테이블은 hr_mgr 스키마에 있습니다. 왜냐하면 emp_admin 패키지는 소유자 권한을 가지고 hr_mgr 가 소유하고 있기 때문입니다.

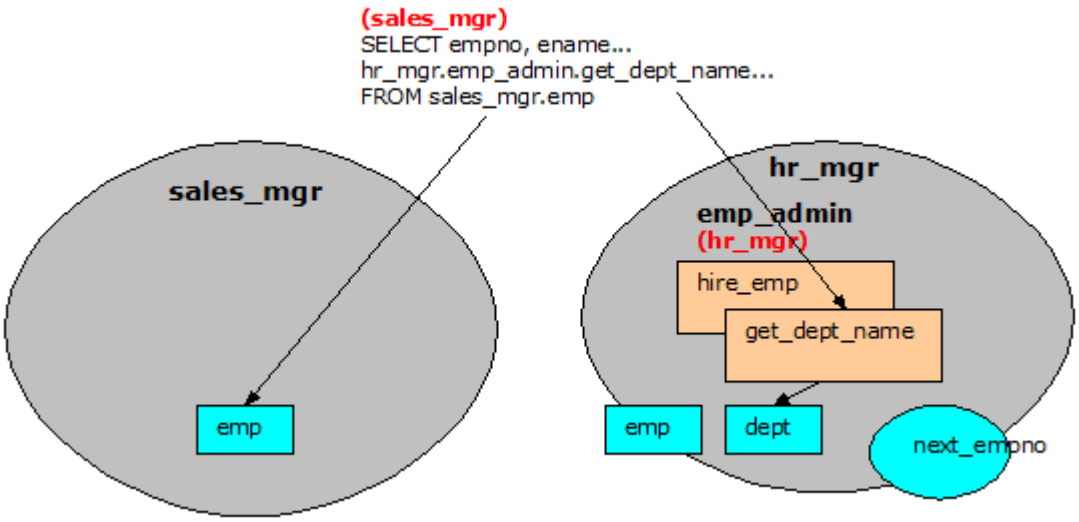


**그림 4 소유자 권한 패키지**

**9 단계 - emp_admin 패키지 내 hire_emp 프로그램 실행**

sales_mgr 사용자로 접속하는 동안, emp_admin 패키지의 hire_emp 프로시저를 실행합니다.

```
EXEC hr_mgr.emp_admin.hire_emp (9001, 'ALICE', 'SALESMAN', 8000, TRUNC (SYSDATE), 1000,7369,40);
```

이 그림은 emp_admin 소유자 권한 패키지 내 hire_emp 프로시저가 hr_mgr 에 속하는 emp 테이블을 업데이트한다는 것을 의미합니다.

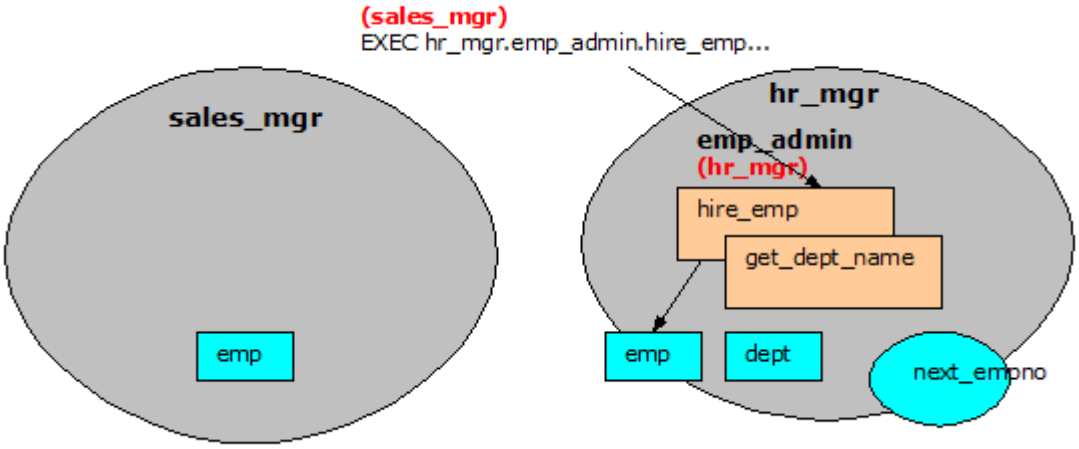


**그림 5 소유자 권한 패키지**

이제 사용자 hr_mgr 로 접속합니다. 다음 SELECT 커맨드는 hr_mgr emp 테이블에 추가된 새로운 직원을 확인합니다. 여기서 emp_admin 패키지는 소유자 권한을 가지며, hr_mgr 은 emp_admin 소유자입니다.

```
   W c - hr_mgr
   SELECT empno, ename, hiredate, sal, deptno, hr_mgr.emp_admin.get_dept_name (deptno) FROM
hr_mgr.emp;

   empno | ename | hiredate | sal | deptno | get_dept_name
   -------+--------+--------------------+---------+-- ------+---------------
   7369 | SMITH | 17 - DEC - 80 00:00:00 | 800.00 | 20 | RESEARCH
   7499 | ALLEN | 20 - FEB - 81 00:00:00 | 1600.00 | 30 | SALES
   7521 | WARD | 22 - FEB - 81 00:00:00 | 1250.00 | 30 | SALES
   7566 | JONES | 02 - APR - 81 00:00:00 | 2975.00 | 20 | RESEARCH
   7654 | MARTIN | 28 - SEP - 81 00:00:00 | 1250.00 | 30 | SALES
   7698 | BLAKE | 01 - MAY - 81 00:00:00 | 2850.00 | 30 | SALES
   7782 | CLARK | 09 - JUN - 81 00:00:00 | 2450.00 | 10 | ACCOUNTING
   7788 | SCOTT | 19 - APR - 87 00:00:00 | 3000.00 | 20 | RESEARCH
   7839 | KING | 17 - NOV - 81 00:00:00 | 5000.00 | 10 | ACCOUNTING
   7844 | TURNER | 08 - SEP - 81 00:00:00 | 1500.00 | 30 | SALES
   7876 | ADAMS | 23 - MAY - 87 00:00:00 | 1100.00 | 20 | RESEARCH
   7900 | JAMES | 03 - DEC - 81 00:00:00 | 950.00 | 30 | SALES
   7902 | FORD | 03 - DEC - 81 00:00:00 | 3000.00 | 20 | RESEARCH
   7934 | MILLER | 23 - JAN - 82 00:00:00 | 1300.00 | 10 | ACCOUNTING
   9001 | ALICE | 08 - NOV - 07 00:00:00 | 8000.00 | 40 | OPERATIONS
   (15 rows)
```

### 4.3 변수 선언

4.2.1 절에서 설명한 것처럼, SPL 은 블록 구조의 언어입니다. 블록의 첫 번째 영역은 선언부입니다. 선언 부는 변수, 정의, 커서 및 블록을 포함하는 SPL 문장에서 사용되는 다른 형식의 정의를 포함합니다. 본 절에서는 변수 선언에 대해 자세히 설명합니다.

### 4.3.1 변수 선언

일반적으로 블록 내에서 사용되는 모든 변수는 블록 선언 부에서 선언되어야 합니다. 변수 선언은 변수에 지정된 이름과 데이터 형식으로 이루어집니다. (데이터 형식에 대해서는 3.2 절 참조) 선택적으로, 변수를 선언할 때 기본 값으로 초기화 할 수 있습니다.

변수 선언의 일반적인 구문은 다음과 같습니다.

*name* *type* [(: = | DEFAULT) *(expression* | NULL)];

*name* 은 변수에 할당된 식별자입니다. *type* 은 변수에 지정된 데이터 형식입니다. 만약 [: = *expression* ]이 지정되어 있으면, 블록이 실행될 때 변수에 기본값이 할당됩니다. 만약 이 항목이 없는 경우, 변수는 SQL null 값으로 초기화됩니다.

기본값은 블록이 실행될 때마다 평가됩니다. 예를 들어 DATE 형식의 변수에 SYSDATE 를 지정하면, 이 변수는 프로시저와 함수가 사전에 컴파일 됐던 시간이 아닌 현재 호출 시간을 가집니다.

다음 프로시저는 문자열과 숫자 표현의 기본값을 사용하는 변수 선언의 예를 보여줍니다.

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
p_deptno NUMBER
)
IS
todays_date DATE : = SYSDATE;
rpt_title VARCHAR2 (60) : = 'Report For Department #'| | p_deptno
| | 'on'| | todays_date;
base_sal INTEGER : = 35525;
base_comm_rate NUMBER : = 1.33333;
base_annual NUMBER : = ROUND (base_sal * base_comm_rate, 2);
BEGIN
DBMS_OUTPUT.PUT_LINE (rpt_title);
DBMS_OUTPUT.PUT_LINE ( 'Base Annual Salary :'| | base_annual);
END;
```

다음은 위의 프로시저 출력이 변수 선언으로 기본값이 확실하게 변수로 할당되었다는 것을 보여줍니다.

```
EXEC dept_salary_rpt (20);

Report For Department # 20 on 10 - JUL - 07 16:44:45
Base Annual Salary : 47366.55
```

### 4.3.2 변수 선언에서 % TYPE 사용

데이터베이스 테이블 값을 변수로 SPL 프로그램에서 선언하고 싶은 경우가 있습니다. 테이블 열을 SPL 변수 간의 호환되도록 하기 위해, 2 개의 데이터 형식은 동일해야 합니다.

그러나, 드물지만 테이블 정의에 변화가 발생할 수 있습니다. 만약 열의 데이터 형식을 변경하는 경우에는 SPL 프로그램 변수에도 대응하는 변화가 필요합니다.

변수 선언에 사용할 데이터 형식을 코딩 하는 대신, 열 특성 % TYPE 을 대신 사용할 수 있습니다. 점(dot) 표기에 의한 자격 있는 이름이나 이전에 선언했던 변수 이름 앞에 % TYPE 를 지정해야 합니다. 열의 데이터 형식 및 % TYPE 이 앞에 붙은 변수는 변수로 선언됩니다. 만약 지정된 열과 변수의 데이터 유형이 변경 되더라도 선언 코드를 수정하지 않고, 관련 변수는 새로운 데이터 형식으로 전환됩니다.

**참고 :** % TYPE 속성은 형식 인자의 정의에서 사용될 수 있습니다.

*name (table. column* | *variable)* % TYPE;

*Name 은* 선언되는 변수와 형식 인자의 식별자입니다. *Column 은 table 이나 뷰의* 열 이름입니다. *Variable 는 name 으로* 식별하는 미리 정의된 변수 이름입니다.

다음 예제는 프로시저 직원 수를 사용해서, emp 테이블을 쿼리하고 직원 데이터를 표시하여, 해당 직원이 근무하는 부서의 모든 직원의 평균 임금을 계산합니다. 그리고 선택한 직원의 급여를 해당 부서의 평균 급여와 비교합니다.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
p_empno IN NUMBER
)
IS
v_ename VARCHAR2 (10);
v_job VARCHAR2 (9);
v_hiredate DATE;
v_sal NUMBER (7,2);
v_deptno NUMBER (2);
v_avgsal NUMBER (7,2);
BEGIN
SELECT ename, job, hiredate, sal, deptno
INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
FROM emp WHERE empno = p_empno;
DBMS_OUTPUT.PUT_LINE ( 'Employee # :'| | p_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | v_ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | v_job);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | v_hiredate);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | v_sal);
DBMS_OUTPUT.PUT_LINE ( 'Dept # :'| | v_deptno);

SELECT AVG (sal) INTO v_avgsal
FROM emp WHERE deptno = v_deptno;
IF v_sal> v_avgsal THEN
DBMS_OUTPUT.PUT_LINE ( 'Employee' 's salary is more than the department'
```

```
| | 'average of'| | v_avgsal);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'Employee' 's salary does not exceed the department'
| | 'average of'| | v_avgsal);
END IF;
END;
```

위의 예제에서 프로시저의 선언 영역으로 emp 테이블의 데이터 형식을 명시적으로 코딩하지 않고 쓸 수 있습니다.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
p_empno IN emp.empno % TYPE
)
IS
v_ename emp.ename % TYPE;
v_job emp.job % TYPE;
v_hiredate emp.hiredate % TYPE;
v_sal emp.sal % TYPE;
v_deptno emp.deptno % TYPE;
v_avgsal v_sal % TYPE;
BEGIN
SELECT ename, job, hiredate, sal, deptno
INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
FROM emp WHERE empno = p_empno;
DBMS_OUTPUT.PUT_LINE ( 'Employee # :'| | p_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | v_ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | v_job);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | v_hiredate);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | v_sal);
DBMS_OUTPUT.PUT_LINE ( 'Dept # :'| | v_deptno);

SELECT AVG (sal) INTO v_avgsal
FROM emp WHERE deptno = v_deptno;
IF v_sal> v_avgsal THEN
DBMS_OUTPUT.PUT_LINE ( 'Employee' 's salary is more than the department'
| | 'average of'| | v_avgsal);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'Employee' 's salary does not exceed the department'
| | 'average of'| | v_avgsal);
END IF;
END;
```

**참고**: p_empno 은 **%** TYPE 을 사용하여 정의한 형식 인자의 예를 보여줍니다. v_avgsal 는 테이블 열 대신 다른 변수를 참조하는 % TYPE 사용법을 나타냅니다.

다음은 이 프로시저 실행 결과의 예입니다.

```
EXEC emp_sal_query (7698);

Employee # : 7698
Name : BLAKE
Job : MANAGER
Hire Date : 01 - MAY - 81 00:00:00
Salary : 2850.00
Dept # : 30
Employee 's salary is more than the department average of 1566.67
```

### 4.3.3 레코드 선언에서의 % ROWTYPE 사용

% TYPE 속성을 통해 열의 데이터 형식에 의존하는 변수를 쉽게 만들 수 있습니다. % ROWTYPE 속성을 통해 지정한 테이블의 모든 열 필드의 레코드를 정의할 수 있습니다. 각 필드는 해당 열의 데이터 형식입니다.

**참고:** 레코드의 필드는 NOT NULL 항목이나 DEFAULT 항목으로 정의되는 다른 열의 속성 중 어느 것도 상속 받지 않습니다.

*Record 는* 이름은 필드 집합에 지시를 내립니다. *Field 는* 변수와 마찬가지로 식별자와 데이터 형식을 갖습니다. 그러나 레코드에 속하는 속성을 가지며, 레코드 이름을 수식어로서 점 표기법을 사용하여 참조합니다.

레코드는 % ROWTYPE 속성을 사용하여 선언됩니다. % ROWTYPE 속성은 테이블 이름 앞에 붙습니다. 지정한 테이블의 각 열은, 열과 같은 데이터 형식의 레코드 이름이 지정된 필드로 정의됩니다.

*record* *table* % ROWTYPE;

*Record* 은 레코드에 할당된 식별자입니다. *Table 은* 열을 레코드의 필드를 정의하는 이름입니다. 뷰 또한 레코드를 정의하기 위해 사용됩니다.

다음은 이전 영역의 emp_sal_query 프로시저가 emp 테이블의 열을 별도의 변수로 정의하지 않고, r_emp 레코드를 emp % ROWTYPE 을 이용하여 만들면 어떻게 변경될 수 있는지 보여줍니다.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
p_empno IN emp.empno % TYPE
)
IS
r_emp emp % ROWTYPE;
v_avgsal emp.sal % TYPE;
BEGIN
SELECT ename, job, hiredate, sal, deptno
INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
FROM emp WHERE empno = p_empno;
DBMS_OUTPUT.PUT_LINE ( 'Employee # :'| | p_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | r_emp.ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | r_emp.job);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | r_emp.hiredate);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | r_emp.sal);
DBMS_OUTPUT.PUT_LINE ( 'Dept # :'| | r_emp.deptno);

SELECT AVG (sal) INTO v_avgsal
FROM emp WHERE deptno = r_emp.deptno;
IF r_emp.sal> v_avgsal THEN
```

```
DBMS_OUTPUT.PUT_LINE ( 'Employee' 's salary is more than the department'
| | 'average of'| | v_avgsal);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'Employee' 's salary does not exceed the department'
| | 'average of'| | v_avgsal);
END IF;
END;
```

### 4.3.4 사용자 정의 레코드 형과 레코드 변수

레코드는 % ROWTYPE 속성을 사용하여 테이블의 정의를 기반으로 선언할 수 있는 부분을 4.3.3 절에서 설명했습니다. 본 절에서는 어떤 특정한 테이블 정의에 얽매이지 않는 새로운 레코드 구조를 정의하는 방법을 설명합니다,

TYPE IS RECORD 구문은 레코드 형식 정의를 만드는 데 사용합니다. *record type 은* 하나 이상의 레코드와 각각의 데이터 형식으로 구성됩니다. 레코드 형식 자체가 데이터 조작에 이용되는 것은 아닙니다.

다음은 레코드 형식을 정의하는 구문입니다.

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

*rectype* 레코드 형식에 할당된 식별자입니다. *field_1, field_2 ...* 레코드 형식 필드에 할당된 식별자입니다. *datatype_1, data type_2 ...은* 각각 *field_1, field_2 ...* 데이터 형식입니다.

*record variable* 아니면 그냥 *record* 레코드 형식의 인스턴스입니다. 레코드는 레코드 형식으로 선언됩니다. 필드 이름이나 형식 등의 레코드 속성은 레코드 형식에서 상속됩니다.

다음은 레코드 선언 구문입니다.

*record rectype*

*record* 레코드 변수에 할당된 식별자입니다. *Rectype 은* 이전에 정의된 레코드 형식의 식별자입니다. 한번 선언되면 레코드는 데이터를 보유하기 위해 사용할 수 있습니다.

레코드 필드를 참조하기 위해서는 점 표기법을 사용합니다.

*record.field*

*record* 은 이전에 선언한 레코드 변수이며, *field 는 record 가* 정의한 레코드 유형에 속하는 필드의 식별자입니다.

다시 emp_sal_query 을 변경합니다. - 이번에는 사용자 정의 레코드 형과 레코드 변수를 사용하십시오.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
p_empno IN emp.empno % TYPE
)
IS
TYPE emp_typ IS RECORD (
ename emp.ename % TYPE,
job emp.job % TYPE,
hiredate emp.hiredate % TYPE,
sal emp.sal % TYPE,
deptno emp.deptno % TYPE
);
r_emp emp_typ;
v_avgsal emp.sal % TYPE;
BEGIN
SELECT ename, job, hiredate, sal, deptno
INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
FROM emp WHERE empno = p_empno;
DBMS_OUTPUT.PUT_LINE ( 'Employee # :'| | p_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | r_emp.ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | r_emp.job);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | r_emp.hiredate);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | r_emp.sal);
DBMS_OUTPUT.PUT_LINE ( 'Dept # :'| | r_emp.deptno);

SELECT AVG (sal) INTO v_avgsal
FROM emp WHERE deptno = r_emp.deptno;
IF r_emp.sal> v_avgsal THEN
DBMS_OUTPUT.PUT_LINE ( 'Employee' 's salary is more than the'
| | 'department average of'| | v_avgsal);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'Employee' 's salary does not exceed the'
| | 'department average of'| | v_avgsal);
END IF;
END;
```

데이터 형식의 이름을 지정하는 대신, % TYPE 속성을 레코드 타입을 정의하는 필드 데이터 형식으로 사용될 수 있는 것을 주의하십시오.

다음은 프로시저를 실행 결과입니다.

```
EXEC emp_sal_query (7698);

Employee # : 7698
Name : BLAKE
Job : MANAGER
Hire Date : 01 - MAY - 81 00:00:00
Salary : 2850.00
Dept # : 30
Employee 's salary is more than the department average of 1566.67
```

### 4.4 기본 문장

이번 절에서는 SPL 프로그램에서 사용되는 프로그래밍 문장에 대해 설명합니다.

#### 4.4.1 NULL

가장 간단한 문장은 NULL 문장입니다. 이 문장은 실행 가능한 문장이지만, 아무것도 하지 않습니다.

```
NULL;
```

다음은 가장 간단하고도 효과적인 SPL 프로그램입니다.

```
BEGIN
NULL;
END;
```

NULL 문은 IF - THEN - ELSE 문장 분기와 같은 실행 가능한 문장이 필요한 플레이스홀더로 유용합니다.

예를 들면 다음과 같습니다:

```
CREATE OR REPLACE PROCEDURE divide_it (
p_numerator IN NUMBER,
p_denominator IN NUMBER,
p_result OUT NUMBER
)
IS
BEGIN
IF p_denominator = 0 THEN
NULL;
ELSE
p_result : = p_numerator / p_denominator;
END IF;
END;
```

#### 4.4.2 지정

대입 문장은 변수와 OUT, IN OUT 모드의 형식 인자로 구성됩니다. : = 왼쪽 위치를 지정하고 오른쪽 위치에 수식을 지정합니다.

*variable* : = *expression;*

*variable* 은 이전에 선언한 변수, O UT 형식 인자 또는 IN OUT 형식 인자를 위한 식별자입니다. *expression* 는 단일 값을 생성하는 식입니다. 이 표현에 의해 생성된 값은 *variable* 에 적합한 데이터 형식을 가지고 있어야 합니다.

4.3 절의 dept_salary_rpt 예제는 변수 선언에서 대입 문장을 사용한 것을 나타냅니다. 이 예제와의 차이점은 프로시저 실행 영역에서 대입 문장의 대표적인 사용 방법입니다.

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
p_deptno NUMBER
)
IS
todays_date DATE;
rpt_title VARCHAR2 (60);
base_sal INTEGER;
base_comm_rate NUMBER;
base_annual NUMBER;
BEGIN
todays_date : = SYSDATE;
rpt_title : = 'Report For Department #'| | p_deptno | | 'on'
| | todays_date;
base_sal : = 35525;
base_comm_rate : = 1.33333;
base_annual : = ROUND (base_sal * base_comm_rate, 2);

DBMS_OUTPUT.PUT_LINE (rpt_title);
DBMS_OUTPUT.PUT_LINE ( 'Base Annual Salary :'| | base_annual);
END;
```

### 4.4.3 SELECT INTO

SELECT INTO 문은 SQL SELECT 커맨드 SPL 의 변형입니다. 그 차이는 다음과 같습니다 :

- SELECT INTO는 그 결과를 SPL프로그램 문장에서 유효한 변수와 레코드에 할당하도록 설계되었습니다
- SELECT INTO에 접근할 수 있는 결과는 대부분 1행입니다

위를 제외하고, WHERE, ORDER BY, GROUP BY, HAVING 와 같은 SELECT 커맨드의 모든 항목은 SELECT INTO 에도 유효합니다. 다음은 SELECT INTO 의 2 가지 변형입니다.

*SELECT* select_expressions INTO target FROM ...;

*target 은* 콤마로 분리된 간단한 변수의 리스트입니다. *select_expressions 과* 문장의 나머지 부분은 SELECT 커맨드와 같습니다. 검색된 값은 데이터 형식, 수 그리고 결과 구조의 순서가 일치해야 합니다. 그렇지 않으면 런타임 오류가 발생합니다.

SELECT * *INTO* record FROM table -;

*record 은* 미리 선언된 레코드 변수입니다.

만일 쿼리가 0 행으로 리턴되면, null 값이 *target 으로* 설정됩니다. 만일 쿼리가 여러 행에 리턴되면, 첫 번째 행이 *target 로* 설정하고 나머지 부분은 무시됩니다. (ORDER BY 를 사용하지 않는 한, "첫 행"은 명확하게 정의되지 않으므로 주의하시기 바랍니다.)

**참고:** 행이 리턴되지 않거나 하나 이상의 행이 리턴되는 경우 모두, SPL 에서 예외가 발생합니다.

**참고**: SELECT INTO 에 BULK COL LECT 항목을 사용하면, 컬렉션에 반환된 1 줄 이상의 결과 집합을 허용됩니다. SELECT INTO 문장의 BULK COLLECT 항목 이용 방법에 대한 자세한 내용은 4.9.4.1 절을 참조하십시오.

EXCEPTION 블록 지정이 성공했는지 여부(여기에는 적어도 1 줄 이상의 행을 반환하는)을 결정하기 위해 WHEN NO_DATA_FOUND 항목을 사용할 수 있습니다.

emp_sal_quer y 프로시저의 버전은 결과 집합을 레코드에 반환하는 SELECT INTO 의 변형을 사용합니다. 또한 WHEN NO_DATA_FOUND 조건식을 가진 EXCEPTION 블록이 추가되었다는 것을 주의하십시오.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
p_empno IN emp.empno % TYPE
)
IS
r_emp emp % ROWTYPE;
v_avgsal emp.sal % TYPE;
BEGIN
SELECT * INTO r_emp
FROM emp WHERE empno = p_empno;
DBMS_OUTPUT.PUT_LINE ( 'Employee # :'| | p_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | r_emp.ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | r_emp.job);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | r_emp.hiredate);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | r_emp.sal);
DBMS_OUTPUT.PUT_LINE ( 'Dept # :'| | r_emp.deptno);

SELECT AVG (sal) INTO v_avgsal
FROM emp WHERE deptno = r_emp.deptno;
IF r_emp.sal> v_avgsal THEN
DBMS_OUTPUT.PUT_LINE ( 'Employee' 's salary is more than the'
| | 'department average of'| | v_avgsal);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'Employee' 's salary does not exceed the'
| | 'department average of'| | v_avgsal);
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Employee #'| | p_empno | | 'not found');
END;
```

만일 검색이 존재하는 직원 없이 실행된다면, 결과는 다음과 같습니다.

```
EXEC emp_sal_query (0);

Employee # 0 not found
```

SELECT INTO 의 EXCEPTION 영역에서 다른 조건 항목은 TOO_MANY_ROWS 예외입니다. 만약 SELECT INTO 문장에서 하나 이상의 행이 검색되면, SPL 에서 예외가 발생합니다.

다음 블록이 실행되면 특정한 부서에 많은 직원이 등록되어 있기 때문에, TOO_MANY_ROWS 예외가 발생합니다.

```
DECLARE
v_ename emp.ename % TYPE;
BEGIN
SELECT ename INTO v_ename FROM emp WHERE deptno = 20 ORDER BY ename;
EXCEPTION
WHEN TOO_MANY_ROWS THEN
DBMS_OUTPUT.PUT_LINE ( 'More than one employee found');
DBMS_OUTPUT.PUT_LINE ( 'First employee returned is'| | v_ename);
END;

More than one employee found
First employee returned is ADAMS
```

**참고 :** 예외 처리에 대한 자세한 내용은 4.5.5 절을 참조하십시오.

4.4.4 INSERT

SQL 언어에서 이용할 수 있는 INSERT 커맨드는 SPL 프로그램에서도 사용할 수 있습니다.

SPL 언어 표현식은 SQL INSERT 커맨드로 허용되는 표현의 모든 위치에서 사용할 수 있습니다. 따라서 SPL 변수 및 인자를 삽입 연산에 값을 전달할 수 있습니다.

다음은 호출 프로그램에서 데이터를 전달하여 새로운 직원의 삽입을 수행하는 프로시저의 예입니다.

```
CREATE OR REPLACE PROCEDURE emp_insert (
p_empno IN emp.empno % TYPE,
p_ename IN emp.ename % TYPE,
p_job IN emp.job % TYPE,
p_mgr IN emp.mgr % TYPE,
p_hiredate IN emp.hiredate % TYPE,
p_sal IN emp.sal % TYPE,
p_comm IN emp.comm % TYPE,
p_deptno IN emp.deptno % TYPE
)
IS
BEGIN
INSERT INTO emp VALUES (
p_empno,
```

```
p_ename,
p_job,
p_mgr,
p_hiredate,
p_sal,
p_comm,
p_deptno);

DBMS_OUTPUT.PUT_LINE ( 'Added employee ...');
DBMS_OUTPUT.PUT_LINE ( 'Employee # :'| | p_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | p_ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | p_job);
DBMS_OUTPUT.PUT_LINE ( 'Manager :'| | p_mgr);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | p_hiredate);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | p_sal);
DBMS_OUTPUT.PUT_LINE ( 'Commission :'| | p_comm);
DBMS_OUTPUT.PUT_LINE ( 'Dept # :'| | p_deptno);
DBMS_OUTPUT.PUT_LINE ('----------------------');
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'OTHERS exception on INSERT of employee #'
| | p_empno);
DBMS_OUTPUT.PUT_LINE ( 'SQLCODE :'| | SQLCODE);
DBMS_OUTPUT.PUT_LINE ( 'SQLERRM :'| | SQLERRM);
END;
```

만약 예외가 발생하면 프로시저에서 만들어지는 모든 데이터베이스의 변경은 자동으로 롤백됩니다. 이 경우, WHEN OTHERS 구문의 EXCEPTION 영역의 모든 예외를 가져옵니다. 2 가지 변수가 나타납니다. SQLCODE 는 발생한 예외를 확인하는 숫자입니다. SQLERRM 는 오류를 설명하는 텍스트 메시지입니다. 예외 처리에 대한 자세한 내용은 4.5.5 절을 참조하십시오.

다음은 이 프로시저를 실행할 때 출력을 나타냅니다.

```
EXEC emp_insert (9503, 'PETERSON', 'ANALYST', 7902, '31 - MAR - 05 ', 5000, NULL, 40);

Added employee ...
Employee # : 9503
Name : PETERSON
Job : ANALYST
Manager : 7902
Hire Date : 31 - MAR - 05 00:00:00
Salary : 5000
Dept # : 40
----------------------

SELECT * FROM emp WHERE empno = 9503;

empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+----------+---------+------+-------------- ------+---------+------+--------
9503 | PETERSON | ANALYST | 7902 | 31 - MAR - 05 00:00:00 | 5000.00 | | 40
(1 row)
```

참고: INSERT 커맨드는 FORALL 문장에 포함될 수 있습니다. FORALL 문장은 1 개의 INSERT 커맨드를 하나 이상의 컬렉션에 전달된 값으로부터 여러 행을 삽입하도록 합니다. FORALL 문장에 대한 정보는 4.9.3 절을 참조하십시오.

### 4.4.5 UPDATE

SQL 언어에서 사용할 수 UPDATE 커맨드는 SPL 프로그램에서도 사용할 수 있습니다.

SPL 언어의 표현식은 SQL UPDATE 커맨드에서 허용되는 표현식의 모든 곳에서 사용할 수 있습니다. 따라서 SPL 변수와 인자는 업데이트 연산에서 값을 전달할 수 있습니다.

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
p_empno IN emp.empno % TYPE,
p_sal IN emp.sal % TYPE,
p_comm IN emp.comm % TYPE
)
IS
BEGIN
UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno;

IF SQL % FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Updated Employee # :'| | p_empno);
DBMS_OUTPUT.PUT_LINE ( 'New Salary :'| | p_sal);
DBMS_OUTPUT.PUT_LINE ( 'New Commission :'| | p_comm);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'Employee #'| | p_empno | | 'not found');
END IF;
END;
```

만약 행이 업데이트된 경우는 SQL % FOUND 조건식이 "true"를 반환하고, 그렇지 않은 경우는 "false"을 반환합니다. SQL % FOUND 과 기타 유사한 표현에 대한 설명은 4.4.8 절을 참조하십시오.

다음은 이 프로시저를 사용하여 직원을 갱신하고 있습니다.

```
EXEC emp_comp_update (9503, 6540, 1200);

Updated Employee # : 9503
New Salary : 6540
New Commission : 1200

SELECT * FROM emp WHERE empno = 9503;

empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+----------+---------+------+-------------- ------+---------+---------+--------
9503 | PETERSON | ANALYST | 7902 | 31 - MAR - 05 00:00:00 | 6540.00 | 1200.00 | 40
(1 row)
```

**참고**: UPDATE 커맨드는 FORALL 문장에 포함됩니다. FORALL 문장은 1 개의 UPDATE 커맨드에서 1 개 이상의 컬렉션에 전달된 값으로 여러 행을 업데이트하는 것을 허용합니다. 4.9.3 절에서 FORALL 문장에 대한 정보를 참조하십시오.

4.4.6 DELETE

SQL 언어에서 사용할 수 DELETE 커맨드는 SPL 프로그램에서도 사용할 수 있습니다.

SPL 언어의 표현은 SQL DELETE 커맨드에서 허용되는 표현의 모든 위치에서 사용할 수 있습니다. 따라서 SPL 변수와 인자는 삭제 연산에 값을 전달할 수 있습니다.

```
CREATE OR REPLACE PROCEDURE emp_delete (
p_empno IN emp.empno % TYPE
)
IS
BEGIN
DELETE FROM emp WHERE empno = p_empno;

IF SQL % FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Deleted Employee # :'| | p_empno);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'Employee #'| | p_empno | | 'not found');
END IF;
END;
```

만약 행을 삭제한 경우, SQL%FOUND 조건식이 "true"를 반환하고, 그렇지 않은 경우 "false"을 반환합니다. SQL%FOUND 과 기타 유사한 표현에 대한 설명은 4.4.8 절을 참조하십시오.

다음은 이 프로시저를 통해 직원을 제거합니다.

```
EXEC emp_delete (9503);

Deleted Employee # : 9503

SELECT * FROM emp WHERE empno = 9503;

empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+----- -+--------
(0 rows)
```

**참고**: DELETE 커맨드는 FORALL 문장에 포함됩니다. FORALL 문장은 1 개의 DELETE 커맨드에서 1 개 이상의 컬렉션에 전달된 값으로 여러 행을 제거하는 것을 허용합니다. FORALL 문장에 대한 정보는 4.9.3 절을 참조하십시오.

**4.4.7 RETURNING INTO 구문 사용**

INSERT, UPDATE, DELETE 커맨드는 RETURNING INTO 구문을 선택적으로 사용할 수 있습니다. 이 구문은 SPL 프로그램이 새로이 추가, 변경 또는 삭제된 값을 INSERT, UPDATE 및 DELETE 커맨드 각각의 결과로 볼 수 있도록 합니다.

다음이 그 형식입니다.

```
(insert | update | delete)
  RETURNING (* | expr_1 [, expr_2] ...)
   INTO (     record | field_1 [, field_2] ...);
```

*Insert 는* 효과적인 INSERT 커맨드입니다. *update 는* 효과적인 UPDATE 커맨드입니다. *delete 는* 효과적인 DELETE 커맨드입니다. 만약 *가 지정되면, INSERT, UPDATE 및 DELETE 커맨드에 영향을 받은 행의 값은 INTO 키워드의 레코드와 오른쪽 필드에 할당됩니다. (주 : * 사용은 Postgres Plus Advanced Server 의 확장이며, Oracle 와 호환되지 않습니다. ) *expr_1, expr_2* ...는 INSERT, UPDATE 및 DELETE 커맨드로 처리된 행에 대한 수식입니다. 평가 결과는 INTO 키워드의 오른쪽 필드나 레코드로 할당됩니다. *record 는* 숫자와 순서가 일치하는 필드로 구성된 레코드 식별자에서 데이터 형식이 RETURNING 항목의 값과 일치합니다. *field_1, field_2* ...는 숫자와 순서가 일치하는 변수이고, 데이터 형식은 RETURNING 항목의 값 집합과 일치합니다.

만약 INSERT, UPDATE 및 DELETE 커맨드가 1 줄 이상의 결과 집합을 반환하면, SQLCODE 01422, query returned more than one row 라는 예외가 발생합니다. 만약 결과 집합에 행이 없으면, INTO 키워드를 따르는 변수는 null 로 설정됩니다.

**참고**: RETURNING INTO 의 변형에서 BULK COLLECT 항목을 사용할 경우, 컬렉션에 반환된 1 행 이상의 결과 집합을 허용합니다. BULK COLLECT 항목 이용 방법에 대한 자세한 내용은 4.9.4 절을 참조하십시오.

다음은 4.4.5 절에서 소개한 emp_comp_update 프로시저, RETURNING INTO 항목을 추가하여 수정한 예제입니다.

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
p_empno IN emp.empno % TYPE,
p_sal IN emp.sal % TYPE,
p_comm IN emp.comm % TYPE
)
IS
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
v_job emp.job % TYPE;
v_sal emp.sal % TYPE;
v_comm emp.comm % TYPE;
v_deptno emp.deptno % TYPE;
BEGIN
UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
RETURNING
```

```
empno,
ename,
job,
sal,
comm,
deptno
INTO
v_empno,
v_ename,
v_job,
v_sal,
v_comm,
v_deptno;

IF SQL % FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Updated Employee # :'| | v_empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | v_ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | v_job);
DBMS_OUTPUT.PUT_LINE ( 'Department :'| | v_deptno);
DBMS_OUTPUT.PUT_LINE ( 'New Salary :'| | v_sal);
DBMS_OUTPUT.PUT_LINE ( 'New Commission :'| | v_comm);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'Employee #'| | p_empno | | 'not found');
END IF;
END;
```

다음은 4.4.4 절의 emp_insert 프로시저에 의해 만들어진 직원 ID 9503 이 아직 테이블에 존재한다는 것을 가정한, 프로시저의 출력 결과입니다.

```
EXEC emp_comp_update (9503, 6540, 1200);

Updated Employee # : 9503
Name : PETERSON
Job : ANALYST
Department : 40
New Salary : 6540.00
New Commission : 1200.00
```

다음 예제는 레코드 형식을 이용한 RETURNING INTO 구문을 4.4.6 절의 emp_delete 프로시저에 추가하고 수정합니다.

```
CREATE OR REPLACE PROCEDURE emp_delete (
p_empno IN emp.empno % TYPE
)
IS
r_emp emp % ROWTYPE;
BEGIN
DELETE FROM emp WHERE empno = p_empno
RETURNING
*
INTO
r_emp;
```

```
IF SQL % FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Deleted Employee # :'| | r_emp.empno);
DBMS_OUTPUT.PUT_LINE ( 'Name :'| | r_emp.ename);
DBMS_OUTPUT.PUT_LINE ( 'Job :'| | r_emp.job);
DBMS_OUTPUT.PUT_LINE ( 'Manager :'| | r_emp.mgr);
DBMS_OUTPUT.PUT_LINE ( 'Hire Date :'| | r_emp.hiredate);
DBMS_OUTPUT.PUT_LINE ( 'Salary :'| | r_emp.sal);
DBMS_OUTPUT.PUT_LINE ( 'Commission :'| | r_emp.comm);
DBMS_OUTPUT.PUT_LINE ( 'Department :'| | r_emp.deptno);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'Employee #'| | p_empno | | 'not found');
END IF;
END;
```

다음은 이 프로시저의 출력 결과입니다.

```
EXEC emp_delete (9503);

Deleted Employee # : 9503
Name : PETERSON
Job : ANALYST
Manager : 7902
Hire Date : 31 - MAR - 05 00:00:00
Salary : 6540.00
Commission : 1200.00
Department : 40
```

### 4.4.8 결과 상태 획득

커맨드의 효과를 확인하기 위해 사용되는 몇 가지 특성이 있습니다. SQL % FOUND 는 INSERT, UPDATE 및 DELETE 커맨드에서 적어도 1 개의 행이 변경되거나 SELECT INTO 커맨드가 1 이상의 행이 검색되면 true 를 반환하는 Boolean 형식입니다.

다음의 익명 블록은 행을 삽입하고, 행이 삽입된 것을 볼 수 있습니다.

```
BEGIN
INSERT INTO emp (empno, ename, job, sal, deptno) VALUES (
9001, 'JONES', 'CLERK', 850.00, 40);
IF SQL % FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Row has been inserted');
END IF;
END;

Row has been inserted
```

SQL % ROWCOUNT 는 INSERT, UPDATE 및 DELETE 커맨드로 처리된 행의 수를 제공합니다. 다음 예제는 지금 삽입된 행을 업데이트하여 SQL % ROWCOUNT 의 값을 표시합니다.

```
BEGIN
UPDATE emp SET hiredate = '03 - JUN - 07 'WHERE empno = 9001;
DBMS_OUTPUT.PUT_LINE ( '# rows updated :'| | SQL % ROWCOUNT);
```

```
END;

# rows updated : 1
```

SQL % NOTFOUND 는 SQL % FOUND 와 반대입니다. 만약 INSERT, UPDATE 및 DELETE 커맨드가 1 행도 처리하지 않거나 SELECT INTO 커맨드가 1 행도 찾을 수 없다면, SQL % NOTFOUND 는 true 를 반환합니다.

```
BEGIN
UPDATE emp SET hiredate = '03 - JUN - 07 'WHERE empno = 9000;
IF SQL % NOTFOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'No rows were updated');
END IF;
END;

No rows were updated
```

### 4.5 제어 구조

SPL 의 프로그램 문장을 SQL 의 절차적 보완으로 만드는 것에 대해 본 절에서 설명합니다.

#### 4.5.1 IF 문장

IF 문장은 조건에 따라 커맨드를 실행시킵니다. SPL 은 다음과 같은 4 개의 IF 형식이 있습니다. :

- IF … THEN
- IF … THEN … ELSE
- IF … THEN … ELSE IF
- IF … THEN … ELSIF … THEN … ELSE

##### 4.5.1.1 IF - THEN

*IF* boolean - *expression THEN*
*statements*
END IF;

IF - THEN 문장은 가장 간단한 IF 형식입니다. THEN 과 END IF 사이의 문장은 조건이 true 일 때 실행됩니다. 반대의 경우, 건너 뜁니다.

다음의 예제에서 IF-THEN 문장은 수수료를 가지는 직원을 표시하고 검사하는데 사용됩니다.

```
DECLARE
v_empno emp.empno % TYPE;
v_comm emp.comm % TYPE;
CURSOR emp_cursor IS SELECT empno, comm FROM emp;
```

```
BEGIN
OPEN emp_cursor;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO COMM');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH emp_cursor INTO v_empno, v_comm;
EXIT WHEN emp_cursor % NOTFOUND;
-
- Test whether or not the employee gets a commission
-
IF v_comm IS NOT NULL AND v_comm> 0 THEN
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| |
TO_CHAR (v_comm, '$ 99999.99'));
END IF;
END LOOP;
CLOSE emp_cursor;
END;
```

**다음은 이 프로그램의 출력입니다.**

```
EMPNO COMM
----- -------
7499 $ 300.00
7521 $ 500.00
7654 $ 1400.00
```

#### 4.5.1.2 IF - THEN - ELSE

*IF* boolean - *expression THEN*
  *statements*
ELSE
  *statements*
END IF;

IF - THEN - ELSE 문장은 IF - THEN 이외 조건 평가가 false 인 경우, 대신하는 문장의 집합을 지정함으로써 IF-THEN 에 IF-THEN-ELSE 문장을 추가할 수 있습니다.

다음 예제는 만일 직원이 수수료를 받을 수 없는 경우, IF - THEN - ELSE 구문은 Non - commission 텍스트를 볼 수 있도록 변경합니다.

```
DECLARE
v_empno emp.empno % TYPE;
v_comm emp.comm % TYPE;
CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
OPEN emp_cursor;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO COMM');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH emp_cursor INTO v_empno, v_comm;
```

```
EXIT WHEN emp_cursor % NOTFOUND;
-
- Test whether or not the employee gets a commission
-
IF v_comm IS NOT NULL AND v_comm> 0 THEN
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| |
TO_CHAR (v_comm, '$ 99999.99'));
ELSE
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | 'Non - commission');
END IF;
END LOOP;
CLOSE emp_cursor;
END;
```

다음은 이 프로그램의 출력입니다.

```
EMPNO COMM
----- -------
7369 Non - commission
7499 $ 300.00
7521 $ 500.00
7566 Non - commission
7654 $ 1400.00
7698 Non - commission
7782 Non - commission
7788 Non - commission
7839 Non - commission
7844 Non - commission
7876 Non - commission
7900 Non - commission
7902 Non - commission
7934 Non - commission
```

#### 4.5.1.3 IF - THEN - ELSE IF

IF 문장은 대신할 IF 문장을 호출하기 위해서 중첩될 수 있습니다. 이는 외부 IF 문장의 조건이 true 인지 false 인지에 따라 결정됩니다.

다음 예제에서는 외부 IF - THEN - ELSE 문장은 직원이 수수료를 가지고 있는지 여부를 테스트합니다. 내부 IF - THEN - ELSE 문은 직원의 총 보상액이 회사의 평균을 초과하는지 여부를 테스트합니다.

```
DECLARE
v_empno emp.empno % TYPE;
v_sal emp.sal % TYPE;
v_comm emp.comm % TYPE;
v_avg NUMBER (7,2);
CURSOR emp_cursor IS SELECT empno, sal, comm FROM emp;
BEGIN
-
- Calculate the average yearly compensation in the company
-
```

```
SELECT AVG ((sal + NVL (comm, 0)) * 24) INTO v_avg FROM emp;
DBMS_OUTPUT.PUT_LINE ( 'Average Yearly Compensation :'| |
TO_CHAR (v_avg, '$ 999,999.99'));
OPEN emp_cursor;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO YEARLY COMP');
DBMS_OUTPUT.PUT_LINE ('----- ----------');
LOOP
FETCH emp_cursor INTO v_empno, v_sal, v_comm;
EXIT WHEN emp_cursor % NOTFOUND;
-
- Test whether or not the employee gets a commission
-
IF v_comm IS NOT NULL AND v_comm> 0 THEN
-
- Test if the employee 's compensation with commission exceeds the average
-
IF (v_sal + v_comm) * 24> v_avg THEN
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| |
TO_CHAR ((v_sal + v_comm) * 24 '$ 999999.99') | |
'Exceeds Average');
ELSE
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| |
TO_CHAR ((v_sal + v_comm) * 24 '$ 999999.99') | |
'Below Average');
END IF;
ELSE
-
- Test if the employee 's compensation without commission exceeds the average
-
IF v_sal * 24> v_avg THEN
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| |
TO_CHAR (v_sal * 24 '$ 999999.99') | | 'Exceeds Average');
ELSE
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| |
TO_CHAR (v_sal * 24 '$ 999999.99') | | 'Below Average');
END IF;
END IF;
END LOOP;
CLOSE emp_cursor;
END;
```

**참고:** 이 프로그램의 논리는 커서 정의의 SELECT 커맨드 내의 NVL 함수를 이용하여 직원의 연간 보상 금액을 계산함으로써 간단해집니다. 그러나 이 예제의 목적은 IF 문장을 어떻게 쓸 것인가를 보여줍니다.

다음은 이 프로그램의 출력입니다.

```
Average Yearly Compensation : $ 53,528.57
EMPNO YEARLY COMP
----- ----------
7369 $ 19,200.00 Below Average
7499 $ 45,600.00 Below Average
7521 $ 42,000.00 Below Average
7566 $ 71,400.00 Exceeds Average
```

```
7654 $ 63,600.00 Exceeds Average
7698 $ 68,400.00 Exceeds Average
7782 $ 58,800.00 Exceeds Average
7788 $ 72,000.00 Exceeds Average
7839 $ 120,000.00 Exceeds Average
7844 $ 36,000.00 Below Average
7876 $ 26,400.00 Below Average
7900 $ 22,800.00 Below Average
7902 $ 72,000.00 Exceeds Average
7934 $ 31,200.00 Below Average
```

이 형식을 사용하면, 실제로는 외부의 IF 문장의 ELSE 부분 내에 IF 문을 중첩합니다. 따라서 중첩된 IF 와 부모 IF-ELSE 를 위한 END IF 문장을 필요합니다.

#### 4.5.1.4 IF - THEN - ELSIF - ELSE

*IF* boolean - *expression THEN*
    *statements*
*[ELSIF* boolean - *expression THEN*
    *statements*
*[ELSIF* boolean - *expression THEN*
    *statemen ts]* ...]
[ELSE
    *statements]*
END IF;

IF - THEN - ELSIF - ELSE 은 1 개의 문장으로 많은 후보를 확인하는 방법을 제공합니다.
공식적으로 이것은 중첩된 IF - THEN - ELSE - IF - THEN 커맨드와 같지만, END IF 가 필요할 때만 입니다.

다음 예제는 IF - THEN - ELSIF - ELSE 문장을 사용하여 보상금 $ 25000 범위에서 직원 수를 계산합니다.

```
DECLARE
v_empno emp.empno % TYPE;
v_comp NUMBER (8,2);
v_lt_25K SMALLINT : = 0;
v_25K_50K SMALLINT : = 0;
v_50K_75K SMALLINT : = 0;
v_75K_100K SMALLINT : = 0;
v_ge_100K SMALLINT : = 0;
CURSOR emp_cursor IS SELECT empno (sal + NVL (comm, 0)) * 24 FROM emp;
BEGIN
OPEN emp_cursor;
LOOP
FETCH emp_cursor INTO v_empno, v_comp;
EXIT WHEN emp_cursor % NOTFOUND;
IF v_comp <25000 THEN
```

```
v_lt_25K : = v_lt_25K + 1;
ELSIF v_comp <50000 THEN
v_25K_50K : = v_25K_50K + 1;
ELSIF v_comp <75000 THEN
v_50K_75K : = v_50K_75K + 1;
ELSIF v_comp <100000 THEN
v_75K_100K : = v_75K_100K + 1;
ELSE
v_ge_100K : = v_ge_100K + 1;
END IF;
END LOOP;
CLOSE emp_cursor;
DBMS_OUTPUT.PUT_LINE ( 'Number of employees by yearly compensation');
DBMS_OUTPUT.PUT_LINE ( 'Less than 25000 :'| | v_lt_25K);
DBMS_OUTPUT.PUT_LINE ('25000 - 49,9999 : '| | v_25K_50K);
DBMS_OUTPUT.PUT_LINE ('50000 - 74,9999 : '| | v_50K_75K);
DBMS_OUTPUT.PUT_LINE ('75000 - 99,9999 : '| | v_75K_100K);
DBMS_OUTPUT.PUT_LINE ('100000 and over : '| | v_ge_100K);
END;
```

**다음은 이 프로그램의 출력입니다.**

```
Number of employees by yearly compensation
Less than 25,000 : 2
25,000 - 49,9999 : 5
50,000 - 74,9999 : 6
75,000 - 99,9999 : 0
100,000 and over : 1
```

### 4.5.2 CASE 식

CASE 표현식은 표현식 내의 CASE 식을 대신하는 값을 반환합니다.

CASE 식에는 2 가지 형식이 있습니다. 하나는 *검색된* CASE 로 불리고, 다른 하나는 *selector* 를 사용합니다.

#### 4.5.2.1 선택 CASE 식

선택 CASE 식은 Selector 로 불리는 식과 1 개 이상의 WHEN 구문이 일치되도록 시도합니다. *result* 는 CASE 식이 사용된 내용에서 형식이 호환되는 식입니다. 만약 일치하면, THEN 항목에 해당하는 값이 CASE 표현식에서 반환됩니다. 일치하지 않으면 ELSE 에 해당하는 값을 반환합니다. ELSE 가 생략되면 CASE 식은 null 을 반환합니다.

*CASE* selector - *expression*
*WHEN* match - *expression THEN*
    *result*
*[WHEN* match - *expression THEN*
    *result*

*[WHEN* match - *expression THEN*
*result* ...]
[ELSE
*result* ]
END;

*match - expression* 은 CASE 식 안에서 나타난 순서로 평가됩니다. *result* 는 CASE 식을 실행하는 식의 내용과 동일 식입니다. 먼저 *match - expression* 이 *selector - expression* 과 일치하면, 해당 THEN 절의 *result* 가 CASE 문장의 값으로 되돌아갑니다. 만약 어떤 *match - expression* 도 *selector - expression* 와 일치하지 않는 경우, ELSE 를 따르는 *result* 를 반환합니다. ELSE 가 지정되지 않으면, CASE 식은 null 을 반환합니다.

다음은 부서 번호를 기반으로 하는 변수에 부서 이름을 할당하기 위하여 선택 CASE 식을 사용하고 있습니다.

```
DECLARE
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
v_deptno emp.deptno % TYPE;
v_dname dept.dname % TYPE;
CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
OPEN emp_cursor;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME DEPTNO DNAME');
DBMS_OUTPUT.PUT_LINE ('----- ------- ------ ----------');
LOOP
FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
EXIT WHEN emp_cursor % NOTFOUND;
v_dname : =
CASE v_deptno
WHEN 10 THEN 'Accounting'
WHEN 20 THEN 'Research'
WHEN 30 THEN 'Sales'
WHEN 40 THEN 'Operations'
ELSE 'unknown'
END;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | RPAD (v_ename 10) | |
''| | v_deptno | | ''| | v_dname);
END LOOP;
CLOSE emp_cursor;
END;
```

다음은 이 프로그램의 출력입니다.

```
EMPNO ENAME DEPTNO DNAME
----- ------- ------ ----------
7369 SMITH 20 Research
7499 ALLEN 30 Sales
7521 WARD 30 Sales
7566 JONES 20 Research
```

```
7654 MARTIN 30 Sales
7698 BLAKE 30 Sales
7782 CLARK 10 Accounting
7788 SCOTT 20 Research
7839 KING 10 Accounting
7844 TURNER 30 Sales
7876 ADAMS 20 Research
7900 JAMES 30 Sales
7902 FORD 20 Research
7934 MILLER 10 Accounting
```

#### 4.5.2.2 검색 CASE 식

검색 CASE 식은 하나 이상의 Boolean 식을 사용해서 리턴되는 결과값을 결정합니다.

*CASE* WHEN boolean - *expression THEN*
    *result*
*[WHEN* boolean - *expression THEN*
    *result*
*[WHEN* boolean - *expression THEN*
    *result* ...]
[ELSE
    *result*    ]
END;

*boolean - expression 은* CASE 식 안에서 나온 순서로 평가됩니다. *result 는* CASE 식을 실행하는 식의 내용과 동일 식입니다. 만약 첫 번째 *boolean – expression 이* true 가 아닐 경우, CASE 식의 값인 해당 THEN 항목의 값을 *result* 에 리턴합니다. 만약 *boolean-expression* 중 true 가 없을 때는 ELSE 다음의 *result* 에 반환합니다. ELSE 가 지정되지 않았다면, CASE 문장은 null 을 반환합니다.

다음 예제는 부서 번호를 기반으로 하는 변수에 부서 이름을 할당하기 위하여 검색 CASE 식을 사용하고 있습니다.

```
DECLARE
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
v_deptno emp.deptno % TYPE;
v_dname dept.dname % TYPE;
CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
OPEN emp_cursor;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME DEPTNO DNAME');
DBMS_OUTPUT.PUT_LINE ('----- ------- ------ ----------');
LOOP
FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
EXIT WHEN emp_cursor % NOTFOUND;
```

```
v_dname : =
CASE
WHEN v_deptno = 10 THEN 'Accounting'
WHEN v_deptno = 20 THEN 'Research'
WHEN v_deptno = 30 THEN 'Sales'
WHEN v_deptno = 40 THEN 'Operations'
ELSE 'unknown'
END;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | RPAD (v_ename 10) | |
''| | v_deptno | | ''| | v_dname);
END LOOP;
CLOSE emp_cursor;
END;
```

다음은 이 프로그램의 출력입니다.

```
EMPNO ENAME DEPTNO DNAME
----- ------- ------ ----------
7369 SMITH 20 Research
7499 ALLEN 30 Sales
7521 WARD 30 Sales
7566 JONES 20 Research
7654 MARTIN 30 Sales
7698 BLAKE 30 Sales
7782 CLARK 10 Accounting
7788 SCOTT 20 Research
7839 KING 10 Accounting
7844 TURNER 30 Sales
7876 ADAMS 20 Research
7900 JAMES 30 Sales
7902 FORD 20 Research
7934 MILLER 10 Accounting
```

### 4.5.3 CASE 문장

CASE 문장은 지정된 검색 조건이 true 일 때, 1 개 이상의 문장을 수행합니다. CASE 문장은 그 자체가 독립적인 문장이지만, 전에 언급한 바와 같이 CASE 식은 표현식의 한 부분으로 표현되어야만 합니다.

CASE 문장에는 2 가지 형식이 있습니다. 하나는 *검색* CASE 이고, 다른 하나는 *선택* CASE 입니다.

#### 4.5.3.1 선택 CASE 문장

선택 CASE 문은 1 개 이상의 WHEN 항목에 지정되는 표현식을 선택하는 것으로, 표현식과 일치하도록 시도됩니다. 일치하는 문장을 찾는다면, 하나나 그 이상의 문장이 실행됩니다.

*CASE* selector - *expression*

*WHEN* match - *expression THEN*

```
    statements
[WHEN match - expression THEN
    statements
[WHEN match - expression THEN
    statements ...]
[ELSE
   statements  ]
END CASE;
```

*selector - expression* 은 각 *match - expression* 과 같은 값을 반환합니다. *match - expression* 은 CASE 문장 속에서 나타나는 순서대로 평가됩니다. *statements* 는 1 개 이상의 SPL 프로그램에서 각각 세미콜론으로 종료됩니다. *selector - expression* 의 값이 처음으로 *match - expression* 과 일치 됐을 때, 대응하는 THEN 절 문장이 실행되고, 계속해서 다음 END CASE 키워드에서 제어됩니다. 일치하지 않을 경우, ELSE 을 따르는 문장이 실행됩니다. 일치하는 항목이 없거나 ELSE 항목이 없는 경우는 예외가 발생합니다.

다음 예제는 부서 번호를 기반으로 하는 변수에 부서 이름과 위치를 할당하기 위하여 선택 CASE 문장을 사용하고 있습니다.

```
DECLARE
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
v_deptno emp.deptno % TYPE;
v_dname dept.dname % TYPE;
v_loc dept.loc % TYPE;
CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
OPEN emp_cursor;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME DEPTNO DNAME'
| | 'LOC');
DBMS_OUTPUT.PUT_LINE ('----- ------- ------ ----------'
| | '---------');
LOOP
FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
EXIT WHEN emp_cursor % NOTFOUND;
CASE v_deptno
WHEN 10 THEN v_dname : = 'Accounting';
v_loc : = 'New York';
WHEN 20 THEN v_dname : = 'Research';
v_loc : = 'Dallas';
WHEN 30 THEN v_dname : = 'Sales';
v_loc : = 'Chicago';
WHEN 40 THEN v_dname : = 'Operations';
v_loc : = 'Boston';
ELSE v_dname : = 'unknown';
v_loc : = '';
END CASE;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | RPAD (v_ename 10) | |
''| | v_deptno | | ''| | RPAD (v_dname 14) | | ''| |
v_loc);
END LOOP;
```

```
CLOSE emp_cursor;
END;
```

다음은 이 프로그램의 출력입니다.

```
EMPNO ENAME DEPTNO DNAME LOC
----- ------- ------ ---------- ---------
7369 SMITH 20 Research Dallas
7499 ALLEN 30 Sales Chicago
7521 WARD 30 Sales Chicago
7566 JONES 20 Research Dallas
7654 MARTIN 30 Sales Chicago
7698 BLAKE 30 Sales Chicago
7782 CLARK 10 Accounting New York
7788 SCOTT 20 Research Dallas
7839 KING 10 Accounting New York
7844 TURNER 30 Sales Chicago
7876 ADAMS 20 Research Dallas
7900 JAMES 30 Sales Chicago
7902 FORD 20 Research Dallas
7934 MILLER 10 Accounting New York
```

#### 4.5.3.2 검색 CASE 문장

검색 CASE 문장은 하나 이상 Boolean 식을 실행하고 결과 집합을 얻기 문장을 결정하는 데 사용됩니다.

*CASE* WHEN boolean - *expression THEN*
    *statements*
*[WHEN* boolean - *expression THEN*
    *statements*
*[WHEN* boolean - *expression THEN*
    *statements* ...]
[ELSE
    *statements]*
END CASE;

*boolean - expression 은* CASE 문장에서 차례로 평가됩니다. 우선 처음 *boolean – expression* 이 *" true"일 경우,* 상응하는 THEN 구문 내의 문장이 실행되고, 계속해서 다음 END CASE 키워드에 들어갑니다. 만약 *boolean - expression* 중 "true"라고 평가되는 것이 없으면, ELSE 을 따르는 문장이 실행됩니다. 만약 *boolean - expression* 중 "true"라고 평가되는 것이 없고, ELSE 항목이 없는 경우는 예외가 발생합니다.

다음의 예제는 부서 번호를 기반으로 하는 변수에서 부서 이름과 위치를 할당하기 위하여 검색 CASE 문장을 사용하고 있습니다.

```
DECLARE
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
v_deptno emp.deptno % TYPE;
v_dname dept.dname % TYPE;
v_loc dept.loc % TYPE;
CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
OPEN emp_cursor;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME DEPTNO DNAME'
| | 'LOC');
DBMS_OUTPUT.PUT_LINE ('----- ------- ------ ----------'
| | '---------');
LOOP
FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
EXIT WHEN emp_cursor % NOTFOUND;
CASE
WHEN v_deptno = 10 THEN v_dname : = 'Accounting';
v_loc : = 'New York';
WHEN v_deptno = 20 THEN v_dname : = 'Research';
v_loc : = 'Dallas';
WHEN v_deptno = 30 THEN v_dname : = 'Sales';
v_loc : = 'Chicago';
WHEN v_deptno = 40 THEN v_dname : = 'Operations';
v_loc : = 'Boston';
ELSE v_dname : = 'unknown';
v_loc : = '';
END CASE;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | RPAD (v_ename 10) | |
''| | v_deptno | | ''| | RPAD (v_dname 14) | | ''| |
v_loc);
END LOOP;
CLOSE emp_cursor;
END;
```

다음은 이 프로그램의 출력입니다.

```
EMPNO ENAME DEPTNO DNAME LOC
----- ------- ------ ---------- ---------
7369 SMITH 20 Research Dallas
7499 ALLEN 30 Sales Chicago
7521 WARD 30 Sales Chicago
7566 JONES 20 Research Dallas
7654 MARTIN 30 Sales Chicago
7698 BLAKE 30 Sales Chicago
7782 CLARK 10 Accounting New York
7788 SCOTT 20 Research Dallas
7839 KING 10 Accounting New York
7844 TURNER 30 Sales Chicago
7876 ADAMS 20 Research Dallas
7900 JAMES 30 Sales Chicago
7902 FORD 20 Research Dallas
```

```
7934 MILLER 10 Accounting New York
```

### 4.5.4 루프

LOOP, EXIT, CONTINUE, WHILE 와 FOR 문장을 사용하여, SPL 프로그램에서 일련의 커맨드를 반복하도록 계획할 수 있습니다.

#### 4.5.4.1 LOOP

```
LOOP
    statements
END LOOP;
```

LOOP 는 EXIT 나 RETURN 문장에 의해 종료될 때까지 무한정 반복되는 조건 없는 루프를 정의합니다.

#### 4.5.4.2 EXIT

EXIT *[WHEN* expression ];

가장 안쪽의 루프가 끝나면, 다음의 END LOOP 문장이 실행됩니다.

WHEN 이 있는 경우, 조건이 true 일 때만 루프 종료가 발생합니다. 그렇지 않으면, EXIT 다음 줄로 제어권이 넘어갑니다.

EXIT 는 모든 타입의 루프로부터 빠르게 종료되도록 사용합니다. 이는 조건 없는 루프에서 제한 없이 사용할 수 있습니다.

아래는 10 번 반복되는 간단한 루프의 예제입니다. 그 후 EXIT 문장을 사용해서 종료합니다.

```
DECLARE
v_counter NUMBER (2);
BEGIN
v_counter : = 1;
LOOP
EXIT WHEN v_counter> 10;
DBMS_OUTPUT.PUT_LINE ( 'Iteration #'| | v_counter);
v_counter : = v_counter + 1;
END LOOP;
END;
```

다음은 이 프로그램의 출력 결과입니다.

```
Iteration # 1
Iteration # 2
Iteration # 3
```

```
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

#### 4.5.4.3 CONTINUE

CONTINUE 문은 안쪽의 문장을 생략하고, 바로 외부 루프를 실행하는 방법을 제공합니다.

CONTINUE 문장이 실행되면, 가장 안쪽 루프 다음 반복이 시작되고, 루프가 끝날 때까지 CONTINUE 문장 다음의 모든 문장을 건너뜁니다. 즉, 루프 제어 식 (만일 있으면)으로 돌아가고, 루프 본문은 다시 실행됩니다.

WHEN 이 사용되면, WHEN 구문에서 지정된 식이 true 일 때만, 다음 루프가 시작됩니다. 그와 반대의 경우는 제어권이 CONTINUE 문장 다음으로 이동합니다.

CONTINUE 는 루프 밖에서 사용되는 것은 아닙니다.

다음 예제는 앞의 예제를 변형한 것으로 CONTINUE 문장을 사용하여 홀수를 표시를 하지 않고 넘어갑니다.

```
DECLARE
v_counter NUMBER (2);
BEGIN
v_counter : = 0;
LOOP
v_counter : = v_counter + 1;
EXIT WHEN v_counter> 10;
CONTINUE WHEN MOD (v_counter, 2) = 1;
DBMS_OUTPUT.PUT_LINE ( 'Iteration #'| | v_counter);
END LOOP;
END;
```

다음은 이 프로그램의 출력 결과입니다.

```
Iteration # 2
Iteration # 4
Iteration # 6
Iteration # 8
Iteration # 10
```

#### 4.5.4.4 WHILE

*WHILE* expression LOOP
    *statements*
END LOOP;

WHILE 문장은 조건 표현식이 true 인 동안 문장 전체를 반복합니다. 조건은 루프 본문에 들어가기 전에만 검사합니다.

다음 예제는 루프의 종료를 EXIT 문장 대신, WHILE 문장에서 통제하고 있다는 점을 제외하고 동일합니다.

**참고** : 조건식은 언제 루프를 종료해야 하는지 결정을 변경해야 합니다. EXIT 문장은 그 조건식이 true 면 루프를 종료합니다. WHILE 문장은 그 조건식이 false 면 루프를 종료합니다. (또는 시작되지 않습니다.)

```
DECLARE
v_counter NUMBER (2);
BEGIN
v_counter : = 1;
WHILE v_counter <= 10 LOOP
DBMS_OUTPUT.PUT_LINE ( 'Iteration #'| | v_counter);
v_counter : = v_counter + 1;
END LOOP;
END;
```

이 예에서는 이전 예제와 같은 결과를 얻을 수 있습니다.

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

#### 4.5.4.5 FOR (정수 FOR 루프)

*FOR* name IN expression .. *expression LOOP*
    *statements*
END LOOP;

형식 FOR 는 정수 값의 범위를 반복 루프하도록 만듭니다.

*name* 변수는 INTEGER 형식으로 자동으로 정의되며 루프 내부에만 존재합니다. 범위를 가진 2 개의 식은 루프에 들어갈 때 한 번만 계산됩니다.

반복 간격은 +1, *name 은* 왼쪽.. *expression* 값으로 시작하고, *name* 값이 오른쪽.. *expression 의* 값을 초과할 경우 종료됩니다. 이처럼 2 개의 *expression 은* 다음의 규칙을 따릅니다.

*start - value .. end - value*

다음은 1 에서 10 까지 반복하는 FOR 루프를 사용하여 WHILE 루프를 보여주는 예제입니다.

```
BEGIN
FOR i IN 1 .. 10 LOOP
DBMS_OUTPUT.PUT_LINE ( 'Iteration #'| | i);
END LOOP;
END;
```

다음은 FOR 분 결과입니다.

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

시작하는 값이 마지막 값보다 크면 루프 시스템은 전혀 작동하지 않습니다. 또한 다음 예제와 같이 오류도 발생하지 않습니다.

```
BEGIN
FOR i IN 10 .. 1 LOOP
DBMS_OUTPUT.PUT_LINE ( 'Iteration #'| | i);
END LOOP;
END;
```

이 루프는 절대 실행되지 않으며, 출력되지 않습니다.

**참고** : SPL 은 CURSOR FOR 루프를 지원합니다. (4.7.7 절 참조).

### 4.5.5 오류 처리

기본적으로, SPL 프로그램에서 발생한 오류는 프로그램 실행으로 중단됩니다. BEGIN 블록과 EXCEPTION 문을 사용하면, 오류를 포착하여 복구할 수 있습니다. 그 구문은 일반적인 BEGIN 블록 구문을 확장합니다.

```
[DECLARE
    declarations    ]
```

```
BEGIN
    statements
EXCEPTION
    WHEN condition [OR condition ... THEN
        handler_statements
[WHEN condition [OR condition] ... THEN
        handler_statements] ...
END;
```

오류가 발생하지 않으면, 이러한 형태의 블록은 간단하게 전체 *statements* 를 실행하고, END 다음 문장으로 제어권이 이동됩니다. 그러나 *statements* 내부 오류가 발생하면, 향후 *statements* 의 작업이 중단되고 EXCEPTIO 리스트로 제어가 이동합니다. 리스트는 발생한 오류와 일치하는 첫 번째 *condition* 을 검색합니다. 일치하는 것이 있으면, 해당 *handler_statements* 를 실행하고, 그런 후 END 다음 문장으로 제어가 이동합니다. 일치하는 것이 없는 경우, EXCEPTION 구문이 없더라도 에러가 파급됩니다. EXCEPTION 을 포함하는 블록은 오류를 잡을 수 있지만, 실패하면 서브프로그램의 처리가 중단됩니다.

OTHERS 라는 특별한 조건 이름은 모든 에러 형식에 부합합니다. 조건 이름은 대소문자를 구분하지 않습니다.

선택된 *handler_statements* 내부에서 새로운 에러가 발생하면, EXCEPTION 구문은 에러를 포착할 수 없지만 오류는 외부에 파급됩니다. 둘러싼 EXCEPTION 구문은 그 오류를 포착할 수 있습니다.

다음의 테이블은 사용될 수 있는 조건 이름을 열거합니다.

**Table 4-2 예외 조건 이름**

| 조건 이름 | 설명 |
|---|---|
| CASE_NOT_FOUND | CASE 문장에 case 문장이 없는데 true 가 되고, ELSE 조건이 없습니다. |
| CURSOR_ALREADY_OPEN | 이미 열려있는 커서를 열려고 시도합니다. |
| INVALID_CURSOR | 열려있지 않은 커서에 접근하려고 합니다. |
| NO_DATA_FOUND | 선택 표준을 만족하는 행이 없습니다. |
| OTHERS | 예외 절의 이전 조건에서 발견되지 않은 예외를 잡습니다. |
| TOO_MANY_ROWS | 한 행만이 반환되는 선택 표준을 만족하는 행이 하나 이상입니다. |
| ZERO_DIVIDE | 0 으로 나누기를 시도합니다. |

### 4.5.6 어플리케이션 에러 발생

RAISE_APPLICATION_ERROR 프로시저는 SPL 프로그램에서 예외의 원인의 처리를 의도적으로 중단시키는 기능을 제공합니다. 예외는 4.5.5 절에서 설명한 것과 같은 방법으로 처리됩니다. 또한, RAISE_APPLICATION_ERROR 프로시저는 프로그램이 예외를 확인하기 위해 사용자 정의 코드와 오류 메시지를 생성할 수 있습니다.

RAISE_APPLICATION_ERROR *(error_number, message);*

*error_number 은* 정수 값이나 프로시저의 실행될 때, SQLCODE 라는 이름의 변수에 반환되는 식입니다. *Message 는* 문자열이나 SQLERRM 라는 변수에 반환되는 식입니다. SQLCODE 과 SQLERRM 변수에 대한 자세한 내용은 4.10 절을 참조하십시오.

다음 예제는 누락된 정보에 의존하는 다른 코드와 메시지를 표시하기 위해서 직원으로부터 RAISE_APPLICATION_ERROR 프로시저를 사용합니다.

```
CREATE OR REPLACE PROCEDURE verify_emp (
p_empno NUMBER
)
IS
v_ename emp.ename % TYPE;
v_job emp.job % TYPE;
v_mgr emp.mgr % TYPE;
v_hiredate emp.hiredate % TYPE;
BEGIN
SELECT ename, job, mgr, hiredate
INTO v_ename, v_job, v_mgr, v_hiredate FROM emp
WHERE empno = p_empno;
IF v_ename IS NULL THEN
RAISE_APPLICATION_ERROR (20010, 'No name for'| | p_empno);
END IF;
IF v_job IS NULL THEN
RAISE_APPLICATION_ERROR (20020, 'No job for'| | p_empno);
END IF;
IF v_mgr IS NULL THEN
RAISE_APPLICATION_ERROR (20030, 'No manager for'| | p_empno);
END IF;
IF v_hiredate IS NULL THEN
RAISE_APPLICATION_ERROR (20040, 'No hire date for'| | p_empno);
END IF;
DBMS_OUTPUT.PUT_LINE ( 'Employee'| | p_empno | |
'validated without errors');
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'SQLCODE :'| | SQLCODE);
DBMS_OUTPUT.PUT_LINE ( 'SQLERRM :'| | SQLERRM);
END;
```

다음은 직원 레코드로부터 관리자 번호가 없는 경우의 출력입니다.

```
EXEC verify_emp (7839);

SQLCODE : 20030
SQLERRM : EDB - 20030 : No manager for 7839
```

### 4.6 동적 SQL

*동적 SQL 은* 커맨드가 실행될 때까지 알 수 없는 SQL 커맨드를 실행할 수 있는 기능을 제공하는 기술입니다. 이전에는 SPL 프로그램에서 SQL 커맨드는 정적 SQL 로 표시했습니다. 정적 SQL 은 전체 커맨드(변수의 예외를 제외하고)로 알려져 있으며, 프로그램 자체가 실행을 시작하기 전, 프로그램 코드화 되어 있습니다. 동적 SQL 을 사용하면 프로그램이 실행 중일 때 실행되는 SQL 을 변경할 수 있습니다.

또한 동적 SQL 은 CREATE TABLE 과 같이 데이터 정의 커맨드를 SPL 프로그램에서 수행하는 유일한 방법입니다.

하지만 동적 SQL 의 작업 성능은 정적 SQL 에 비해 느리다는 것을 주의하십시오.

EXECUTE IMMEDIATE 커맨드는 SQL 커맨드를 동적으로 실행하는 데 사용됩니다.

*EXECUTE* IMMEDIATE sql_expression;
[INTO *(variable* [...] | *record)]*
*[USING* expression [...]

*sql_expression 는* 동적으로 실행하는 SQL 커맨드를 포함하는 문자열 식입니다. *Variable 은* 일반적 SELECT 커맨드로부터, *sql_expression 의* SQL 커맨드의 결과에서 생성된 결과 집합의 출력을 받습니다. 변수의 수, 순서와 데이터 형식은 결과 집합 필드의 수, 순서, 그리고 형식 호환이 일치해야 합니다. 그 대신에, 레코드는 레코드의 필드가 집합필드의 수, 순서와 형식 호환이 일치하는 만큼 지정할 수 있습니다. INTO 항목을 사용하면 반드시 1 개의 행이 결과 집합으로 반환되지만, 반대의 경우는 예외가 발생합니다. USING 항목을 사용하면, *expression 의* 값을 *플레이스홀더에* 전달할 수 있습니다. 플레이스홀더는 변수가 사용되는 *sql_expression 의* SQL 커맨드 안에 포함되어 나타납니다. 플레이스홀더는 콜론 (:) 접두사 - : *name 을* 식별자로 나타냅니다. 평가한 식의 수, 순서, 결과의 데이터 형식은 *sql_expression* 플레이스홀더의 개수, 순서, 및 형식과 일치해야만 합니다. 플레이스홀더는 SPL 프로그램의 어디서나 정의되지는 않습니다. *sql_expression* 안에서만 나타납니다.

다음 예제는 문자열에 의한 기본적인 동적 SQL 커맨드를 보여줍니다.

```
DECLARE
v_sql VARCHAR2 (50);
```

```
BEGIN
EXECUTE IMMEDIATE 'CREATE TABLE job (jobno NUMBER (3),'| |
'jname VARCHAR2 (9))';
v_sql : = 'INSERT INTO job VALUES (100, '' ANALYST '')';
EXECUTE IMMEDIATE v_sql;
v_sql : = 'INSERT INTO job VALUES (200, '' CLERK '')';
EXECUTE IMMEDIATE v_sql;
END;
```

다음은 SQL 문자열의 USING 절에서 플레이스홀더에 값을 전달합니다.

```
DECLARE
v_sql VARCHAR2 (50) : = 'INSERT INTO job VALUES'| |
'(: p_jobno : p_jname)';
v_jobno job.jobno % TYPE;
v_jname job.jname % TYPE;
BEGIN
v_jobno : = 300;
v_jname : = 'MANAGER';
EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
v_jobno : = 400;
v_jname : = 'SALESMAN';
EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
v_jobno : = 500;
v_jname : = 'PRESIDENT';
EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
END;
```

다음은 INTO 과 USING 구문을 모두 보여줍니다. SELECT 커맨드의 마지막 실행은 각 변수가 아닌 레코드에 그 결과가 반환됩니다.

```
DECLARE
v_sql VARCHAR2 (60);
v_jobno job.jobno % TYPE;
v_jname job.jname % TYPE;
r_job job % ROWTYPE;
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'JOBNO JNAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
v_sql : = 'SELECT jobno, jname FROM job WHERE jobno = : p_jobno';
EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 100;
DBMS_OUTPUT.PUT_LINE (v_jobno | | ''| | v_jname);
EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 200;
DBMS_OUTPUT.PUT_LINE (v_jobno | | ''| | v_jname);
EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 300;
DBMS_OUTPUT.PUT_LINE (v_jobno | | ''| | v_jname);
EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 400;
DBMS_OUTPUT.PUT_LINE (v_jobno | | ''| | v_jname);
EXECUTE IMMEDIATE v_sql INTO r_job USING 500;
DBMS_OUTPUT.PUT_LINE (r_job.jobno | | ''| | r_job.jname);
END;
```

다음은 위의 익명 블록으로부터의 출력입니다.

```
JOBNO JNAME
----- -------
100 ANALYST
200 CLERK
300 MANAGER
400 SALESMAN
500 PRESIDENT
```

### 4.7 정적 커서

쿼리 전체를 수행하는 것이 아니라 쿼리를 캡슐화하는 *커서*를 설정하고, 쿼리 결과를 한번에 한 행씩 읽습니다. 따라서 SPL 프로그램 로직 결과 집합로부터 1 행 단위로 데이터를 읽고 해당 행 데이터를 처리하고, 다음 커맨드를 처리하는 반복 가능합니다.

#### 4.7.1 커서 선언

커서를 이용하기 위해서는, SPL 프로그램 선언 영역에서 처음으로 커서를 선언해야 합니다. 커서 선언은 다음과 같습니다.

*CURSOR* name IS query;

*name 은* 커서를 참조하는데 사용하는 식별자이며, 그 결과는 후에 프로그램 설정됩니다.
*query 는* SQL SELECT 커맨드에서 커서의 검색을 지정합니다.

참고: 파라미터를 사용해서 이 구문을 확장합니다. 4.7.8 절에 더 자세히 설명되어 있습니다.

다음은 커서 선언의 같은 예입니다.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
CURSOR emp_cur_1 IS SELECT * FROM emp;
CURSOR emp_cur_2 IS SELECT empno, ename FROM emp;
CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
ORDER BY empno;
BEGIN
...
END;
```

#### 4.7.2 커서 열기

커서를 사용하여 행을 제거하기 전에 공개되어야만 합니다. 이 과정은 OPEN 문장을 사용합니다.

*OPEN* name;

*name 은* SPL 프로그램의 선언 영역에서 미리 선언된 커서의 식별자입니다. OPEN 문은 이미 열린 커서에 대해서는 실행되지 않습니다. 그것은 열린 그대로 존재합니다.

다음은 커서를 선언과 대응하는 OPEN 문장을 나타냅니다.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
ORDER BY empno;
BEGIN
OPEN emp_cur_3;
...
END;
```

### 4.7.3 커서로부터 행 가져오기

커서가 한번 오픈되면, FETCH 문장을 사용하여 커서의 결과로부터 행을 검색할 수 있습니다.

*FETCH* name I NTO *(record* | *variable* [, *variable_2* ...);

*name 은* 사전에 오픈된 커서 식별자입니다. *record 은* 사전에 정의한 레코드 (예 : *table* % ROWTYPE 을 사용하는)의 식별자입니다. *variable, variable_2* ...는 반입하는 행의 필드 데이터를 받는 SPL 변수입니다. *record* 또는 *variable, variable_2* ... 필드는 수와 순서가 일치해야 합니다. 필드는 커서 정의한 검색 SEL ECT 목록을 반환합니다. SELECT 목록 필드의 데이터 형식이 일치해야 하며, *record* 필드의 데이터 형식과 *variable, variable_2* ... 데이터 형식과 명시적으로 호환되어야 합니다.

**참고**: FETCH INTO 에서 BULK COLLECT 항목을 사용할 경우에는 컬렉션에 반환된 1 행 이상의 결과를 반환합니다. FETCH INTO 문장의 BULK COLLECT 항목 이용 방법에 대한 자세한 내용은 4.9.4.2 절을 참조하십시오.

다음은 FETCH 문장을 보여줍니다

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
v_empno NUMBER (4);
v_ename VARCHAR2 (10);
CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
ORDER BY empno;
BEGIN
OPEN emp_cur_3;
FETCH emp_cur_3 INTO v_empno, v_ename;
...
END;
```

대상 변수의 데이터 형식을 명시적으로 선언하는 대신, %TYPE 을 사용할 수 있습니다. 이 방법은 만약 데이터베이스 열이 변경 되더라도, SPL 프로그램의 대상 변수 선언을 변경할 필요가 없습니다. % TYPE 은 자동으로 지정된 열의 데이터 형식을 가져옵니다

```
CREATE OR REPLACE PROCEDURE cursor_example
```

```
IS
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
ORDER BY empno;
BEGIN
OPEN emp_cur_3;
FETCH emp_cur_3 INTO v_empno, v_ename;
...
END;
```

테이블의 모든 열이 테이블에 정의된 순서대로 검색하는 경우, FETCH 문장이 검색 결과에 대한 데이터를 저장하는 레코드를 정의하기 위하여 % ROWTYPE 을 사용할 수 있습니다. 점 표기법을 사용하면 레코드의 각 필드에 액세스할 수 있습니다.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
v_emp_rec emp % ROWTYPE;
CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
OPEN emp_cur_1;
FETCH emp_cur_1 INTO v_emp_rec;
DBMS_OUTPUT.PUT_LINE ( 'Employee Number :'| | v_emp_rec.empno);
DBMS_OUTPUT.PUT_LINE ( 'Employee Name :'| | v_emp_rec.ename);
...
END;
```

#### 4.7.4 커서 종료

커서 결과 집합에서 모든 필요한 열을 검색하면, 커서는 종료되어야만 합니다. 한번 닫히면 결과 집합에 더 이상 액세스할 수 없습니다. CLOSE 문장은 다음과 같습니다.

*CLOSE* name;

*name 은* 현재 열린 커서 식별자입니다. 한번 커서가 닫히면, 다시 종료될 수 없습니다. 그러나 커서가 한 번 닫히면, 닫힌 커서로 다시 OPEN 문장을 발행할 수 있습니다. 그리고 그 후에 새로운 쿼리 결과 집합을 검색하기 위해 FETCH 문장을 사용할 수 있는 쿼리 결과 세트가 작성됩니다.

다음은 CLOSE 문장의 사용법을 설명합니다.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
v_emp_rec emp % ROWTYPE;
CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
OPEN emp_cur_1;
FETCH emp_cur_1 INTO v_emp_rec;
DBMS_OUTPUT.PUT_LINE ( 'Employee Number :'| | v_emp_rec.empno);
DBMS_OUTPUT.PUT_LINE ( 'Employee Name :'| | v_emp_rec.ename);
```

```
CLOSE emp_cur_1;
END;
```

이 프로시저는 실행되면 다음과 같은 출력을 생성합니다. 직원 ID 7369 의 SMITH 은 결과 집합의 첫 번째 줄입니다.

```
EXEC cursor_example;

Employee Number : 7369
Employee Name : SMITH
```

### 4.7.5 커서에서 % ROWTYPE 이용

% ROWTYPE 속성을 이용하여 커서와 커서 변수에서 검색한 모든 열에 해당하는 필드가 포함된 레코드를 정의할 수 있습니다. 필드는 해당 열에 대한 데이터 형식을 갖습니다. % ROWTYPE 속성은 커서 이름 또는 커서 변수 이름이 접두사입니다.

*record* *cursor* % ROWTYPE;

*record* 는 레코드에 할당된 식별자입니다. *cursor* 는 현재 범위 안에 명시적으로 선언된 커서입니다.

다음은 어떤 직원이 어느 부서에서 일하고 있다는 정보를 얻기 위하여 % ROWTYPE 을 커서와 함께 사용하고 있습니다.

```
CREATE OR REPLACE PROCEDURE emp_info
IS
CURSOR empcur IS SELECT ename, deptno FROM emp;
myvar empcur % ROWTYPE;
BEGIN
OPEN empcur;
LOOP
FETCH empcur INTO myvar;
EXIT WHEN empcur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (myvar.ename | | 'works in department'
| | myvar.deptno);
END LOOP;
CLOSE empcur;
END;
```

다음은 이 프로시저부터의 출력입니다.

```
EXEC emp_info;

SMITH works in department 20
ALLEN works in department 30
WARD works in department 30
JONES works in department 20
MARTIN works in department 30
```

```
BLAKE works in department 30
CLARK works in department 10
SCOTT works in department 20
KING works in department 10
TURNER works in department 30
ADAMS works in department 20
JAMES works in department 30
FORD works in department 20
MILLER works in department 10
```

### 4.7.6 커서 속성

각 커서는 프로그램이 커서 상태를 시험할 수 있는 관련 속성 집합을 가지고 있습니다. 이러한 특성은 % ISOPEN, % FOUND, % NOTFOUND 및 % ROWCOUNT 입니다. 이러한 특성에 대해 다음 절에서 설명합니다.

#### 4.7.6.1 % ISOPEN 특성

% ISOPEN 속성은 커서가 오픈되어 있는지 여부를 테스트하는 데 사용됩니다.

*cursor_name* % ISOPEN

*cursor_name 은* 커서 이름으로 커서가 오픈 될 때, BOOLEAN 데이터 형식의 "true"가 리턴됩니다. 그렇지 않을 경우, "false"를 리턴합니다.

다음은 % ISOPEN 를 사용한 예입니다.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
...
CURSOR emp_cur_1 IS SELECT * FROM emp;
...
BEGIN
...
IF emp_cur_1 % ISOPEN THEN
NULL;
ELSE
OPEN emp_cur_1;
END IF;
FETCH emp_cur_1 INTO ...
...
END;
```

#### 4.7.6.2 % FOUND

% FOUND 속성은 지정된 커서의 결과 집합으로부터 반환되는 행이 검색되는지 여부를 테스트하는 데 사용됩니다.

*cursor_name* % FOUND

*cursor_name* 는 커서 이름으로 커서가 FETCH 후 결과 집합에서 행이 검색되면 BOOLEAN 데이터 형식의 "true"을 반환합니다.

결과 집합에서 마지막 행이 FETCH 되면, 다음 FETCH 에서 % FOUND 는 "false"을 반환합니다.
결과 집합 행이 없으면, 첫 번째 FETCH 도 "false"를 반환합니다.

커서가 오픈되기 전이나 닫힌 후에 %FOUND 참고는 INVALID_CURSOR 예외가 발생합니다.

% FOUND 는 커서가 오픈되나 처음 FETCH 전에는 null 을 반환합니다.

다음 예는 % FOUND 를 사용합니다.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
v_emp_rec emp % ROWTYPE;
CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
OPEN emp_cur_1;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
FETCH emp_cur_1 INTO v_emp_rec;
WHILE emp_cur_1 % FOUND LOOP
DBMS_OUTPUT.PUT_LINE (v_emp_rec.empno | | ''| | v_emp_rec.ename);
FETCH emp_cur_1 INTO v_emp_rec;
END LOOP;
CLOSE emp_cur_1;
END;
```

이전 프로시저를 실행하면 결과는 다음과 같습니다.

```
EXEC cursor_example;

EMPNO ENAME
----- ------
7369 SMITH
7499 ALLEN
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7782 CLARK
7788 SCOTT
7839 KING
7844 TURNER
7876 ADAMS
7900 JAMES
7902 FORD
7934 MILLER
```

#### 4.7.6.3 % NOTFOUND 속성

% NOTFOUND 속성은 % FOUND 와 논리적으로 반대입니다.

*cursor_name* % NOTFOUND

*cursor_name 는* 커서 이름으로 커서가 FETCH 된 후 결과 집합에서 행을 검색되면 BOOLEAN 데이터 형식의 "false"을 반환합니다.

결과 집합의 마지막 행이 FETCH 되면 다음 FETCH 에서 % NOTFOUND 은 "true"을 반환합니다. 결과 집합 행이 없으면 첫 번째 FETCH 도 "ture"을 반환합니다.

커서가 오픈되기 전이나 닫힌 후에 % NOTFOUND 참고하는 것은 INVALID_CURSOR 예외가 발생합니다.

% NOTFOUND 는 커서가 오픈되나 처음 FETCH 전에는 null 을 반환합니다.

다음 예는 % NOTFOUND 를 사용합니다.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
v_emp_rec emp % ROWTYPE;
CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
OPEN emp_cur_1;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH emp_cur_1 INTO v_emp_rec;
EXIT WHEN emp_cur_1 % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_emp_rec.empno | | ''| | v_emp_rec.ename);
END LOOP;
CLOSE emp_cur_1;
END;
```

위의 예제뿐 아니라 이 프로시저를 실행하면 동일한 결과를 생성합니다.

```
EXEC cursor_example;

EMPNO ENAME
----- ------
7369 SMITH
7499 ALLEN
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7782 CLARK
7788 SCOTT
```

```
7839 KING
7844 TURNER
7876 ADAMS
7900 JAMES
7902 FORD
7934 MILLER
```

#### 4.7.6.4 % ROWCOUNT

% ROWCOUNT 속성은 지정된 커서에서 FETCH 하는 행 수를 나타내는 정수를 반환합니다.

*curs or_name* % ROWCOUNT

*cursor_name 는* 커서의 이름이고, % ROWCOUNT 는 위와 같이 검색하는 행 수를 반환합니다. 마지막 행이 처리되면 커서가 종료되기 전까지는 행 수의 합계가 유지됩니다. 이 단계에서 보면 % ROWCOUNT 는 INVALID_CURSOR 예외를 던집니다.

% ROWCOUNT 참조하는 것은 커서가 오픈되기 전과 닫힌 후에 INVALID_CURSOR 예외가 발생합니다.

커서가 열리고 처음 FETCH 전에 보면 % ROWCOUNT 는 0 을 반환합니다. 또한 % ROWCOUNT 는 처음부터 결과 집합에 행이 없으면 첫 번째 FETCH 후 0 을 반환합니다.

다음 예는 % ROWCOUNT 를 사용합니다.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
v_emp_rec emp % ROWTYPE;
CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
OPEN emp_cur_1;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH emp_cur_1 INTO v_emp_rec;
EXIT WHEN emp_cur_1 % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_emp_rec.empno | | ''| | v_emp_rec.ename);
END LOOP;
DBMS_OUTPUT.PUT_LINE ('**********************');
DBMS_OUTPUT.PUT_LINE (emp_cur_1 % ROWCOUNT | | 'rows were retrieved');
CLOSE emp_cur_1;
END;
```

이 프로시저는 직원 명부를 끝까지 검색하고 합계를 표시합니다.

```
EXEC cursor_example;

EMPNO ENAME
----- -------
```

```
7369 SMITH
7499 ALLEN
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7782 CLARK
7788 SCOTT
7839 KING
7844 TURNER
7876 ADAMS
7900 JAMES
7902 FORD
7934 MILLER
**********
14 rows were retrieved
```

#### 4.7.6.5 커서 상태와 특성 요약

다음 테이블은 커서 상태와 커서 속성에 의해 반환되는 값을 요약한 것입니다.

**표 4-2 커서 속성**

| 커서 상태 | % ISOPEN | % FOUND | % NOTFOUND | % ROWCOUNT |
|---|---|---|---|---|
| 오픈 전 | False | INVALID_CURSOR 예외 | INVALID_CURSOR 예외 | INVALID_CURSOR 예외 |
| 오픈 후하고 처음 FETCH 이전 | True | Null | Null | 0 |
| 처음 성공했다 FETCH 다음 | True | True | False | 1 |
| n 번째 성공하는 FETCH 다음 (마지막 행) | True | True | False | n |
| n +1 번째 FETCH 다음 (마지막 행 다음) | True | False | True | n |
| 닫힌 후에 | False | INVALID_CURSOR 예외 | INVALID_CURSOR 예외 | INVALID_CURSOR 예외 |

4.7.7 커서 FOR 루프

커서 예제와 같이, 커서의 결과 집합을 처리하기 위해서는 커서를 여는 문장과 결과 집합의 각 행을 검색, 결과 집합 종료 테스트, 마지막으로 커서를 닫는 문장을 포함하는 프로그래밍 로직이 필요합니다. *커서 FOR 루프는* 이러한 개별 코드 문장을 배제한 루프 구문입니다.

커서 FOR 루프는 미리 선언한 커서를 오픈하고 커서 결과 집합의 모든 행을 반입하고, 커서를 닫습니다

커서 FOR 루프를 생성하는 형식은 다음과 같습니다.

*FOR* record IN cursor
LOOP
    *statements*
END LOOP;

*record 는* 정의에 명확하게 선언된 레코드에 할당된 식별자입니다. *cursor* % ROWTYPE.
*cursor 은* 미리 선언된 커서의 이름입니다. *Statements 는* 1 개 이상의 SPL 문장입니다. 적어도 1 문장 이상이 필요합니다.

다음은 4.7.6.3 절 예를 커서 FOR 루프를 사용하여 수정한 것입니다.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
FOR v_emp_rec IN emp_cur_1 LOOP
DBMS_OUTPUT.PUT_LINE (v_emp_rec.empno | | ''| | v_emp_rec.ename);
END LOOP;
END;
```

다음 출력에서 보듯이 같은 결과를 얻을 수 있습니다.

```
EXEC cursor_example;

EMPNO ENAME
----- -------
7369 SMITH
7499 ALLEN
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7782 CLARK
7788 SCOTT
7839 KING
7844 TURNER
7876 ADAMS
```

```
7900 JAMES
7902 FORD
7934 MILLER
```

### 4.7.8 매개변수  커서

사용자는 인자를 받아들이는 정적 커서를 선언하고, 커서가 오픈되면, 이러한 인자 값을 전달합니다.  다음 예제에서는 emp 테이블의 급여가 지정된 값보다 적은 직원의 이름과 급여를 확인하는 매개변수 커서를 선언합니다.

```
DECLARE
my_record emp % ROWTYPE;
CURSOR c1 (max_wage NUMBER) IS
SELECT * FROM emp WHERE sal <max_wage;
BEGIN
OPEN c1 (2000);
LOOP
FETCH c1 INTO my_record;
EXIT WHEN c1 % NOTFOUND;
DBMS_OUTPUT.PUT_LINE ( 'Name ='| | my_record.ename | | ', salary ='
| | my_record.sal);
END LOOP;
CLOSE c1;
END;
```

이 경우 max_wage 에게 2000 값을 전달하면 급여가 2000 이하의 모든 직원들의 이름과 급여가 나타납니다. 이 쿼리의 결과는 다음과 같습니다.

```
Name = SMITH, salary = 800.00
Name = ALLEN, salary = 1600.00
Name = WARD, salary = 1250.00
Name = MARTIN, salary = 1250.00
Name = TURNER, salary = 1500.00
Name = ADAMS, salary = 1100.00
Name = JAMES, salary = 950.00
Name = MILLER, salary = 1300.00
```

## 4.8 REF 커서와 커서 변수

이 절에서는 이전에 설명한 정적 커서보다 더 유연한 또 다른 커서를 제시합니다.

### 4.8.1 REF 커서 개요

*커서 변수는* 실제 검색 결과 집합에 대한 포인터를 가진 커서입니다.  결과 집합은 커서 변수를 사용하는 OPEN FOR 문장의 실행에 의해 결정됩니다.

커서 변수는 정적 커서와 같이 하나의 특정한 쿼리와 엮여 있지 않습니다. 동일한 커서 변수는 다른 쿼리를 포함하는 OPEN FOR 문장에 여러 차례 공개됩니다. 매번 쿼리로부터 새로운 결과 집합이 생성되고, 커서 변수에 액세스할 수 있습니다.

REF CURSOR 형식은 저장 프로시저와 함수로부터 인자로 전달됩니다. 반환 형식도 REF CURSOR 형식입니다. 이것은 프로그램 사이에 커서 변수를 전달하여, 독립된 프로그램으로 커서 작업을 모듈화하도록 합니다.

### 4.8.2 커서 변수 선언

SPL 는 내장된 데이터 형식 SYS_REFCURSOR 과 REF CURSOR 형식을 생성하고 해당 형식의 변수를 선언하는 2 가지 방법으로 커서 변수 선언을 지원합니다. SYS_REFCURSOR 는 REF CURSOR 형식이며, 그것에 관계하는 모든 결과 집합을 제공합니다. 이는 *weakly - typed* REF CURSOR 라고 합니다.

단순한 SYS_REFCURSOR 선언과 사용자 정의 REF CURSOR 변수와는 다릅니다. 여는 커서 및 커서의 검색과 닫는 커서와 같은 나머지 사용법은 두 커서 형식과 동일합니다. 이 장의 나머지 부분에서는 주로 SYS_REFCURSOR 커서 사용 예제를 다룹니다. 사용자 정의 REF CURSOR 를 사용하기 위해 예제를 변경하려면 선언 영역을 변경해야 합니다.

**참고:** *Strongly - typed* REF CURSOR 는 필드의 데이터 형식이 일치되도록 선언한 수와 필드의 순서를 필요로 하며, 선택적으로 결과 집합을 반환합니다.

#### 4.8.2.1 SYS_REFCURSOR 커서 변수 선언

다음은 SYS_REFCURSOR 변수를 선언하는 구문입니다.

*name SYS_REFCURSOR;*

*name 는* 커서 변수에 할당된 식별자입니다.

다음은 SYS_REFCURSOR 변수 선언의 예입니다.

```
DECLARE
emp_refcur SYS_REFCURSOR;
...
```

4.8.2.2 사용자 정의 REF CURSOR 형 변수 선언

2 가지 선언 단계는 사용자 정의 REF CURSOR 를 사용하기 위해서 수행해야 합니다.

● 참조되는 커서 TYPE 을 만든다

● TYPE 을 바탕으로 실제 커서 변수를 선언한다
사용자 정의 REF CURSOR 형식을 만드는 구문은 다음과 같습니다.
*TYPE* cursor_type_name IS REF CURSOR [RETURN return_type];

다음은 커서 변수를 선언하는 예입니다.

```
DECLARE
TYPE emp_cur_type IS REF CURSOR RETURN emp % ROWTYPE;
my_rec emp_cur_type;
...
```

### 4.8.3 커서 변수 열기

한 번 커서 변수가 선언되면, 관련되는 SELECT 커맨드를 가지고 오픈되어야 합니다. OPEN FOR 문은 결과 집합을 생성하는 데 사용되는 SELECT 커맨드를 지정합니다.

*OPEN* name FOR query;

*name 는* 사전에 선언된 커서 변수의 식별자입니다. *query 는* 문장이 실행될 때, 결과 집합을 결정하는 SELECT 커맨드입니다. OPEN FOR 문장이 실행되면 커서 변수의 값을 결과 집합을 확인합니다.

다음 예제는 선택된 부서로부터 직원의 수와 이름의 목록을 담은 결과 집합입니다. 변수 및 인자는 표현식을 기술할 수 있는 어디라도 SELECT 커맨드를 사용할 수 있는 것에 주의하십시요. 이 경우, 인자는 부서 번호를 같게 하는 테스트에 사용할 수 있습니다.

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
p_deptno emp.deptno % TYPE
)
IS
emp_refcur SYS_REFCURSOR;
BEGIN
OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
...
```

### 4.8.4 커서 변수에서 행 가져오기

커서 변수가 오픈되면 FETCH 문을 사용하여 결과 집합에서 행을 찾습니다. 결과 집합에서 행을 검색할 FETCH 문장을 사용하는 방법에 대한 자세한 내용은 4.7.3 를 참조하십시오.

다음 예제는 FETCH 문장이 이전 예제에 추가되고, 결과 집합은 2 개의 변수가 반환되어 표시됩니다. 커서 속성은 커서 변수와 함께 정적 커서 상태를 결정하는 데 이용되고 있었습니다. 커서 속성에 대한 자세한 내용은 4.7.6 를 참조하십시오.

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
```

```
p_deptno emp.deptno % TYPE
)
IS
emp_refcur SYS_REFCURSOR;
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
BEGIN
OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH emp_refcur INTO v_empno, v_ename;
EXIT WHEN emp_refcur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | v_ename);
END LOOP;
...
```

### 4.8.5 커서 변수 닫기

결과 집합을 릴리즈하기 위해 CLOSE 문장을 사용하는 방법은 4.7.4 절에 설명했습니다.

**참고**: 정적 커서와 달리 커서 변수는 그것을 다시 오픈하기 전에 닫을 필요가 없습니다. 열기 전에 결과 집합을 잃게 됩니다.

다음 예제는 CLOSE 문장을 추가해서 완성됐습니다.

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
p_deptno emp.deptno % TYPE
)
IS
emp_refcur SYS_REFCURSOR;
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
BEGIN
OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH emp_refcur INTO v_empno, v_ename;
EXIT WHEN emp_refcur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | v_ename);
END LOOP;
CLOSE emp_refcur;
END;
```

다음은 프로시저를 실행할 때 출력합니다.

```
EXEC emp_by_dept (20)

EMPNO ENAME
----- -------
7369 SMITH
```

```
7566 JONES
7788 SCOTT
7876 ADAMS
7902 FORD
```

#### 4.8.6 제한 사용

다음은 커서 변수를 사용할 때의 제한입니다.

- 비교 연산자는 커서 변수가 동일하거나 같지 않은지, null 이거나 null 이 아닌 지를 시험하기 위해서 사용할 수 없다.
- 커서 변수에는 null 이 할당되지 않는다.
- 커서 변수의 값은 데이터베이스 열에 포함되지 않는다.
- 정적 커서와 커서 변수는 교환이 불가능합니다. 예를 들어, 정적 커서는 OPEN FOR 문장에서는 사용할 수 없다.

또한, 다음은 프로시저와 함수 내 커서 변수 작업에서 프로시저와 함수의 인자로서 커서 변수가 된 인자 형태를 보여줍니다.

**표 4-3 된 커서 변수 인자 형태**

| 작업 | IN | IN OUT | OUT |
|---|---|---|---|
| OPEN | No | Yes | No |
| FETCH | Yes | Yes | No |
| CLOSE | Yes | Yes | No |

따라서, 예를 들어 프로시저의 형식 인자로 선언한 커서 변수에서 OPEN FOR, FETCH, CLOSE 3 개의 모든 작업을 실행할 경우, 해당 인자는 IN OUT 형식으로 선언되어야 합니다.

#### 4.8.7 예제

다음은 커서 변수를 사용하는 예입니다.

##### 4.8.7.1 함수에서 REF 커서를 반환

다음 예제에서 커서 변수는 주어진 일을 가진 직원을 조회하기 위해 오픈되어 있습니다. 결과 집합을 호출하는 함수에서 사용할 수 있도록, 커서 변수는 이 함수의 RETURN 문장으로 지정되어있다는 것을 주의하십시오.

```
CREATE OR REPLACE FUNCTION emp_by_job (p_job VARCHAR2)
RETURN SYS_REFCURSOR
IS
```

```
emp_refcur SYS_REFCURSOR;
BEGIN
OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
RETURN emp_refcur;
END;
```

이 함수는 함수의 호출을 익명 블록의 선언 영역에서 커서 변수에 할당한 다음 익명 블록에서 실행됩니다. 결과 집합은 커서 변수를 사용하여 얻어진 다음 닫혀집니다.

```
DECLARE
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
v_job emp.job % TYPE : = 'SALESMAN';
v_emp_refcur SYS_REFCURSOR;
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'EMPLOYEES WITH JOB'| | v_job);
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
v_emp_refcur : = emp_by_job (v_job);
LOOP
FETCH v_emp_refcur INTO v_empno, v_ename;
EXIT WHEN v_emp_refcur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | v_ename);
END LOOP;
CLOSE v_emp_refcur;
END;
```

다음은 이 익명 블록을 실행한 출력입니다.

```
EMPLOYEES WITH JOB SALESMAN
EMPNO ENAME
----- -------
7499 ALLEN
7521 WARD
7654 MARTIN
7844 TURNER
```

#### 4.8.7.2 커서 연산 모듈화

다음은 커서 변수에 대한 다양한 연산을 다른 프로그램으로 모듈화할 수 있다는 것을 보여줍니다.

다음과 같은 경우는 주어진 종류의 모든 행을 검색하는 SELECT 커맨드로 주어진 커서 변수를 오픈합니다

```
CREATE OR REPLACE PROCEDURE open_all_emp (
p_emp_refcur IN OUT SYS_REFCURSOR
)
IS
BEGIN
OPEN p_emp_refcur FOR SELECT empno, ename FROM emp;
END;
```

다음과 같은 경우는 지정된 커서 변수를 주어진 부서가 아닌 모든 행을 검색하는 SELECT 커맨드로 오픈합니다.

```
CREATE OR REPLACE PROCEDURE open_emp_by_dept (
p_emp_refcur IN OUT SYS_REFCURSOR,
p_deptno emp.deptno % TYPE
)
IS
BEGIN
OPEN p_emp_refcur FOR SELECT empno, ename FROM emp
WHERE deptno = p_deptno;
END;
```

세 번째 변화는, 다른 테이블에서 모든 행을 검색하는 SELECT 커맨드로 주어진 커서 변수를 선보였다.  함수의 반환 값은 열린 커서 변수라는 것을 주의하십시오.

```
CREATE OR REPLACE FUNCTION open_dept (
p_dept_refcur IN OUT SYS_REFCURSOR
) RETURN SYS_REFCURSOR
IS
v_dept_refcur SYS_REFCURSOR;
BEGIN
v_dept_refcur : = p_dept_refcur;
OPEN v_dept_refcur FOR SELECT deptno, dname FROM dept;
RETURN v_dept_refcur;
END;
```

이 프로시저는 직원 번호와 이름으로 구성된 커서 변수의 결과 집합을 획득하고 봅니다.

```
CREATE OR REPLACE PROCEDURE fetch_emp (
p_emp_refcur IN OUT SYS_REFCURSOR
)
IS
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH p_emp_refcur INTO v_empno, v_ename;
EXIT WHEN p_emp_refcur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | v_ename);
END LOOP;
END;
```

이 프로시저는 부서 번호와 이름이 저장된 커서 변수의 결과 집합을 반입하여 봅니다

```
CREATE OR REPLACE PROCEDURE fetch_dept (
p_dept_refcur IN SYS_REFCURSOR
)
IS
v_deptno dept.deptno % TYPE;
v_dname dept.dname % TYPE;
```

```
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'DEPT DNAME');
DBMS_OUTPUT.PUT_LINE ('---- ---------');
LOOP
FETCH p_dept_refcur INTO v_deptno, v_dname;
EXIT WHEN p_dept_refcur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_deptno | | ''| | v_dname);
END LOOP;
END;
```

이 프로시저는 주어진 커서 변수를 닫습니다.

```
CREATE OR REPLACE PROCEDURE close_refcur (
p_refcur IN OUT SYS_REFCURSOR
)
IS
BEGIN
CLOSE p_refcur;
END;
```

다음의 익명 블록은 이전에 설명된 모든 프로그램을 실행합니다.

```
DECLARE
gen_refcur SYS_REFCURSOR;
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'ALL EMPLOYEES');
open_all_emp (gen_refcur);
fetch_emp (gen_refcur);
DBMS_OUTPUT.PUT_LINE ('****************');

DBMS_OUTPUT.PUT_LINE ( 'EMPLOYEES IN DEPT # 10');
open_emp_by_dept (gen_refcur, 10);
fetch_emp (gen_refcur);
DBMS_OUTPUT.PUT_LINE ('****************');

DBMS_OUTPUT.PUT_LINE ( 'DEPARTMENTS');
fetch_dept (open_dept (gen_refcur));
DBMS_OUTPUT.PUT_LINE ('*****************');

close_refcur (gen_refcur);
END;
```

다음은 익명 블록의 출력입니다.

```
ALL EMPLOYEES
EMPNO ENAME
----- -------
7369 SMITH
7499 ALLEN
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
```

```
7782 CLARK
7788 SCOTT
7839 KING
7844 TURNER
7876 ADAMS
7900 JAMES
7902 FORD
7934 MILLER
****************
EMPLOYEES IN DEPT # 10
EMPNO ENAME
----- -------
7782 CLARK
7839 KING
7934 MILLER
****************
DEPARTMENTS
DEPT DNAME
---- ---------
10 ACCOUNTING
20 RESEARCH
30 SALES
40 OPERATIONS
*****************
```

### 4.8.8 REF 커서로 동적 쿼리

Postgres Plus Advanced Server 은 OPEN FOR USING 문장을 통해 동적 쿼리도 지원합니다.
문자열과 문자열 변수는 OPEN FOR USING 문장의 SELECT 커맨드로 제공됩니다.

*OPEN* name FOR dynamic_string
*[USING* bind_arg [, *bind_arg_2]* ...];

*name 은* 미리 선언한 커서 변수의 식별자입니다. *dynamic_string 는* SELECT 커맨드(세미콜론으로 끝나지 않음)를 포함하는 문자열이나 문자열 변수입니다. *bind_arg, bind_arg_2* ...는 커서 변수가 오픈되면, SELECT 커맨드 변수를 플레이스홀더에 전달하는 데 사용되는 바인딩 인자입니다.
플레이스홀더는 콜론을 접두사로 하는 식별자입니다.

다음은 문자열을 사용한 동적 쿼리.

```
CREATE OR REPLACE PROCEDURE dept_query
IS
emp_refcur SYS_REFCURSOR;
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
BEGIN
OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = 30'| |
'AND sal> = 1500';
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
```

```
FETCH emp_refcur INTO v_empno, v_ename;
EXIT WHEN emp_refcur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | v_ename);
END LOOP;
CLOSE emp_refcur;
END;
```

다음은 이 프로시저를 실행할 때의 출력입니다.

```
EXEC dept_query;

EMPNO ENAME
----- -------
7499 ALLEN
7698 BLAKE
7844 TURNER
```

다음 예제에서는 이전 쿼리가 쿼리 매개 변수에 전달하는 바인딩 인자를 사용하도록 수정됩니다.

```
CREATE OR REPLACE PROCEDURE dept_query (
p_deptno emp.deptno % TYPE,
p_sal emp.sal % TYPE
)
IS
emp_refcur SYS_REFCURSOR;
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
BEGIN
OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = : dept'
| | 'AND sal> = : sal'USING p_deptno, p_sal;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH emp_refcur INTO v_empno, v_ename;
EXIT WHEN emp_refcur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | v_ename);
END LOOP;
CLOSE emp_refcur;
END;
```

다음은 출력 결과입니다.

```
EXEC dept_query (30, 1500);

EMPNO ENAME
----- -------
7499 ALLEN
7698 BLAKE
7844 TURNER
```

마지막은 SELECT 를 전달하는데 가장 유연한 문자열 변수가 사용됩니다.

```
CREATE OR REPLACE PROCEDURE dept_query (
```

```
p_deptno emp.deptno % TYPE,
p_sal emp.sal % TYPE
)
IS
emp_refcur SYS_REFCURSOR;
v_empno emp.empno % TYPE;
v_ename emp.ename % TYPE;
p_query_string VARCHAR2 (100);
BEGIN
p_query_string : = 'SELECT empno, ename FROM emp WHERE'| |
'deptno = : dept AND sal> = : sal';
OPEN emp_refcur FOR p_query_string USING p_deptno, p_sal;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH emp_refcur INTO v_empno, v_ename;
EXIT WHEN emp_refcur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_empno | | ''| | v_ename);
END LOOP;
CLOSE emp_refcur;
END;
EXEC dept_query (20, 1500);

EMPNO ENAME
----- -------
7566 JONES
7788 SCOTT
7902 FORD
```

### 4.9 컬렉션

*컬렉션은* 동일한 데이터 형식의 일정한 데이터의 집합입니다. 일반적으로 데이터 항목은 스칼라 필드입니다. 그러나 구조와 레코드 형식의 필드로 구성된 데이터 타입이 집합의 요소를 위해 같은 한 레코드 형식일 수도 있습니다..

가장 일반적으로 알려져 있는 유형은 배열입니다. Postgres Plus Advanced Server 에서 지원하는 컬렉션 타입은 이전에 Oracle 에서 *index-by table 로* 불렸지만, 여기서는 *associative array(*연관 배열)이라고 합니다.

#### 4.9.1 연관 배열

연관 배열은 고유 키와 값으로 구성된 컬렉션 형식입니다. 일반적으로 연관 배열에서는 키는 숫자일 필요 없이, 문자열일 수 있습니다.

연관 배열은 다음과 같은 특징을 가지고 있습니다.

● 연관 배열 형식은 배열 변수가 배열 형식으로 선언된 후에 정의되어야만 한다. 데이터 조작은 배열 변수에서 이루어진다.

● 배열을 초기화할 필요가 없다. 바로 배열로 값이 할당된다.

● 키는 BINARY_INTEGER 이 지정되어 있을 경우, 음수, 양수와 0 이 될 수 있다.

● 배열 요소 수의 제한은 사전에 정의되지 않는다. 요소가 추가될 때마다 동적으로 확장된다.

● 배열은 드문드문 있을 수 있다. 키 값의 할당에 간격이 있을 수 있다.

● 할당되지 않은 요소 값에 대한 참조는 SQLCODE 1403, access of uninitialized index 예외가 발생한다.

TYPE IS TABLE INDEX BY 문장은 연관 배열 형식의 정의에 사용됩니다.

*TYPE* assoctype IS TABLE OF *(datatype* | *rectype)*
INDEX BY (BINARY_INTEGER | VARCHAR2 *(n));*

*assoctype 은* 배열 형식에 할당된 식별자입니다. *datatype 은* VARCHAR2 나 NUMBER 와 같은 스칼라 데이터 형식입니다. *rectype 은* 미리 정의된 레코드 형식입니다. n 은 문자 형식 키의 최대값입니다.

배열을 사용하려면, *변수는* 배열 형식으로 선언해야 합니다. 다음은 배열 변수를 선언하는 구문입니다.

*array assoctype*

*array* 는 관련된 배열에 대한 식별자입니다. *assoctype 은* 미리 선언된 배열 형식의 식별자입니다.

배열 요소는 다음과 같은 형식으로 참조됩니다.

*array (n). field]*

*array 는* 미리 선언된 배열의 식별자입니다. n 은 정수입니다. 만약 *array 의* 배열 형식을 레코드 형식으로부터 정의할 경우,[. *field*]*가* 레코드의 각각의 요소를 참조하는 데 사용됩니다. 여기서 *field 는* 배열 형식을 정의한 레코드 형식 내에 선언되어있습니다. 그 대신에, 모든 레코드는 [. *field*]를 생략함으로써 참조할 수 있습니다.

다음은 emp 테이블에서 처음 10 명의 직원 이름을 읽고, 배열에 저장한 다음 배열로부터 결과를 표시합니다.

```
DECLARE
TYPE emp_arr_typ IS TABLE OF VARCHAR2 (10) INDEX BY BINARY_INTEGER;
```

```
emp_arr emp_arr_typ;
CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
i INTEGER : = 0;
BEGIN
FOR r_emp IN emp_cur LOOP
i : = i + 1;
emp_arr (i) : = r_emp.ename;
END LOOP;
FOR j IN 1 .. 10 LOOP
DBMS_OUTPUT.PUT_LINE (emp_arr (j));
END LOOP;
END;
```

위의 예제는 다음과 같은 출력을 합니다.

```
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
```

위의 예제를 배열 정의 레코드를 사용하도록 수정했습니다.

```
DECLARE
TYPE emp_rec_typ IS RECORD (
empno NUMBER (4),
ename VARCHAR2 (10)
);
TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
emp_arr emp_arr_typ;
CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
i INTEGER : = 0;
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
FOR r_emp IN emp_cur LOOP
i : = i + 1;
emp_arr (i). empno : = r_emp.empno;
emp_arr (i). ename : = r_emp.ename;
END LOOP;
FOR j IN 1 .. 10 LOOP
DBMS_OUTPUT.PUT_LINE (emp_arr (j). empno | | ''| |
emp_arr (j). ename);
END LOOP;
END;
```

다음은 이 익명 블록의 출력입니다.

```
EMPNO ENAME
```

```
----- -------
7369 SMITH
7499 ALLEN
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7782 CLARK
7788 SCOTT
7839 KING
7844 TURNER
```

emp_rec_typ 레코드 형식 대신, emp % ROWTYPE 속성을 emp_arr_typ 을 정의하는 데 사용하는 예제입니다.

```
DECLARE
TYPE emp_arr_typ IS TABLE OF emp % ROWTYPE INDEX BY BINARY_INTEGER;
emp_arr emp_arr_typ;
CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
i INTEGER : = 0;
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
FOR r_emp IN emp_cur LOOP
i : = i + 1;
emp_arr (i). empno : = r_emp.empno;
emp_arr (i). ename : = r_emp.ename;
END LOOP;
FOR j IN 1 .. 10 LOOP
DBMS_OUTPUT.PUT_LINE (emp_arr (j). empno | | ''| |
emp_arr (j). ename);
END LOOP;
END;
```

결과는 앞의 예제와 동일합니다.

레코드의 각 필드를 별도로 할당하는 대신, r_emp 에서 emp_arr 로 레코드 단계 지정이 가능합니다.

```
DECLARE
TYPE emp_rec_typ IS RECORD (
empno NUMBER (4),
ename VARCHAR2 (10)
);
TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
emp_arr emp_arr_typ;
CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
i INTEGER : = 0;
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
FOR r_emp IN emp_cur LOOP
i : = i + 1;
emp_arr (i) : = r_emp;
```

```
END LOOP;
FOR j IN 1 .. 10 LOOP
DBMS_OUTPUT.PUT_LINE (emp_arr (j). empno | | ''| |
emp_arr (j). ename);
END LOOP;
END;
```

다음 예제와 같이 배열의 키는 문자열 데이터일 수 있습니다.

```
DECLARE
TYPE job_arr_typ IS TABLE OF NUMBER INDEX BY VARCHAR2 (9);
job_arr job_arr_typ;
BEGIN
job_arr ( 'ANALYST') : = 100;
job_arr ( 'CLERK') : = 200;
job_arr ( 'MANAGER') : = 300;
job_arr ( 'SALESMAN') : = 400;
job_arr ( 'PRESIDENT') : = 500;
DBMS_OUTPUT.PUT_LINE ( 'ANALYST :'| | job_arr ( 'ANALYST'));
DBMS_OUTPUT.PUT_LINE ( 'CLERK :'| | job_arr ( 'CLERK'));
DBMS_OUTPUT.PUT_LINE ( 'MANAGER :'| | job_arr ( 'MANAGER'));
DBMS_OUTPUT.PUT_LINE ( 'SALESMAN :'| | job_arr ( 'SALESMAN'));
DBMS_OUTPUT.PUT_LINE ( 'PRESIDENT :'| | job_arr ( 'PRESIDENT'));
END;
```

```
ANALYST : 100
CLERK : 200
MANAGER : 300
SALESMAN : 400
PRESIDENT : 500
```

### 4.9.2 Collection Methods

*Collection methods 은* 컬렉션의 데이터 처리에 편리하고, 컬렉션에 대한 유용한 정보를 제공하는 함수입니다. 다음 영역에서는 이들도 방법을 설명합니다.

#### 4.9.2.1 COUNT

COUNT 는 컬렉션의 요소 수를 반환하는 메소드입니다. COUNT 를 사용하기 위한 구문은 다음과 같습니다.

*collection.* COUNT

*collection is the identifier of a collection variable.*

*collection 은* 컬렉션 변수의 식별자입니다.

다음 예제에서 배열은 간격을 두고 배치됩니다. (예를 들면, 정의된 요소의 순서대로 "간격"이 있습니다.) COUNT 는 값이 주어진 요소에만 이용할 수 있습니다.

```
DECLARE
TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
sparse_arr sparse_arr_typ;
BEGIN
sparse_arr (-100) = -100;
sparse_arr (-10) = -10;
sparse_arr (0) : = 0;
sparse_arr (10) : = 10;
sparse_arr (100) = 100;
DBMS_OUTPUT.PUT_LINE ( 'COUNT :'| | sparse_arr.COUNT);
END;
```

다음 출력은 COUNT 에는 5 가지의 요소가 포함되어있다는 것을 보여줍니다.

```
COUNT : 5
```

#### 4.9.2.2 FIRST

FIRST 는 컬렉션의 첫 번째 요소의 인덱스를 반환하는 메소드입니다. FIRST 를 사용하는 구문은 다음과 같습니다.

*collection.* FIRST

*collection 은* 컬렉션 변수의 식별자입니다.

다음은 배열의 첫 번째 요소를 표시합니다.

```
DECLARE
TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
sparse_arr sparse_arr_typ;
BEGIN
sparse_arr (-100) = -100;
sparse_arr (-10) = -10;
sparse_arr (0) : = 0;
sparse_arr (10) : = 10;
sparse_arr (100) = 100;
DBMS_OUTPUT.PUT_LINE ( 'FIRST element :'| | sparse_arr (sparse_arr.FIRST));
END;

FIRST element : -100
```

#### 4.9.2.3 LAST

LAST 는 컬렉션의 마지막 요소의 인덱스를 반환하는 메소드입니다. LAST 를 사용하는 형식은 다음과 같습니다.

*collection.* LAST

*collection 은* 컬렉션 변수의 식별자입니다.

다음은 배열의 마지막 요소를 표시합니다.

```
DECLARE
TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
sparse_arr sparse_arr_typ;
BEGIN
sparse_arr (-100) = -100;
sparse_arr (-10) = -10;
sparse_arr (0) : = 0;
sparse_arr (10) : = 10;
sparse_arr (100) = 100;
DBMS_OUTPUT.PUT_LINE ( 'LAST element :'| | sparse_arr (sparse_arr.LAST));
END;

LAST element : 100
```

### 4.9.3 FORALL 문장 사용

컬렉션은 DELETE, INSERT 또는 UPDATE 커맨드를 각각 새 값으로 DML 커맨드를 실행하는 것 보다, 반복적으로 수행하는 모든 값을 한 번에 데이터베이스 서버에 전달하여 DML 커맨드를 실행하기 위해 좀더 효율적으로 이용할 수 있습니다. DML 커맨드는 FORALL 문장에서 지정한 방법으로 실행됩니다. 또한 하나 이상의 컬렉션은 커맨드를 실행할 때마다 다른 값을 할당하고, DML 커맨드에 전달합니다.

*FORALL* index IN lower_bound .. *upper_bound*
*(ins ert* | *update* | *delete);*

*Index 는* 컬렉션의 새 위치에서 *lower_bound 에서 upper_bound 까지* 정수 값으로 *insert, update, delete DML* 커맨드를 반복합니다.

**참고 :** FORALL 문장 반복하는 동안 예외가 발생할 경우, FORALL 문장 실행으로 시작한 모든 업데이트가 자동으로 롤백이 됩니다. 이 작업은 Oracle 과 호환되지 않습니다. Oracle 에서는 예외가 발생하기 전, 업데이트를 처리하거나 롤백하는 방법을 제어하기 위해 COMMIT 또는 ROLLBACK 커맨드를 명시적으로 사용하도록 허용합니다.

다음은 FORALL 문으로 emp 테이블에 3 명의 새로운 직원을 삽입하는 INSERT 커맨드 사용 예제입니다.

```
DECLARE
TYPE empno_tbl IS TABLE OF emp.empno % TYPE INDEX BY BINARY_INTEGER;
TYPE ename_tbl IS TABLE OF emp.ename % TYPE INDEX BY BINARY_INTEGER;
TYPE job_tbl IS TABLE OF emp.ename % TYPE INDEX BY BINARY_INTEGER;
TYPE sal_tbl IS TABLE OF emp.ename % TYPE INDEX BY BINARY_INTEGER;
TYPE deptno_tbl IS TABLE OF emp.deptno % TYPE INDEX BY BINARY_INTEGER;
t_empno EMPNO_TBL;
t_ename ENAME_TBL;
t_job JOB_TBL;
```

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

SELECT * FROM emp WHERE empno> 9000;

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

다음은 FORALL 문장에서 이 3 명의 급여를 갱신하고 있습니다.

```
DECLARE
TYPE empno_tbl IS TABLE OF emp.empno % TYPE INDEX BY BINARY_INTEGER;
TYPE sal_tbl IS TABLE OF emp.ename % TYPE INDEX BY BINARY_INTEGER;
t_empno EMPNO_TBL;
t_sal SAL_TBL;
BEGIN
t_empno (1) : = 9001;
t_sal (1) = 3350.00;
t_empno (2) : = 9002;
t_sal (2) = 2000.00;
t_empno (3) : = 9003;
t_sal (3) : = 4100.00;
FORALL i IN t_empno.FIRST .. t_empno.LAST
UPDATE emp SET sal = t_sal (i) WHERE empno = t_empno (i);
END;

SELECT * FROM emp WHERE empno> 9000;

empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+--------+---------+-----+----------+------ ---+------+--------
9001 | JONES | ANALYST | | | 3350.00 | | 40
```

```
9002 | LARSEN | CLERK | | | 2000.00 | | 40
9003 | WILSON | MANAGER | | | 4100.00 | | 40
(3 rows)
```

마지막 예제는 FORALL 문장에서 이들 3 명의 직원을 제거합니다.

```
DECLARE
TYPE empno_tbl IS TABLE OF emp.empno % TYPE INDEX BY BINARY_INTEGER;
t_empno EMPNO_TBL;
BEGIN
t_empno (1) : = 9001;
t_empno (2) : = 9002;
t_empno (3) : = 9003;
FORALL i IN t_empno.FIRST .. t_empno.LAST
DELETE FROM emp WHERE empno = t_empno (i);
END;

SELECT * FROM emp WHERE empno> 9000;

empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+----- -+--------
(0 rows)
```

4.9.4 BULK COLLECT 구문 사용

많은 행으로 구성된 결과 집합을 반환하는 SQL 커맨드는 효율적으로 실행할 수 없습니다. 왜냐하면 모든 결과 집합을 데이터베이스 서버와 클라이언트 사이에 전송하기 위해 특정 컨텍스트 교환이 발생하기 때문입니다. 이러한 비효율성은 클라이언트가 접근할 수 있는 메모리에 모든 결과 집합을 모으는 컬렉션을 통해 경감시킬 수 있습니다. BULK COLLECT 구문은 컬렉션에 결과 집합을 모으는 것을 지정하는 데 사용됩니다.

BULK COLLEC 절은 SELECT INTO 및 FETCH INTO 커맨드와 DELETE, INSERT, UPDATE 커맨드의 RETURNING INTO 구문과 함께 사용됩니다. 각각 다음 절에서 설명됩니다.

4.9.4.1 SELECT BULK COLLECT

BULK COLLECT 절은 SELECT INTO 문장과 함께 다음과 같이 사용됩니다. (SELECT INTO 문장에 대한 자세한 내용은 4.4.3 절을 참조하십시오.)

*SELECT* select_expressions BULK COLLECT INTO *collection*
[...] FROM ...;

만약 1 개의 컬렉션이 지정되어 있으면, *collection 은* 하나의 필드 컬렉션 또는 레코드 형식의 컬렉션입니다. 만약 1 개 이상의 컬렉션이 지정되어 있으면, 각 *collection 은* 하나의 필드로 구성되어 있어야만 합니다. *select_expressions 는* 목표 커넥션에 모든 필드의 형식 호환성, 수량, 순서가 일치해야 합니다.

다음은 원하는 컬렉션이 하나의 필드로 구성되는 연관 배열인 BULK COLLECT 구문의 사용 방법을 보여줍니다.

```
DECLARE
TYPE empno_tbl IS TABLE OF emp.empno % TYPE INDEX BY BINARY_INTEGER;
TYPE ename_tbl IS TABLE OF emp.ename % TYPE INDEX BY BINARY_INTEGER;
TYPE job_tbl IS TABLE OF emp.job % TYPE INDEX BY BINARY_INTEGER;
TYPE hiredate_tbl IS TABLE OF emp.hiredate % TYPE INDEX BY BINARY_INTEGER;
TYPE sal_tbl IS TABLE OF emp.sal % TYPE INDEX BY BINARY_INTEGER;
TYPE comm_tbl IS TABLE OF emp.comm % TYPE INDEX BY BINARY_INTEGER;
TYPE deptno_tbl IS TABLE OF emp.deptno % TYPE INDEX BY BINARY_INTEGER;
t_empno EMPNO_TBL;
t_ename ENAME_TBL;
t_job JOB_TBL;
t_hiredate HIREDATE_TBL;
t_sal SAL_TBL;
t_comm COMM_TBL;
t_deptno DEPTNO_TBL;
BEGIN
SELECT empno, ename, job, hiredate, sal, comm, deptno BULK COLLECT
INTO t_empno, t_ename, t_job, t_hiredate, t_sal, t_comm, t_deptno
FROM emp;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME JOB HIREDATE'| |
'SAL'| | 'COMM DEPTNO');
DBMS_OUTPUT.PUT_LINE ('----- ------- --------- --------- '| |
'-------- '| | '-------- ------');
FOR i IN 1 .. t_empno.COUNT LOOP
DBMS_OUTPUT.PUT_LINE (t_empno (i) | | ''| |
RPAD (t_ename (i), 8) | | ''| |
RPAD (t_job (i), 10) | | ''| |
TO_CHAR (t_hiredate (i), 'DD - MON - YY') | | ''| |
TO_CHAR (t_sal (i), '99999 .99 ') | |' '| |
TO_CHAR (NVL (t_comm (i), 0), '99999 .99 ') | |' '| |
t_deptno (i));
END LOOP;
END;

EMPNO ENAME JOB HIREDATE SAL COMM DEPTNO
----- ------- --------- --------- -------- -------- ---- -
7369 SMITH CLERK 17 - DEC - 80 800.00 .00 20
7499 ALLEN SALESMAN 20 - FEB - 81 1600.00 300.00 30
7521 WARD SALESMAN 22 - FEB - 81 1250.00 500.00 30
7566 JONES MANAGER 02 - APR - 81 2975.00 .00 20
7654 MARTIN SALESMAN 28 - SEP - 81 1250.00 1400.00 30
7698 BLAKE MANAGER 01 - MAY - 81 2850.00 .00 30
7782 CLARK MANAGER 09 - JUN - 81 2450.00 .00 10
7788 SCOTT ANALYST 19 - APR - 87 3000.00 .00 20
7839 KING PRESIDENT 17 - NOV - 81 5000.00 .00 10
7844 TURNER SALESMAN 08 - SEP - 81 1500.00 .00 30
7876 ADAMS CLERK 23 - MAY - 87 1100.00 .00 20
7900 JAMES CLERK 03 - DEC - 81 950.00 .00 30
7902 FORD ANALYST 03 - DEC - 81 3000.00 .00 20
7934 MILLER CLERK 23 - JAN - 82 1300.00 .00 10
```

다음은 동일한 결과를 제공합니다. 그러나 레코드 형식의 정의 % ROWTYPE 속성을 사용한 연관 배열을 사용합니다.

```
DECLARE
TYPE emp_tbl IS TABLE OF emp % ROWTYPE INDEX BY BINARY_INTEGER;
t_emp EMP_TBL;
BEGIN
SELECT * BULK COLLECT INTO t_emp FROM emp;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME JOB HIREDATE'| |
'SAL'| | 'COMM DEPTNO');
DBMS_OUTPUT.PUT_LINE ('----- ------- --------- --------- '| |
'-------- '| | '-------- ------');
FOR i IN 1 .. t_emp.COUNT LOOP
DBMS_OUTPUT.PUT_LINE (t_emp (i). empno | | ''| |
RPAD (t_emp (i). ename, 8) | | ''| |
RPAD (t_emp (i). job, 10) | | ''| |
TO_CHAR (t_emp (i). hiredate, 'DD - MON - YY') | | ''| |
TO_CHAR (t_emp (i). sal, '99999 .99 ') | |' '| |
TO_CHAR (NVL (t_emp (i). comm, 0), '99999 .99 ') | |' '| |
t_emp (i). deptno);
END LOOP;
END;

EMPNO ENAME JOB HIREDATE SAL COMM DEPTNO
----- ------- --------- --------- -------- -------- ---- -
7369 SMITH CLERK 17 - DEC - 80 800.00 .00 20
7499 ALLEN SALESMAN 20 - FEB - 81 1600.00 300.00 30
7521 WARD SALESMAN 22 - FEB - 81 1250.00 500.00 30
7566 JONES MANAGER 02 - APR - 81 2975.00 .00 20
7654 MARTIN SALESMAN 28 - SEP - 81 1250.00 1400.00 30
7698 BLAKE MANAGER 01 - MAY - 81 2850.00 .00 30
7782 CLARK MANAGER 09 - JUN - 81 2450.00 .00 10
7788 SCOTT ANALYST 19 - APR - 87 3000.00 .00 20
7839 KING PRESIDENT 17 - NOV - 81 5000.00 .00 10
7844 TURNER SALESMAN 08 - SEP - 81 1500.00 .00 30
7876 ADAMS CLERK 23 - MAY - 87 1100.00 .00 20
7900 JAMES CLERK 03 - DEC - 81 950.00 .00 30
7902 FORD ANALYST 03 - DEC - 81 3000.00 .00 20
7934 MILLER CLERK 23 - JAN - 82 1300.00 .00 10
```

#### 4.9.4.2 FETCH BULK COLLECT

BULK COLLECT 절은 FETCH 문장과 함께 사용할 수 있습니다. (FETCH 문장에 대한 자세한 내용은 4.7.3 절을 참조하십시오.) 한번에 하나의 행을 결과 집합에서 반환하는 대신, FETCH BULK COLLECT 은 LIMIT 절 제약이 아니라면, 결과 집합에서 즉시 모든 행을 지정된 컬렉션으로 반환합니다.

*FETCH* name BULK COLLECT INTO *collection* [...] *[LIMIT* n];

만일 하나의 컬렉션이 지정된 경우, *collection 은* 하나의 필드거나 레코드 형식의 컬렉션입니다. 만약 1 개 이상의 컬렉션이 지정되면, 각 *collection 은* 하나의 필드로 구성되어야 합니다. *name*

에 지정된 커서 SELECT 목록의 표현식은 목표의 컬렉션의 모든 필드의 형식 호환성과 수량, 순서가 일치해야 합니다. 만약 *LIMIT* n 가 지정되면, 각 FETCH 를 실행하여 컬렉션에 반환되는 행의 수는 *n 을* 초과할 수 없습니다.

다음은 배열로 행을 회수하는데 FETCH BULK COLLECT 문장을 사용하고 있습니다.

```
DECLARE
TYPE emp_tbl IS TABLE OF emp % ROWTYPE INDEX BY BINARY_INTEGER;
t_emp EMP_TBL;
CURSOR emp_cur IS SELECT * FROM emp;
BEGIN
OPEN emp_cur;
FETCH emp_cur BULK COLLECT INTO t_emp;
CLOSE emp_cur;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME JOB HIREDATE'| |
'SAL'| | 'COMM DEPTNO');
DBMS_OUTPUT.PUT_LINE ('----- ------- --------- --------- '| |
'-------- '| | '-------- ------');
FOR i IN 1 .. t_emp.COUNT LOOP
DBMS_OUTPUT.PUT_LINE (t_emp (i). empno | | ''| |
RPAD (t_emp (i). ename, 8) | | ''| |
RPAD (t_emp (i). job, 10) | | ''| |
TO_CHAR (t_emp (i). hiredate, 'DD - MON - YY') | | ''| |
TO_CHAR (t_emp (i). sal, '99999 .99 ') | |' '| |
TO_CHAR (NVL (t_emp (i). comm, 0), '99999 .99 ') | |' '| |
t_emp (i). deptno);
END LOOP;
END;

EMPNO ENAME JOB HIREDATE SAL COMM DEPTNO
----- ------- --------- --------- -------- -------- ---- -
7369 SMITH CLERK 17 - DEC - 80 800.00 .00 20
7499 ALLEN SALESMAN 20 - FEB - 81 1600.00 300.00 30
7521 WARD SALESMAN 22 - FEB - 81 1250.00 500.00 30
7566 JONES MANAGER 02 - APR - 81 2975.00 .00 20
7654 MARTIN SALESMAN 28 - SEP - 81 1250.00 1400.00 30
7698 BLAKE MANAGER 01 - MAY - 81 2850.00 .00 30
7782 CLARK MANAGER 09 - JUN - 81 2450.00 .00 10
7788 SCOTT ANALYST 19 - APR - 87 3000.00 .00 20
7839 KING PRESIDENT 17 - NOV - 81 5000.00 .00 10
7844 TURNER SALESMAN 08 - SEP - 81 1500.00 .00 30
7876 ADAMS CLERK 23 - MAY - 87 1100.00 .00 20
7900 JAMES CLERK 03 - DEC - 81 950.00 .00 30
7902 FORD ANALYST 03 - DEC - 81 3000.00 .00 20
7934 MILLER CLERK 23 - JAN - 82 1300.00 .00 10
```

#### 4.9.4.3 RETURNING BULK COLLEC T

BULK COLLECT 절을 DELETE, INSERT, UPDATE 커맨드의 RETURNING INTO 구문에 추가할 수 있습니다. (RETURNING INTO 항목에 대한 자세한 내용은 4.4.7 절을 참조하십시오.)

*(insert | update | delete)*

RETURNING (* | *expr_1* [, *expr_2]* ...)
*BULK* COLLECT INTO collection [...];

*Insert, update, delete* 는 각각 INSERT, UPDATE, DELETE 커맨드이며, 4.4.4, 4.4.5, 4.4.6 절에서 설명하고 있습니다. 만약 1 개의 컬렉션이 지정된 경우, *collection* 은 단일 필드로 구성된 레코드 형식의 컬렉션입니다. 만약 하나 이상의 컬렉션이 지정되면, 각 *collection* 은 하나의 필드로 구성되어 있어야 합니다. RETURNING 키워드를 따르는 표현식은 대상 컬렉션의 모든 필드의 형식 호환성과 수량, 순서가 동일해야 합니다. 만약 *가 지정되면, 영향 받는 테이블의 모든 열이 반환됩니다. (*의 이용은 Postgres Plus Advanced Server 확장으로서 Oracle 과 호환되지 않습니다.)

emp 테이블을 복사하여 만든 clerkemp 테이블은 다음 절의 예제에서 사용됩니다.

```
CREATE TABLE clerkemp AS SELECT * FROM emp WHERE job = 'CLERK';

SELECT * FROM clerkemp;

empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+--------+-------+------+------------------ --+---------+------+--------
7369 | SMITH | CLERK | 7902 | 17 - DEC - 80 00:00:00 | 800.00 | | 20
7876 | ADAMS | CLERK | 7788 | 23 - MAY - 87 00:00:00 | 1100.00 | | 20
7900 | JAMES | CLERK | 7698 | 03 - DEC - 81 00:00:00 | 950.00 | | 30
7934 | MILLER | CLERK | 7782 | 23 - JAN - 82 00:00:00 | 1300.00 | | 10
(4 rows)
```

다음은 전체의 급여를 1.5 로 나누고, 직원 번호와 이름, 새로운 급여를 연관 배열에 저장하며 마지막으로 배열의 내용을 표시합니다.

```
DECLARE
TYPE empno_tbl IS TABLE OF emp.empno % TYPE INDEX BY BINARY_INTEGER;
TYPE ename_tbl IS TABLE OF emp.ename % TYPE INDEX BY BINARY_INTEGER;
TYPE sal_tbl IS TABLE OF emp.sal % TYPE INDEX BY BINARY_INTEGER;
t_empno EMPNO_TBL;
t_ename ENAME_TBL;
t_sal SAL_TBL;
BEGIN
UPDATE clerkemp SET sal = sal * 1.5 RETURNING empno, ename, sal
BULK COLLECT INTO t_empno, t_ename, t_sal;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME SAL');
DBMS_OUTPUT.PUT_LINE ('----- ------- -------- ');
FOR i IN 1 .. t_empno.COUNT LOOP
DBMS_OUTPUT.PUT_LINE (t_empno (i) | | ''| | RPAD (t_ename (i), 8) | |
''| | TO_CHAR (t_sal (i), '99999 .99 '));
END LOOP;
END;

EMPNO ENAME SAL
----- ------- --------
7369 SMITH 1,200.00
```

```
7876 ADAMS 1,650.00
7900 JAMES 1,425.00
7934 MILLER 1,950.00
```

다음은 이전 예제와 같은 기능을 제공하지만, 직원번호와 이름, 새로운 급여를 저장하는 레코드 형식을 가진 정의된 컬렉션을 사용합니다.

```
DECLARE
TYPE emp_rec IS RECORD (
empno emp.empno % TYPE,
ename emp.ename % TYPE,
sal emp.sal % TYPE
);
TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
t_emp EMP_TBL;
BEGIN
UPDATE clerkemp SET sal = sal * 1.5 RETURNING empno, ename, sal
BULK COLLECT INTO t_emp;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME SAL');
DBMS_OUTPUT.PUT_LINE ('----- ------- -------- ');
FOR i IN 1 .. t_emp.COUNT LOOP
DBMS_OUTPUT.PUT_LINE (t_emp (i). empno | | ''| |
RPAD (t_emp (i). ename, 8) | | ''| |
TO_CHAR (t_emp (i). sal, '99, 999.99 '));
END LOOP;
END;

EMPNO ENAME SAL
----- ------- --------
7369 SMITH 1,200.00
7876 ADAMS 1,650.00
7900 JAMES 1,425.00
7934 MILLER 1,950.00
```

다음은 clerkemp 테이블에서 모든 행을 삭제하고, 배열로 삭제된 행 정보를 반환하고 표시합니다.

```
DECLARE
TYPE emp_rec IS RECORD (
empno emp.empno % TYPE,
ename emp.ename % TYPE,
job emp.job % TYPE,
hiredate emp.hiredate % TYPE,
sal emp.sal % TYPE,
comm emp.comm % TYPE,
deptno emp.deptno % TYPE
);
TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
r_emp EMP_TBL;
BEGIN
DELETE FROM clerkemp RETURNING empno, ename, job, hiredate, sal,
comm, deptno BULK COLLECT INTO r_emp;
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME JOB HIREDATE'| |
'SAL'| | 'COMM DEPTNO');
DBMS_OUTPUT.PUT_LINE ('----- ------- --------- --------- '| |
```

```
'-------- '| | '-------- ------');
FOR i IN 1 .. r_emp.COUNT LOOP
DBMS_OUTPUT.PUT_LINE (r_emp (i). empno | | ''| |
RPAD (r_emp (i). ename, 8) | | ''| |
RPAD (r_emp (i). job, 10) | | ''| |
TO_CHAR (r_emp (i). hiredate, 'DD - MON - YY') | | ''| |
TO_CHAR (r_emp (i). sal, '99999 .99 ') | |' '| |
TO_CHAR (NVL (r_emp (i). comm, 0), '99999 .99 ') | |' '| |
r_emp (i). deptno);
END LOOP;
END;

EMPNO ENAME JOB HIREDATE SAL COMM DEPTNO
----- ------- --------- --------- -------- -------- ---- -
7369 SMITH CLERK 17 - DEC - 80 1200.00 .00 20
7876 ADAMS CLERK 23 - MAY - 87 1650.00 .00 20
7900 JAMES CLERK 03 - DEC - 81 1425.00 .00 30
7934 MILLER CLERK 23 - JAN - 82 1950.00 .00 10
```

### 4.10 에러 메시지

메시지를 출력하기 위해서는 DBMS_OUTPUT.PUT_LINE 문장을 사용합니다.

DBMS_OUTPUT.PUT_LINE *(message);*

*Message 는* 문자열로 평가되는 식입니다.

이 예제는 사용자의 출력 화면에 메시지가 나타납니다.

```
DBMS_OUTPUT.PUT_LINE ( 'My name is John');
```

특별한 변수 SQLCODE 과 SQLERRM 은 각각 숫자 코드와 문자 메시지를 가지고 있습니다.
그것은 마지막으로 실행된 SQL 커맨드를 실행합니다. 만약 프로그램에서 0 으로 나누기와 같은 다른 에러가 발생하면, 이 변수는 에러에 대한 정보를 갖게 됩니다.

## 제 5장 트리거

이 장에서는 Postgres Plus Advanced Server에 있는 트리거를 설명합니다. 프로시저와 함수와 마찬가지로 트리거는 SPL 언어입니다.

### 5.1 개요

트리거는 이름이 부여되고, 결합된 테이블과 관련된 데이터베이스에 저장된 SPL 코드 블록입니다. 이벤트가 테이블에 발생하면 SPL 코드 블록이 실행됩니다. 트리거는 코드 블록이 실행되면 시작됩니다.

트리거를 시작하는 이벤트는 직접 또는 간접적인 삽입, 업데이트 및 복원 어떤 조합에도 테이블에 대해 실행되는 경우 발생합니다. 테이블이 SQL INSERT, UPDATE, DELETE 명령의 대상이면 해당 인서트, 업데이트 및 복원 이벤트를 트리거 이벤트로 정의하면, 트리거를 직접 시작할 수 있습니다. 트리거를 시작하는 이벤트는 CREATE TRIGGER 명령으로 정의합니다.

다른 테이블에서 이벤트가 시작된 것은 트리거 이벤트가 발생하면 트리거는 간접적으로 시작할 수 있습니다. 예를 들면, ON DELETE CASCADE 조항에 정의된 외부 키가 있는 테이블에 트리거가 정의되면 부모 테이블의 행이 삭제되면 부모 테이블의 모든 자식 테이블만큼 삭제됩니다. 삭제가 자식 테이블의 트리거 이벤트라면 자식 테이블의 삭제는 트리거를 시작합니다.

### 5.2 트리거의 유형

Postgres Plus Advanced Server 는 *row-level* 와 *statement-level* 트리거를 지원하고 있습니다. row-level 트리거는 트리거 이벤트에 따라 영향을 받는 각 라인마다 한 번 발행합니다. 예를 들면, 삭제가 테이블의 트리거 이벤트로 정의되고 테이블에서 5 행 삭제 단일 DELETE 명령이 실행되면, 트리거는 각 라인마다 한 번, 총 5 회 시작합니다.

대조적으로, statement-level 는 트리거 이벤트에 영향을 받은 행의 개수에 상관없이 트리거 명령문 마다 한 번 시작합니다. 하나의 DELETE 명령 5 행 삭제 위의 예에서는 문을 트리거는 한 번만 시작합니다.

그것이 명령문 수준 트리거의 경우 트리거 코드 블록을 트리거 성명에서 "전" 또는 "다음"을, row-level 트리거의 경우 트리거 명령문에 영향을 받는 각 행의 "이전" 또는 "후" 실행 여부 에 대한 일련의 동작을 정의할 수 있습니다.

Row-level 전 트리거는 각 영향을 받는 줄에 트리거 동작이 이루어지기 전에 트리거 코드 블록이 실행됩니다. 문 수준 이전 트리거는 트리거 문이 열리는 동작 전에 트리거 코드 블록이 실행됩니다.

이후 row-level 유발, 각 영향을 받는 줄에 트리거 동작이 일어난다 후 트리거 코드 블록이 실행됩니다. 문 수준 다음 트리거는 트리거 문이 열리는 동작 후에 트리거 코드 블록이 실행됩니다.

### 5.3 트리거 발생

CREATE TRIGGER 명령은 데이터베이스에 저장된 트리거의 정의 및 명령입니다.

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

*name* 은 트리거 이름입니다. [OR REPLACE ] 이 지정한 동일한 이름의 트리거가 이미 스키마에 존재하는 경우에, 새로운 트리거는 기존의 것으로 대체합니다. [OR REPLACE ] 이 지정되지 않는 경우, 새로운 트리거는 같은 스키마에 같은 이름의 기존하는 것으로 대체 것은 아닙니다. BEFORE 가 지정되어 있으면 이전 트리거로 트리거가 정의됩니다. AFTER 가 정의 되었다면, 다음 트리거로 트리거가 정의됩니다. INSERT, UPDATE 및 DELETE 중 하나는 각각 삽입, 업데이트, 삭제 트리거 이벤트에 대한 정의로 지정되어 있어야 합니다. 하나 또는 잔여 트리거 이벤트 키워드 양측은 OR 키워드로 구분 정의해야 합니다. 이 경우 이들도 트리거 이벤트로 정의됩니다. *table* 은 트리거 이벤트가 트리거를 시작하는 테이블의 이름입니다. [ FOR EACH ROW ] 를 사용하면 트리거는 row-level 트리거로 정의됩니다. [FOR EACH ROW ] 이 생략되면 트리거는 명령문을 트리거로 정의됩니다. *declarations* 커서 또는 형식 선언 변수입니다. *statements* 는 SPL 프로그램 명령문 입니다. BEGIN - END 블록은 EXCEPTION 섹션을 포함합니다.

트리거 발생에 대한 자세한 내용은 CREATE TRIGGER 명령을 참조하십시오.

### 5.4 트리거 변수

트리거 코드 블록에서 일부 특수 변수를 사용할 수 있습니다.

#### NEW

NEW 는 row-level 트리거 삽입 및 업데이트 작업에서 새 테이블 행을 가리키는 임시 기록입니다. 이 변수는 문 수준 트리거와 row-level 트리거로 제거 작업에는 적용되지 않습니다.

사용방법: NEW. *column, column* 는 트리거가 정의된 테이블의 열 이름입니다.

: NEW. *column* 초기 내용을 삽입하는 새로운 행 또는 row-level 전 트리거에서 사용된 오래된 것으로 대체할 새 줄에서 열 이름 값입니다. 이후 row-level 트리거로 사용하면 영향을 받는 행에서 동작하도록 이미 일어나고 있기 때문에 이 값은 이미 테이블에 저장됩니다.

트리거 코드 블록은: NEW. *column* 는 다른 변수와 같은 방식으로 사용됩니다. 값 지정은: NEW. *column 은* row-level 전 트리거 코드 블록에서 줄 새로운 삽입 및 갱신되는 값이 지정됩니다.

**OLD**

OLD 은 row-level 트리거 업데이트 및 삭제 작업에 오래된 테이블 행을 가리키는 임시 기록입니다. 이 변수는 문 수준 트리거와 row-level 트리거로 삽입 작업에는 적용되지 않습니다.

사용방법: OLD. *column, column* 는 트리거가 정의된 테이블의 열 이름입니다.

: OLD. *column* 초기 내용은 삭제된 행 또는 row-level 전 트리거에서 사용된 새로운 것으로 대체되는 이전 행에 열 이름 값입니다. 이후 row-level 트리거로 사용하면 영향을 받는 행에서 동작하도록 이미 일어나고 있기 때문에 이 값은 테이블에 저장되는 것은 아닙니다.

트리거 코드 블록은: OLD. *column 은* 다른 변수들과 같은 방식으로 사용됩니다. : OLD. *column* 값을 지정은 트리거 동작에 영향을 받지 않습니다.

**IN SERTING**

INSERTING 은 구문으로 삽입 작업 트리거를 시작하면 참 (true)을 반환합니다. 그렇지 않다면 거짓 (false)을 반환합니다.

**UPDATING**

UPDATING 은 구문으로 업데이트 작업을 트리거를 시작하면 (true)을 반환합니다. 그렇지 않다면 거짓 (false)을 반환합니다.

**DELETING**

DELETING 은 구문으로 삭제 작업을 트리거를 시작하면 참 (true)을 반환합니다. 그렇지 않다면 거짓 (false)을 반환합니다.

**5.5 트랜잭션과 예외 처리**

트리거는 트리거 링 문이 실행중인 동일한 거래의 일환으로 항상 실행됩니다. 트리거 코드 블록에서 예외가 발생 없으면 커밋 된 트리거 링 문이 거래에 포함되어있는 경우에만 트리거에

있는 모든 DML 명령의 결과는 커밋 됩니다. 따라서, 트랜잭션이 롤백 되면 트리거에 있는 모든 DML 명령의 결과는 롤백 됩니다.

예외로, 트리거 코드 블록에 낼 경우 트리거에 있는 모든 DML 명령의 결과가 아직 롤백 되고 있는데도, 그것은 예외 처리 구역으로 파악 처리됩니다. 그러나 트리거 문 자체는 거래 캡슐화 작용 힘 롤백 하지 않은 경우 롤백 되지 않습니다.

처리되지 않은 예외가 트리거 코드 블록에 낼 경우 트리거를 캡슐 화하는 거래가 중단 롤백 합니다. 그래서 트리거에 있는 모든 DML 명령 및 트리거 링 문의 결과는 자체적으로 모든 롤백 합니다.

### 5.6 트리거 예제

다음 섹션에서는 각 유형의 트리거의 예를 설명합니다.

#### 5.6.1 명령문 수준 이전 트리거

다음은 emp 테이블에 삽입 작업 이전에 메시지를 표시하는 간단한 문 수준 이전 트리거 예제입니다.

```
CREATE OR REPLACE TRIGGER emp_alert_trig
BEFORE INSERT ON emp
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'New employees are about to be added');
END;
```

다음은 몇 가지 새로운 행이 명령 단일 실행에 삽입되는 것과 같이, INSERT 가 구성되어있습니다. 7900 에서 7999 까지 직원 ID 의 각 행에 1000 증가되는 직원 ID 에 해당하는 행이 삽입됩니다. 그 아래는 새 3 행이 삽입된 명령 실행 결과입니다.

```
INSERT INTO emp (empno, ename, deptno) SELECT empno + 1000, ename, 40
FROM emp WHERE empno BETWEEN 7900 AND 7999;

New employees are about to be added

SELECT empno, ename, deptno FROM emp WHERE empno BETWEEN 8900 AND 8999;

EMPNO ENAME DEPTNO
---------- ---------- ----------
8900 JAMES 40
8902 FORD 40
8934 MILLER 40
```

"New employees are about to be added"메시지는 결과가 새로운 3 행 추가되는 트리거의 시작이기도 한 번만 나타납니다.

### 5.6.2 statement-level 이후

다음은 statement-level 후에 트리거의 예제입니다. emp 테이블에 삽입, 업데이트, 삭제 작업이 일어날 때마다 날짜, 사용자 활동 기록 empauditlog 테이블에 행이 추가됩니다.

```
CREATE TABLE empauditlog (
audit_date DATE,
audit_user VARCHAR2 (20),
audit_desc VARCHAR2 (20)
);
CREATE OR REPLACE TRIGGER emp_audit_trig
AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
v_action VARCHAR2 (20);
BEGIN
IF INSERTING THEN
v_action : = 'Added employee (s)';
ELSIF UPDATING THEN
v_action : = 'Updated employee (s)';
ELSIF DELETING THEN
v_action : = 'Deleted employee (s)';
END IF;
INSERT INTO empauditlog VALUES (SYSDATE, USER,
v_action);
END;
```

다음 명령 시퀀스는 2 행 2 개의 INSERT 명령을 사용하여 emp 테이블에 넣습니다. sal 과 comm 쌍방의 행렬은 1 개의 UPDATE 명령으로 업데이트됩니다. 그리고 두 줄은 1 개의 DELETE 명령으로 삭제됩니다.

```
INSERT INTO emp VALUES (9001, 'SMITH', 'ANALYST', 7782, SYSDATE, NULL,
NULL, 10);

INSERT INTO emp VALUES (9002, 'JONES', 'CLERK', 7782, SYSDATE, NULL, NULL,
10);

UPDATE emp SET sal = 4000.00, comm = 1200.00 WHERE empno IN (9001, 9002);

DELETE FROM emp WHERE empno IN (9001, 9002);

SELECT TO_CHAR (AUDIT_DATE, 'DD - MON - YY HH24 : MI : SS') AS "AUDIT
DATE"

audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;

AUDIT DATE AUDIT_USER AUDIT_DESC
------------------ -------------------- ------------ --------
31 - MAR - 05 14:59:48 SYSTEM Added employee (s)
31 - MAR - 05 15:00:07 SYSTEM Added employee (s)
31 - MAR - 05 15:00:19 SYSTEM Updated employee (s)
31 - MAR - 05 15:00:34 SYSTEM Deleted employee (s)
```

empauditlog 테이블의 내용은 많은 시간 트리거가 시작했는지 보여줍니다. 하나는 2 회 삽입 하나의 갱신 (2 행 갱신 후), 그리고 하나의 제거 (2 행이 삭제 후).

### 5.6.3 row-level 전 트리거

다음은 30 개 부서에 소속 신입 사원의 비율을 산출하고, emp 테이블에 삽입으로 다 수준 이전 트리거의 예입니다.

```
CREATE OR REPLACE TRIGGER emp_comm_trig
BEFORE INSERT ON emp
FOR EACH ROW
BEGIN
IF : NEW.deptno = 30 THEN
: NEW.comm : = : NEW.sal * .4;
END IF;
END;
```

다음은 두 명의 직원을 추가하고 트리거가 비율을 산출하고, 새로운 직원 행들의 일부로 추가되었다는 것을 의미합니다.

```
INSERT INTO emp VALUES (9005, 'ROBERS', 'SALESMAN', 7782, SYSDATE,
3000.00, NULL, 30);
INSERT INTO emp VALUES (9006, 'ALLEN', 'SALESMAN', 7782, SYSDATE, 4500.00,
NULL, 30);
SELECT * FROM emp WHERE empno IN (9005, 9006);
EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO
---------- ---------- --------- ---------- --------- - -------- ---------
- ----------
9005 ROBERS SALESMAN 7782 01 - APR - 05 3000 1200 30
9006 ALLEN SALESMAN 7782 01 - APR - 05 4500 1800 30
```

### 5.6.4 row-level 트리거 이후

아래는 row-level 후에 트리거의 예입니다. 새로운 직원 행이 삽입되는 경우 트리거는 직원의 jobhist 테이블에 새 행을 추가합니다. 기존 직원이 업데이트가 되는 경우 트리거는 최신 jobhist 행 enddate 열 (enddate 는 null 로오고 하나인 추정)를 현재 날짜로 설정하고 직원에 해당하는 정보에 해당 jobhist 줄을 삽입합니다.

마지막으로 트리거는 empchglog 테이블 행 작동 묘사와 함께 추가합니다.

```
CREATE TABLE empchglog (
chg_date DATE,
chg_desc VARCHAR2 (30)
);
CREATE OR REPLACE TRIGGER emp_chg_trig
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW
DECLARE
v_empno emp.empno % TYPE;
```

```
  v_deptno emp.deptno % TYPE;
  v_dname dept.dname % TYPE;
  v_action VARCHAR2 (7);
  v_chgdesc jobhist.chgdesc % TYPE;
  BEGIN
  IF INSERTING THEN
  v_action : = 'Added';
  v_empno : = : NEW.empno;
  v_deptno : = : NEW.deptno;
  INSERT INTO jobhist VALUES (: NEW.empno, SYSDATE, NULL,
  : NEW.job : NEW.sal : NEW.comm : NEW.deptno, 'New Hire');
  ELSIF UPDATING THEN
  v_action : = 'Updated';
  v_empno : = : NEW.empno;
  v_deptno : = : NEW.deptno;
  v_chgdesc : = '';
  IF NVL (: OLD.ename, '- null -')! = NVL (: NEW.ename, '- null -') THEN
  v_chgdesc : = v_chgdesc | | 'name';
  END IF;
  IF NVL (: OLD.job, '- null -')! = NVL (: NEW.job, '- null -') THEN
  v_chgdesc : = v_chgdesc | | 'job';
  END IF;
  IF NVL (: OLD.sal, -1)! = NVL (: NEW.sal, -1) THEN
  v_chgdesc : = v_chgdesc | | 'salary';
  END IF;
  IF NVL (: OLD.comm, -1)! = NVL (: NEW.comm, -1) THEN
  v_chgdesc : = v_chgdesc | | 'commission,';
  END IF;
  IF NVL (: OLD.deptno, -1)! = NVL (: NEW.deptno, -1) THEN
  v_chgdesc : = v_chgdesc | | 'department,';
  END IF;
  v_chgdesc : = 'Changed'| | RTRIM (v_chgdesc ',');
  UPDATE jobhist SET enddate = SYSDATE WHERE empno = : OLD.empno
  AND enddate IS NULL;
  INSERT INTO jobhist VALUES (: NEW.empno, SYSDATE, NULL,
  : NEW.job : NEW.sal : NEW.comm : NEW.deptno, v_chgdesc);
  ELSIF DELETING THEN
  v_action : = 'Deleted';
  v_empno : = : OLD.empno;
  v_deptno : = : OLD.deptno;
  END IF;

  INSERT INTO empchglog VALUES (SYSDATE,
  v_action | | 'employee #'| | v_empno);
  END;
```

하단에 있는 명령의 첫 시퀀스 안에 두 명의 직원은 두 개의 개별 INSERT 명령을 사용하여 추가되고 양측은 하나의 UPDATE 명령을 사용하여 업데이트됩니다. jobhist 테이블의 내용은 각각의 영향을 받은 행 트리거 동작을 보여줍니다. 두 개의 새로운 직원을 위한 새로운 신입 사원 항목과 두 명의 직원 비율 갱신을 위한 두 개의 비율 기록을 변화합니다. empchglog 테이블에 트리거가 총 4 회 시작되고 2 개의 행에 한 번 각 동작도 보여주고 있습니다.

```
  INSERT INTO emp VALUES (9003, 'PETERS', 'ANALYST', 7782, SYSDATE, 5000.00,
NULL, 40);
```

```
  INSERT INTO emp VALUES (9004, 'AIKENS', 'ANALYST', 7782, SYSDATE, 4500.00,
NULL, 40);

  UPDATE emp SET comm = sal * 1.1 WHERE empno IN (9003, 9004);

  SELECT * FROM jobhist WHERE empno IN (9003, 9004);

  EMPNO STARTDATE ENDDATE JOB SAL COMM DEPTNO CHGDESC
  ---------- --------- --------- --------- ---------- --- ------- ---------
- ------------------
  9003 31 - MAR - 05 31 - MAR - 05 ANALYST 5000 40 New Hire
  9004 31 - MAR - 05 31 - MAR - 05 ANALYST 4500 40 New Hire
  9003 31 - MAR - 05 ANALYST 5000 5500 40 Changed commission
  9004 31 - MAR - 05 ANALYST 4500 4950 40 Changed commission

  SELECT * FROM empchglog;

  CHG_DATE CHG_DESC
  --------- ------------------------------
  31 - MAR - 05 Added employee # 9003
  31 - MAR - 05 Added employee # 9004
  31 - MAR - 05 Updated employee # 9003
  31 - MAR - 05 Updated employee # 9004
```

마지막으로, 두 직원이 하나의 DELETE 명령으로 삭제됩니다. empchglog 테이블은 각 직원의 제거에 트리거가 2 회 시작되었음을 현재 보여줍니다.

```
  DELETE FROM emp WHERE empno IN (9003, 9004);

  SELECT * FROM empchglog;

  CHG_DATE CHG_DESC
  --------- ------------------------------
  31 - MAR - 05 Added employee # 9003
  31 - MAR - 05 Added employee # 9004
  31 - MAR - 05 Updated employee # 9003
  31 - MAR - 05 Updated employee # 9004
  31 - MAR - 05 Deleted employee # 9003
  31 - MAR - 05 Deleted employee # 9004
```

**제 6장 패키지**

이 장에서는 Postgres Plus Advanced Serve 의 패키지 개념을 설명합니다. 패키지는 함수, 프로시저, 변수, 커서, 사용자 정의 레코드 형식 레코드를 수집하는 이름으로, 일반적인 한정자 - 패키지 ID 를 사용하여 참조됩니다. 패키지에는 다음과 같은 특징이 있습니다.

- 패키지는 관련된 목표를 수행 편성된 함수나 프로시저의 편리한 방법을 제공합니다. 패키지 함수나 프로시저를 사용 하는 권한은 모든 패키지에 주어진 1 개의 권한에 따라 달라집니다. 모든 패키지 프로그램은 일반적인 명칭으로는 언급되고 있지 않습니다.

- 패키지에 있는 몇 가지 함수 프로시저 변수 형식 등은 *public* 으로 선언될 수 있습니다. 공용 요소는 참조되며 패키지 EXECUTE 권한이 있는 다른 프로그램에서 볼 수 있습니다. 공용 함수와 프로시저 내용은 그 서명 (프로그램 이름, 만약 있다면 매개 변수, 함수의 반환 형식) 만 보입니다. 이 함수와 프로시저의 SPL 코드는 다른 사람이 액세스할 수 없습니다. 따라서, 패키지를 이용하는 응용 프로그램은 서명 된 정보에 의존합니다. (자체적으로 프로시저 논리 안에는 없습니다).
- 패키지에 있는 다른 함수 프로시저 변수 형식 등은 *Private* 선언할 수 있습니다. 개인 요소는 찾아볼 수 패키지의 기능과 프로시저에 의해 사용될 수 있습니다. 다른 외부 응용 프로그램은 없습니다. 개인 요소는 패키지 프로그램에만 사용합니다.
- 함수와 프로시저 이름은 패키지에 overloading 수 있습니다. 하나 이상의 함수 / 프로시저는 서로 다른 서명에 동일한 이름으로 정의할 수 있습니다. 다른 종류의 입력에서 동일한 작업을 수행하는 같은 이름의 프로그램을 만들기 위한 기능을 제공합니다.

### 6.1 패키지 구성 요소

패키지는 2 개의 주요 부분으로 구성됩니다:

- *패키지 규격*: 공용 인터페이스 (이들은 패키지 외부에서 볼 수 있는 부분입니다). 패키지 사양 범위에 속하는 모든 데이터베이스 개체를 공개합니다.
- 패키지 본체 패키지 사양에서 선전한 모든 데이터베이스 개체의 실제 구현을 포함합니다.

패키지 본문은 패키지 사양 내역을 구현합니다. 구현 세부 사항 및 애플리케이션에는 보이지 않는 개인 선언을 포함합니다. 따라서, 사양 변경 없이, 디버그 기능 강화, 교체가 가능합니다. 마찬가지로 구현 세부 정보가 응용 프로그램에는 보이지 않기 때문에 프로그램을 부르고 다시 컴파일 하지 않고 본체를 변경할 수 있습니다.

#### 6.1.1 패키지 사양 구문

다음은 패키지 사양 구문 입니다:

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

*package_name 은* 패키지에 할당된 ID 입니다. AUTHID 항이 생략되거나 DEFINER 이 지정되면 패키지 소유자의 권리 및 검색 경로는 데이터 객체에 대한 액세스 권한을 결정하고 무조건 데이터베이스 객체 참조를 결정하기 위해 각각 사용됩니다. CURRENT_USER 의 지정이 완료되면 패키지의 프로그램을 실행하는 현재 사용자의 권리 및 검색 경로는 권한을 결정하고 무조건 개체 참조를 결정하는 데 사용됩니다. *declaration* 공용 변수의 식별자 입니다. 공용 변수는 *package_name.variable* 구문을 사용하여 패키지 외부에서 접근할 수 있습니다. 지정되지 않은 하나 이상의 공용 변수가 있을 가능성이 있습니다. 공용 변수 정의는 프로시저와 함수 선언 전에 해야 합니다. *declaration* 다음 중 하나에 있습니다.

- 변수 선언 (섹션 4.3 를 참조하시기 바랍니다)
- 레코드 선언 (섹션 4.3.4 를 참조하십시오)
- 배열 선언 (섹션 4.9.1 를 참조하십시오)
- REF CURSOR 로 커서 변수 선언 (섹션 4.8 를 참조하시기 바랍니다)
- TYPE 기록, 모음과 REF CURSOR 를 정의합니다.

*proc_name* 는 공용 프로시저 식별자 입니다. 공용 프로시저는 구문 *package_name.proc_name [(...)]*를 사용하여 패키지 외부에서 시작할 수 있습니다. 프로시저 형식 매개 변수는 *parm1, parm2* ...입니다. *datatype1, datatyp e2, ..., parm1, parm2* ...은 데이터 형식입니다. IN, IN OUT, 그리고 OUT 각 형식 매개 변수 모드입니다. 아무것도 지정하지 않으면 기본값은 IN 입니다.

*func_name* 은 공용 함수의 ID 입니다. 공용 함수는 구문 *package_name.func_name [(...)]* 를 사용하여 패키지 외부에서 시작할 수 있습니다. 함수의 형식 매개 변수, *parm1, parm2* ...입니다. *datatype1, datatype2* ... *parm1, parm2* ...은 데이터 형식입니다. IN, IN OUT, 과 OUT 각 형식 매개 변수 매개 변수 모드입니다. 아무것도 지정하지 않으면 기본값은 IN 입니다. *return_type* 은 함수의 반환 데이터 형식입니다. IN 매개 변수를 지정 실수 IN 매개 변수를 대신 사용하는 기본값으로 초기화 할 수 있습니다.

### 6.1.2 패키지 본문의 구문

다음은 패키지 본문의 구문입니다:

```
CREATE [OR REPLACE] PACKAGE BODY package_name
(IS | AS)
[   private_declaration;    ] ...
[(PROCEDURE proc_name
[(parm1 [IN | IN OUT | OUT] datatype1
[, parm2 [IN | IN OUT | OUT] datatype2 ...)
(IS | AS)
[proc_declaration] ...
BEGIN
       statement; ...
[EXCEPTION
WHEN ... THEN
         statement; ...]
```

```
END;
|
FUNCTION func_name
[(parm1 [IN | IN OUT | OUT] datatype1
[, parm2 [IN | IN OUT | OUT] datatype2 ...)
RETURN return_type
(IS | AS)
[func_declaration] ...
BEGIN
       statement; ...
[EXCEPTION
WHEN ... THEN
         statement; ...]
END; }...]
[BEGIN
       init_statement; ...]
END [    package_name    ];
```

*package_name* 이것이 패키지 본문이 있는 패키지의 이름입니다. 기존 패키지사양과 동일한 이름입니다.

*private_declaration* 패키지의 모든 기능 및 프로시저를 통해 액세스할 수 있는 전용 변수의 식별자입니다. 항상 하나 이상의 공용 변수가 있을 수 있습니다. *private_declaration* 다음 중 하나에 있습니다.

- 변수 선언 (섹션 4.3 를 참조하시기 바랍니다)
- 레코드 선언 (섹션 4.3.4 를 참조하십시오)
- 배열 선언 (섹션 4.9 를 참조하십시오)
- REF CURSOR 로 커서 변수 선언 (섹션 4.8 를 참조하시기 바랍니다)
- TYPE 기록, 모음과 REF CURSOR 를 정의합니다

*proc_name* 이 패키지 사양 중 선언한 공용프로시저 식별자와 동일 형식 매개 변수 이름 *(parm1, parm2, ...)*, 데이터 유형 *(datatype1, datatype2, ...)* 매개 변수 형태, 형식 매개 변수의 순서와 형식 매개 변수)의 서명이 공용프로시저 선언의 서명과 정확하게 일치하는 경우 *proc_name* 이 공용프로시저의 본문을 정의합니다.

앞의 단락으로 작성된 조건이 참이 아닌 경우 *proc_name 은* 공용프로시저를 정의합니다.

*parm1, parm2, ...*는 프로시저 형식의 매개변수입니다. *datatype1, datatype2 ...*은 각 *parm1, parm2 ...*의 자료유형입니다. IN, IN OUT 다고 OUT 는 각 형식 매개 변수 매개 변수 모드입니다. 아무것도 지정되지 않는 경우, 기본값은 IN 입니다. IN 매개 변수는 지정 실수 IN 매개 변수를 대신 사용하는 기본값으로 초기화됩니다.

*proc_variable* 는 *proc_name 의/* 프로시저 내에서만 액세스할 수 있는 변수의 식별자 입니다. 항상 하나 이상의 변수가 있을 수 있습니다. *Datatype 은* *proc_variable* 자료 유형입니다. *statement* 는 SPL 프로그램 문장입니다.

*func _name* 이 패키지 사양 안에 선언된 공용 함수의 식별자와 동일 형식 매개 변수 이름 *(parm1, parm2,* ...), 데이터 유형 *(datatype1, datatype2,* ...) 매개 변수 형태, 형식 매개 변수의 순서와 형식 매개 변수)의 서명이 공용프로시저 선언의 서명과 정확하게 일치하는 경우, *func _name* 이 공용 함수의 본문을 정의합니다.

앞의 단락으로 작성된 조건이 완전하지 않을 경우, *func _name* 공용 함수 정의합니다.

*parm1, parm2* ... 함수 형식 매개변수입니다. *datatype1, datatype2* ...은 각 *parm1, parm2* ... 자료 유형입니다. IN, IN OUT 과 OUT 는 각 형식 매개 변수 매개 변수 모드입니다. 아무것도 지정하지 않으면 기본값은 IN 입니다. *return_type* 은 함수의 반환 데이터 형식입니다.

*func_variable* 는 *func _name* 함수 내에서만 액세스할 수 있는 변수의 식별자 입니다. 항상 하나 이상의 변수가 있을 수 있습니다. *datatype* 는 *func_variable* 자료 유형입니다. *statement* , SPL 프로그램 문장입니다.

*init_statement* 는 패키지 시스템 초기화 부 문장입니다. 초기화 부분을 지정하는 경우, 적어도 1 개의 문장 포함해야 합니다. 우선 패키지를 참조하면 초기화 부 문장은 사용자 세션 마다 한 번씩 실행됩니다

## 6.2 패키지 구성

여기서는 패키지를 생성하고 데이터베이스에 저장하십시오. 여기서 패키지는 실행 가능한 코드가 아닐 수 있습니다. 좀더 정확히 말하면, 그들은 사용되는 코드 창고입니다. 패키지를 사용하면 실제로 패키지 요소에 대한 참조를 실행하거나 만들 수 있습니다. 이 정보는 패키지 사양에 포함됩니다.

### 6.2.1 패키지 생성 사용

패키지 규격은 패키지에 외부에서 참조 할 수 있는 모든 요소에 대한 정의를 포함하고 있습니다. 이것은 패키지 공용 성분이라는 동작이 패키지로 제공됩니다. 다음은 패키지 사양입니다.

```
-
- Package specification for the 'emp_admin'packa    ge.
-
CREATE OR REPLACE PACKAGE emp_admin
IS

FUNCTION get_dept_name (
p_deptno NUMBER DEFAULT 10
)
RETURN VARCHAR2;
FUNCTION update_emp_sal (
```

```
p_empno NUMBER,
p_raise NUMBER
)
RETURN NUMBER;
PROCEDURE hire_emp (
p_empno NUMBER,
p_ename VARCHAR2,
p_job VARCHAR2,
p_sal NUMBER,
p_hiredate DATE DEFAULT sysdate,
p_comm NUMBER DEFAULT 0,
p_mgr NUMBER,
p_deptno NUMBER DEFAULT 10
);
PROCEDURE fire_emp (
p_empno NUMBER
);

END emp_admin;
```

여기서는 패키지 사양 emp_admi 를 만들었습니다. 이 패키지 사양은 두 함수와 두 저장형 프로시저로 구성됩니다. 편의상 OR REPLACE 절을 CREATE PACK AGE 문장에 추가할 수 있습니다.

### 6.2.2 패키지 본문 생성

패키지 본문은 패키지 사양 뒤에 있는 실제 구현을 포함합니다. 위의 패키지 사양 emp_admin 이 사양을 구현하는 패키지 본문을 만듭니다. 본체는 사양에 함수와 저장형 프로시저 구현을 포함합니다.

```
-
- Package body for the 'emp_admin'package.
-
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
-
- Function that queries the 'dept'table based on the department
- number and returns the corresponding department name.
-
FUNCTION get_dept_name (
p_deptno IN NUMBER DEFAULT 10
)
RETURN VARCHAR2
IS
v_dname VARCHAR2 (14);
BEGIN
SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
RETURN v_dname;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Invalid department number'| | p_deptno);
RETURN '';
END;
-
```

```
- Function that updates an employee 's salary based on the
- employee number and salary increment / decrement passed
- as IN parameters. Upon successful completion the function
- returns the new updated salary.
-
FUNCTION update_emp_sal (
p_empno IN NUMBER,
p_raise IN NUMBER
)
RETURN NUMBER
IS
v_sal NUMBER : = 0;
BEGIN
SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
v_sal : = v_sal + p_raise;
UPDATE emp SET sal = v_sal WHERE empno = p_empno;
RETURN v_sal;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ( 'Employee'| | p_empno | | 'not found');
RETURN -1;
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'The following is SQLERRM :');
DBMS_OUTPUT.PUT_LINE (SQLERRM);
DBMS_OUTPUT.PUT_LINE ( 'The following is SQLCODE :');
DBMS_OUTPUT.PUT_LINE (SQLCODE);
RETURN -1;
END;
-
- Procedure that inserts a new employee record into the 'emp'table.
-
PROCEDURE hire_emp (
p_empno NUMBER,
p_ename VARCHAR2,
p_job VARCHAR2,
p_sal NUMBER,
p_hiredate DATE DEFAULT sysdate,
p_comm NUMBER DEFAULT 0,
p_mgr NUMBER,
p_deptno NUMBER DEFAULT 10
)
AS
BEGIN
INSERT INTO emp (empno, ename, job, sal, hiredate, comm, mgr, deptno)
VALUES (p_empno, p_ename, p_job, p_sal,
p_hiredate, p_comm, p_mgr, p_deptno);
END;
-
- Procedure that deletes an employee record from the 'emp'table based
- on the employee number.
-
PROCEDURE fire_emp (
p_empno NUMBER
)
AS
BEGIN
DELETE FROM emp WHERE empno = p_empno;
END;
END;
```

### 6.3 패키지 참조

유형 항목, 하위 프로그램의 참조는 패키지 사양 사이 선언된 점 표기법을 사용합니다. 예를 들면:

```
package_name. type_name
package_name. item_name
package_name. subprogram_name
```

패키지 사양 emp_admin 에서 함수를 시작하려면 다음 SQL 명령을 실행합니다.

```
SELECT emp_admin.get_dept_name(10) FROM DUAL;
```

여기서는 패키지 emp_admin 안에서 선언한 get_dept_name 함수를 시작합니다. 부서 번호를 논점화 해서 기능으로 전달, 부서 이름을 반환합니다. 반환 값은 ACCOUNTING 에서 부서 번호 10 에 해당합니다.

### 6.4 사용자 정의 유형 패키지 사양

다음은 패키지의 문맥 사이 앞 장에서 설명하는 사용자 정의 유형을 포함합니다.

emp_rpt 패키지 규격은 레코드 유형 emprec_typ 약화 형식 REF CURSOR 의 emp_refcur 을 두 함수와 두 프로시저와 함께 액세스할 수 있는 선언을 보여줍니다. 함수 open_emp_by_dept 는 REF CURSOR 유형 EMP_REFCUR 을 반환합니다. 프로시저 fetch_emp 과 close_refcur 은 형식 매개 변수의 약점 형식의 REF CURSOR 를 선언합니다. 레코드 유형과 REF CURSOR 정보는 각 섹션 4.3.4 과 4.8 을 참조하십시오.

```
CREATE OR REPLACE PACKAGE emp_rpt
IS
TYPE emprec_typ IS RECORD (
empno NUMBER (4),
ename VARCHAR (10)
);
TYPE emp_refcur IS REF CURSOR;

FUNCTION get_dept_name (
p_deptno IN NUMBER
) RETURN VARCHAR2;
FUNCTION open_emp_by_dept (
p_deptno IN emp.deptno % TYPE
) RETURN EMP_REFCUR;
PROCEDURE fetch_emp (
p_refcur IN OUT SYS_REFCURSOR
);
PROCEDURE close_refcur (
```

```
p_refcur IN OUT SYS_REFCURSOR
);
END emp_rpt;
```

패키지 본문은 어떤 개인 변수 (정적 커서 dept_cur 테이블 유형 depttab_typ 테이블 변수 t_dept 정수 변수 t_dept_max, 레코드 변수 r_emp) 선언을 나타냅니다. 정적 커서, 배열, 레코드 변수에 대한 자세한 내용은 해당 섹션 4.7, 4.9.1, 4.3.4 를 참조하십시오.

```
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
CURSOR dept_cur IS SELECT * FROM dept;
TYPE depttab_typ IS TABLE of dept % ROWTYPE
INDEX BY BINARY_INTEGER;
t_dept DEPTTAB_TYP;
t_dept_max INTEGER : = 1;
r_emp EMPREC_TYP;

FUNCTION get_dept_name (
p_deptno IN NUMBER
) RETURN VARCHAR2
IS
BEGIN
FOR i IN 1 .. t_dept_max LOOP
IF p_deptno = t_dept (i). deptno THEN
RETURN t_dept (i). dname;
END IF;
END LOOP;
RETURN 'Unknown';
END;

FUNCTION open_emp_by_dept (
p_deptno IN emp.deptno % TYPE
) RETURN EMP_REFCUR
IS
emp_by_dept EMP_REFCUR;
BEGIN
OPEN emp_by_dept FOR SELECT empno, ename FROM emp
WHERE deptno = p_deptno;
RETURN emp_by_dept;
END;

PROCEDURE fetch_emp (
p_refcur IN OUT SYS_REFCURSOR
)
IS
BEGIN
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH p_refcur INTO r_emp;
EXIT WHEN p_refcur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (r_emp.empno | | ''| | r_emp.ename);
END LOOP;
END;

PROCEDURE close_refcur (
```

```
p_refcur IN OUT SYS_REFCURSOR
)
IS
BEGIN
CLOSE p_refcur;
END;
BEGIN
OPEN dept_cur;
LOOP
FETCH dept_cur INTO t_dept (t_dept_max);
EXIT WHEN dept_cur % NOTFOUND;
t_dept_max : = t_dept_max + 1;
END LOOP;
CLOSE dept_cur;
t_dept_max : = t_dept_max - 1;
END emp_rpt;
```

이 패키지는 전용 테이블 변수 t_dept 개인 정적 커서를 사용합니다 dept_cur 초기화 부분을 포함하고 있습니다. t_dept 는 get_dept_name 함수 부서이름 참조 테이블로 유용합니다.

open_emp_by_dept 함수는 사원 번호와 부서 이름을 결과로 설정된 REF CURSOR 변수를 반환합니다. 그리고 REF CURSOR 변수는 개별 행을 결과 목록의 검색 및 명부를 하기 위하여 fetch_emp 프로시저에 전달할 수 있습니다. 마지막으로 close_refcur 절차는 이 결과 목록에 관련된 RE F CURSOR 변수를 닫는 데 사용할 수 있습니다.

다음 익명 블록은 패키지의 기능과 프로시저를 수행합니다. 익명 블록의 선언 부분은 커서 변수 v_emp_cur 패키지 공용 REF CURSOR 유형 EMP_REFCUR 선언에 유의하시기 바랍니다. v_emp_cur 패키지 함수와 프로시저 간에 전달되는 결과 목록에 대한 포인터를 포함하고 있습니다.

```
DECLARE
v_deptno dept.deptno % TYPE DEFAULT 30;
v_emp_cur emp_rpt.EMP_REFCUR;
BEGIN
v_emp_cur : = emp_rpt.open_emp_by_dept (v_deptno);
DBMS_OUTPUT.PUT_LINE ( 'EMPLOYEES IN DEPT #'| | v_deptno | |
':'| | emp_rpt.get_dept_name (v_deptno));
emp_rpt.fetch_emp (v_emp_cur);
DBMS_OUTPUT.PUT_LINE ('**********************');
DBMS_OUTPUT.PUT_LINE (v_emp_cur % ROWCOUNT | | 'rows were retrieved');
emp_rpt.close_refcur (v_emp_cur);
END;
```

다음은 익명 블록의 결과입니다.

```
EMPLOYEES IN DEPT # 30 : SALES
EMPNO ENAME
----- -------
7499 ALLEN
7521 WARD
7654 MARTIN
7698 BLAKE
```

```
7844 TURNER
7900 JAMES
**********
6 rows were retrieved
```

다음 익명 블록에서는 동일한 결과를 얻을 수 있을 것입니다. 다른 방법을 설명합니다. 패키지 프로시저의 fetch_emp 와 close_refcur 를 사용하는 대신,이 프로그램 논리는 익명 블록에 직접 코드화할 수 있습니다. 익명 블록 선언 부분에서는 레코드 변수 r_emp 패키지 공용 레코드 유형을 사용하여 선언한 EMPREC_TYP 추가 주의하십시오.

```
DECLARE
v_deptno dept.deptno % TYPE DEFAULT 30;
v_emp_cur emp_rpt.EMP_REFCUR;
r_emp emp_rpt.EMPREC_TYP;
BEGIN
v_emp_cur : = emp_rpt.open_emp_by_dept (v_deptno);
DBMS_OUTPUT.PUT_LINE ( 'EMPLOYEES IN DEPT #'| | v_deptno | |
':'| | emp_rpt.get_dept_name (v_deptno));
DBMS_OUTPUT.PUT_LINE ( 'EMPNO ENAME');
DBMS_OUTPUT.PUT_LINE ('----- -------');
LOOP
FETCH v_emp_cur INTO r_emp;
EXIT WHEN v_emp_cur % NOTFOUND;
DBMS_OUTPUT.PUT_LINE (r_emp.empno | | ''| |
r_emp.ename);
END LOOP;
DBMS_OUTPUT.PUT_LINE ('**********');
DBMS_OUTPUT.PUT_LINE (v_emp_cur % ROWCOUNT | | 'rows were retrieved');
CLOSE v_emp_cur;
END;
```

다음은 이 익명블록의 결과입니다.

```
EMPLOYEES IN DEPT # 30 : SALES
EMPNO ENAME
----- -------
7499 ALLEN
7521 WARD
7654 MARTIN
7698 BLAKE
7844 TURNER
7900 JAMES
**********
6 rows were retrieved
```

### 6.5 패키지 제거

모든 패키지 또는 간단하게 패키지 본문을 삭제하는 구문은 다음과 같습니다:

```
DROP PACKA GE [BODY] package_name;
```

키워드 BODY 가 생략되면 패키지 사양 및 패키지 본문 모두 삭제됩니다. 즉 전체 패키지를 제거합니다. 키워드 BODY 가 지정되면 패키지 본문만이 삭제됩니다. 패키지 규격은 그대로 유지됩니다. *package_name* 는 제거할 패키지의 ID 입니다.

다음 문장은 *emp_admin* 패키지 본체 단지 해제합니다:

```
DROP PACKAGE BODY emp_admin;
```

다음 문장은 *emp_admin* 패키지 전체를 삭제합니다:

```
DROP PACKAGE emp_admin;
```

**제 7 장**
**통합 패키지**

이 장은 Postgres Plus Advanced Server에서 제공되는 통합 패키지를 설명합니다.
특정 패키지는, 슈퍼유저가 아닌 사용자는 패키지의 함수나 프로시저를 사용하기 전에 명시적으로 패키지의 EXECUTE 권한을 부여 받아야 합니다.
대부분의 통합 패키지는, EXECUTE 권한은 기본값으로 PUBLIC에 부여 받습니다. 권한부여는 GRANT 명령을 참조하십시오.
전체 통합 패키지는, 통합 패키지에 권한을 부여하거나 해제할 때, 지정된 특별한 시스템 사용자에 의해 소유됩니다.

### 7.1 DBMS_ALERT

DBMS_ALERT 패키지는 경고의 송수신을 등록하는 기능을 제공합니다.
DBMS_ALERT 패키지에서 사용 가능한 프로시저와 함수는 다음 표와 같습니다.

**Table 7-1 DBMS_ALERT 함수/프로시저**

| 함수/프로시저 | 반환 형식 | 설명 |
|---|---|---|
| REGISTER(*name*) | n/a | 경고를 받을 수 있도록 *name*을 등록한다. |
| REMOVE(*name*) | n/a | 경고 *name*를 등록을 취소한다. |
| REMOVEALL | n/a | 모든 경고를 제거한다. |
| SIGNAL(*name*, *message*) | n/a | *message* 함께 경고 *name* 알림 |
| WAITANY(*name* OUT, *message* OUT, *status* OUT, *timeout*) | n/a | 등록된 경고의 발생을 기다린다. |
| WAITONE(*name*, *message* OUT, *status* OUT, *timeout*) | n/a | *지정 경고 name*의 발생을 기다린다. |

### 7.1.1 REGISTER

REGISTER 프로시저는 현재 세션에서 지정된 경고의 알림을 받을 수 있도록 합니다.

REGISTER *(name VARCHAR2)*

**매개 변수**

*name*

등록된 경고 명.

**예**

다음 익명 블록은 지정된 경고, alert_test를 등록하고 신호를 기다립니다.

```
DECLARE
 v_name VARCHAR2 (30) : = 'alert_test';
 v_msg VARCHAR2 (80);
 v_status INTEGER;
 v_timeout NUMBER (3) : = 120;
 BEGIN
 DBMS_ALERT.REGISTER (v_name);
 DBMS_OUTPUT.PUT_LINE ( 'Registered for alert'| | v_name);
 DBMS_OUTPUT.PUT_LINE ( 'Waiting for signal ...');
 DBMS_ALERT.WAITONE (v_name, v_msg, v_status, v_timeout);
 DBMS_OUTPUT.PUT_LINE ( 'Alert name :'| | v_name);
 DBMS_OUTPUT.PUT_LINE ( 'Alert msg :'| | v_msg);
 DBMS_OUTPUT.PUT_LINE ( 'Alert status :'| | v_status);
 DBMS_OUTPUT.PUT_LINE ( 'Alert timeout :'| | v_timeout | | 'seconds');
 DBMS_ALERT.REMOVE (v_name);
 END;

Registered for alert alert_test
Waiting for signal...
```

### 7.1.2 REMOVE

REMOVE 프로시저는 지정된 경고 세션의 등록을 취소합니다.

REMOVE *(name VARCHAR2)*

**매개 변수**

*name*

등록을 취소하는 경고의 명칭.

### 7.1.3 REMOVEALL

REMOVEALL 프로시저는 모든 경고 세션의 등록을 취소합니다.

REMOVEALL

### 7.1.4 SIGNAL

SIGNAL 절차는 지정된 경고의 발생을 알립니다.

SIGNAL *(name VARCHAR2, message VARCHAR2)*

**매개 변수**

*name*

경고의 이름.

*message*

경고와 함께 정보를 전달.

**예**

다음 익명 블록은 alert_test 경고를 알립니다.

```
DECLARE
v_name VARCHAR2 (30) : = 'alert_test';
BEGIN
DBMS_ALERT.SIGNAL (v_name, 'This is the message from'| | v_name);
DBMS_OUTPUT.PUT_LINE ( 'Issued alert for'| | v_name);
END;

Issued alert for alert_test
```

### 7.1.5 WAITANY

WAITANY 프로시저는 등록된 경고들 중 하나의 경고의 발생을 기다립니다.

WAITANY *(name OUT VARCHAR2, message OUT VARCHAR2,*

*status OUT INTEGER, timeout NUMBER)*

**매개 변수**

*name*

경고의 이름을 받는 변수이다.

*message*

SIGNAL 프로시저 통해 보내진 메시지를 받는 변수이다.

*status*

연산에서 반환된 상태 코드입니다. 가능한 값 : 0 - 경고 발생; 1 -. 타임아웃 발생.

*timeo ut*

초 단위의 경고를 기다리는 시간.

**예**

다음 익명 블록은 WAITANY 프로시저로 지정된 경고 alert_test 또는 any_alert을 받습니다.

```
DECLARE
v_name VARCHAR2 (30);
v_msg VARCHAR2 (80);
v_status INTEGER;
v_timeout NUMBER (3) : = 120;
BEGIN
DBMS_ALERT.REGISTER ( 'alert_test');
DBMS_ALERT.REGISTER ( 'any_alert');
DBMS_OUTPUT.PUT_LINE ( 'Registered for alert alert_test and any_alert');
DBMS_OUTPUT.PUT_LINE ( 'Waiting for signal ...');
DBMS_ALERT.WAITANY (v_name, v_msg, v_status, v_timeout);
```

```
DBMS_OUTPUT.PUT_LINE ( 'Alert name :'| | v_name);
DBMS_OUTPUT.PUT_LINE ( 'Alert msg :'| | v_msg);
DBMS_OUTPUT.PUT_LINE ( 'Alert status :'| | v_status);
DBMS_OUTPUT.PUT_LINE ( 'Alert timeout :'| | v_timeout | | 'seconds');
DBMS_ALERT.REMOVEALL;
END;

Registered for alert alert_test and any_alert
Waiting for signal ...
```

익명 블록의 다음 세션에서는 any_alert에 신호를 발행합니다.

```
DECLARE
v_name VARCHAR2 (30) : = 'any_alert';
BEGIN
DBMS_ALERT.SIGNAL (v_name, 'This is the message from'| | v_name);
DBMS_OUTPUT.PUT_LINE ( 'Issued alert for'| | v_name);
END;

Issued alert for any_alert
```

제어는 처음 익명 블록을 반환하고, 코드의 나머지 부분이 실행됩니다.

```
Registered for alert alert_test and any_alert
Waiting for signal ...
Alert name : any_alert
Alert msg : This is the message from any_alert
Alert status : 0
Alert timeout : 120 seconds
```

### 7.1.6 WAITONE

WAITONE 프로시저는 지정된 등록된 경고 발생을 기다립니다.

WAITONE *(name VARCHAR2, message OUT VARCHAR2,*

*status OUT INTEGER, timeout NUMBER)*

**매개 변수**

*name*

경고 명.

*message*

SIGNAL 프로시저를 통해 보내진 메시지를 받는 변수이다.

*status*

작업에서 반환하는 상태 코드입니다. 가능한 값 : 0 - 경고 발생; 1 -. 타임아웃 발생.

*timeout*

초 단위의 경고를 기다리는 시간.

**예**

다음 익명 블록은 WAITONE 프로시저로 지정된 경고 alert_test을 받는 것을 제외하고, WAITANY의 경우와 비슷합니다.

```
DECLARE
v_name VARCHAR2 (30) : = 'alert_test';
```

```
v_msg VARCHAR2 (80);
v_status INTEGER;
v_timeout NUMBER (3) : = 120;
BEGIN
DBMS_ALERT.REGISTER (v_name);
DBMS_OUTPUT.PUT_LINE ( 'Registered for alert'| | v_name);
DBMS_OUTPUT.PUT_LINE ( 'Waiting for signal ...');
DBMS_ALERT.WAITONE (v_name, v_msg, v_status, v_timeout);
DBMS_OUTPUT.PUT_LINE ( 'Alert name :'| | v_name);
DBMS_OUTPUT.PUT_LINE ( 'Alert msg :'| | v_msg);
DBMS_OUTPUT.PUT_LINE ( 'Alert status :'| | v_status);
DBMS_OUTPUT.PUT_LINE ( 'Alert timeout :'| | v_timeout | | 'seconds');
DBMS_ALERT.REMOVE (v_name);
END;

Registered for alert alert_test
Waiting for signal ...
```

두 번째 세션의 익명 블록을 통해 보내진 alert_test에 보낸 신호:

```
DECLARE
v_name VARCHAR2 (30) : = 'alert_test';
BEGIN
DBMS_ALERT.SIGNAL (v_name, 'This is the message from'| | v_name);
DBMS_OUTPUT.PUT_LINE ( 'Issued alert for'| | v_name);
END;

Issued alert for alert_test
```

첫 번째 세션은 경고되고, 제어는 익명 블록에 반환하고, 코드의 나머지 부분이 실행됩니다.

```
Registered for alert alert_test
Waiting for signal ...
Alert name : alert_test
Alert msg : This is the message from alert_test
Alert status : 0
Alert timeout : 120 seconds
```

### 7.1.7 종합적인 예

다음의 예에서는, dept 테이블이나 emp 테이블이 변경되는 경우, 경고를 보내는 두 개의 트리거를 사용합니다. 익명 블록은 이 경고를 기다리고 경고를 받은 후 메시지를 표시합니다.
다음은 dept과 emp 테이블의 트리거입니다.:

```
CREATE OR REPLACE TRIGGER dept_alert_trig
AFTER INSERT OR UPDATE OR DELETE ON dept
DECLARE
v_action VARCHAR2 (25);
BEGIN
```

```
IF INSERTING THEN
v_action : = 'added department (s)';
ELSIF UPDATING THEN
v_action : = 'updated department (s)';
ELSIF DELETING THEN
v_action : = 'deleted department (s)';
END IF;
DBMS_ALERT.SIGNAL ( 'dept_alert', USER | | v_action | | 'on'| |
SYSDATE);
END;

CREATE OR REPLACE TRIGGER emp_alert_trig
AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
v_action VARCHAR2 (25);
BEGIN
IF INSERTING THEN
v_action : = 'added employee (s)';
ELSIF UPDATING THEN
v_action : = 'updated employee (s)';
ELSIF DELETING THEN
v_action : = 'deleted employee (s)';
END IF;
DBMS_ALERT.SIGNAL ( 'emp_alert', USER | | v_action | | 'on'| |
SYSDATE);
END;
```

dept과 emp 테이블에 갱신이 다른 세션에서 발생하는 동안 다음 익명 블록이 세션에서 실행됩니다.

```
DECLARE
v_dept_alert VARCHAR2 (30) : = 'dept_alert';
v_emp_alert VARCHAR2 (30) : = 'emp_alert';
v_name VARCHAR2 (30);
v_msg VARCHAR2 (80);
v_status INTEGER;
v_timeout NUMBER (3) : = 60;
BEGIN
DBMS_ALERT.REGISTER (v_dept_alert);
DBMS_ALERT.REGISTER (v_emp_alert);
DBMS_OUTPUT.PUT_LINE ( 'Registered for alerts dept_alert and emp_alert');
DBMS_OUTPUT.PUT_LINE ( 'Waiting for signal ...');
LOOP
DBMS_ALERT.WAITANY (v_name, v_msg, v_status, v_timeout);
EXIT WHEN v_status! = 0;
DBMS_OUTPUT.PUT_LINE ( 'Alert name :'| | v_name);
DBMS_OUTPUT.PUT_LINE ( 'Alert msg :'| | v_msg);
DBMS_OUTPUT.PUT_LINE ( 'Alert status :'| | v_status);
DBMS_OUTPUT.PUT_LINE ('-----------------------------------' | |
'-------------------------');
END LOOP;
DBMS_OUTPUT.PUT_LINE ( 'Alert status :'| | v_status);
DBMS_ALERT.REMOVEALL;
END;

Registered for alerts dept_alert and emp_alert
Waiting for signal ...
```

다음의 변화가 사용자 mary 통해 이루어 집니다. :

```
INSERT INTO dept VALUES (50, 'FINANCE', 'CHICAGO');
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
```

다음의 변화가 사용자 john을 통해 이루어 집니다. :

```
INSERT INTO dept VALUES (60, 'HR', 'LOS ANGELES');
```

다음은 트리거의 신호를 받은 익명 블록에 의해 실행되는 출력입니다. :

```
Registered for alerts dept_alert and emp_alert
Waiting for signal ...
Alert name : dept_alert
Alert msg : mary added department (s) on 25 - OCT - 07 16:41:01
Alert status : 0
------------------------------------------------- -----------
Alert name : emp_alert
Alert msg : mary added employee (s) on 25 - OCT - 07 16:41:02
Alert status : 0
------------------------------------------------- -----------
Alert name : dept_alert
Alert msg : john added department (s) on 25 - OCT - 07 16:41:22
Alert status : 0
------------------------------------------------- -----------
Alert status : 1
```

### 7.2 DBMS_OUTPUT

DBMS_OUTPUT 패키지는 메시지 버퍼에서 메시지를 검색하거나 메시지 버퍼 또는 메시지(텍스트 라인) 송수신 기능을 제공합니다. 메시지 버퍼는 단일 세션에 한정됩니다. 세션에서 메시지 전송을 하려면 DBMS_PIPE 패키지를 사용하십시오.
DBMS_OUTPUT 패키지에서 사용 가능한 프로시저와 함수는 다음 표와 같습니다.

**표 7-2 DBMS_OUTPUT 함수 / 프로시저**

| 함수 / 프로시저 | 반환 형식 | 설명 |
|---|---|---|
| DISABLE | n/a | 메시지 송수신 기능을 해제한다. |
| ENABLE(*buffer_size*) | n/a | 메시지 송수신 기능을 활성화한다. |
| GET_LINE(*line* OUT, *status* OUT) | n/a | 메시지 버퍼로부터 라인을 가져온다. |
| GET_LINES(*lines* OUT, *numlines* IN OUT) | n/a | 메시지 버퍼로부터 복수의 라인을 가져온다. |
| NEW_LINE | n/a | 라인 종료 에 문자시퀀스를 넣는다. |
| PUT(*item*) | n/a | 라인 종료 에 문자 시퀀스없이 부분적 라인을 넣는다. |

| | | |
|---|---|---|
| PUT_LINE(*item*) | n/a | 라인 종료 에 문자 시퀀스와 완전한 라인을 넣는다. |
| SERVEROUTPUT(*stdout*) | n/a | PUT, PUT_LINE, 또는 NEW_LINE에서의 표준 출력 또는 메시지 버퍼 중 하나를 직접 메시지로 한다. |

다음의 표는 the DBMS_OUTPUT 패키지에 사용 가능한 공용 변수의 리스트입니다.

**Table 7-3 DBMS_OUTPUT 공용 변수**

| 공용 변수 | 데이터 형 | 값 | 설명 |
|---|---|---|---|
| chararr | TABLE | | 메시지 라인 |

### 7.2.1 CHARARR

CHARARR는 복수의 메시지 라인을 저장합니다.

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

### 7.2.2 DISABLE

DISABLE 프로시저는 메시지 버퍼를 제거합니다. DISABLE 프로시저가 실행되었을 때 버퍼 메시지는 접근할 수 없게 됩니다. PUT, PUT_LINE 또는 N EW_LINE 프로시저에서 나중에 보내진 메시지는 삭제됩니다. PUT, PUT_LINE 또는 NEW_LINE 프로시저가 실행되고 메시지가 잘못되면 오류가 반환되지 않습니다.

메시지의 송수신을 다시 허용하려면 ENABLE 프로시저 또는 SERVEROUTPUT (TRUE) 프로시저를 이용하세요.

DISABLE

**예**

다음 익명 블록은 현재 세션에서 메시지 송수신을 해제합니다.

```
BEGIN
DBMS_OUTPUT.DISABLE;
END;
```

### 7.2.3 ENABLE

ENABLE 프로시저는 메시지 버퍼에 대한 메시지 전송 및 메시지 버퍼에서 메시지 수신 기능을 활성화합니다. SERVEROUTPUT (TRUE) 실행도 내재적으로 ENABLE 프로시저를 수행합니다.

PUT, PUT_LINE 또는 NEW_LINE에서 보내진 메시지의 대상은 SERVEROUTPUT 상태에 따라 달라집니다.

● SERVEROUTPUT 마지막 상태가 "참 (true)"의 경우 메시지는 명령 라인 표준 출력으로 이동합니다.

● SERVEROUTPUT 마지막 상태가 "거짓 (false)"의 경우 메시지는 메시지 버퍼로 이동합니다.

ENABLE *[(length INTEGER)*

**매개 변수**

*length*

메시지 버퍼바이트의 최대 길이. *length* 2000미만이 지정되면 버퍼 크기는 2000으로 설정됩니다.

**예**

다음 익명 블록은 메시지를 사용합니다. SERVEROUTPUT (TRUE)을 지정하고 표준 출력합니다.

```
BEGIN
DBMS_OUTPUT.ENABLE;
DBMS_OUTPUT.SERVEROUTPUT (TRUE);
DBMS_OUTPUT.PUT_LINE ( 'Messages enabled');
END;

Messages enabled
```

간단하게 SERVEROUTPUT (TRUE)을 사용하면 같은 결과를 얻을 수 있습니다.

```
BEGIN
DBMS_OUTPUT.SERVEROUTPUT (TRUE);
DBMS_OUTPUT.PUT_LINE ('Messages enabled');
END;

Messages enabled
```

다음 익명 블록은 메시지를 사용합니다. 그러나 SERVEROUTPUT (FALSE) 지정 메시지를 메시지 버퍼에 맞춥니다.

```
BEGIN
DBMS_OUTPUT.ENABLE;
DBMS_OUTPUT.SERVEROUTPUT (FALSE);
DBMS_OUTPUT.PUT_LINE ('Message sent to buffer');
END;
```

### 7.2.4 GET_LINE

GET_LINE 프로시저는 메시지 버퍼 텍스트 라인을 회복하는 기능을 제공합니다. 라인 종료 문자 시퀀스로 종료된 텍스트는 회복됩니다. 그것은 PUT_LINE 또는 일련의 NEW_LINE 호출 후에 PUT 호출을 사용하여 생성된 완전한 라인입니다.

GET_LINE *(line OUT VARCHAR2, status OUT INTEGER)*

**매개 변수**

*line*

메시지 버퍼에서 텍스트 라인을 받는 변수이다.

*status*

메시지 버퍼 라인이 반환되면 1, 상환 라인이 없다면 0. 이 값은 Oracle 호환이 아니므로 유의하시기 바랍니다. Oracle는 메시지 버퍼 라인이 반환되면 0, 상환 라인이 없다면 1을 반환합니다.

**예**

다음 익명 블록은 각 라인의 쉼표로 구분된 문자열을 메시지 버퍼에 emp 테이블을 내보냅니다.

```
DECLARE
v_emprec VARCHAR2 (120);
CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
DBMS_OUTPUT.SERVEROUTPUT (FALSE);
FOR i IN emp_cur LOOP
v_emprec : = i.empno | | ','| | i.ename | | ','| | i.job | | ','| |
NVL (LTRIM (TO_CHAR (i.mgr, '9999 ')),'') | | ','| | i.hiredate | |
','| | i.sal | | ','| |
NVL (LTRIM (TO_CHAR (i.comm, '9990 .99 ')),'') | | ','| | i.deptno;
DBMS_OUTPUT.PUT_LINE (v_emprec);
END LOOP;
END;
```

다음 익명 블록은 메시지 버퍼를 읽고, 앞의 예제를 통해 명령을 내보낸 라인을 표시합니다.

```
DECLARE
v_line VARCHAR2 (100);
v_status INTEGER : = 1;
BEGIN
DBMS_OUTPUT.SERVEROUTPUT (TRUE);
WHILE v_status = 1 LOOP
DBMS_OUTPUT.GET_LINE (v_line, v_status);
DBMS_OUTPUT.PUT_LINE (v_line);
END LOOP;
END;

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 00:00:00950.00, 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
7369, SMITH, CLERK ,7902,17 - DEC - 80 00:00:00800.00, 20
```

### 7.2.5 GET_LINES

GET_LINES 프로시저는 집합에 메시지 버퍼로부터 하나 이상의 텍스트 라인의 회복기능을 제공합니다. 라인 종료 문자 시퀀스에 의해 종료된 텍스트만이 회복됩니다. 그것은 PUT_LINE 또는 일련의 NEW_LINE 호출 후에 PUT 호출을 사용하여 생성된 완전한 라인입니다.

GET_LINES(*lines* OUT CHARARR, *numlines* IN OUT INTEGER)

**매개변수**

*lines*

메시지 버퍼로부터 텍스트의 라인을 받는 테이블. 라인의 설명으로 CHARARR 를 보세요.

*numlines IN*

메시지 버퍼로부터 회수된 라인의 수

*numlines OUT*

메시지 버퍼로부터 회수된 실제 라인의 수. 입력 값이 numlines의 출력 값보다 더 적을 경우, 메시지 버퍼에 남은 라인은 더 이상 없습니다.

실제 라인의 수는 메시지 버퍼로부터 회복됩니다. *Numlines*의 출력값이 입력값보다 작은 경우, 더 이상 메시지 버퍼에 왼쪽 라인은 없습니다.

**예**

다음의 예는, 정렬 내의 메시지 버퍼에 대체된 emp 테이블의 행을 저장하는 GET_LINES 프로시저를 사용합니다.

```
EXEC DBMS_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE
   v_emprec       VARCHAR2(120);
   CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
   DBMS_OUTPUT.ENABLE;
   FOR i IN emp_cur LOOP
      v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||
         NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
         ',' || i.sal || ',' ||
         NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
      DBMS_OUTPUT.PUT_LINE(v_emprec);
   END LOOP;
END;

DECLARE
   v_lines        DBMS_OUTPUT.CHARARR;
   v_numlines      INTEGER := 14;
   v_status       INTEGER := 0;
BEGIN
   DBMS_OUTPUT.GET_LINES(v_lines,v_numlines);
   FOR i IN 1..v_numlines LOOP
      INSERT INTO messages VALUES(v_numlines, v_lines(i));
   END LOOP;
END;

SELECT msg FROM messages;

                         msg
------------------------------------------------------------------
 7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
 7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30
 7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
 7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20
 7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
 7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
 7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10
 7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
```

```
 7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10
 7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
 7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
 7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
 7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
 7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
(14 rows)
```

### 7.2.6 NEW_LINE

NEW_LINE 프로시저는 메시지 버퍼의 라인 종료 문자 시퀀스에 기록합니다.

NEW_LINE

### 7.2.7 PUT

PUT 프로시저는 메시지 버퍼에 문자열을 기록합니다. 문자열의 라인 종료 문자 시퀀스에 기록됩니다. 라인 종료 문자 시퀀스에 추가하기 위해 NEW_LINE 프로시저를 사용합니다.

PUT *(string VARCHAR2)*

**매개 변수**

*string*

메시지 버퍼에 기록되는 텍스트입니다.

**예**

다음 예제는 emp 테이블에서 직원의 쉼표로 구분된 목록을 보기 위해 PUT 프로시저를 사용합니다.

```
DECLARE
CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
FOR i IN emp_cur LOOP
DBMS_OUTPUT.PUT (i.empno);
DBMS_OUTPUT.PUT (',');
DBMS_OUTPUT.PUT (i.ename);
DBMS_OUTPUT.PUT (',');
DBMS_OUTPUT.PUT (i.job);
DBMS_OUTPUT.PUT (',');
DBMS_OUTPUT.PUT (i.mgr);
DBMS_OUTPUT.PUT (',');
DBMS_OUTPUT.PUT (i.hiredate);
DBMS_OUTPUT.PUT (',');
DBMS_OUTPUT.PUT (i.sal);
DBMS_OUTPUT.PUT (',');
DBMS_OUTPUT.PUT (i.comm);
DBMS_OUTPUT.PUT (',');
DBMS_OUTPUT.PUT (i.deptno);
DBMS_OUTPUT.NEW_LINE;
END LOOP;
END;

7369, SMITH, CLERK ,7902,17 - DEC - 80 00:00:00800.00, 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 00:00:00950.00, 30
7902, FORD, ANALYST ,7566,03 - DEC - 81 00:00:00,3000.00, 20
7934, MILLER, CLERK ,7782,23 - JAN - 82 00:00:00,1300.00, 10
```

### 7.2.8 PUT_LINE

PUT_LINE 프로시저를 라인 종료 문자 시퀀스가 포함된 메시지 버퍼, 1 라인을 씁니다.

PUT_LINE *(line VARCHAR2)*

**매개 변수**

*line*

메시지 버퍼에 기록되는 텍스트입니다.

**예**

다음 예제는 emp 테이블에서 직원의 쉼표로 구분된 목록을 보기 위해 PUT_LINE 프로시저를 사용합니다.

```
DECLARE
v_emprec VARCHAR2 (120);
CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
FOR i IN emp_cur LOOP
v_emprec : = i.empno | | ','| | i.ename | | ','| | i.job | | ','| |
NVL (LTRIM (TO_CHAR (i.mgr, '9999 ')),'') | | ','| | i.hiredate | |
','| | i.sal | | ','| |
NVL (LTRIM (TO_CHAR (i.comm, '9990 .99 ')),'') | | ','| | i.deptno;
DBMS_OUTPUT.PUT_LINE (v_emprec);
END LOOP;
END;

7369, SMITH, CLERK ,7902,17 - DEC - 80 00:00:00800.00, 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 00:00:00950.00, 30
7902, FORD, ANALYST ,7566,03 - DEC - 81 00:00:00,3000.00, 20
7934, MILLER, CLERK ,7782,23 - JAN - 82 00:00:00,1300.00, 10
```

### 7.2.9 SERVEROUTPUT

SERVEROUTPUT 프로시저는 명령 라인의 표준 출력 또는 메시지 버퍼로 메시지를 보내는 기능을

제공합니다. SERVEROUTPUT (TRUE) 설정도 ENABLE의 내재적인 실행을 합니다.
SERVEROUTPUT의 기본값 설정은 실행에 의존합니다.
예를 들면, Oracle SQL * Plus에서, SERVEROUTPUT (FALSE) 은 초기값입니다. PSQL에서는 SERVEROUTPUT (TRUE)이 초기값입니다. 또한 Oracle SQL * Plus는, Postgres Plus Advanced Server에 구현된 것처럼, 저장된 프로시저가 아닌, SQL * Plus의 SET 명령을 사용하여 설정이 제어되는 것을 기억하십시오.

SERVEROUTPUT *(stdout BOOLEAN)*

**매개 변수**

*stdout*

PUT, PUT_LINE과 NEW_LINE 명령이 명령 라인의 표준 출력에 텍스트를 보내기 위해 "참 (true)"으로 설정합니다. 메시지 버퍼에 텍스트를 보내려면, "거짓 (false)"으로 설정한다.

**예**

다음 익명 블록은 첫 번째 메시지를 명령 라인에 보내고, 두 번째 메시지를 메시지 버퍼에 전송합니다.

```
BEGIN
DBMS_OUTPUT.SERVEROUTPUT (TRUE);
DBMS_OUTPUT.PUT_LINE ( 'This message goes to the command line');
DBMS_OUTPUT.SERVEROUTPUT (FALSE);
DBMS_OUTPUT.PUT_LINE ( 'This message goes to the message buffer');
END;

This message goes to the command line
```

같은 세션에서 다음과 익명의 블록이 실행되는 경우, 위 예제의 메시지 버퍼에 저장된 메시지는 플래시되고 새로운 메시지와 마찬가지로 명령 라인에 표시됩니다.

```
BEGIN
DBMS_OUTPUT.SERVEROUTPUT (TRUE);
DBMS_OUTPUT.PUT_LINE ( 'Flush messages from the buffer');
END;

This message goes to the message buffer
Flush messages from the buffer
```

### 7.3 DBMS_PIPE

DBMS_PIPE 패키지는 파이프를 통해 동일한 데이터베이스 클러스터에 연결된 세션 내에서 또는 세션 사이에 메시지를 보내는 기능을 제공합니다.
DBMS_PIPE 패키지에서 사용할 수 있는 함수와 프로시저는 다음 표와 같다.

**표 7-3 DBMS_PIPE 함수 / 프로시저**

| 함수 / 프로시저 | 반환 형식 | 설명 |
|---|---|---|
| CREATE_PIPE(*pipename* [*maxpipesize* ] | INTEGER | *Private0*/ 참인 경우 개인 파이프(기본값), |

| | | |
|---|---|---|
| [,*private* ]) | | *Private*이 거짓인 경우 공용 파이프를 명시적으로 생성한다. |
| NEXT_ITEM_TYPE | INTEGER | 수신 메시지의 다음 항목의 데이터 유형을 결정한다 |
| PACK_MESSAGE *(dataitem)* | n / a | 세션의 로컬 메시지 버퍼에 *dataitem*을 사용한다. |
| PURGE (*pipename*) | n / a | 지정된 파이프에서 받지 못한 메시지를 지운다. |
| RECEIVE_MESSAGE(*pipename* [,*timeout* ]) | INTEGER | 지정된 파이프에서 메시지를 검색한다. |
| REMOVE_PIPE(*pipename*) | INTEGER | 명시적으로 만든 파이프 제거한다. |
| RESET_BUFFER | n / a | 로컬 메시지 버퍼를 다시 설정한다. |
| SEND_MESSAGE(*pipename* [,*timeout* ] [,*maxpipesize* ]) | INTEGER | 파이프에 메세지를 보낸다. |
| UNIQUE_SESSION_NAME | VARCHAR2 | 고유 세션 이름을 가져온다. |
| UNPACK_MESSAGE(*item* OUT) | n / a | 호환 가능한 변수, *item* 의 메시지에서 다음 데이터 항목을 회복한다 |

파이프는 암시 혹은 명시로 분류됩니다. CREATE_PIPE 함수가 생성한 파이프 이름이 먼저 만들어진 경우 암시적 파이프가 만들어집니다. 예를 들면, 존재하지 않는 파이프 이름을 사용하여 SEND_MESSAGE 함수를 실행하면 새 암시적 파이프는 그 명칭으로 만들어집니다. 명시적 파이프는 첫 번째 매개 변수가 새로운 파이프 이름을 지정하는 CREATE_PIPE 함수를 사용하여 만들어집니다.

파이프는 공개 또는 비공개로 분류됩니다. 개인 파이프는 파이프를 만들기 사용자만 접근할 수 있습니다. 다른 사용자에 의해 만들어진 파이프는 슈퍼유저조차 접근할 수 없습니다. DBMS_PIPE 패키지를 방문한 사용자는 공용 파이프에 접근할 수 있습니다.

공용 파이프는 CREATE_PIPE 함수의 세 번째 매개 변수가 "거짓 (false)"으로 설정된 경우에만 만들어집니다. CREATE_PIPE 함수는 세 번째 매개 변수를 "참 (true)"으로 설정되거나 생략될 경우에는 개인 파이프를 만들 수 있습니다. 전체 암시적 파이프는 개인 전용입니다.

메시지의 개별 데이터 항목 또는 "라인"은, 현재 세션에 유일한 로컬 메시지 버퍼에 처음으로 작성됩니다. PACK_MESSAGE 프로시저는 세션 로컬 메시지 버퍼 메시지를 작성합니다. SEND_MESSAGE 함수는 파이프를 통해 메시지를 보내는 데 사용됩니다.

메시지 수신은 역방향 연산이 포함되고 있습니다. RECEIVE_MESSAGE 함수는 지정된 파이프에서 메시지를 받는 데 사용됩니다. 메시지는 세션의 로컬 메시지 버퍼에 기록됩니다. UNPACK_MESSAGE 프로시저는 메시지 버퍼의 프로그램 변수에서 메시지 데이터 항목으로의 전송에 사용됩니다. 파이프에 여러 메시지가 포함되어있는 경우, RECEIVE_MESSAGE은 *FIFO* (First In -

First Out) 순으로 메시지를 받고 있습니다.
각 세션은 PACK_MESSAGE 프로시저로 작성된 메시지 및 RECEIVE_MESSAGE 함수에 의해 회복된 개별 메시지 버퍼를 유지합니다. 따라서, 메시지는 모두 같은 세션에서 회복될 수 있습니다. 그러나 연속 RECEIVE_MESSAGE 호출이 실행되면, 마지막 RECEIVE_MESSAGE 호출에서 메시지만 로컬 메시지 버퍼가 유지됩니다.

### 7.3.1 CREATE_PIPE

CREATE_PIPE 함수는 지정된 이름으로 명시적 공용 파이프 또는 명시적 개인 파이프를 생성합니다.

*status INTEGER CREATE_PIPE (name VARCHAR2* [, *length INTEGER]*
[, *private BOOLEAN)*

**매개 변수**

*name*

파이프의 명칭.

*length*

바이트 매개변수로 파이프의 최대 허용량. 기본값은 8192 바이트.

*private*

"거짓 (false)" 지정 공용 파이프를 생성.

"참 (true)" 지정 전용 파이프를 생성. 이것은 기본값입니다.

*status*

연산에 의한 반환 상태 코드. 0은 생성의 성공을 나타냅니다.

**예**

개인 파이프 messages 생성:

```
DECLARE
v_status INTEGER;
BEGIN
v_status : = DBMS_PIPE.CREATE_PIPE ( 'messages');
DBMS_OUTPUT.PUT_LINE ( 'CREATE_PIPE status :'| | v_status);
END;

CREATE_PIPE status : 0
```

공용 파이프 mailbox 생성:

```
DECLARE
v_status INTEGER;
BEGIN
v_status : = DBMS_PIPE.CREATE_PIPE ( 'mailbox', 8192, FALSE);
DBMS_OUTPUT.PUT_LINE ( 'CREATE_PIPE status :'| | v_status);
END;

CREATE_PIPE status : 0
```

### 7.3.2 NEXT_ITEM_TYPE

NEXT_ITEM_TYPE 함수는 세션 로컬 메시지 버퍼에 회수된 메시지 중 데이터 유형의 다음 데이터 항목을 식별하는 정수 코드를 반환합니다. 항목이 각각 UNPACK_MESSAGE 프로시저에서 로컬 메시지 버퍼로 이동하는 동시에, NEXT_ITEM_TYPE 함수는 다음 이용 가능한 항목을 위해 데이터 유형 코드를 반환합니다. 항목이 메시지에 더 이상 남아 있지 않으면 0 코드가 반환됩니다.

*typecode INTEGER NEXT_ITEM_TYPE*

**매개 변수**

*typecode*

표 7-4에서 보여지는 것처럼 다음 데이터 항목의 데이터 형식을 식별하는 고유 코드.

**표 7-4 NEXT_ITEM_TYPE 데이터 형식 코드**

| 유형 코드 | 데이터 형식 |
|---|---|
| 0 | 더 이상 데이터 항목이 없는 |
| 9 | NUMBER |
| 11 | VARCHAR2 |
| 13 | DATE |
| 23 | RAW |

*** 참고**

**표의 유형 코드 목록은 Oracle과 호환성이 없습니다. Oracle 다른 번호 순서를 데이터 유형에 할당하고 있습니다.**

**예**

다음은 NUMBER 항목, VARCHAR2 항목, DATE 항목, RAW 항목을 포함하는 파이프를 보여줍니다. 두 번째 익명 블록은 각 항목의 유형 코드를 보여주는 NEXT_ITEM_TYPE 함수를 사용합니다.

```
DECLARE
v_number NUMBER : = 123;
v_varchar VARCHAR2 (20) : = 'Character data';
v_date DATE : = SYSDATE;
v_raw RAW (4) : = '21222324 ';
v_status INTEGER;
BEGIN
DBMS_PIPE.PACK_MESSAGE (v_number);
DBMS_PIPE.PACK_MESSAGE (v_varchar);
DBMS_PIPE.PACK_MESSAGE (v_date);
DBMS_PIPE.PACK_MESSAGE (v_raw);
v_status : = DBMS_PIPE.SEND_MESSAGE ( 'datatypes');
DBMS_OUTPUT.PUT_LINE ( 'SEND_MESSAGE status :'| | v_status);
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'SQLERRM :'| | SQLERRM);
DBMS_OUTPUT.PUT_LINE ( 'SQLCODE :'| | SQLCODE);
END;

SEND_MESSAGE status : 0

DECLARE
v_number NUMBER;
```

```
v_varchar VARCHAR2 (20);
v_date DATE;
v_timestamp TIMESTAMP;
v_raw RAW (4);
v_status INTEGER;
BEGIN
v_status : = DBMS_PIPE.RECEIVE_MESSAGE ( 'datatypes');
DBMS_OUTPUT.PUT_LINE ( 'RECEIVE_MESSAGE status :'| | v_status);
DBMS_OUTPUT.PUT_LINE ('----------------------------------');

v_status : = DBMS_PIPE.NEXT_ITEM_TYPE;
DBMS_OUTPUT.PUT_LINE ( 'NEXT_ITEM_TYPE :'| | v_status);
DBMS_PIPE.UNPACK_MESSAGE (v_number);
DBMS_OUTPUT.PUT_LINE ( 'NUMBER Item :'| | v_number);
DBMS_OUTPUT.PUT_LINE ('----------------------------------');

v_status : = DBMS_PIPE.NEXT_ITEM_TYPE;
DBMS_OUTPUT.PUT_LINE ( 'NEXT_ITEM_TYPE :'| | v_status);
DBMS_PIPE.UNPACK_MESSAGE (v_varchar);
DBMS_OUTPUT.PUT_LINE ( 'VARCHAR2 Item :'| | v_varchar);
DBMS_OUTPUT.PUT_LINE ('----------------------------------');

v_status : = DBMS_PIPE.NEXT_ITEM_TYPE;
DBMS_OUTPUT.PUT_LINE ( 'NEXT_ITEM_TYPE :'| | v_status);
DBMS_PIPE.UNPACK_MESSAGE (v_date);
DBMS_OUTPUT.PUT_LINE ( 'DATE Item :'| | v_date);
DBMS_OUTPUT.PUT_LINE ('----------------------------------');

v_status : = DBMS_PIPE.NEXT_ITEM_TYPE;
DBMS_OUTPUT.PUT_LINE ( 'NEXT_ITEM_TYPE :'| | v_status);
DBMS_PIPE.UNPACK_MESSAGE (v_raw);
DBMS_OUTPUT.PUT_LINE ( 'RAW Item :'| | v_raw);
DBMS_OUTPUT.PUT_LINE ('----------------------------------');

v_status : = DBMS_PIPE.NEXT_ITEM_TYPE;
DBMS_OUTPUT.PUT_LINE ( 'NEXT_ITEM_TYPE :'| | v_status);
DBMS_OUTPUT.PUT_LINE ('----------------------------------');
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'SQLERRM :'| | SQLERRM);
DBMS_OUTPUT.PUT_LINE ( 'SQLCODE :'| | SQLCODE);
END;

RECEIVE_MESSAGE status : 0
----------------------------------
NEXT_ITEM_TYPE : 9
NUMBER Item : 123
----------------------------------
NEXT_ITEM_TYPE : 11
VARCHAR2 Item : Character data
----------------------------------
NEXT_ITEM_TYPE : 13
DATE Item : 02 - OCT - 07 11:11:43
----------------------------------
NEXT_ITEM_TYPE : 23
RAW Item : 21222324
----------------------------------
NEXT_ITEM_TYPE : 0
```

### 7.3.3 PACK_MESSAGE

PACK_MESSAGE 프로시저는 세션 로컬 메시지 버퍼 데이터 항목을 위치시킵니다. PACK_MESSAGE 은 SEND_MESSAGE을 호출하기 전에 적어도 한 번 실행됩니다.

PACK_MESSAGE *(dataitem* (DATE | NUMBER | VARCHAR2 | RAW))

RECEIVE_MESSAGE 호출을 사용해 메시지가 회복되면 데이터 항목을 얻기 위해 UNPACK_MESSAGE 프로시저를 이용하세요.

**매개 변수**

*dataitem*

적절한 매개 변수의 데이터 유형을 나타내는 평가. 세션 로컬 메시지 버퍼에 값이 추가됩니다.

### 7.3.4 PURGE

PURGE 프로시저는 지정된 암시적 파이프로부터 받지 못한 메시지를 삭제합니다.

PURGE *(name VARCHAR2)*

명시적 파이프를 제거하려면 REMOVE_PIPE 함수를 사용하십시오.

**매개 변수**

*name*

파이프의 명칭.

**예**

두 개의 메시지가 파이프로 전송됩니다.

```
DECLARE
v_status INTEGER;
BEGIN
DBMS_PIPE.PACK_MESSAGE ( 'Message # 1');
v_status : = DBMS_PIPE.SEND_MESSAGE ( 'pipe');
DBMS_OUTPUT.PUT_LINE ( 'SEND_MESSAGE status :'| | v_status);

DBMS_PIPE.PACK_MESSAGE ( 'Message # 2');
v_status : = DBMS_PIPE.SEND_MESSAGE ( 'pipe');
DBMS_OUTPUT.PUT_LINE ( 'SEND_MESSAGE status :'| | v_status);
END;

SEND_MESSAGE status : 0
SEND_MESSAGE status : 0
```

첫 메시지를 받고 해독합니다 :

```
DECLARE
v_item VARCHAR2 (80);
v_status INTEGER;
BEGIN
v_status : = DBMS_PIPE.RECEIVE_MESSAGE ( 'pipe', 1);
DBMS_OUTPUT.PUT_LINE ( 'RECEIVE_MESSAGE status :'| | v_status);
DBMS_PIPE.UNPACK_MESSAGE (v_item);
DBMS_OUTPUT.PUT_LINE ( 'Item :'| | v_item);
END;

RECEIVE_MESSAGE status : 0
Item : Message # 1
```

파이프를 삭제합니다.

```
EXEC DBMS_PIPE.PURGE ( 'pipe');
```

다음 메시지 회수를 시도합니다. RECEIVE_MESSAGE 호출을 사용할 수 있는 메시지가 없어, 시간 초과했음을 나타내는 상태 코드 1을 반환합니다.

```
DECLARE
v_item VARCHAR2 (80);
v_status INTEGER;
BEGIN
v_status : = DBMS_PIPE.RECEIVE_MESSAGE ( 'pipe', 1);
DBMS_OUTPUT.PUT_LINE ( 'RECEIVE_MESSAGE status :'| | v_status);
END;

RECEIVE_MESSAGE status : 1
```

### 7.3.5 RECEIVE_MESSAGE

RECEIVE_MESSAGE 함수는 지정된 파이프에서 메시지를 가져옵니다.

*status INTEGER RECEIVE_MESSAGE (name VARCHAR2*

[,*timeout INTEGER)*

**매개 변수**

*name*

파이프의 명칭.

*timeout*

대기 시간 (초). 기본값은 86400000 (1000 일).

*status*

작업에서 반환하는 상태 코드입니다.

가능한 상태 코드 :

**Table 7-5 RECEIVE_MESSAGE 상태 코드**

| 상태 코드 | 설명 |
|---|---|
| 0 | 성공 |
| 1 | 시간 초과 |
| 2 | 버퍼에 메시지가 너무 큼 |

### 7.3.6 REMOVE_PIPE

REMOVE_PIPE 함수는 명시적 개인 파이프 또는 명시적 공용 파이프를 제거합니다.

*status INTEGER REMOVE_PIPE (name VARCHAR2)*

명시적으로 만든 파이프 (CREATE_PIPE 함수에 의해 생성된 파이프)를 제거하기 위해

REMOVE_PIPE 함수를 사용합니다.

**매개 변수**

*name*

파이프의 명칭.

*status*

작업에서 반환하는 상태 코드입니다.

지정된 파이프가 존재하지 않고, 0의 상태 코드가 반환됩니다.

**예**

두 개의 메시지가 파이프로 전송됩니다.

```
DECLARE
 v_status INTEGER;
 BEGIN
 v_status : = DBMS_PIPE.CREATE_PIPE ( 'pipe');
 DBMS_OUTPUT.PUT_LINE ( 'CREATE_PIPE status :'| | v_status);

 DBMS_PIPE.PACK_MESSAGE ( 'Message # 1');
 v_status : = DBMS_PIPE.SEND_MESSAGE ( 'pipe');
 DBMS_OUTPUT.PUT_LINE ( 'SEND_MESSAGE status :'| | v_status);

 DBMS_PIPE.PACK_MESSAGE ( 'Message # 2');
 v_status : = DBMS_PIPE.SEND_MESSAGE ( 'pipe');
 DBMS_OUTPUT.PUT_LINE ( 'SEND_MESSAGE status :'| | v_status);
 END;

 CREATE_PIPE status : 0
 SEND_MESSAGE status : 0
 SEND_MESSAGE status : 0
```

첫 메시지를 받고 해독합니다. :

```
 DECLARE
 v_item VARCHAR2 (80);
 v_status INTEGER;
 BEGIN
 v_status : = DBMS_PIPE.RECEIVE_MESSAGE ( 'pipe', 1);
 DBMS_OUTPUT.PUT_LINE ( 'RECEIVE_MESSAGE status :'| | v_status);
 DBMS_PIPE.UNPACK_MESSAGE (v_item);
 DBMS_OUTPUT.PUT_LINE ( 'Item :'| | v_item);
 END;

 RECEIVE_MESSAGE status : 0
 Item : Message # 1
```

파이프를 제거합니다.

```
 SELECT DBMS_PIPE.REMOVE_PIPE ( 'pipe') FROM DUAL;

 remove_pipe
 -------------
 0
```

```
(1 row)
```

다음 메시지 회수를 시도합니다. RECEIVE_MESSAGE 호출은 파이프를 삭제한 것에 따른 시간 변경을 나타내는 상태 코드 1을 반환합니다.

```
DECLARE
v_item VARCHAR2 (80);
v_status INTEGER;
BEGIN
v_status : = DBMS_PIPE.RECEIVE_MESSAGE ( 'pipe', 1);
DBMS_OUTPUT.PUT_LINE ( 'RECEIVE_MESSAGE status :'| | v_status);
END;

RECEIVE_MESSAGE status : 1
```

### 7.3.7 RESET_BUFFER

RESET_BUFFER 프로시저는 세션의 로컬 메시지 버퍼 "포인터 (pointer)"를 버퍼의 처음으로 돌려보내도록 다시 설정합니다. 이것은 RESET_BUFFER 호출보다 이전에 존재하는 메시지 버퍼의 데이터 항목에 덮어쓰도록, 다음 PACK_MESSAGE 호출하는 결과의 원인이 되는 효과를 가집니다.

RESET_BUFFER

**예**

John에게 메시지가 로컬 메시지 버퍼에 쓰이고 있습니다. 그것은 RESET_BUFFER를 호출하여 Bob에게 메시지로 대체됩니다. 메시지는 파이프로 전송됩니다.

```
DECLARE
v_status INTEGER;
BEGIN
DBMS_PIPE.PACK_MESSAGE ( 'Hi, John');
DBMS_PIPE.PACK_MESSAGE ( 'Can you attend a meeting at 3:00, today?');
DBMS_PIPE.PACK_MESSAGE ( 'If not, is tomorrow at 8:30 ok with you?');
DBMS_PIPE.RESET_BUFFER;
DBMS_PIPE.PACK_MESSAGE ( 'Hi, Bob');
DBMS_PIPE.PACK_MESSAGE ( 'Can you attend a meeting at 9:30, tomorrow?');
v_status : = DBMS_PIPE.SEND_MESSAGE ( 'pipe');
DBMS_OUTPUT.PUT_LINE ( 'SEND_MESSAGE status :'| | v_status);
END;

SEND_MESSAGE status : 0
```

받은 메시지에 Bob에게 보내는 메시지가 있습니다.

```
DECLARE
v_item VARCHAR2 (80);
v_status INTEGER;
BEGIN
v_status : = DBMS_PIPE.RECEIVE_MESSAGE ( 'pipe', 1);
DBMS_OUTPUT.PUT_LINE ( 'RECEIVE_MESSAGE status :'| | v_status);
DBMS_PIPE.UNPACK_MESSAGE (v_item);
DBMS_OUTPUT.PUT_LINE ( 'Item :'| | v_item);
DBMS_PIPE.UNPACK_MESSAGE (v_item);
DBMS_OUTPUT.PUT_LINE ( 'Item :'| | v_item);
```

```
END;

RECEIVE_MESSAGE status : 0
Item : Hi, Bob
Item : Can you attend a meeting at 9:30, tomorrow?
```

### 7.3.8 SEND_MESSAGE

SEND_MESSAGE 함수는 세션의 로컬 메시지 버퍼 지정된 파이프에 메시지를 보냅니다.

*status SEND_MESSAGE (name VARCHAR2 ', timeout INTEGER]*

*[, length INTEGER)*

**매개 변수**

*name*

파이프의 명칭.

*timeout*

대기 시간 (초). 기본값은 86400000 (1000 일).

*length*

파이프 바이트의 용량. 기본값은 8192 바이트.

*status*

작업에서 반환하는 상태 코드입니다.

가능한 상태 코드 :

**Table 7-6 SEND_MESSAGE 상태 코드**

| 상태 코드 | 설명 |
|---|---|
| 0 | 성공 |
| 1 | 시간 제한 |
| 3 | 함수의 붕괴 |

7.3.9 UNIQUE_SESSION_NAME

UNIQUE_SESSION_NAME 함수는 이름 (현재 세션과 관련된)을 반환합니다.

*name VARCHAR2 UNIQUE_SESSION_NAME*

**매개 변수**

*name*

독특한 세션 이름.

**Examples**

다음 익명 블록은 독특한 세션 이름을 회수하고 표시합니다.

```
DECLARE
v_session VARCHAR2 (30);
```

```
BEGIN
v_session : = DBMS_PIPE.UNIQUE_SESSION_NAME;
DBMS_OUTPUT.PUT_LINE ( 'Session Name :'| | v_session);
END;

Session Name : PG $ PIPE $ 5 $ 2752
```

7.3.10 UNPACK_MESSAGE

UNPACK_MESSAGE 프로시저는 로컬 루멧세지 버퍼에서 지정된 프로그램 변수까지 메시지의 데이터 항목을 복사합니다. 메시지는 UNPACK_MESSAGE를 사용하기 전에 RECEIVE_MESSAGE 함수 로컬 메시지 버퍼에 배치됩니다.

UNPACK_MESSAGE *(dataitem OUT* (DATE | NUMBER | VARCHAR2 | RAW))

**매개 변수**

*dataitem*

로컬 메시지 버퍼에서 데이터 항목을 받는 호환 가능한 변수.

**7.3.11 종합적인 예**

다음 예제에서는 파이프를 "사서함"으로 사용합니다. 프로시저는 사서함을 생성하고, 여러 항목의 메시지를 사서함 (3 항목까지)에 추가합니다. 패키지 이름 mailbox에 있는 사서함의 전체 내용을 볼 수 있습니다.

```
CREATE OR REPLACE PACKAGE mailbox
IS
PROCEDURE create_mailbox;
PROCEDURE add_message (
p_mailbox VARCHAR2,
p_item_1 VARCHAR2,
p_item_2 VARCHAR2 DEFAULT 'END',
p_item_3 VARCHAR2 DEFAULT 'END'
);
PROCEDURE empty_mailbox (
p_mailbox VARCHAR2,
p_waittime INTEGER DEFAULT 10
);
END mailbox;

CREATE OR REPLACE PACKAGE BODY mailbox
IS
PROCEDURE create_mailbox
IS
v_mailbox VARCHAR2 (30);
v_status INTEGER;
BEGIN
v_mailbox : = DBMS_PIPE.UNIQUE_SESSION_NAME;
v_status : = DBMS_PIPE.CREATE_PIPE (v_mailbox, 1000, FALSE);
IF v_status = 0 THEN
DBMS_OUTPUT.PUT_LINE ( 'Created mailbox :'| | v_mailbox);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'CREATE_PIPE failed - status :'| |
v_status);
END IF;
```

```
END create_mailbox;

PROCEDURE add_message (
p_mailbox VARCHAR2,
p_item_1 VARCHAR2,
p_item_2 VARCHAR2 DEFAULT 'END',
p_item_3 VARCHAR2 DEFAULT 'END'
)
IS
v_item_cnt INTEGER : = 0;
v_status INTEGER;
BEGIN
DBMS_PIPE.PACK_MESSAGE (p_item_1);
v_item_cnt : = 1;
IF p_item_2! = 'END'THEN
DBMS_PIPE.PACK_MESSAGE (p_item_2);
v_item_cnt : = v_item_cnt + 1;
END IF;
IF p_item_3! = 'END'THEN
DBMS_PIPE.PACK_MESSAGE (p_item_3);
v_item_cnt : = v_item_cnt + 1;
END IF;
v_status : = DBMS_PIPE.SEND_MESSAGE (p_mailbox);
IF v_status = 0 THEN
DBMS_OUTPUT.PUT_LINE ( 'Added message with'| | v_item_cnt | |
'item (s) to mailbox'| | p_mailbox);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'SEND_MESSAGE in add_message failed -'| |
'status :'| | v_status);
END IF;
END add_message;

PROCEDURE empty_mailbox (
p_mailbox VARCHAR2,
p_waittime INTEGER DEFAULT 10
)
IS
v_msgno INTEGER DEFAULT 0;
v_itemno INTEGER DEFAULT 0;
v_item VARCHAR2 (100);
v_status INTEGER;
BEGIN
v_status : = DBMS_PIPE.RECEIVE_MESSAGE (p_mailbox, p_waittime);
WHILE v_status = 0 LOOP
v_msgno : = v_msgno + 1;
DBMS_OUTPUT.PUT_LINE ('****** Start message # '| | v_msgno | |
'******');
BEGIN
LOOP
v_status : = DBMS_PIPE.NEXT_ITEM_TYPE;
EXIT WHEN v_status = 0;
DBMS_PIPE.UNPACK_MESSAGE (v_item);
v_itemno : = v_itemno + 1;
DBMS_OUTPUT.PUT_LINE ( 'Item #'| | v_itemno | | ':'| |
v_item);
END LOOP;
DBMS_OUTPUT.PUT_LINE ('******* End message # '| | v_msgno | |
'*******');
DBMS_OUTPUT.PUT_LINE ('*');
```

```
v_itemno : = 0;
v_status : = DBMS_PIPE.RECEIVE_MESSAGE (p_mailbox, 1);
END;
END LOOP;
DBMS_OUTPUT.PUT_LINE ( 'Number of messages received :'| | v_msgno);
v_status : = DBMS_PIPE.REMOVE_PIPE (p_mailbox);
IF v_status = 0 THEN
DBMS_OUTPUT.PUT_LINE ( 'Deleted mailbox'| | p_mailbox);
ELSE
DBMS_OUTPUT.PUT_LINE ( 'Could not delete mailbox - status :'
| | v_status);
END IF;
END empty_mailbox;
END mailbox;
```

다음은 mailbox에서 프로시저의 실행을 설명합니다. 첫 번째 프로시저에서는 UNIQUE_SESSION_NAME 함수에 의해 생성된 이름을 사용하여 공용 파이프를 만듭니다.

```
EXEC mailbox.create_mailbox;

 Created mailbox : PG $ PIPE $ 13 $ 3940
```

사서함 이름을 사용하고, mailbox 패키지와 DBMS_PIPE 패키지에 접근하는 동일한 데이터베이스 관리자는 메시지를 추가할 수 있습니다 :

```
EXEC mailbox.add_message ( 'PG $ PIPE $ 13 $ 3940', 'Hi, John', 'Can you
attend a meeting at 3:00, today ?','-- Mary');

Added message with 3 item (s) to mailbox PG $ PIPE $ 13 $ 3940

EXEC mailbox.add_message ( 'PG $ PIPE $ 13 $ 3940', 'Don''t forget to
submit your report', 'Thanks ,','-- Joe');

 Added message with 3 item (s) to mailbox PG $ PIPE $ 13 $ 3940
```

마지막으로, 사서함 내용은 비워둘 수 있습니다:

```
EXEC mailbox.empty_mailbox ( 'PG $ PIPE $ 13 $ 3940');

****** Start message # 1 ******
Item # 1 : Hi, John
Item # 2 : Can you attend a meeting at 3:00, today?
Item # 3 : - Mary
******* End message # 1 *******
*
****** Start message # 2 ******
Item # 1 : Don't forget to submit your report
Item # 2 : Thanks,
Item # 3 : Joe
******* End message # 2 *******
*
Number of messages received : 2
Deleted mailbox PG $ PIPE $ 13 $ 3940
```

### 7.4 UTL_FILE

UTL_FILE 패키지는 운영 체제의 파일 시스템에서 파일 읽기 및 쓰기 기능을 제공합니다. 비 슈퍼유저는 패키지의 함수와 프로시저를 사용하기 전에 슈퍼유저를 UTL_FILE 패키지의 EXECUTE 권한을 부여해야 합니다.

예를 들어, 다음 명령은 mary에 권한을 부여합니다:

```
GRANT EXECUTE ON PACKAGE SYS.UTL_FILE TO mary;
```

또한 운영 체제의 사용자 이름 enterprisedb는, UTL_FILE 함수나 프로시저를 사용하여 디렉토리와 파일에 접근하기 위해 적절한 읽기/쓰기 허가를 얻어야 합니다. 필요한 파일 권한이 적시에 없으면, UTL_FILE 함수나 프로시저 중 예외가 발생합니다.

파일 쓰기 및 읽기 작업은 파일 참조를 사용합니다. *file handle* 은 UTL_FILE 패키지의 공용 변수 UTL_FILE.FILE_TYPE 로 정의됩니다. FOPEN 함수 호출에서 반환되는 파일 핸들을 받으려면 FILE_TYPE 변수를 선언해야 합니다. 그리고, 파일 핸들은 이후의 모든 파일 작업에 사용됩니다.

파일 시스템의 디렉토리 검색은 CREATE DIRECTORY 명령을 사용하여 디렉토리에 할당된 별칭이나 디렉토리 이름을 사용합니다.

UTL_FILE 패키지에서 사용 가능한 프로시저와 함수를 다음 표에 열거합니다.

**Table 7-7 UTL_FILE 함수 / 프로시저**

| 함수 / 프로시저 | 반환 형식 | 설명 |
|---|---|---|
| FCLOSE(*file* IN OUT) | n/a | *File*에 의해 식별된 지정된 파일을 닫는다. |
| FCLOSE_ALL | n/a | 모든 열린 파일을 닫는다. |
| FCOPY(*location*, *filename*, *dest_dir*, *dest_file* [, *start_line* [, *end_line* ] ]) | n/a | 그리고 *Location*에 의해 파일에 식별된 디렉토리, *dest_file*, *dest_dir*, 시작 라인부터, *start_line*, *end_line*으로부터 *Filename*을 복사한다 |
| FFLUSH(*file*) | n/a | 버퍼의 데이터가 *file*에 의해 식별된 파일 내의 디스크에 기록되게 한다.. |
| FOPEN(*location*, *filename*, *open_mode* [,*max_linesize* ]) | FILE_TYPE | *location* 에 의해 식별된 디렉토리의 파일, *filename*을 연다. |
| FREMOVE(*location*, *filename*) | n/a | 파일 시스템으로부터 지정된 파일을 제거한다. |
| FRENAME(*location*, *filename*, *dest_dir*, *dest_file* [,*overwrite* ]) | n/a | 지정된 파일을 이름을 다시 짓는다. |
| GET_LINE(*file*, *buffer* OUT) | n/a | Reads a line of text into variable, *buffer*, from the file identified by *file*.<br>*file*로 식별된 파일로부터, 변수, *buffer*, 의 |

| | | |
|---|---|---|
| | | 텍스트 라인을 읽는다. |
| IS_OPEN(*file*) | BOOLEAN | 주어진 파일을 열 것인지 아닌지 결정한다. |
| NEW_LINE(*file* [, *lines* ]) | n/a | 파일에 라인 종료 문자 시퀀스를 작성한다. |
| PUT(*file*, *buffer*) | n/a | 주어진 파일에 *buffer*를 작성한다. PUT은 라인 종료 문자 시퀀스를 작성하지 않는다. |
| PUT_LINE(*file*, *buffer*) | n/a | 주어진 파일에 *buffer*를 작성한다 라인 종료 문자 시퀀스는 PUT_LINE 프로시저에 의해 더해진다. |
| PUTF(*file*, *format* [, *arg1* ] [, ...]) | n/a | 주어진 파일에 서식화된 문자열을 작성한다.<br>다섯 개 까지, 치환매개변수, *arg1,...arg5*는 *format*에 대체로 지정된다. |

### 7.4.1 FCLOSE

FCLOSE 프로시저는 오픈 파일을 닫습니다.

FCLOSE *(filetype FILE_TYPE)*

**매개 변수**

*filetype*

닫혀져야 할 파일의 파일 핸들을 포함하는 변수 형식 FILE_TYPE

### 7.4.2 FCLOSE_ALL

FLCLOSE_ALL 프로시저는 열려있는 모든 파일을 닫습니다. 닫아야 할 열린 파일이 없어도 프로시저 실행이 성공합니다.

FCLOSE_ALL

### 7.4.3 FCOPY

FCOPY 함수는 파일의 텍스트를 다른 파일에 복사합니다.

VOID FCOPY *(dirname VARCHAR2, filename VARCHAR2,*
*destdir VARCHAR2, destfile VARCHAR2*
[, *begin INTEGER* [, *end INTEGER])*

EnterpriseDB의 FCOPY은 Oracle와 호환되지 않는 것을 주의하십시오. Oracle에서는 FCOPY은 함

수가 아닌 프로시저로 구현됩니다.

**매개 변수**

*dirname*

복사할 파일을 포함하는 디렉토리, pg_catalog.edb_ dir.dirname에 저장된 디렉토리 이름.

*filename*

복사할 원본 파일 이름.

*destdir*

파일이 복사되는 디렉토리, pg_catalog.edb_dir.dirname에 저장된 디렉토리 이름.

*destfile*

대상 파일 이름.

*begin*

복사가 시작되는 소스 파일의 라인 번호. 기본값은 1.

*end*

복사할 원본 파일의 마지막 라인 번호. 생략 또는 해제된 경우 파일의 마지막 라인까지 복사합니다.

**예**

다음은 emp 테이블에서 직원 쉼표로 구분된 목록이 포함된 파일
C : ₩ TEMP ₩ EMPDIR ₩ empfile.csv 사본을 만듭니다.
그런 다음 복사 empcopy.csv가 나열됩니다. FCOPY은 유효 함수이므로 PERFORM 명령을 사용해 시작된다는 점에 주의하십시오. PERFORM 명령은 Oracle 호환이 아닙니다.

```
CREATE DIRECTORY empdir AS 'C : ₩ TEMP / EMPDIR';

DECLARE
v_empfile UTL_FILE.FILE_TYPE;
v_src_dir VARCHAR2 (50) : = 'empdir';
v_src_file VARCHAR2 (20) : = 'empfile.csv';
v_dest_dir VARCHAR2 (50) : = 'empdir';
v_dest_file VARCHAR2 (20) : = 'empcopy.csv';
v_emprec VARCHAR2 (120);
v_count INTEGER : = 0;
BEGIN
PERFORM UTL_FILE.FCOPY (v_src_dir, v_src_file, v_dest_dir, v_dest_file);
v_empfile : = UTL_FILE.FOPEN (v_dest_dir, v_dest_file, 'r');
DBMS_OUTPUT.PUT_LINE ( 'The following is the destination file, '''| |
v_dest_file | | '''');
LOOP
UTL_FILE.GET_LINE (v_empfile, v_emprec);
DBMS_OUTPUT.PUT_LINE (v_emprec);
v_count : = v_count + 1;
END LOOP;
EXCEPTION
WHEN NO_DATA_FOUND THEN
UTL_FILE.FCLOSE (v_empfile);
DBMS_OUTPUT.PUT_LINE (v_count | | 'records retrieved');
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'SQLERRM :'| | SQLERRM);
DBMS_OUTPUT.PUT_LINE ( 'SQLCODE :'| | SQLCODE);
```

```
END;

The following is the destination file, 'empcopy.csv'
7369, SMITH, CLERK ,7902,17 - DEC - 80, 800, 20
7499, ALLEN, SALESMAN ,7698,20 - FEB - 81, 1600,300,30
7521, WARD, SALESMAN ,7698,22 - FEB - 81, 1250,500,30
7566, JONES, MANAGER ,7839,02 - APR - 81, 2975, 20
7654, MARTIN, SALESMAN ,7698,28 - SEP - 81, 1250,1400,30
7698, BLAKE, MANAGER ,7839,01 - MAY - 81, 2850, 30
7782, CLARK, MANAGER ,7839,09 - JUN - 81, 2450, 10
7788, SCOTT, ANALYST ,7566,19 - APR - 87, 3000, 20
7839, KING, PRESIDENT, ,17 - NOV - 81, 5000, 10
7844, TURNER, SALESMAN ,7698,08 - SEP - 81, 1500,0,30
7876, ADAMS, CLERK ,7788,23 - MAY - 87, 1100, 20
7900, JAMES, CLERK ,7698,03 - DEC - 81, 950, 30
7902, FORD, ANALYST ,7566,03 - DEC - 81, 3000, 20
7934, MILLER, CLERK ,7782,23 - JAN - 82, 1300, 10
14 records retrieved
```

### 7.4.4 FFLUSH

FFLUSH 프로시저는 쓰기 버퍼 파일에 기록되지 않은 데이터를 플래시합니다.

FFLUSH *(filetype FILE_TYPE)*

**매개 변수**

*filetype*

파일 핸들을 포함하는 변수 형식 FILE_TYPE

**예**

NEW_LINE 프로시저가 호출된 후, 각 라인이 플래시됩니다.

```
DECLARE
v_empfile UTL_FILE.FILE_TYPE;
v_directory VARCHAR2 (50) : = 'empdir';
v_filename VARCHAR2 (20) : = 'empfile.csv';
CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
v_empfile : = UTL_FILE.FOPEN (v_directory, v_filename, 'w');
FOR i IN emp_cur LOOP
UTL_FILE.PUT (v_empfile, i.empno);
UTL_FILE.PUT (v_empfile ,',');
UTL_FILE.PUT (v_empfile, i.ename);
UTL_FILE.PUT (v_empfile ,',');
UTL_FILE.PUT (v_empfile, i.job);
UTL_FILE.PUT (v_empfile ,',');
UTL_FILE.PUT (v_empfile, i.mgr);
UTL_FILE.PUT (v_empfile ,',');
UTL_FILE.PUT (v_empfile, i.hiredate);
UTL_FILE.PUT (v_empfile ,',');
UTL_FILE.PUT (v_empfile, i.sal);
UTL_FILE.PUT (v_empfile ,',');
UTL_FILE.PUT (v_empfile, i.comm);
UTL_FILE.PUT (v_empfile ,',');
UTL_FILE.PUT (v_empfile, i.deptno);
UTL_FILE.NEW_LINE (v_empfile);
UTL_FILE.FFLUSH (v_empfile);
END LOOP;
DBMS_OUTPUT.PUT_LINE ( 'Created file :'| | v_filename);
```

```
UTL_FILE.FCLOSE (v_empfile);
END;
```

### 7.4.5 FOPEN

FOPEN 함수는 I / O를 위해 파일을 엽니다.

*filetype FILE_TYPE FOPEN (dirname VARCHAR2, filename VARCHAR2,*
*mode VARCHAR2 ', linesize INTEGER)*

**매개 변수**

*dirname*

열릴 파일을 포함하는 디렉토리, pg_catalog.edb_dir.dirname에 저장된 디렉토리 이름.

*filename*

열릴 파일 이름.

*mode*

열릴 파일을 형태.

형태는 다음과 같다: a - 파일에 추가, r - 파일에서 읽기, w - 파일에 내보낼.

*linesize*

라인의 최대 길이. 읽기 모드에서는 *max_linesize* 를 초과하는 라인을 읽으려 시도하면 예외가 발생합니다. 쓰기, 추가 모드에서는 *max_linesize* 를 넘는 라인을 기록하려고 하면 예외가 발생합니다. 라인 종료 문자는 최대 라인의 크기가 초과 여부 결정에 포함되지 않습니다. 이 행동은 Oracle 호환이 되지 않습니다. Oracle은 라인 종료 문자를 카운트 합니다.

*filetype*

열린 파일 핸들을 포함하는 변수 형식 FILE_TYPE

7.4.6 FREMOVE

FREMOVE 함수는 시스템에서 파일을 삭제합니다.

VOID FREMOVE *(dirname VARCHAR2, filename VARCHAR2)*

삭제된 파일이 없으면 예외가 발생합니다.

**매개 변수**

*location*

제거될 디렉토리에 포함된 파일의 pg_catalog.edb_dir.dirname에 저장된 디렉토리의 이름 입니다.

*filename*

삭제되는 파일 이름입니다.

**예**

다음은 파일 empfile.csv를 삭제합니다.

```
 DECLARE
v_directory VARCHAR2 (50) : = 'empdir';
v_filename VARCHAR2 (20) : = 'empfile.csv';
BEGIN
PERFORM UTL_FILE.FREMOVE (v_directory, v_filename);
```

```
DBMS_OUTPUT.PUT_LINE ( 'Removed file :'| | v_filename);
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ( 'SQLERRM :'| | SQLERRM);
DBMS_OUTPUT.PUT_LINE ( 'SQLCODE :'| | SQLCODE);
END;

Removed file : empfile.csv
```

### 7.4.7 FRENAME

FRENAME 프로시저는 지정된 파일 이름을 변경합니다. 이것은 한 위치에서 다른 위치로 파일을 효과적으로 이동합니다.

FRENAME(*location* VARCHAR2, *filename* VARCHAR2,
*dest_dir* VARCHAR2, *dest_file* VARCHAR2, [ *overwrite* BOOLEAN ])

**매개 변수**

*location*

pg_catalog.edb_dir.dirname에 저장된, 이름이 변경될 파일을 포함하는 디렉토리 명

*filename*

변경될 소스 파일의 이름

*dest_dir*

pg_catalog.edb_dir.dirname에 저장된, 이름이 변경된 파일에 존재하기 위한 디렉토리 명

*dest_file*

원래 파일의 새 이름

*overwrite*

참으로 설정되면, dest_dir 에 있는 dest_file 라는 이름으로 존재하는 파일을 대체한다. 거짓으로 설정되면, 예외로 한다. 이것은 기본값이다.

**예**

다음은, emp 테이블로부터 쉼표로 구분된 직원의 목록을 포함하는 C:₩TEMP₩EMPDIR₩empfile.csv 파일의 이름을 변경합니다. 이름이 변경된 파일 C:₩TEMP₩NEWDIR₩newemp.csv 는 열거됩니다.

```
CREATE DIRECTORY "newdir" AS 'C:/TEMP/NEWDIR';
DECLARE
   v_empfile      UTL_FILE.FILE_TYPE;
   v_src_dir      VARCHAR2(50) := 'empdir';
   v_src_file     VARCHAR2(20) := 'empfile.csv';
   v_dest_dir     VARCHAR2(50) := 'newdir';
   v_dest_file    VARCHAR2(50) := 'newemp.csv';
   v_replace      BOOLEAN := FALSE;
   v_emprec       VARCHAR2(120);
   v_count        INTEGER := 0;
BEGIN
```

```
    UTL_FILE.FRENAME(v_src_dir,v_src_file,v_dest_dir,
       v_dest_file,v_replace);
    v_empfile := UTL_FILE.FOPEN(v_dest_dir,v_dest_file,'r');
    DBMS_OUTPUT.PUT_LINE('The following is the renamed file, ''' ||
       v_dest_file || '''');
    LOOP
       UTL_FILE.GET_LINE(v_empfile,v_emprec);
       DBMS_OUTPUT.PUT_LINE(v_emprec);
       v_count := v_count + 1;
    END LOOP;
    EXCEPTION
       WHEN NO_DATA_FOUND THEN
          UTL_FILE.FCLOSE(v_empfile);
          DBMS_OUTPUT.PUT_LINE(v_count || ' records retrieved');
       WHEN OTHERS THEN
          DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
          DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

The following is the renamed file, 'newemp.csv'
7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30
7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10
7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10
7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
14 records retrieved
```

### 7.4.8 GET_LINE

GET_LINE 프로시저는 라인 종료 터미네이터를 포함하지 않고, 주어진 파일로부터 텍스트 라인을 읽습니다. A NO_DATA_FOUND 예외는 더 이상 읽을 라인이 없을 때 주어집니다.

GET_LINE(*file* FILE_TYPE, *buffer* OUT VARCHAR2)

**매개변수**

*file*

열린 파일의 파일 핸들러를 포함하는 FILE_TYPE 형의 변수

*buffer*

파일로부터 라인을 받기 위한 변수

**예**

다음의 익명 블록은 empfile.csv 파일에서 레코드를 표시하고 읽습니다.

```
DECLARE
    v_empfile      UTL_FILE.FILE_TYPE;
    v_directory    VARCHAR2(50) := 'empdir';
    v_filename     VARCHAR2(20) := 'empfile.csv';
    v_emprec       VARCHAR2(120);
    v_count        INTEGER := 0;
BEGIN
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'r');
    LOOP
       UTL_FILE.GET_LINE(v_empfile,v_emprec);
       DBMS_OUTPUT.PUT_LINE(v_emprec);
       v_count := v_count + 1;
    END LOOP;
    EXCEPTION
       WHEN NO_DATA_FOUND THEN
          UTL_FILE.FCLOSE(v_empfile);
          DBMS_OUTPUT.PUT_LINE('End of file ' || v_filename || ' - ' ||
             v_count || ' records retrieved');
       WHEN OTHERS THEN
          DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
          DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30
7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10
7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10
7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
End of file empfile.csv - 14 records retrieved
```

### 7.4.9 IS_OPEN

IS_OPEN 함수는 주어진 파일을 열지, 열지 않을 것인지 결정합니다.

*status* BOOLEAN IS_OPEN(*file* FILE_TYPE)

**매개변수**

*file*

테스트 될 파일의 파일 핸들러를 포함하는 FILE_TYPE 형의 변수

*status*

"True" if the given file is open, "false" otherwise.

주어진 파일을 열면 '참', 그렇지 않으면 '거짓'

### 7.4.10 NEW_LINE

NEW_LINE 프로시저는 파일에 라인 종료 문자 시퀀스를 씁니다.

NEW_LINE(*file* FILE_TYPE [, *lines* INTEGER ])

**매개변수**

*file*

라인 종료 문자 시퀀스가 쓰여진 파일의 파일 핸들러를 포함하는 FILE_TYPE 형의 변수

*lines*

쓰여진 라인 종료 문자 시퀀스의 수. 기본값은 1.

**예**

직원의 두줄 간격 목록을 포함하는 파일이 쓰여집니다.

```
DECLARE
   v_empfile      UTL_FILE.FILE_TYPE;
   v_directory    VARCHAR2(50) := 'empdir';
   v_filename     VARCHAR2(20) := 'empfile.csv';
   CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
   v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
   FOR i IN emp_cur LOOP
      UTL_FILE.PUT(v_empfile,i.empno);
      UTL_FILE.PUT(v_empfile,',');
      UTL_FILE.PUT(v_empfile,i.ename);
      UTL_FILE.PUT(v_empfile,',');
      UTL_FILE.PUT(v_empfile,i.job);
      UTL_FILE.PUT(v_empfile,',');
      UTL_FILE.PUT(v_empfile,i.mgr);
      UTL_FILE.PUT(v_empfile,',');
      UTL_FILE.PUT(v_empfile,i.hiredate);
      UTL_FILE.PUT(v_empfile,',');
      UTL_FILE.PUT(v_empfile,i.sal);
      UTL_FILE.PUT(v_empfile,',');
      UTL_FILE.PUT(v_empfile,i.comm);
      UTL_FILE.PUT(v_empfile,',');
      UTL_FILE.PUT(v_empfile,i.deptno);
      UTL_FILE.NEW_LINE(v_empfile,2);
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
   UTL_FILE.FCLOSE(v_empfile);
END;

Created file: empfile.csv
```

파일이 보여집니다.

```
C:\TEMP\EMPDIR>TYPE empfile.csv

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
```

```
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```

### 7.4.11 PUT

PUT 프로시저는 문자열을 주어진 파일에 씁니다. 라인 종료 문자 시퀀스는 문자열의 끝에 쓰여지지 않습니다. 라인 종료 문자 시퀀스에 추가하기 위해서 NEW_LINE 프로시저를 사용하세요.

PUT(*file* FILE_TYPE, *buffer* { DATE | NUMBER | TIMESTAMP |
VARCHAR2 })

**매개변수**

*file*

주어진 문자열이 작성될 파일의 파일 핸들러를 포함하는 FILE_TYPE 형의 변수

*buffer*

지정된 파일에 작성될 텍스트

**예**

다음의 예는 emp 테이블로부터 직원의 쉼표 처리 된 파일을 생성하기 위해 PUT 프로시저를 사용합니다.

```
DECLARE
   v_empfile      UTL_FILE.FILE_TYPE;
   v_directory    VARCHAR2(50) := 'empdir';
   v_filename     VARCHAR2(20) := 'empfile.csv';
   CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
   v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
   FOR i IN emp_cur LOOP
      UTL_FILE.PUT(v_empfile,i.empno);
      UTL_FILE.PUT(v_empfile,',');
      UTL_FILE.PUT(v_empfile,i.ename);
      UTL_FILE.PUT(v_empfile,',');
      UTL_FILE.PUT(v_empfile,i.job);
```

```
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.mgr);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.hiredate);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.sal);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.comm);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.deptno);
        UTL_FILE.NEW_LINE(v_empfile);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
END;

Created file: empfile.csv
```

다음은 위에서 생성된 empfile.csv의 내용입니다.

```
C:\TEMP\EMPDIR>TYPE empfile.csv

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

### 7.4.12 PUT_LINE

PUT_LINE 프로시저는 라인 종료 문자 시퀀스를 포함하는 주어진 파일에 하나의 라인을 씁니다.

PUT_LINE(*file* FILE_TYPE, *buffer* { DATE | NUMBER | TIMESTAMP |
VARCHAR2 })

**매개변수**

*file*

주어진 라인이 작성될 파일의 파일 핸들러를 포함하는 FILE_TYPE 형의 변수

*buffer*

지정된 파일에 작성될 텍스트

**예**

다음의 예는 emp 테이블로부터 직원의 쉼표 처리된 파일을 생성하기 위해 PUT_LINE 프로시저를

사용합니다.

```
DECLARE
   v_empfile      UTL_FILE.FILE_TYPE;
   v_directory    VARCHAR2(50) := 'empdir';
   v_filename     VARCHAR2(20) := 'empfile.csv';
   v_emprec       VARCHAR2(120);
   CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
   v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
   FOR i IN emp_cur LOOP
      v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||
         NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
         ',' || i.sal || ',' ||
         NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
      UTL_FILE.PUT_LINE(v_empfile,v_emprec);
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
   UTL_FILE.FCLOSE(v_empfile);
END;
```

다음은 위에서 생성된 empfile.csv 의 내용입니다.

```
C:\TEMP\EMPDIR>TYPE empfile.csv

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

### 7.4.13 PUTF

PUTF 프로시저는 서식화된 문자열을 주어진 파일에 작성합니다.

PUTF(*file* FILE_TYPE, *format* VARCHAR2 [, *arg1* VARCHAR2]
[, ...])

**매개변수**

*file*

서식화된 라인이 작성될 파일의 파일 핸들러를 포함하는 FILE_TYPE 형의 변수

*format*

파일에 작성된 텍스트를 서식화하는 문자열. 특별한 문자 시퀀스, %s, 는 arg 값에 의해 대체됩니다. 특별한 문자 시퀀스, ₩n,는 새로운 라인을 가리킵니다.
그러나 Postgres Plus Advanced 서버에, 새로운 라인 문자는 한 개의- ₩₩n 대신 두 개의 연속적인 백슬래시가 지정됩니다. 이 특성은 Oracle 호환성이 없습니다.

*arg1*

각 %s 의 발생에 대체되는 문자열 서식에 5개 인자, *arg1,...arg5*
첫 번째 arg는 첫 번째 %s 발생에 대체되고, 두 번째 arg는 두 번째 %s의 발생에 대체됩니다.

**예**

다음의 익명 블록은 emp 테이블로부터 데이터를 포함하는 서식화된 출력물을 산출합니다. Oracle 호환성이 없는 문자열 서식에 새로운 라인 문자 시퀀스에, E 리터럴 구문의 사용과 두 개의 백슬러시를 사용합니다.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
    v_format        VARCHAR2(200);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    v_format := E'%s %s, %s\\nSalary: $%s Commission: $%s\\n\\n';
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
    FOR i IN emp_cur LOOP
        UTL_FILE.PUTF(v_empfile,v_format,i.empno,i.ename,i.job,i.sal,
            NVL(i.comm,0));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

Created file: empfile.csv
```

다음은 위에서 생성된 empfile.csv 의 내용입니다.

```
C:\TEMP\EMPDIR>TYPE empfile.csv

7369 SMITH, CLERK
Salary: $800.00 Commission: $0

7499 ALLEN, SALESMAN
Salary: $1600.00 Commission: $300.00

7521 WARD, SALESMAN
Salary: $1250.00 Commission: $500.00
```

```
7566 JONES, MANAGER
Salary: $2975.00 Commission: $0

7654 MARTIN, SALESMAN
Salary: $1250.00 Commission: $1400.00

7698 BLAKE, MANAGER
Salary: $2850.00 Commission: $0

7782 CLARK, MANAGER
Salary: $2450.00 Commission: $0

7788 SCOTT, ANALYST
Salary: $3000.00 Commission: $0

7839 KING, PRESIDENT
Salary: $5000.00 Commission: $0

7844 TURNER, SALESMAN
Salary: $1500.00 Commission: $0.00

7876 ADAMS, CLERK
Salary: $1100.00 Commission: $0

7900 JAMES, CLERK
Salary: $950.00 Commission: $0

7902 FORD, ANALYST
Salary: $3000.00 Commission: $0

7934 MILLER, CLERK
Salary: $1300.00 Commission: $0
```

**8장 오픈 클라이언트 라이브러리**

오픈 클라이언트 라이브러리 (Open Client Library)은 오라클 호출 인터페이스 (OCI)와 애플리케이션의 상호 운용성을 제공합니다. 그 동안 오라클 "잠금상태"로 되어 응용 프로그램이 응용 프로그램 코드에 약간의 수정 또는 그대로 Postgres Plus Advanced Server 또는 Oracle 데이터베이스에서 다 동작할 수 있다. 오픈 클라이언트 라이브러리(OCL)은 처음부터 C 언어로 만들어졌습니다.

다음 그림은 오픈 클라이언트 라이브러리와 오라클 호출 인터페이스 애플리케이션 구조의 비교입니다.

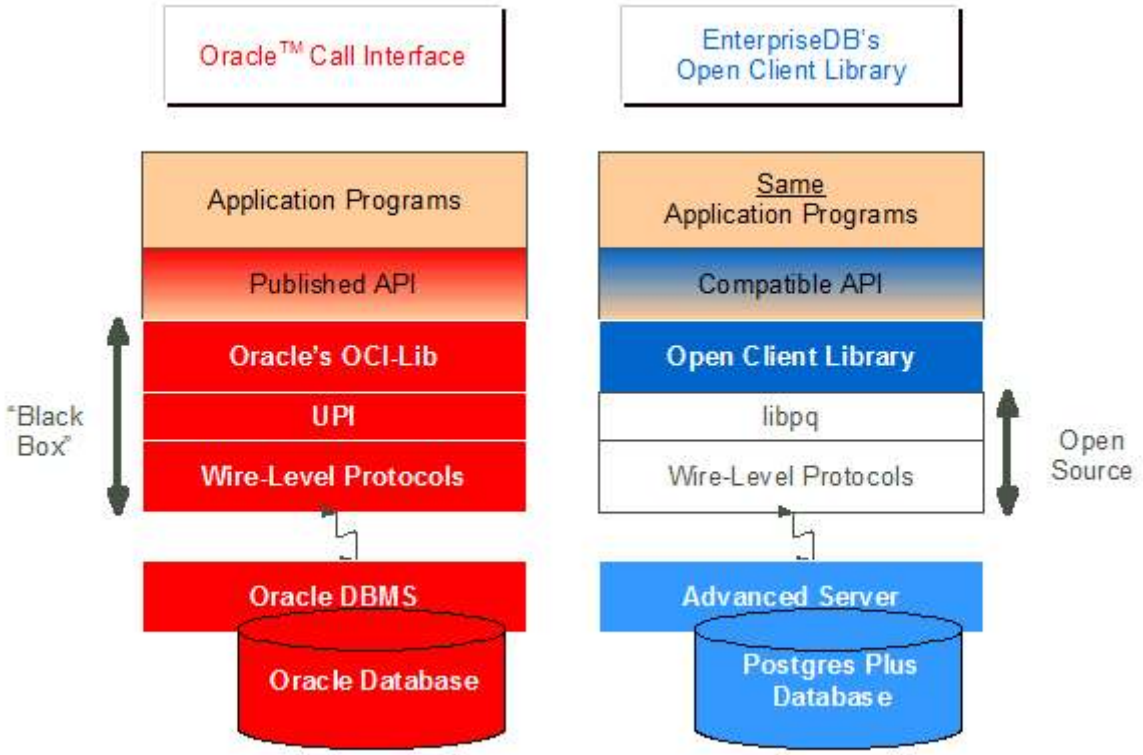


**그림 6 오픈 클라이언트 라이브러리**

다음은 오픈 클라이언트 라이브러리에서 지원되는 함수 목록입니다. 거의 모든 헤더 파일은 사용자가 제공할 수 있습니다. Postgres Plus Advanced Server 는 그런 파일은 제공하지 않습니다.

Table 8-1 연결, 인증, 초기화

| 함수 | 설명 |
|---|---|
| OCIEnvCreate | OCI 환경을 생성. |
| OCIEnvInit | OCI 환경 핸들 초기화. |
| OCIInitialize | OCI 환경 초기화. |
| OCILogoff | 세션의 개방. |
| OCILogon | 접속 연결 발생. |
| OCILogon2 | 다양한 모드 세션 접속 발생. |
| OCIServerAttach | 데이터 소스에 접근 경로의 설립. |
| OCIServerDetach | 데이터 소스에 액세스 경로 개방. |
| OCISessionBegin | 사용자 세션 생성. |
| OCISessionEnd | 사용자 세션 종료. |
| OCISessionGet | 세션 풀에서 세션 받음. |
| OCISessionRelease | 세션의 개방. |

| | |
|---|---|
| OCITerminate | 공유 메모리 서브 시스템에서 개방. |

Table 8-2 핸들, 기술어 함수

| 함수 | 설명 |
|---|---|
| OCIAttrGet | 핸들 속성을 받음. |
| OCIAttrSet | 핸들 속성 설정. |
| OCIDescriptorAlloc | 기술어 할당 및 초기화. |
| OCIDescriptorFree | 할당된 기술어 개방. |
| OCIHandleAlloc | 핸들의 할당과 초기화. |
| OCIHandleFree | 할당된 핸들 개방. |
| OCIParamGet | 매개변수 기술어 가져오기. |
| OCIParamSet | 매개변수 기술어를 설정. |

Table 8-3 바인딩 정의 기술 함수

| 함수 | 설명 |
|---|---|
| OCIBindByName | 이름으로 바인딩. |
| OCIBindByPos | 위치에 의한 바인딩. |
| OCIBindDynamic | 바인딩 후에 추가 속성을 설정. |
| OCIBindArrayOfStruct | 대량 작업을 위한 배열 구조를 바인딩. |
| OCIDefineByPos | 출력 변수의 관련성을 정의. |
| OCIDefineDynamic | 정의에 대한 추가 특성을 설정. |
| OCIDescribeAny | 기존 스키마 개체를 작성. |
| OCIStmtGetBindInfo | 바인딩 및 표시기 변수 이름과 핸들. |

Table 8-4 문 함수

| 함수 | 설명 |
|---|---|
| OCIStmtExecute | 준비한 SQL 문을 실행. |
| OCIStmtFetch | 행 데이터 가져오기 (사용되지 않음). |
| OCIStmtFetch2 | 행 데이터 가져오기. |
| OCIStmtPre pare | SQL 문을 준비. |
| OCIStmtPrepare2 | SQL 문을 준비. |
| OCIStmtRelease | 명령문 핸들 릴리스 |

Table 8-5 트랜잭션 함수

| 함수 | 설명 |
|---|---|
| OCITransCommit | 트랜잭션 커밋. |
| OCITransRollback | 트랜잭션 롤백. |

Table 8-6 기타 기능

| 함수 | 설명 |
|---|---|
| OCIClientVersion | 클라이언트 버전을 반환 |
| OCIErrorGet | 오류 메시지를 반환. |
| OCIPasswordChange | 암호 변경. |
| OCIPing | 연결과 서버가 실행중인 확인. |
| OCIServerVersion | Oracle 버전 문자열 받았습니다. |

Table 8-7 날짜, 날짜와 시간 함수

| 함수 | 설명 |
|---|---|
| OCIDateAddDays | 일 덧셈, 뺄셈. |
| OCIDateAddMonths | 월 덧셈, 뺄셈. |
| OCIDateAssign | 일시 할당 |
| OCIDateCheck | 지정한 날짜가 유효한지 확인하십시오. |
| OCIDateCompare | 2 개의 날짜 비교. |
| OCIDateDaysBetween | 2 개의 날짜 사이의 날짜의 수를 찾습니다. |
| OCIDateFromText | 문자열을 날짜로 변환. |
| OCIDateGetDate | 날짜에서 날짜 부분을 받았습니다. |
| OCIDateGetTime | 날짜에서 시간 부분을 받았습니다. |
| OC IDateLastDay | 달의 마지막 날 날짜를 받았습니다. |
| OCIDateNextDay | 다음날 날짜를 받았습니다. |
| OCIDateSetDate | 날짜의 날짜 부분을 조정. |
| OCIDateSetTime | 날짜 시간 부분을 조정. |
| OCIDateSysDate | 현재 시스템 날짜와 시간을 받았습니다. |
| OCIDateToText | 날짜를 문자열로 변환. |
| OCIDateTimeAssign | 날짜와 시간을 할당. |
| OCIDateTimeCheck | 시간이 유효한지 확인하십시오. |
| OCIDateTimeCompare | 2 개의 시간을 비교. |
| OCIDateTimeConstruct | 시간 기술어를 구축. |
| OCIDateTimeConvert | 한 날짜 형식을 다른 변환 |
| OCIDateTimeFromArray | 사이즈 OCI_DT_ARRAYLEN 배열을 OCIDateTime |

| | |
|---|---|
| | 기술어로 변환. |
| OCIDateTimeFromText | 지정된 형식에 따라 주어진 문자열을 OCIDateTime 기술어를 오라클 날짜 형식으로 변환 |
| OCIDateTimeGetDate | 날짜 값에서 날짜 부분을 받았습니다. |
| OCIDateTimeGetTime | 일시 값에서 시간 부분을 받았습니다. |
| OCIDateTimeGetTimeZoneName | 날짜 값에서 시간대 이름 부분을 받았습니다. |
| OCIDateTimeGetTimeZoneOffset | 날짜 값에서 시간대 (시, 분) 부분을 받았습니다. |
| OCIDateTimeSubtract | 2 개의 날짜 값을 입력하고 그 차이를 간격으로 반환합니다. |
| OCIDateTimeSysTimeStamp | 시스템의 현재 날짜와 시간을 시간대와 함께 타임 스탬프로 받았습니다. |
| OCIDateTimeToArray | OCIDateTime 설명자를 배열로 변환 |
| OCIDateTimeToText | 지정된 형식에 따라 임의의 날짜를 문자열로 변환. |

Table 8-8 NUMBER 함수

| 함수 | 설명 |
|---|---|
| OCINumberAbs | 절대값을 계산. |
| OCINumberAdd | NUMBER 를 가산. |
| OCINumberArcCos | arc 코사인을 계산. |
| OCINumberArcSin | arc 사인을 조사. |
| OCINumberArcTan | arc 탄젠트 계산. |
| OCINumberArcTan2 | 2 개의 NUMBER arc 탄젠트 계산. |
| OCINumberAssign | 한 NUMBER 을 다른 사람에게 할당. |
| OCINumberCeil | 상한 값을 계산. |
| OCINumberCmp | NUMBER 비교. |
| OCINumberCos | 코사인을 계산. |
| OCINumberDec | NUMBER 를 감소. |
| OCINumberDiv | 2 개의 NUMBER 나누기. |
| OCINumberExp | e 를 지정된 NUMBER 에서 올리기. |
| OCINumberFloor | NUMBER 의 하한선을 계산. |
| OCINumberFromInt | 정수를 Oracle NUMBER 로 변환. |
| OCINumberFromReal | 실수를 Orac le NUMBER 로 변환. |
| OCINumberFromText | 문자열을 Oracle NUMBER 로 변환. |
| OCINumberHypCos | 코사인 값을 계산. |
| OCINumberHypSin | 사인 값을 계산. |
| OCINumberHypTan | 탄젠트 값을 계산. |
| OCINumberInc | NUMBER 를 증가 |
| OCINumberIntPower | 정수 능력을 주어진 자료를 올립니다 |

| | |
|---|---|
| OCINumberIsInt | NUMBER 이 정수 있는지 여부를 테스트. |
| OCINumberIsZero | NUMBER 가 0 있는지 확인 |
| OCINumberLn | 자연 로그 (기수 e)를 계산. |
| OCINumberLog | 모든 기수에 대한 로그를 계산. |
| OCINumberMod | 나눈 나머지 (계수)를 받았습니다. |
| OCINumberMul | NUMBER 를 곱하여. |
| OCINumberNeg | NUMBER 의 음수. |
| OCINumberPower | 바닥 *e 을* 지수 |
| OCINumberPrec | NUMBER 를 지정된 소수 자릿수로 반올림. |
| OCINumberRound | NUMBER 를 지정된 소수 자릿수로 반올림. |
| OCINumberSet Pi | NUMBER 를 Pi 초기화. |
| OCINumberSetZero | NUMBER 를 0 으로 초기화. |
| OCINumberShift | 10 을 곱하여 소수점 이하를 지정 자릿수. |
| OCINumberSign | NUMBER 부호를 받았습니다. |
| OCINumberSin | 사인 계산. |
| OCINumberSqrt | NUMBER 의 제곱근을 계산. |
| OCINumberSub | NUMBER s 감소. |
| OCINumberTan | 탄젠트 계산. |
| OCINumberToInt | NUMBER 을 정수로 변환. |
| OCIN umberToReal | NUMBER 를 실수로 변환. |
| OCINumberToRealArray | NUMBER 배열을 실제 배열로 변환. |
| OCINumberToText | NUMBER 를 문자열로 변환. |
| OCINumberTrunc | NUMBER 를 지정된 소수 자릿수로 내림. |

Table 8-9 문자열 함수

| 함수 | 설명 |
|---|---|
| OCIStringAllocSize | 문자열 메모리 크기가 할당된 바이트 받았습니다. |
| OCIStringAssign | 문자열을 문자열로 할당. |
| OCIStringAssignText | 문자열을 문자열로 할당. |
| OCIStringPtr | 문자열 포인터를 받았습니다. |
| OCIStringResize | 문자열 메모리 크기 변경. |
| OCIStringSi ze | 문자의 크기를 받았습니다. |

Table 8-10 카트리지, 파일 I / O 인터페이스 함수

| 함수 | 설명 |
|---|---|
| OCIFileClose | 열린 파일을 닫는다. |
| OCIFileExists | 파일이 존재하는지 테스트. |

| | |
|---|---|
| OCIFileFlush | 버퍼의 데이터를 파일에 씁니다. |
| OCIFileGetLength | 파일의 길이 취득. |
| OCIFileInit | OCIFil e 패키지를 초기화. |
| OCIFileOpen | 파일을 열기. |
| OCIFileRead | 버퍼에 파일의 읽기. |
| OCIFileSeek | 파일의 현재 위치를 변경. |
| OCIFileTerm | OCIFile 패키지를 종료. |
| OCIFileWrite | buflen 바이트를 파일에 씁니다. |

Table 8-11 지원하는 데이터 형식

| 함수 | 설명 |
|---|---|
| ANSI_DATE | AN SI 날짜 |
| SQLT_AFC | ANSI 고정 길이 문자 |
| SQLT_AVC | ANSI 가변 길이 문자 |
| SQLT_BDOUBLE | 이진 더블 |
| SQLT_BIN | 이진 데이터 |
| SQLT_BFLOAT | 이진 부동 소수점 |
| SQLT_CHR | 문자열 |
| SQLT_DAT | Oracle 날짜 |
| SQLT_DATE | ANSI 날짜 |
| SQLT_FLT | 부동 소수 |
| SQLT_INT | 정수 |
| SQLT_LBI | 롱 이진 |
| SQLT_LNG | 길다 |
| SQLT_LVB | 더욱 긴 바이너리 |
| SQLT_LVC | 더욱 긴 (문자) |
| SQLT_NUM | Oracle 숫자 |
| SQLT_ODT | OCI 날짜 형식 |
| SQLT_STR | 제로 종단 문자열 |
| SQLT_TIMESTAMP | 타임 스탬프 |
| SQLT_TIMESTAMP_TZ | 시간대 갖춘 시점 |
| SQLT_TIMESTAMP_LTZ | 현지 시간대가 내장된 시점 |
| SQLT_UIN | 부호 없는 정수 |
| SQLT_VBI | VCS 형식 이진 |
| SQLT_VCS | 가변 길이 문자 |
| SQLT_VNU | 길이 바이트 갖춘 몇 값 |
| SQLT _VST | OCI 문자열 |

## 제9장 오라클 뷰 목록

Oracle 뷰 목록은 오라클 데이터베이스의 오라클 데이터 사전과 호환되는 방법으로, 오라클과 호환된 데이터베이스 개체의 정보를 제공합니다.

### 9.1 ALL_OBJECTS

ALL_OBJECTS 보기는 다음과 같은 데이터베이스 개체에 대한 정보를 제공합니다. 테이블, 인덱스, 뷰, 트리거, 함수, 패키지, 패키지 시스템. SPL 트리거, 함수, 프로시저 패키지, 패키지 본문이 표시되지 않으므로 주의해야 한다. PL / pgSQL 트리거 함수는 ALL_OBJECTS 보기에 표시되지 않습니다.

표 9-1 ALL_OBJECTS

| 이름 | 형 | 설명 |
|---|---|---|
| owner | VARCHAR2 | 개체 소유자의 사용자 이름. |
| schemaname | VARCHAR2 | 개체가 속해있는 스키마의 이름. |
| object_name | VARCHAR2 | 개체 이름 |
| object_type | VARCHAR2 | 객체 유형 - 가능한 값은: INDEX, FU NCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, TABLE, TRIGGER, 그리고 VIEW. |
| status | VARCHAR2 | 이 개체의 상태가 유효한지. 항상 VALID 로 설정 |

### 9.2 ALL_SOURCE

ALL_SOURCE 보기는 다음 프로그램 소스 코도 목록을 제공합니다. 함수 프로시저 트리거, 패키지 규격, 패키지 시스템입니다.

표 9-2 ALL_ SOURCE

| 이름 | 형 | 설명 |
|---|---|---|
| owner | VARCHAR2 | 프로그램 소유자의 사용자 이름 |
| schemaname | VARCHAR2 | 프로그램이 속해있는 스키마의 이름 |
| name | VARCHAR2 | 프로그램 이름 |
| type | VARCHAR2 | 학위 유형 - 가능한 값은: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, 그리고 TRIGGER. |
| line | INTEGER | 주어진 프로그램 소스 코드의 행수. |
| text | VARCHAR2 | 소스 코드 라인. |

### 9.3 ALL_SYNONYMS

ALL_SYNONYMS 보기는 모든 동의어 정보를 제공합니다.

**표 9-3 ALL_ SYNONYMS**

| 이름 | 형 | 설명 |
|---|---|---|
| owner | VARCHAR2 | 동의어 소유자의 사용자 이름. |
| synonym_name | VARCHAR2 | 동의어의 이름. |
| object_owner | VARCHAR2 | 동의어가 정의된 개체 소유자의 사용자 이름. |
| object_name | VARCHAR2 | 동의어가 정의되는 개체의 이름. |
| synacl | VARCHAR2 | 현재 사용하지 않는 - 항상 null. |
| status | VARCHAR2 | 이 동의어 상태가 유효한지. 항상 VALID 로 셋팅. |

### 9.4 ALL_TAB_COLUMNS

ALL_TAB_COLUMNS 보기는 모든 사용자 정의 테이블의 모든 열 정보를 제공합니다.

**표 9-4 ALL_ TAB_COLUMNS**

| 이름 | 형 | 설명 |
|---|---|---|
| owner | VARCHAR2 | 테이블 소유자의 사용자 이름. |
| schemaname | VARCHAR2 | 테이블이 속해있는 스키마의 이름. |
| table_name | VARCHAR2 | 테이블 이름. |
| column_name | VARCHAR2 | 열 이름. |
| data_type | VARCHAR2 | 열 데이터 형식 |
| data_length | INTEGER | 텍스트 줄의 길이. |
| data_precision | INTEGER | Numeric 열 정확도 - NUMBER 열이 자릿수 이진수 (INTEGER, REAL, DO UBLE PRECISION 등) 열은 비트 수 . |
| data_scale | INTEGER | NUMBER 열의 자릿수. |
| column_id | INTEGER | 테이블에서 열 상대적 위치. |
| nullable | CHARACTER | 열에 NULL 을 허용할지 여부를 나타냅니다. - 가능한 값은 : Y - 열은 null 이 가능; N - 열은 null 이 불가능하다. |
| data_default | VARCHAR2 | 열에 할당된 초기값. |

### 9.5 ALL_TABLES

ALL_T ABLES 보기는 모든 사용자 정의 테이블의 정보를 제공합니다.

**표 9-5 ALL_ TABLES**

| 이름 | 형 | 설명 |
|---|---|---|
| owner | VARCHAR2 | 테이블 소유자의 사용자 이름. |
| schemaname | VARCHAR2 | 테이블이 곳에 속하는 스키마의 이름. |
| table_name | VARCHAR2 | 테이블 이름. |
| table_space | VARCHAR2 | 기본 테이블 스페이스를 제외하고 테이블이 있는 테이블 스페이스의 이름입니다. |
| status | VARCHAR2 | 테이블의 상태가 유효한지. 항상 VALID 로 셋팅. |

### 9.6 ALL_USERS

ALL_USERS 보기는 모든 사용자의 정보를 제공합니다.

**표 9-6ALL_ USERS**

| 이름 | 형 | 설명 |
|---|---|---|
| u sername | VARCHAR2 | 아이디. |
| user_id | VARCHAR2 | 사용자에게 할당된 숫자의 사용자 ID. |

### 9.7 ALL_VIEW_COLUMNS

ALL_VIEW _COLUMNS 보기는 모든 사용자 정의보기의 모든 열에 대한 정보를 제공합니다.

**표 9-7 ALL_ VIEW_COLUMNS**

| 이름 | 형 | 설명 |
|---|---|---|
| owner | VARCHAR2 | 뷰 소유자의 사용자 이름. |
| schemaname | VARCHAR2 | 뷰가 속한 스키마의 이름. |
| view_name | VARCHAR2 | 뷰 이름. |
| column_name | VARCHAR2 | 열 이름. |
| data_type | VARCHAR2 | 열 데이터 형식입니다. |
| data_length | INTEGER | 텍스트 형식 열 텍스트 길이. |
| data_precision | INTEGER | Numeric 열 정확도 - NUMBER 열이 자릿수 이진수 (INTEGER, REAL, DO UBLE PRECISION 등) 열은 비트 수 . |
| data_scale | INTEGER | NUMBER 열의 자릿수. |
| column_id | INTEGER | 열에서 뷰의 상대적 위치. |
| nullable | CHARACTER | 열에 NULL 을 허용할지 여부를 나타냅니다. - 가능한 값은: Y - 열은 null 이 가능; N - 열은 null 이 불가능하다. |

| | | |
|---|---|---|
| data_default | VARCHAR2 | 열에 할당된 초기값. |

### 9.8 ALL_VIEWS

ALL_VIEW S 는 모든 사용자 정의보기의 정보를 제공합니다.

**표 9-8 ALL_ VIEWS**

| 이름 | 형 | 설명 |
|---|---|---|
| owner | VARCHAR2 | 뷰 소유자의 사용자 이름. |
| schemaname | VARCHAR2 | 뷰 소속 스키마의 이름. |
| view_name | VARCHAR2 | 뷰 이름. |
| status | VARCHAR2 | 테이블의 상태가 유효한지. 항상 VALID 로 셋팅. |

### 9.9 DBA_ROLE_PRIVS

DBA_ROLE_PRIVS 은 사용자에게 권리를 준 모든 역할에 대한 정보를 제공합니다. 줄은 사용자가 권한을 준 각 역할마다 생성됩니다.

**표 9-9 DBA_ROLE_PRIVS**

| 이름 | 형 | 설명 |
|---|---|---|
| grantee | VARCHAR2 | 역할의 권한이 주어진 사용자 이름. |
| granted_role | VARCHAR2 | 권리가 주어진 역할의 이름. |
| admin_option | VARCHAR2 | 역할이 관리자 옵션 권위 경우 YES, 그렇지 않으면 NO. |
| default_role | VARCHAR2 | 만약 권한을 가진 사용자가 세션을 시작하면 자동으로 목록이 활성화될 경우 YES, 그렇지 않으면 NO. 현재는 항상 YES. |

### 9.10 DBA_ROLES

DBA_ROLES 보기는 전체 목록에 대한 정보를 NOLOGIN 속성 (그룹)과 함께 제공합니다.

**표 9-10 DBA_ROLS**

| 이름 | 형 | 설명 |
|---|---|---|
| role | VARCHA R2 | 목록이 있는 NOLOGIN 속성 이름 - 예를 들어 그룹. |
| password_required | VARCHAR2 | 목록을 이용하기 위해 암호가 필요한지 여부. |

### 9.11 USER_OBJECTS

현재 사용자가 보유한 표시 전용 개체를 제외하고 ALL_OBJECTS 과 동일합니다.

### 9.12 USER_SOURCE

현재 사용자가 보유한 표시 전용 프로그램을 제외하고 ALL_SOURCE 과동일합니다

### 9.13 USER_SYNONYMS

현재 사용자가 보유한 표시 전용 동의어를 제외하고 ALL_SYNONYMS 과 동일합니다.

### 9.14 USER_TAB_COLUMNS

현재 사용자가 보유한 테이블 표시 전용 열을 제외하고 ALL_TAB_COLUMNS 과 동일합니다.

### 9.15 USER_TABLES

현재 사용자가 보유한 표시 전용 테이블을 제외하고 ALL_TABLES 과 동일합니다.

### 9.16 USER_VIEW_COLUMNS

현재 사용자가 보유한 뷰 디스플레이를 열을 제외하고 ALL_VIEW_COLUMNS 과 동일합니다.

### 9.17 USER_VIEWS

현재 사용자가 보유한 표시 전용보기를 제외하고 ALL_VIEWS ALL_VIEWS 과 동일합니다.

## 10 유틸리티

이 장에 각 섹션은 다양한 유틸리티 프로그램을 설명합니다. 다음과 같은 것을 포함합니다.

- EDB*Plus
- EDB*Loader
- 동적 런타임 수단

### *10.1 EDB*Plus*

EDB*Plus는 명령 라인 사용자가 Postgres Plus Advanced Server 에 인터페이스 하는 것을 제공하는 유틸리티 프로그램 입니다. EDB*Plus는 SQL명령, SPL 익명의 블록과 EDB*Plus 명령을 수락합니다. EDB*Plus 명령은 SQL*Plus과 호환되는 등 다양한 기능을 제공합니다.

- 특정 데이터베이스 객체의 쿼리

- 저장 프로시저를 실행
- SQL 명령의 서식 출력
- 배치 스크립트 실행
- 출력물 기록

다음 섹션은 Postgres Plus Advanced Server 데이터베이스가 EDB*Plus를 사용하여 어떻게 연결하는 지를 설명합니다. 또한, 마지막 섹션은 EDB*Plus 명령의 요약을 제공합니다.

### 10.1.1 EDB*Plus의 시작

EDB*Plus는 애플리케이션 메뉴에서 선택하거나 운영 체제 EDB*Plus 프로그램을 실행하여 명령 라인에서 시작할 수 있습니다.
다음에 EDB*Plus 프로그램은 Postgres Plus Advanced Server 홈 디렉토리의 위치한 edbplus 하위 디렉토리로 부터 edbplus 를 실행하여 호출됩니다.

edbplus [ -S[ILENT ] ] [ *login* | /NOLOG ] [ @*scriptfile*[.*ext* ] ]

**- SILENT**
지정된 경우, EDB*Plus 서명은 모든 프롬프트와 함께 압축되어 있습니다.

***Login***
데이터베이스 서버 및 데이터베이스에 연결에 대한 로그인 정보.
로그인은 다음과 같은 형식을 따릅니다. (로그인 정보 안에 공백이 없어야 합니다.

*username*[/*password*][@{*connectstring* | *variable* } ]

*username은* 데이터베이스를 연결하기 위한 데이터베이스 사용자 이름입니다.
*password 는 username*의 패스워드 입니다. 만약 *패스워드가* 생략되었는데 패스워드가 요구되어진다면 EDB*Plus는 패스워드를 묻는 프롬프트가 뜹니다.

@*connectstring* 또는 @*variable는* 데이터베이스 문자열이나 데이터베이스 연결 문자열을 담고 있는 login.sql 파일 안에 정의된 변수의 연결문자가 지정될지도 모릅니다.
( login.sql 파일은 Postgres Plus Advanced Server 홈 디렉토리의 edbplus 하위 디렉토리 안에서 찾아질 수 있습니다.) 이 두 경우 모두, 데이터베이스 연결 문자열은 다음과 같은 형식을 갖고 있습니다. (연결 문자열에는 공백이 없어야만 합니다.)

***host*[:*port*][/*dbname* ] ]**

*host 는* 데이터베이스 서버에 존재하는 호스트이름입니다. 만약 @*connectstring나* @*variable,* /NOLOG 어느 한 경우도 지정되지 않는다면 기본 호스트는 로컬호스트로 간주 됩니다.
*port는* 데이터베이스 서버의 연결을 받고 있는 포트 번호입니다. 만약 지정되어있지 않다면 기

본값은 5444입니다. *Dbname은* 데이터베이스에 연결하려는 이름입니다. 만약 지정되지 않았다면 기본값은 edb입니다.

**/NOLOG**

만약 /NOLOG이 지정되었다면, EDB*Plus는 데이터베이스 연결을 설정하지 않고 시작될 것입니다. 데이터베이스의 연결이 요구되는 SQL 명령과 EDB*Plus 명령은 이 모드에서 이용되어질 수 없습니다. CONNECT 명령은 /NOLOG옵션과 EDB*Plus를 시작한 후에 데이터베이스에 연결될 수 있습니다.

***scriptfile*[*.ext* ]**

*scriptfile은* EDB*Plus 시작 후에 자동적으로 실행되는 SQL 과/또는 EDB*Plus 명령을 포함하는 현재 작업중인 디렉토리에 존재하는 파일 이름입니다.
*ext*는 파일이름 확장입니다. 만약 파일이름 확장이 sql이라면 .sql 확장은 지정된 *scriptfile* 일 때 생략할지도 모릅니다. 스크립트 파일을 생성할 때, 항상 확장과 함께 파일 이름은 EDB*Plus에 의해 접근 되지 않을 것입니다. (EDB*Plus는 확장 없이 지정된 파일이름에 .sql 확장이 항상 가정될 것입니다.
아래의 예제는 패스워드, password, 포트5444의 로컬호스트에 데이터베이스 서버로 실행되는 edb 데이터베이스의 연결과 함께 enterprisedb 사용자에게 보여줍니다.

```
C:\EnterpriseDB\8.3\edbplus>edbplus enterprisedb/password
Connected to EnterpriseDB 8.3.0.10 (localhost:5444/edb) AS enterprisedb

EDB*Plus: Release 8.3 - Beta (Build 12)
Copyright (c) 2008, EnterpriseDB Corporation.  All rights reserved.

SQL>
```

아래의 예제는 패스워드, password, 포트5444의 로컬호스트에 데이터베이스 서버로 실행되는 edb 데이터베이스의 연결과 함께 enterprisedb 사용자에게 보여줍니다.

```
C:\EnterpriseDB\8.3\edbplus>edbplus
enterprisedb/password@localhost:5445/edb
Connected to EnterpriseDB 8.3.0.10 (localhost:5445/edb) AS enterprisedb

EDB*Plus: Release 8.3 - Beta (Build 12)
Copyright (c) 2008, EnterpriseDB Corporation.  All rights reserved.

SQL>
```

login.sql 파일의 hr_5445 변수를 사용하면, 포트5445 로컬호스트에 hr 데이터베이스의 연결하는 사용 방법을 설명합니다.

```
C:\EnterpriseDB\8.3\edbplus>edbplus enterprisedb/password@hr_5445
```

```
Connected to EnterpriseDB 8.3.0.10 (localhost:5445/hr) AS enterprisedb

EDB*Plus: Release 8.3 - Beta (Build 12)
Copyright (c) 2008, EnterpriseDB Corporation.  All rights reserved.

SQL>
```

다음은 이전의 예제에서 사용되었던 login.sql 파일의 연결을 보여줍니다.

```
define edb="localhost:5445/edb"
define hr_5445="localhost:5445/hr"
```

다음의 예제는 포트5444 서버 로컬호스트에 edb 데이터베이스로 연결한 후에 스크립트 파일, dept_query.sql을 실행합니다.

```
C:\EnterpriseDB\8.3\edbplus>edbplus enterprisedb/password @dept_query
Connected to EnterpriseDB 8.3.0.10 (localhost:5444/edb) AS enterprisedb

SQL>
SELECT * FROM dept;

DEPTNO DNAME         LOC
------ -------------- -------------
    10 ACCOUNTING     NEW YORK
    20 RESEARCH       DALLAS
    30 SALES         CHICAGO
    40 OPERATIONS     BOSTON

SQL>
EXIT
```

다음은 이전의 예제에서 사용되는 dept_query.sql 파일의 콘텐츠 입니다.

```
SET PAGESIZE 9999
SET ECHO ON
SELECT * FROM dept;
EXIT
```

### 10.1.2 명령문 요약

이 섹션은 EDB*Plus 명령문 요약이 있습니다.

#### 10.1.2.1 ACCEPT

ACCEPT 명령은 프롬프트를 표시하고 사용자 키보드 삽입을 위해 기다립니다. 사용자가 지정한 변수에 그 값을 입력합니다.

```
ACC[EPT ] variable
```

다음 예제는 my_name로 이름 지어진 새로운 변수를 생성합니다. John Smith의 값은 수락합니다. 그리고 DEFINE 명령의 사용되는 값을 표시합니다.

```
SQL> ACCEPT my_name
Enter value for my_name: John Smith
SQL> DEFINE my_name
```

```
DEFINE MY_NAME = "John Smith"
```

#### 10.1.2.2 APPEND

APPEND는 SQL 버퍼 안의 현재 라인의 끝으로 텍스트가 주어지는 첨부 명령의 라인 편집기 입니다.

A[PPEND ] *text*

다음 예제에서 SELECT 명령문은 APPEND 명령문을 사용하는 SQL 버퍼에 생성됩니다.
그 두 스페이스는 SQL버퍼에 한 스페이스로 dept과 WHERE을 나누기 위해서 APPEND과 WHERE 절 사이에 위치하는 것에 주의하세요.

```
SQL> APPEND SELECT * FROM dept
SQL> LIST
  1* SELECT * FROM dept
SQL> APPEND  WHERE deptno = 10
SQL> LIST
  1* SELECT * FROM dept WHERE deptno = 10
```

#### 10.1.2.3 CHANGE

CHANGE는 SQL버퍼 현재 라인에 검색과 대체를 수행하는 라인 편집기 변경 명령입니다.

C[HANGE ] /*from*/[*to*/ ]

만약 *to/* 가 지정된다면 현재 라인의 *from* 텍스트의 첫 번째 발생이 텍스트 *to*로 변경됩니다. 만약 *to/*가 생략된다면 현재 라인의 *from* 텍스트의 첫 번째 발생이 삭제됩니다.

다음 명령의 시퀀스가 현재라인 라인3으로 만들어 집니다. 그리고 WHERE 절에 있는 부문 숫자가 20에서 30으로 변경됩니다.

```
SQL> LIST
  1  SELECT empno, ename, job, sal, comm
  2  FROM emp
  3  WHERE deptno = 20
  4* ORDER BY empno
SQL> 3
  3* WHERE deptno = 20
SQL> CHANGE /20/30/
  3* WHERE deptno = 30
SQL> LIST
  1  SELECT empno, ename, job, sal, comm
  2  FROM emp
  3  WHERE deptno = 30
  4* ORDER BY empno
```

#### 10.1.2.4 CLEAR

CLEAR 명령은 SQL 버퍼의 콘텐츠를 제거합니다. COLUMN명령문과 함께 모든 열 정의 세트를 삭

제하거나 화면을 모두 삭제합니다.

```
CL[EAR ] [ BUFF[ER ] | SQL | COL[UMNS ] | SCR[EEN ] ]
BUFFER | SQL
```

SQL 버퍼를 삭제합니다.

COLUMNS
열의 정의를 삭제합니다.

SCREEN
화면을 삭제합니다. 이 것은 만약 옵션이 지정되어 있지 않으면 기본값입니다.

#### 10.1.2.5 COLUMN

COLUMN 명령은 출력의 형식을 제어 합니다. 그 형식의 속성은 현재 세션의 기간 동안 오직 효과 있는 COLUMN명령문 사용에 의해 설정됩니다.

```
COL[UMN ]
  [ column
    { CLE[AR ] |
      { FOR[MAT ] spec |
        HEA[DING ] text |
        { OFF | ON }
      } [...]
    }
  ]
```

COLUMN 명령이 수반하는 옵션 없이 지정되지 않는다면, 세션에 표시 되어지는 효과에 현재 열에 대한 옵션의 포맷을 지정합니다. COLUMN 열의 이름으로 뒤를 잇고 그 뒤에 다음 중 하나가 될 수 있는 것이 뒤를 잇습니다.

- 다른 옵션이 없음
- CLEAR
- FORMAT, HEADING 의 결합 과 OFF 나 ON 중 하나

*column*
테이블 안의 열의 이름은 후에 서식 옵션을 적용할 수 있습니다.
만약 *column* 이 따르는 다른 옵션이 없는 경우에는 현재 열 형식 옵션을 만약 다른 경우에는 *column* 이 표시 됩니다.

CLEAR
CLEAR 옵션은 *column의* 기본값으로 모든 형식 옵션이 되돌아 갑니다. 만약 CLEAR 옵션이 지정되었다면 오직 지정된 옵션이 되어야 합니다.

*spec*

형식 지정은 *column*에 적용될 것이다. 열의 문자에 대해 *spec*은 다음과 같은 형식을 취합니다.

A*n*

*n*은 데이터를 표시하는 문자에서 열 너비를 지정하는 양의 정수입니다. *n*을 초과하는 데이터는 지정된 너비 주변으로 될 것입니다.

숫자 열의 *spec*은 다음 요소로 구성되어 있습니다.

| Element | Description |
|---|---|
| $ | 달러 기호로 표시합니다. |
| , | 지시된 위치에 콤마를 표시합니다. |
| . | 부호 소수점의 위치를 표시합니다. |
| 0 | 0을 표시 합니다. |
| 9 | 가장 큰 숫자의 개수를 표시합니다. |

**Table 10-1 숫자 열 형식 요소**

만약 큰 숫자의 손실은 형식의 오버플로우 때문에 발생합니다. 그 후 모든 # 표시 됩니다.

text

텍스트는 열의 column heading을 사용해야 합니다.

OFF | ON

만약 OFF로 지정되었다면, 형식 옵션을 기본값으로 돌립니다. 하지만 여전히 그 세션 내에서 변수가 있습니다. 만약 ON으로 지정되었다면, 세션이 다시 활성화 되는 *column* 에 대한 이전의 COLUMN 명령문에 의해 형식 옵션이 지정됩니다.

다음의 예제는 job 열의 표시 너비 변경의 효과를 보여줍니다.

```
SQL> SET PAGESIZE 9999
SQL> COLUMN job FORMAT A5
SQL> COLUMN job
COLUMN   JOB  ON
FORMAT   A5
wrapped
SQL> SELECT empno, ename, job FROM emp;

EMPNO ENAME      JOB
----- ---------- -----
 7369 SMITH      CLERK
 7499 ALLEN      SALES
```

```
                MAN

 7521 WARD       SALES
                MAN

 7566 JONES      MANAG
                ER

 7654 MARTIN     SALES
                MAN

 7698 BLAKE      MANAG
                ER

 7782 CLARK      MANAG
                ER
 7788 SCOTT      ANALY
                ST
7839 KING       PRESI
                DENT
7844 TURNER     SALES
                MAN
7876 ADAMS      CLERK
7900 JAMES      CLERK
7902 FORD       ANALY
                ST

7934 MILLER     CLERK

14 rows retrieved.
```

다음 예시는 sal 열의 형식으로 적용됩니다.

```
 SQL> COLUMN sal FORMAT $99,999.00
SQL> COLUMN
COLUMN   JOB  ON
FORMAT   A5
wrapped

COLUMN   SAL  ON
FORMAT   $99,999.00
wrapped
SQL> SELECT empno, ename, job, sal FROM emp;

EMPNO ENAME      JOB          SAL
----- ---------- ----- -----------
 7369 SMITH      CLERK     $800.00
 7499 ALLEN      SALES   $1,600.00
                MAN

 7521 WARD       SALES   $1,250.00
                MAN

 7566 JONES      MANAG   $2,975.00
                ER

 7654 MARTIN     SALES   $1,250.00
                MAN
```

```
 7698 BLAKE      MANAG   $2,850.00
              ER
 7782 CLARK      MANAG   $2,450.00
              ER

 7788 SCOTT      ANALY   $3,000.00
              ST

 7839 KING       PRESI   $5,000.00
              DENT

 7844 TURNER     SALES   $1,500.00
              MAN

 7876 ADAMS      CLERK   $1,100.00
 7900 JAMES      CLERK     $950.00
 7902 FORD       ANALY   $3,000.00
              ST
 7934 MILLER     CLERK   $1,300.00

14 rows retrieved.
```

#### 10.1.2.6 CONNECT

다른 사용자 및/또는 다른 데이터베이스에 연결은 데이터 베이스 연결을 변경합니다.
CONNECT 명령을 따르는 매개변수 사이에 공백이 없어야 합니다.

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

CONNECT 명령 매개변수는 명령 라인으로부터 EDB*Plus 시작을 위해 *login* 매개변수와 똑같은 의미를 갖고 있습니다. *login* 매개변수에 대한 자세한 내용은 제 10.1.1장을 참조하십시오.

다음의 예시에서는 데이터베이스 연결은 smith, 사용자 이름과 포트5445 로컬호스트 데이터베이스 edb으로 변경됩니다.

```
SQL> CONNECT smith/mypassword@localhost:5445/edb
Connected to EnterpriseDB 8.3.0.10 (localhost:5445/edb) AS smith
```

위에 표시된 세션 내에서 그 연결은 사용자 이름이 enterprisedb로 변경됩니다.
또한, 호스트의 기본값은 로컬호스트로 포트 기본값은 5445(이전에 사용된 포트와 똑같지 않음)로 데이터베이스의 기본값은 edb로 된다는 것에 주의하십시오.

```
SQL> CONNECT enterprisedb/password
Connected to EnterpriseDB 8.3.0.10 (localhost:5444/edb) AS enterprisedb
```

#### 10.1.2.7 DEFINE

DEFINE 명령은 생성되거나 사용자 변수의 값으로 대체됩니다. (또한 대리 변수를 불리웁니다.)

```
DEF[INE ] [ variable [ = text ] ]
```

만약 DEFINE 명령문은 매개변수 없이 주어진다면, 현재의 모든 변수와 그들의 값은 표시됩니다.
DEFINE 변수가 주어졌다면, 변수는 그 값이 표시됩니다.

*DEFINE variable = text*는 *text*로 *변수*를 지정합니다. *Text*는 단일 또는 이중 인용 부호로 선택적으로 닫혀 있을 수도 있습니다.

*Text* 스페이스 문자를 포함된 경우 따옴표를 사용해야 합니다.

다음 예제는 두 변수와 dept와 name을 정의합니다.

```
SQL> DEFINE dept = 20
SQL> DEFINE name = 'John Smith'
SQL> DEFINE DEFINE EDB = "localhost:5445/edb"
DEFINE DEPT = "20"
DEFINE NAME = "John Smith"
```

**주의 :** `EDB` 변수는 Postgres Plus Advanced Server 홈 디렉토리의 하위 디렉토리인 `edbplus` 에 위치한 `login.sql` 파일로 읽혀 집니다.

#### 10.1.2.8 DEL

DEL는 SQL버퍼에서 하나나 그 이상의 라인을 삭제하는 명령 라인 편집기 이다.

```
DEL [ n | n m | n * | n L[AST ] | * | * n | * L[AST ] | L[AST ] ]
```

그 매개변수를 SQL버퍼로부터 삭제되는 라인을 지정합니다.
두 매개변수는 라인이 삭제되는 범위의 시작과 끝을 지정합니다.
만약 DEL 명령문은 매개변수가 주어지지 않고 현재 라인이 삭제됩니다.

다음은 매개변수의 의미입니다.

*n*
*n 은 n*번째 라인으로 정수 표시입니다.

*n m*
*n* 과 *m은* 정수입니다. *m* 은 *n*보다 크고 *n*번째 줄부터 *m* 번째 라인까지 나타납니다.

*
현재 라인

LAST

맨 마지막 라인

다음의 예시는 다섯 번째, 여섯 번째 라인의 열 sal과 comm의 각각 SQL버퍼의 SELECT명령으로부터 삭제됩니다.

```
SQL> LIST
1 SELECT
2 empno
3 ,ename
4 ,job
5 ,sal
6 ,comm
7 ,deptno
8* FROM emp
SQL> DEL 5 6
SQL> LIST
  1 SELECT
  2 empno
  3 ,ename
  4 ,job
  5 ,deptno
  6* FROM emp
```

#### 10.1.2.9 DESCRIBE

DESCRIBE 명령은 열의 목록, 데이터 타입, 테이블의 길이 또는 뷰를 표시합니다.
프로시저 또는 함수에 대한 매개변수의 목록; 프로시저와 함수와 패키지에 대한 각각의 매개변수의 목록

```
DESCRIBE [ schema.]object
```

*Schema*
개체가 포함된 스키마의 이름을 설명한 것입니다.

*Object*
테이블, 뷰, 프로시저, 함수, 패키지의 이름이 표시됩니다.

#### 10.1.2.10 DISCONNECT

DISCONNECT 명령은 현재 데이터베이스의 연결을 닫습니다. 그러나 EDB*Plus 종료 되지 않습니다.

```
DISC[ONNECT ]
```

#### 10.1.2.11 EDIT

EDIT 명령은 운영 체제 파일이나 SQL 버퍼의 컨텐츠를 수정하기 위해 외부 편집기를 호출합니다.

```
ED[IT ] [ filename[.ext ] ]
```

*filename*[.*ext* ]
*filename은* 외부 편집기와 함께 열리기 위한 파일의 이름입니다. *ext*은 파일 확장자입니다.
만약 파일 확장자가 sql일 경우 .sql의 확장자는 *filename*이 지정될 때 생략될 수도 있습니다.
EDIT는 확장자 없이 지정된 파일이름에 .sql 확장자로 항상 가정됩니다.
만약 파일이름 매개변수는 EDIT 명령으로부터 생략 됩니다. SQL 버퍼의 컨텐츠는 편집기로 가져올 수 있습니다.

#### 10.1.2.12 EXIT

EXIT 명령은 EDB*Plus 세션을 종료하고 운영체제로 제어를 되돌립니다.
QUIT는 EXIT의 동이어 입니다. 매개변수를 지정하지 않으면 EXIT SUCCESS COMMIT 하는 것과 같습니다..

```
{ EXIT | QUIT } [ SUCCESS | FAILURE | WARNING | value |variable ] [ COMMIT
| ROLLBACK ]

SUCCESS | FAILURE |WARNING
```

반환 코드를 반환하는 운영 체제에 의존하고 성공적인 실행, 실패나 SUCCESS, FAILURE, 와 WARNING 각각에 대한 경고가 나타나는 반환코드에 의존하는 운영체제를 반환합니다.
기본값은 SUCCESS입니다.

*value*
반환코드로서 그 정수 값이 반환됩니다.

*variable*
변수는 DEFINE 명령과 함께 생성됩니다. 그 값은 반환 코드로서 반환됩니다.

```
COMMIT | ROLLBACK
```

만약 COMMIT 지정되었다면, 커밋 되지 않은 업데이트는 종료 시에 커밋 됩니다.
ROLLBACK가 지정되었다면, 커밋 되지 않은 업데이트는 종료 시에 롤 백 됩니다.
기본값은 COMMIT 입니다.

#### 10.1.2.13 GET

GET 명령 SQL 버퍼에서 주어진 파일의 컨텐츠에 로드 됩니다.

```
GET filename[.ext ] [ LIS[T ] | NOL[IST ] ]
```

*filename*[.*ext* ]
*filename*는 SQL 버퍼에 로드 하는 파일의 이름입니다.

*ext*s는 파일의 확장자입니다. 만약 파일의 확장자가 sql이었다면 .sql 확장자는 *filename*을 지정되었을 때 생략될 수도 있습니다.
GET은 항상 확장자 없이 지정된 파일을 .sql 확장자로 가정합니다.

LIST | NOLIST
만약 LIST을 지정된다면, SQL 버퍼의 컨텐츠는 파일이 로드 되어진 후에 표시되어 집니다.
NOLIST가 지정된다면, 목록 없이 표시 됩니다. 기본값은 LIST입니다.

#### 10.1.2.14 HELP

HELP 명령은 주제의 인덱스를 얻거나 특정 주제의 도움을 줍니다.
물음표는 HELP 지정과 동의어 입니다.

```
{ HELP | ? } { INDEX | topic }
```

INDEX
해당 항목의 인덱스를 표시합니다.

*Topic*
특정 주제의 이름, 예를 들어 EDB*Plus 명령, 이것의 도움이 요구 됩니다.

#### 10.1.2.15 HOST

HOST 명령은 EDB*Plus로부터 운영체제 명령을 실행합니다.

```
HO[ST ] os_command
```

*os_command*
운영체제 명령은 실행됩니다.

#### 10.1.2.16 INPUT

INPUT은 현재 라인 후에 SQL 버퍼에 텍스트의 라인의 추가를 라인 명령 편집기입니다.

```
I[NPUT ] text
```

다음은 INPUT의 시퀀스 명령은 SELECT 명령을 구성합니다.

```
SQL> INPUT SELECT empno, ename, job, sal, comm
SQL> INPUT FROM emp
SQL> INPUT WHERE deptno = 20
SQL> INPUT ORDER BY empno
SQL> LIST
 1 SELECT empno, ename, job, sal, comm
 2 FROM emp
 3 WHERE deptno = 20
 4* ORDER BY empno
```

## *10.2 EDB*Loader*

EDB*Loader는 EnterpriseDB에 대한 고성능 대량 데이트 로더입니다. 그것은 Oracle SQL*Loader 호환 인터페이스를 제공합니다. EDB*Loader는 Oracle SQL*Loader 제어 파일을 읽을 수 있고 명령 라인 매개변수의 하위 집합을 받아들입니다.

### 10.2.1 EDB*Loader 8.3의 기능

- 직접 경로 – 빠른 로딩, 비복구되는 데이터 로딩 기능
- 기존 경로 – 완벽하게 복구할 수 있는 로딩 시설
- Delimiter Separated Values (DSV)를 로드 하는 능력
- 일반 파일 시스템 데이터로부터 너비 항목을 고정을 로드하는 능력
- 로드 프로세스 동안 발생된 오류는 로드 실행을 중지할 수도 있습니다. 그러나 사용자는 복구할 숫자의 정의, 중단되기 전에 치명적인 오류를 허용하지 않는 연속성에 걸친 오류를 요구할 수 있다. (However, users can request continuity across errors by defining the number of recoverable, non fatal errors to allow before aborting.)
- UNIQUE 와 NOT NULL 제한은 직접적 로딩을 하는 동안 실행되어 집니다.
- 전통적인 로딩에 대한 모든 제약은 지원됩니다.
- 규칙, 트리거, 상속은 기본적으로 파티션 실행합니다.

### 10.2.2 이 릴리스에서 지원되지 않는 기능:

- 외래 키 제한은 직접적인 경로 로드 동안 지원되지 않습니다.
- 규칙, 트리거, 상속은 직접적인 경로 로드 동안 지원되지 않습니다.
- 비지 않은 테이블로 직접 경로 로드
- 제어 파일 아네 데이터의 사양
- 열은 기본 값 표현을 갖고 있습니다.

### 10.2.3 명령 라인에서 EDB*Loader 시작

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

데이터 옵션 로드:

**-d DBNAME** 데이터베이스 이름

**-p PORT** 데이터 로드를 위해 매개변수를 연결하기 위한 포트 숫자

**USERID**={username[/password]|/}

데이터베이스를 접속할 때 사용하기 위해 사용자명과 패스워드를 지정합니다. 그것 중 하나를 생략한다면 그 유틸리티는 사용자명과 패스워드에 대한 프롬프트 할 것이다. 만약 "/" 이 사용된다면 데이터베이스는 운영체제 인증을 사용하여 연결을 시도할 것입니다. (i.e. OS 사용자명/패스워드 결합)

**CONTROL=**control_ file_name

제어파일의 절대 경로를 포함할 수 있는 이름을 지정합니다. 이 기본 확장자는 *.ctl* 입니다.

**LOG=**log_file_name

로그 파일의 절대 경로를 포함할 수 있는 이름을 지정합니다. 만약 이 매개변수가 지정되지 않았다면 로그 파일 이름은 .log 확장명에도 불구하고 제어 파일을 따릅니다. 또한 제어 파일로 같은 디렉토리 위치에 작성 될 것입니다. 그 기본 확장자는 .log입니다.

**DIRECT=**{TRUE | FALSE}

로드하는데 사용되는 매커니즘을 가리킵니다. 기존에 기반경로 로드의 결과값은 FALSE 입니다. 로딩의 직접 경로 값은 TRUE입니다.

**ERRORS=**error_count

로드 작업이 중단하기 전에 받아들여질 수 있는 에러의 숫자에 제한을 지정합니다. 그 기본 값인 에러 숫자는 50입니다.

**SKIP=**skip_count

로드 되는 것으로부터 데이터 파일의 삽입의 생략되어야 하는 초기 행 수를 지정합니다.

**SKIP_INDEX_MAINTENCE=**{true|false}

만약 매개변수가 true일 때, 인덱스의 유지 보수 로드 하는 직접 경로는 로드의 일부분으로 진행되지 않습니다. 만약 관련 테이블이 그것에 인덱스가 있다면, 무효로 표시됩니다. 다음의 인덱스 재설정은 그것을 유효하게 만들어 주기 위해 인덱스를 할 필요가 있습니다.

**PARFILE=**param_file_name

매개변수 파일의 절대 정로를 포함할 수 있는 이름을 지정합니다. 이 파일은 기본 확장명이 없습니다.
이 매개변수에 언급 될 수 있었던 명령 라인에 모든 매개변수가 허용됩니다.

**BAD=**bad_file_name

나쁜 파일의 절대 경로를 포함 할 수 있는 이름을 지정합니다.
만약 이 매개변수가 지정되지 않았다면, 그 나쁜 파일의 이름은 *.bad* 확장자로 제어 파일 이름이 허용됩니다. 또한 제어 파일로서 같은 디렉토리 위치로 작성된 것입니다. 기본 확장자는 *.bad*입니다.

**DISCARD=**discard_file_name

삭제된 파일(discard file)의 절대 경로를 포함할 수 있는 이름을 지정합니다. 만약 이 매개변수가 지정되지 않았다면 삭제된 파일 이름은 *.dsc* 확장자와 함께 제어 파일을 따릅니다. 또한 제어 파일과 같은 디렉토리 위치로 작성된 것입니다. 기본 확장자는 *.dsc* 입니다.

any 관련 테이블로 로드 할 수 없는 오직 그 레코드입니다. 만약 이 매개변수가 지정되지 않았다면 삭제된 파일 이름은 *.dsc* 확장자와 함께 제어 파일을 따립니다. 또한 제어파일과 같은 디렉토리 위치로 작성된 것입니다. 기본 확장자는 *.dsc* 입니다. (Only those recodrd which cannot be loaded into **any** of the involved tables, because of failure to match the corresponding WHEN clause, will be logged into this file. )
이와 같은 기록은 백엔드에 의해 거부되지 않지만 로딩 선택적 로드 논리에 의해서 된다는 것에 주의하십시오.( Note that such records are not rejected by the backend, but by the selective loading logic only. )

### 10.2.4 EDB*Loader Examples

#### 10.2.4.1 Basic Syntax

```
load data
infile '/tmp/mydata.csv'
badfile '/tmp/mydata.bad'
discardfile '/tmp/mydata.dsc'
into table emp
fields terminated by “,” optionally enclosed by '”'
(empno, empname, sal, deptno)
```

#### 10.2.4.2 구분된 열 구문

Column 구문, FILLER 키워드 플러스. Filler 열은 사용자가 원치 않은 필드를 무시하는데 사용됩니다.

```
load data
insert into table T1
fields terminated by ','
( field1, field2 FILLER, field3)
```

#### 10.2.4.3 Fixed Width Column Syntax

구문 사양 폭이 고정됩니다. 모든 새로운 라인이 항목을 구분하여 문자의 위치 필드에 대한 값을 확인하는 예입니다.

```
load data
insert into table T1
( field1 POSITION (1:2),
  field2 FILLER POSITION (4:8),
  field3 POSITION (60:80)
)
```

#### 10.2.4.4 Multiple Table Syntax

똑 같은 ctl 파일의 복수 테이블에 로드 되는 능력이 선택적 로딩으로 사용되는 예입니다.

```
load data
infile '/tmp/mydata.csv'
append into table emp1 when (1:3) = '100'
(empno POSITION (1:3), sal POSITION (5:7), deptno POSITION (9:11))
append into table emp2 when (1:3) = '200'
(empno POSITION (1:3), sal POSITION (5:7), deptno POSITION (9:11))
Here any record which does not match any of the WHEN clauses will be put
into
the corresponding discardfile
```

### 10.2.5 Notes

1. TRAILING NULLCOLS 지원

만약 이것이 지정되면 입력 데이터 파일의 populated 될 수 없는 어느 열을 따라서 다음과 같은 열이 null로 표시됩니다. (any following columns that cannot be populated from the input data file will be marked as null. e.g.)

```
Into table emp trailing nullcols
(empno, sal, deptno)
```

만약 그 입력 행이 다음과 같은 것을 포함되어 있는 경우

```
1,2
```

그리고 deptno 필드는 현재 입력에 대해 NULL 값이 할당 될 것입니다.

2. INSERT에 따라서 TRUNCATE/REPLACE 키워드가 지원됩니다. 이러한 키워드가 사용하는 경우, 관련 테이블은 로드가 시작하기 전에 종료 될 것입니다.

```
Load data TRUNCATE into table T1 ..
Load data REPLACE into table T1..
```

3. 이식이 가능한 데이터 타입과 같은 문자를 지원합니다. 각 필드 마다 데이터 타입은 지정될 수 있습니다. 지정은 삽입 데이터에 발행하는 컨텐츠 필드에 해석에 대해서 결정됩니다. 예를 들어 봅시다.
만약 INTEGER 데이터 타입이라면 그 데이터 파일은 2진수 형식의 파일을 담고 있습니다. 기본 해석은 문자열이고 1단계에 대해서만 이러한 데이터 형식을 지원할 것입니다.

- CHAR
- INTEGER EXTERNAL
- FLOAT EXTERNAL
- ZONED EXTERNAL
- DATE

## *10.3 동적 실행 최적화*

동적 실행 최적화는 새로운 카탈로그 뷰에 DBA를 허용하고 무엇이 개별적인 세션이나 시스템 전체에 영향을 미치는 "이벤트 대기"인지 결정합니다.
이벤트의 횟수는 발생할 뿐만 아니라 수집되는 기다리는 시간도 보냈습니다. 비율로서 명령을 내림차순으로 기다립니다. 이것을 한눈에 보는 것이 가능합니다. 이벤트는 실행에 영향력을 행사하고 올바른 조치를 취합니다.
이 기능은 발생하므로 기다리는 이벤트의 숫자를 수집합니다. 실제 시스템의 오버헤드를 최소화(검색 없이)하고 매우 작은 발자국과 함께 모듈 방법으로 구현됩니다. 이 기능은 향후 릴리스에 강화될 것으로 예상됩니다.

### 10.3.1 DRI 활성화

타이밍 데이터, 새로운 초기 매개변수, timed_statistics 의 모음의 스위치 하기 위해 추가 되었습니다.
이것은 postgresql.conf 파일에서 세션뿐만 아니라 시스템 레벨을 설정할 수 있는 동적 매개변수 입니다.
유효한 값은 TRUE 또는 FALSE입니다. FALSE가 기본값이 되는 것과 함께
(The valid values are TRUE or FALSE, with FALSE being the default.)
데이터는 각 프로세스의 시작과 끝 그리고 델타가 계산될 때 수집됩니다. 이 때, 각각의 명령문 데이터는 수집되지 않습니다.

### 10.3.2 Catalog views

#### 10.3.2.1 SYSTEM_WAITS

기다리는 횟수와 각 이름을 기다리는 세션 당 총 기다리는 시간을 요약합니다.
(This summarizes the count of waits and the total wait time per session for each wait name.)

또한 기다리는 숫자의 평균과 최대값을 표시합니다.

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

#### 10.3.2.2 SESSION_WAITS

이것은 SYSTEM_WAITS에 의해 요약되는 자세한 데이터 입니다.

```
Column      |     Type      | Modifiers Definition
 ----------------+---------------+-----------+----------------
  backend_id     | bigint       |    session identifier
  wait_count     | bigint       |    number of times the event occur
  avg_wait_time   | numeric      |  average wait time in microseconds
  max_wait_time   | numeric(50,6) |  maximum wait time in microseconds
  total_wait_time | numeric(50,6) |  total wait time in microseconds
  wait_name      | text         |    name of the event
```

#### 10.3.2.3 SESSION_WAIT_HISTORY

이것은 세션에 대해 지난 25 wait 이벤트가 포함 되어 있습니다.

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

아래의 예를 참조 :

```
select * from public.system_waits;
wait_count | avg_wait | max_wait | total_wait |          wait_name
 -----------+----------+----------+------------+---------------------------
---
--
      128 | 0.000999 | 0.006211 |   0.147253 | db file random read
       44 | 0.001440 | 0.020071 |   0.121372 | query plan
       52 | 0.000077 | 0.000211 |   0.003856 | sql parse
      574 | 0.000000 | 0.000001 |   0.000038 | first buf mapping lock
acquire
      157 | 0.000000 | 0.000001 |   0.000022 | xid gen lock acquire
```

```
      128 | 0.000000 | 0.000001 |   0.000012 | buffer free list lock
acquire
       14 | 0.000000 | 0.000000 |   0.000000 | clog control lock acquire
(7 rows)

select * from public.session_waits;
backend_id | wait_count | avg_wait_time | max_wait_time | total_wait_time |
wait_name
-----------+------------+---------------+---------------+-----------------
+--
-------------------------------
 11222 |        43 |     0.003658 |     0.020071 |       0.157311
|query plan
    11222 |        87 |     0.001488 |     0.006211 |       0.129423 |
db file random read
    11222 |        53 |     0.000078 |     0.000211 |       0.004113 |
sql parse
    11350 |         6 |     0.000612 |     0.001760 |       0.003671
|query plan
    11350 |         6 |     0.000102 |     0.000193 |       0.000609 |
sql parse
    11354 |         2 |     0.000209 |     0.000346 |       0.000418
|query plan
    11354 |         1 |     0.000063 |     0.000063 |       0.000063 |
sql parse
    11222 |       537 |     0.000000 |     0.000001 |       0.000038
|first buf mapping lock acquire
    11222 |       157 |     0.000000 |     0.000001 |       0.000021 |
 xid gen lock acquire
    11222 |        87 |     0.000000 |     0.000001 |       0.000009
|buffer free list lock acquire
    11350 |       120 |     0.000000 |     0.000001 |       0.000008
|first buf mapping lock acquire
    11350 |        19 |     0.000000 |     0.000001 |       0.000001 |
xid gen lock acquire
    11349 |         1 |     0.000001 |     0.000001 |       0.000001
|first buf mapping lock acquire
    11222 |        14 |     0.000000 |     0.000000 |       0.000000
|clog control lock acquire
    11354 |         7 |     0.000000 |     0.000000 |       0.000000
|first buf mapping lock acquire
    11354 |         4 |     0.000000 |     0.000000 |       0.000000 |
xid gen lock acquire
    11349 |         1 |     0.000000 |     0.000000 |       0.000000 |
xid gen lock acquire
(17 rows)

      select * from public.session_wait_history;
       backend_id | seq |          wait_name          | elapsed | p1 | p2 |
      p3
      ------------+-----+-----------------------------+---------+----+-
      ---+----
           11222 |   1 | first buf mapping lock acquire |       0 |  0 |  0
      |  0
           11222 |   2 | first buf mapping lock acquire |       0 |  0 |  0
      |  0
           11222 |   3 | first buf mapping lock acquire |       0 |  0 |  0
      |  0
           11222 |   4 | first buf mapping lock acquire |       0 |  0 |  0
```

```
|  0
     11222 |   5 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |   6 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |   7 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |   8 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |   9 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  10 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  11 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  12 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  13 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  14 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  15 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  16 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  17 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  18 | xid gen lock acquire           |      0 |  0 |  0
|  0
     11222 |  19 | query plan                     |   5694 |  0 |  0
|  0
     11222 |  20 | xid gen lock acquire           |      1 |  0 |  0
|  0
     11222 |  21 | sql parse                      |    164 |  0 |  0
|  0
     11222 |  22 | xid gen lock acquire           |      0 |  0 |  0
|  0
     11222 |  23 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  24 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11222 |  25 | first buf mapping lock acquire |      1 |  0 |  0
|  0
     11550 |   1 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11550 |   2 | xid gen lock acquire          |      0 |  0 |  0 |
0
     11550 |   3 | empty wait event              |      0 |  0 |  0 |
0
     11550 |   4 | empty wait event              |      0 |  0 |  0 |
0
 11550 |   5 | empty wait event                  |      0 |  0 |  0 |
0
     11550 |   6 | empty wait event              |      0 |  0 |  0 |
0
     11550 |   7 | empty wait event              |      0 |  0 |  0 |
0
     11550 |   8 | empty wait event              |      0 |  0 |  0 |
0
```

```
     11550 |   9 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  10 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  11 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  12 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  13 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  14 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  15 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  16 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  17 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  18 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  19 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  20 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  21 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  22 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  23 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  24 | empty wait event              |      0 |  0 |  0 |
0
     11550 |  25 | empty wait event              |      0 |  0 |  0 |
0
     11554 |   1 | xid gen lock acquire          |      1 |  0 |  0 |
0
     11554 |   2 | query plan                    |     40 |  0 |  0 |
0
     11554 |   3 | xid gen lock acquire          |      0 |  0 |  0 |
0
     11554 |   4 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |   5 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |   6 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |   7 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |   8 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |   9 | sql parse                     |     30 |  0 |  0 |
0
     11554 |  10 | xid gen lock acquire          |      0 |  0 |  0
|  0
     11554 |  11 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |  12 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |  13 | first buf mapping lock acquire |      0 |  0 |  0
```

```
|  0
     11554 |  14 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |  15 | first buf mapping lock acquire |      1 |  0 |  0
|  0
     11554 |  16 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |  17 | first buf mapping lock acquire |      1 |  0 |  0
|  0
     11554 |  18 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |  19 | xid gen lock acquire           |      0 |  0 |  0
|  0
     11554 |  20 | query plan                     |    486 |  0 |  0
|  0
     11554 |  21 | first buf mapping lock acquire |      1 |  0 |  0
|  0
     11554 |  22 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |  23 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |  24 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11554 |  25 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11555 |   1 | xid gen lock acquire           |      1 |  0 |  0 |
0
     11555 |   2 | query plan                     |     63 |  0 |  0
|  0
     11555 |   3 | xid gen lock acquire           |      0 |  0 |  0 |
0
     11555 |   4 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11555 |   5 | sql parse                      |     59 |  0 |  0 |
0
     11555 |   6 | xid gen lock acquire           |      0 |  0 |  0 |
0
     11555 |   7 | first buf mapping lock acquire |      0 |  0 |  0
|  0
     11555 |   8 | xid gen lock acquire          |      0 |  0 |  0 |
0
     11555 |   9 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  10 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  11 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  12 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  13 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  14 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  15 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  16 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  17 | empty wait event             |      0 |  0 |  0 |
0
```

```
     11555 |  18 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  19 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  20 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  21 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  22 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  23 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  24 | empty wait event             |      0 |  0 |  0 |
0
     11555 |  25 | empty wait event             |      0 |  0 |  0 |
0
(100 rows)
```

### 10.3.3 DRITA 스크립트를 포함한 사용

1. 이 정보를 저장하기 위해 생성될 필요가 있는 함수와 테이블이 있습니다. 그 후 Statspack와 같은 방식으로 보고 합니다.

2. 첫 번째로 정보를 기다리기 위해 시퀀스와 4테이블이 생성되는 snap_tables.sql을 실행합니다. (First, run snap_tables.sql, which creates a sequence and 4 tables to hold wait information.)
3. 8가지 함수를 생성하는 snap_functions.sql 실행합니다.
4. 그 프로세스는 다음과 같다 :
5. Select * from edbsnap();
6. 이 함수는 스냅샷을 시작합니다.
7. 작업(workload)을 실행합니다.
8. Select * from edbsnap();
9. 이 것은 This will take and end snapshot.

```
Select * from get_snaps();
```

이 함수는 찍은 시간과 스냅 ID의 목록을 반환합니다. 지금 하나 나 그 이상의 보고 함수를 실행합니다.

```
select * from sys_rpt(14,15,10) as "System Waits";
```

기다리는 시스템 정보를 되돌립니다.( This returns system wait information.) 인수 1을 삽입합니다. 스냅 ID 2를 시작합니다. 스냅ID와 3를 종료합니다. 위에서 n 번째 행으로 돌아갑니다.

```
select * from sess_rpt(14,15,10) as "Session Waits";
```

기다리는 세션의 정보를 되돌립니다. 인수 1을 삽입합니다. 스냅ID 2를 시작합니다. 스냅ID와 3을 종료합니다. 위에서 n번째 행으로 되돌립니다.

```
select * from sessid_rpt(14,15,10) as "Backend Session Waits";
```

이 함수는 특정 백엔드에 대해 기다리는 세션의 보고서를 제시합니다. 인수 1을 삽입합니다. 스냅ID 2가 시작됩니다. 스냅ID와 3은 종료됩니다. backend_id to return

```
select * from sesshist_rpt (14,15) as "Backend SessHist Waits";
```

이 함수는 특정한 백엔드가 기다리는 세션의 이력의 보고서를 제시합니다. 인수 1을 삽입합니다. 스냅ID 2를 시작합니다. backend_id to return

Blks 열(p3)의 합계는 블록이 읽어지는 숫자 입니다. 현재 이것이 항상 1이 될 것입니다. 다중 블록 읽기는 후에 릴리스에 유효하게 될 것입니다.

사용할 수 있는 모든 스냅 테이블을 제거 –

```
Select * from truncsnap();
```

스냅 테이블 안에 스냅샷의 범위를 제거 –

```
select * from purgesnap(20,21) as "Purging range of snapshots";
```

인수는 : 1. 열 2에서 시작. 열의 끝

#### 10.3.4 개별 이벤트 설명

**wal write :** 백그라운드 프로세스가 버터에 로그 데이터를 쓰는 경우 발생합니다. 일반적으로 값이 높을 것이라 기대하고 있습니다.

**wal flush:** 버퍼를 디스크로 플러시할 때 발생합니다.

**wal file sync:** Occurs when log is being synced to disk.
wal_sync_method 매개변수에 관련이 있는 것 = fsync
더 나은 성능은 open_sync에 이 매개변수를 변경하는 것에 의해 얻을 수 있습니다.

**wal insert lock acquire :** WAL에 쓰는 동안 short-term을 (The short-term lock that is held while writing to the WAL.) 높은 숫자는 wal 버퍼가 너무 작다는 것을 나타낼지도 모릅니다.

**wal write lock acquire:** WAL이 플러싱할 때(The lock that is held when flushing the WAL.) 이것은 wal 플러쉬와 관련 있습니다.

**control file lock acquire:** 파일이 업데이트 되는 것을 제어할 때 발생합니다. 일반적으로 낮은 숫자가 되어야 합니다.

**checkpoint start lock acquire :** 백엔드 프로세스가 체크포인트에 bgwriter 프로세스를 명령할 때 발생합니다.

**clog control lock acquire:** 커밋 로그와 체크포인트에 발생합니다.
**bgwriter communication lock acquire:** bgwriter는 백엔드 프로세스에 전달하는 동안 발생합니다.
**db file index read :** 인덱스를 읽는 동안 발생합니다.
**db file table read :** 테이블을 읽는 동안 발생합니다.
**db file index write :** 인덱스를 쓰는 동안 발생합니다.
**db file table write :** 테이블을 쓰는 동안 발생합니다.
**db file sync :** 백엔드 프로세스 fsync에 파일에 알리는 것이 발생합니다.
This occurs if the backend process has to tell a file to fsync.
**db file extend :** 만약 새로운 블록이 확장의 부분으로 파일 끝 부분에 추가할 필요가 있다면 발생합니다.
**sql parse :** 이것은 명령문의 분석하는 동안 발생합니다.
**query plan :** 명령문은 결정되어진 실행을 계획하는 동안 발생합니다.
**db file random read** – n/a
**lwlock acquire** – n/a
**proc array lock acquire** – n/a
**buffer free list lock acquire** – n/a
**xid gen lock acquire** – n/a
**first buf mapping lock acquire** – n/a
**db file random write** – n/a
**sinval lock acquire** – n/a
**freespace lock acquire** – n/a
**checkpoint start lock acquire** – n/a
**btree vacuum lock acquire** – n/a
**shmem index lock acquire** – n/a
**subtrans control lock acquire** – n/a
**multi xact offset lock acquire** – n/a
**rel cache init lock acquire** – n/a
**multi xact member lock acquire** – n/a

### 10.3.5 추천 튜닝

상위 5 이벤트는 검토 되야 하고 불균형하게 전체에서 큰 비중을 차지하는 것을 검토합니다.
잘 수행되는 시스템에서 사용자 I/O는 기다리는 가장 큰 숫자를 만듭니다. 그러나 그 기다림은 CPU 사용량과 총 시간의 맥락에서 평가 되야 합니다.

다른 말로 하면, 이벤트는 2시간 간격으로 전체 측정의 경우 2분 정도 걸릴 때, 나머지 시간의 CPU 시간을 소모하는 곳에, 이벤트는 지정되지 않을 수 있습니다.
(In other words, an event may not be significant if it takes 2 minutes out of a total measurement interval of 2 hours, where the rest of the time is consumed by CPU time.)

CPU "work" 시간이나 다른 "wait" 시간 응답시간의 구성요소 무엇이든지 상당한 시간이 전체 시간의 비율입니다. 적합성에 대해 보고해야 합니다.
Whichever component of response time, CPU "work" time or some other "wait" time, is a significant percentage of overall time, should be looked at for it's appropriateness.

예를 들어 많은 양의 CPU 시간이 메모리 객체를 교환하는데 메모리가 너무 작은 크기 때문에 사용 될 수 있습니다. (For example, a large amount of CPU time may be inefficiently used to swap objects into and out of memory because the memory has been sized too small.)

물론 모든 성능 검토는 하드웨어, OS, 네트워크, 특히 애플리케이션 sql 명령문을 포함해야 합니다. 이벤트에 대한 특히 높은 읽기단계에 기초한 튜닝 단계를 추천합니다.
(Recommendations on tuning steps based on particularly high readings for above events. )

1. Checkpoint waits : 체크포인트 매개변수는 checkpoint_segments 와 checkpoint_timeout 같이 조정해야 할 필요가 있다는 것을 가리킵니다.

2. WAL-related waits: 너무 작다는 것을 가리킵니다.

3. sql parse: 만약 높으면 준비된 명령문 사용을 시도합니다.

4. db file random read: 만약 높으면 적당한 인덱스와 통계가 존재하는 것을 검사합니다.

5. db file random write : 만약 높다면 bgwriter_delay 감소할 필요가 있습니다.

6. btree random lock acquire : 인덱스는 다시 작성되고 있다는 것을 나타냅니다. 적은 활동 시간 동안 다시 작성 합니다.

**10.3.6 추가 기능과 8.3릴리스에 추가된 테이블**
8.3릴리스에 대해 추가된 5 “snap” 테이블과 추가된 6 기능이 추가되었습니다.
Oracle Statspack/AWR 보고서에 실험되기 위해 사용될 수 있습니다.

스냅샷을 할 때, 몇몇 시스템 카탈로그 테이블로부터 실행 데이터는 이력 테이블로 저장됩니다.

Catalog Table  New DRITA Table

pg_stat_database → edb$stat_database
pg_stat_all_tables → edb$stat_all_tables
pg_stat_io_tables → edb$statio_all_tables
pg_stat_all_indexes → edb$stat_all_indexes
pg_statio_all_indexes → edb$statio_all_indexes

각각 새로운 테이블과 함께 연결되는 기능을 새롭게 보고 됩니다.
그 함수는 주어진 스냅샷 사이에 deltas에 보고 됩니다.

- stat_db_rpt
- stat_tables_rpt
- statio_tables_rpt
- stat_indexes_rpt
- statio_indexes_rpt

개별적으로 실행될 수 있거나 Edbreport 함수를 실행할 수 5가지 함수 모두 호출할 수 있습니다.
개별적으로 호출 될 때 다음의 인자를 가지고 있는 기능 :
stat_db_rpt : bid INT 과 eid INT

이러한 값은 시작과 스냅샷 ID로 마감을 하게 됩니다.

```
Usage: select * from stat_db_rpt(22,23);
```

Stat_tables_rpt, statio_tables_rpt, stat_indexes_rpt, statio_indexes_rpt : 이러한 기능은 같은 인수를 취합니다.

스냅 ID 시작(bid)

스냅 ID 끝 (eid)

행 번호를 반환 Topn

시작 테이블 타입 – 유효한 값은 ALL, USER 또는 SYS. SYS 테이블/인덱스 또는 USER 중 하나, 또는 ALL 와 관련된 선택 데이터 사용을 허용합니다.

```
Usage: select * from stat_tables_rpt(22,23,10,"ALL");
```

스냅샷 22와 23 사이 기간에 pg_stat_all_tables로부터 테이블 통계의 상위10 delta를 보여줄

것입니다.

Edbreport 함수는 다른 기능에서 모든 데이터와 추가적으로 시스템 정보를 포함하고 있습니다. 오직 인수만 시작되고 스냅 ID를 마감했습니다.

행과 stat 타입의 숫자에 제어 topn 함수의 시작에 컨텐츠의 설정 (There are 2 constants set in the beginning of the function that control topn number of rows and stat type.)
10과 ALL 각각 있습니다. 그러나 설정을 변경할 수 있습니다.

```
Usage: select * from edbreport(22,23);
```

# 11 Appendix

## *11.1 Acknowledgements*

The PostgreSQL 8.3 Documentation provided the baseline for portions of this Oracle Compatibility Developer's Guide that is common to PostgreSQL, and is hereby acknowledged: Portions of this EnterpriseDB™ Software and Documentation may utilize the following copyrighted material, the use of which is hereby acknowledged. PostgreSQL Documentation, Database Management System PostgreSQL is Copyright © 1996-2007 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below. Postgres95 is Copyright © 1994-5 by the Regents of the University of California. Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies. **IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.**