

제 0 장 소개

목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 관계형 데이터베이스의 이론적, 물리적인 특징을 토의합니다.
- 오라클 제품군에서 **SQL**과 **PL/SQL**이 어떻게 사용되는지를 기술합니다.

과정 목적

본 과정에서는 RDBMS(Relational Database Management System)와 ORDBMS(Object Relational Database Management System)를 이해 하도록 합니다. 또한 다음을 소개 합니다.

- 오라클의 특정한 SQL 문장
- SQL과 PL/SQL 실행을 위해서 그리고 포매팅과 리포팅을 위해서 사용되는 SQL*Plus
- 오라클의 절차적 언어인 PL/SQL

관계형 데이터베이스 개념

- E. F. Codd 박사가 1970년 데이터베이스 시스템에 대한 관계형 모델을 제안했습니다.
- 이것이 RDBMS(Relational Database Management System)의 시초입니다.
- 관계형 모델은 다음으로 구성됩니다.
 - 개체(object) 혹은 관계(relation)의 집합
 - 관계(relation)에 가해지는 연산자의 집합
 - 정확성과 일관성을 위한 데이터 무결성

관계형 모델

관계형 모델의 원칙은 1970 년 6 월, E. F. Codd 박사의 “A Relational Model of Data for Large Shared Data Banks” 라는 논문에서 처음으로 윤곽을 드러내었고, 이 논문에서 Codd 박사는 [데이터베이스 시스템](#) 에 관계형 모델 도입을 제안했습니다.

그 당시 많이 사용되었던 대중적 모델은 계층형, 네트워크형, 심지어는 단순형태의 파일 데이터 구조 등이었으나 RDBMS (Relational Database Management Systems)가 구조상의 유연성이라든가, 사용상의 편의로 인해 쉽게 대중화 되었습니다. 게다가, 총체적 해결을 제공하는 강력한 응용프로그램 개발과 사용자 제품 군들을 RDBMS 로 가능하게 해주는 Oracle 같은 혁신적인 업체들이 다수 있었습니다.

관계형 모델의 구성요소

- 데이터를 저장하는 관계들과 개체들의 모음
- 다른 관계를 생성하기 위해 관계에 가해지는 일련의 연산자 집합
- 정확성과 [일관성](#)을 위한 데이터 무결성

보다 자세한 정보를 알고자 한다면, 다음을 참조하십시오.

E. F. Codd, *The Relational Model for Database Management Version 2* (Reading, Mass.: Addison-Wesley, 1990).



관계형 데이터베이스

관계형 데이터베이스는 정보 저장을 위해 관계나 2차원 테이블을 이용합니다.

예를 들어, 회사는 그 회사 직원들에 관한 정보를 저장할 필요가 있는데, 관계형 데이터베이스에서는 직원 테이블, 부서 테이블, 급여 테이블 등과 같은 직원들에 대한 각종 정보들을 저장하기 위해 여러 테이블을 생성하게 됩니다.



ER 모델링

실제 시스템에서 데이터는 엔티티(entity) 단위로 분류되어 나누어져 있습니다. ER(Entity Relationship)모델은 업무상의 다양한 엔티티(entity)와 그들간의 관계(relationship)를 그림으로 나타낸 것입니다. ER 모델은 업무 요구나 사양으로부터 유출되며 시스템 개발 생성 주기 단계중 분석 단계에서 만들어집니다. ER 모델은 어떤 한 업무내에서 수행되는 활동이 요구하는 정보를 분리시켜 줍니다. 비록 업무는 그들의 활동에 변화를 가할 수 있지만 정보의 형태는 그대로 남으려는 경향이 있습니다. 그러므로 데이터 구조도 변함없이 그대로 있으려고 합니다.

ER 모델링의 장점

- 조직에 대한 정보를 정확하고 자세하게 문서화
- 정보 요구사항의 범위를 명확히 기술
- 데이터베이스 설계를 쉽게 이해할 수 있는 표본 제공
- 복수 응용프로그램의 통합화를 위한 효과적 프레임워크 제공

주요 구성요소

- 엔티티(entity): 알려져야 하는 어떤 정보에 대한 중대한 것. 예를 들면 부서, 직원, 서열 등이 있습니다.
- 속성(attribute): 엔티티를 기술하거나 한정하는 것들. 예를 들면, 직원 엔티티에 관한 속성은 직원번호, 이름, 근무지역, 고용일자, 부서번호 등이 될 수 있습니다. 각 속성들은 필수적일 수도 있고 선택적일 수도 있어서 이를 optionality 이라 부릅니다.
- 관계(relationship): optionality와 degree를 보여 주는 엔티티간의 명명된 관계로서 예를 들면, 직원과 부서 그리고 주문과 항목 등이 있습니다.

관계형 데이터베이스 용어

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7698	JONES	MANAGER	7839	02-APR-81	2975		20
7664	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7655	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7644	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7566	JAMES	CLERK	7698	03-DEC-81	950		30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7562	FORD	ANALYST	7566	03-DEC-81	3000		20
7565	SMITH	CLERK	7562	17-DEC-80	800		20
7568	SCOTT	ANALYST	7566	09-DEC-82	3000		20
7576	ADAMS	CLERK	7568	12-JAN-83	1100		20
7584	MILLER	CLERK	7582	23-JAN-82	1300		10

관계형 데이터베이스에서 사용되는 용어

관계형 데이터베이스는 한 개 이상의 테이블을 가지고 있습니다. 테이블은 RDBMS에서의 기본 저장 구조입니다. 테이블은 현실속의 어떤 사물에 대해 필요한 정보를 지니고 있는데? 예를 들면, 고용인, 송장, 고객 등입니다.

위의 슬라이드는 EMP table 또는 relation을 보여 주고 있습니다. 각 숫자는 다음을 의미합니다:

1. 특정 종업원에 대한 모든 데이터를 나타내는 단일 row 또는 tuple입니다. 테이블에서 각 행은 중복되지 않는 행을 허용하는 기본 키(primary key)에 의해 식별되어야 합니다. 행의 순서는 무의미합니다. 데이터를 읽어 들일 때 행 순서를 지정할 수 있습니다.
2. 기본 키(primary key)인 종업원 번호를 포함하는 열 (column) 또는 속성 (attribute)입니다. 종업원 번호는 EMP 테이블에서 유일한(unique) 종업원을 식별합니다. 기본 키(primary key)는 값을 꼭 포함해야 합니다.
3. 키 값이 아닌 열입니다. 열은 테이블에서 한 종류의 데이터를 나타냅니다. 위의 예에서는 모든 종업원에 대한 업무명(JOB)이 해당됩니다. 열 순서는 데이터 저장 시에는 무

의미합니다. 데이터를 읽어 들일 때 열 순서를 지정하십시오.

4. 부서번호를 포함하는 열은 **외래 키(foreign key)**입니다. **외래 키**는 테이블 간에 서로 어떻게 관련되었는가를 정의합니다. **외래 키**는 다른 테이블의 기본 키 또는 고유 키를 참조합니다. 위의 예에서, DEPTNO는 DEPT 테이블에서 부서를 unique 하게 식별합니다.

5. 필드(field)는 행과 열의 교차되는 곳에 발견될 수 있습니다. 그 안에는 하나의 값만이 존재합니다.

6. 필드는 그 안에 값을 가지지 않을 수도 있습니다. 이것은 null value라 불립니다. EMP 테이블에서 영업 사원인 종업원만이 COMM (commission) 필드에서 값을 가집니다.

주: Null 값은 후속 장에서 자세히 다룹니다.

관계형 데이터베이스 속성

관계형 데이터베이스는

- **SQL(Structured Query Language)** 문을 사용하여 이용되고 수정되어 집니다.
- 물리적 포인터가 없는 테이블들의 모음을 가지고 있습니다.
- 일련의 연산자를 사용합니다.

관계형 데이터베이스 속성

관계형 데이터베이스에서는 테이블로의 액세스 루트를 지정하지 않으며 또 물리적으로 데이터가 어떻게 나열되는지 알 필요가 없습니다.

데이터베이스를 이용하기 위해서는, 관계형 데이터베이스를 가동시킬 수 있는 **ANSI**(American National Standards Institute) 표준 언어인 **SQL**(Structured Query Language)문을 실행해야 합니다. 이 언어는 관계(relation)를 조합, 분리 시킬 수 있는 일련의 큰 연산자 집합을 지니고 있습니다. 데이터베이스는 SQL문 사용으로 수정도 할 수 있습니다.

SQL을 사용한 RDBMS와의 통신

SQL statement
is entered

```
SQL> SELECT loc  
2 FROM dept;
```

Statement is sent to
database

Database

Data is displayed

```
LOC  
-----  
NEW YORK  
DALLAS  
CHICAGO  
BOSTON
```

SQL 문장

SQL 로 서버와 통신이 가능하며 다음의 장점 등을 지니고 있습니다:

- 효율적입니다.
- 배우기 쉽고 사용하기 쉽습니다.
- 기능적으로 완전합니다. SQL은 테이블에 있는 데이터를 정의, 검색, 조작 할 수 있게 해 줍니다.

Oracle8: 객체 관계형 데이터베이스 관리 시스템

- 사용자가 정의한 데이터 타입과 객체
- 완전히 호환 가능한 관계형
- 대형 객체와 멀티미디어 지원
- 고품격 데이터베이스 서버 특성

Oracle8 에 대하여

Oracle8은 오라클에 의해 개발된 첫 번째 객체가능 데이터베이스 입니다. 이것은 새로운 ORDBMS를 지원 하기 위하여 Oracle7의 **데이터 모델링** 능력을 확장합니다. Oracle8은 객체 지향 프로그래밍, 복잡한 데이터 타입, 복잡한 비즈니스 객체, 실세계와 완전히 양립 가능한 새 엔진을 제공합니다.

Oracle8은 많은 면에서 Oracle7을 확장합니다. 런타임 데이터구조를 더 많이 공유하고, 더 큰 버퍼 캐쉬를 잡고, 제약조건을 지연 가능케 하는 등 OLTP(Online Transaction Processing)의 기능성과 성능을 향상시킵니다. 데이터 웨어하우스 어플리케이션은 삽입, 갱신, 삭제의 병렬 수행, 분할, 병렬 인식질의 최적화 같은 성능 향상 측면에서 장점이 있습니다. NCA(Network Computing Architecture) 프레임워크에서 작동중인 Oracle8은 분산 되고(distributed) 다중으로 고정된(multitiered) **클라이언트-서버**와 웹 기반 어플리케이션을 지원합니다.

Oracle8은 수십만의 동시 사용자를 조정할 수 있고 512 petabytes 까지 지원하며 전통적으로 구조화된 데이터 뿐만 아니라 텍스트, 공간 이미지, 사운드, 비디오, 일련의 시간을 포함하는 어떤 타입의 데이터도 처리할 수 있습니다.

자세한 내용을 알고자 한다면, 다음을 참조하십시오.

Oracle Server Concepts Manual, Release 7.3.

Oracle Server Concepts Manual, Release 8.0.

SQL 문장

SELECT	데이터 검색
INSERT UPDATE DELETE	DML(Data manipulation language)
CREATE ALTER DROP RENAME TRUNCATE	DDL(Data definition language)
COMMIT ROLLBACK SAVEPOINT	트랜잭션 제어
GRANT REVOKE	DCL(Data control language)

SQL 문장

오라클 SQL 은 산업승인표준을 따릅니다. 오라클 회사는 SQL 표준 위원회에 주요 인사를 적극적으로 포함시켜, 변화, 발전하는 표준에 대한 미래 수용을 보증합니다. 산업승인 표준위원회는 [ANSI](#)(American Standards Institute)와 ISO(International Standards Organization) 입니다. [ANSI](#) 와 ISO 는 관계형 데이터베이스 표준 언어로 SQL 을 채택했습니다.

문장	설명
SELECT	데이터베이스로부터 데이터 검색
INSERT UPDATE DELETE	개별적으로 데이터베이스 테이블에서 새로운 행을 입력하고 기존의 행을 변경하고, 원치 않는 행을 제거합니다. 총괄하여 DML (data manipulation language)이라 부릅니다.
CREATE ALTER DROP RENAME TRUNCATE	테이블로부터 데이터 구조를 생성, 변경, 제거를 하며 DDL (data definition language)이라 부릅니다.
COMMIT ROLLBACK SAVEPOINT	DML 명령문으로 만든 변경을 관리. 데이터 변경은 논리적 트랜잭션으로 함께 그룹화 될 수 있습니다.
GRANT REVOKE	오라클 데이터베이스와 그 구조에게 액세스 권한을 제공하거나 제거합니다. 이를 DCL (data control language)이라 부릅니다.

본 과정에 사용되는 테이블

EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLERK	MANAGER	7839	09-JUN-81	1500		10
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7900	JAMES	CLERK	7698	03-DEC-81	950		30
			7698	22-FEB-81	1250	500	30
			7566	03-DEC-81			20
			7902	17-DEC-80			
			7566	09-DEC-82			
			7788	12-JAN-83			
			7782	23-JAN-82			

DEPT

DEPTNO	ENAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

SALGRADE

GRADE	LO SAL	HIS SAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

본 과정에 사용되는 테이블

본 과정에서 3 가지의 주요 테이블이 사용될 것입니다:

- 모든 종업원의 세부 사항을 제공하는 EMP 테이블
- 모든 부서의 세부 사항을 제공하는 DEPT 테이블
- 다양한 등급별 급여의 세부 사항을 제공하는 SALGRADE 테이블

제 1 장 기본적인 SQL 문장 작성

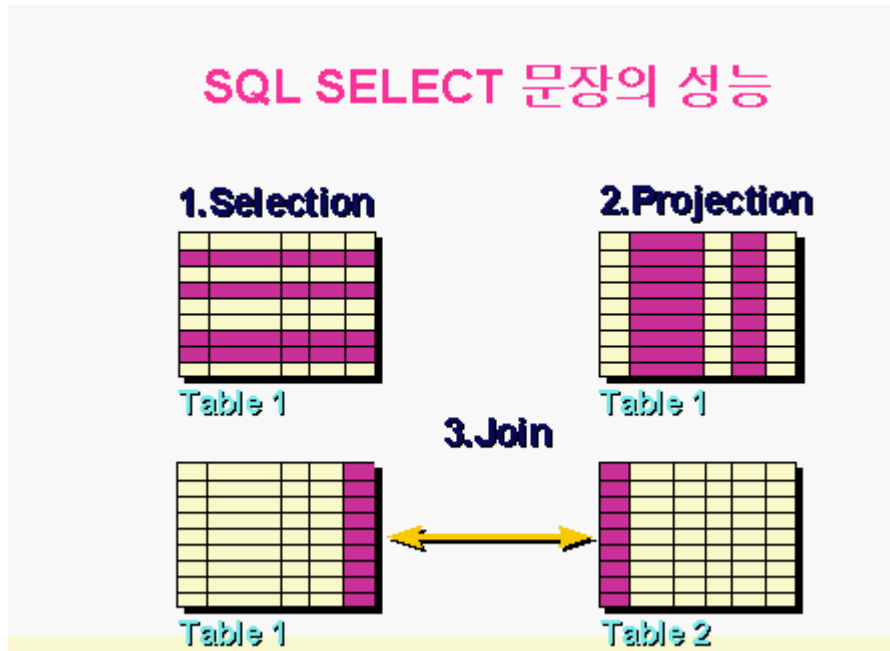
목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- **SQL SELECT** 문장의 성능을 알아봅니다.
- 기본적인 **SELECT** 문장을 실행합니다.

과정 목표

데이터베이스로부터 데이터를 추출하기 위해서 SQL(Structured Query Language) SELECT 문장을 사용해야 합니다. 디스플레이 되는 열을 축소해야 할 필요가 있을 수도 있습니다. 본 과정은 이러한 작업을 수행하는데 필요한 모든 SQL 문장을 기술합니다. 반복적으로 사용될 수 있는 SELECT 문장을 생성하기를 원할 수도 있습니다. 본 과정은 또한 SQL 문장을 실행하기 위한 SQL*Plus 명령어의 사용을 설명합니다.



SQL SELECT 문장의 성능

SELECT 문장은 데이터베이스로부터 정보를 검색합니다. SELECT 문장을 사용하여 다음을 할 수 있습니다:

- *Selection* : 질의에 대해 리턴하고자 하는 테이블의 행을 선택하기 위해 SQL의 selection 기능을 사용할 수 있습니다. 보고자 하는 행을 선택적으로 제한하기 위해 다양한 방법을 사용할 수 있습니다.
- *Projection* : 질의에 대해 리턴하고자 하는 테이블의 열을 선택하기 위해 SQL의 projection 기능을 사용할 수 있습니다. 필요한 만큼의 열을 선택할 수 있습니다.
- *Join* : 공유 테이블 양쪽의 열에 대해 링크를 생성하여 다른 테이블에 저장되어 있는 데이터를 함께 가져오기 위해 SQL의 join 기능을 사용할 수 있습니다.

기본적인 SELECT 문장

```
SELECT  [DISTINCT] {*, column [alias],...}  
FROM    table;
```

- **SELECT** 절에는 열을 나열합니다.
- **FROM** 절에는 테이블을 명시합니다.

기본적인 SELECT 문장

기본적으로 SELECT 문장은 다음을 포함해야 합니다:

- SELECT 절: 디스플레이 시킬 열을 명시합니다.
- FROM 절: SELECT 절에 나열된 열을 포함하는 테이블을 명시합니다.

구문형식에서:

SELECT	하나 이상의 열을 나열합니다.
DISTINCT	중복을 제거합니다.
*	모든 열을 선택합니다.
<i>column</i>	명명된 열을 선택합니다.
<i>alias</i>	선택된 열을 다른 이름으로 변경합니다.
FROM <i>table</i>	열을 포함하는 테이블을 명시합니다.

주: 본 과정 전체를 통하여 keyword, clause 그리고 statement 용어가 사용됩니다.

- *keyword* 는 독립적인 SQL 요소입니다. 예를 들면, SELECT 와 FROM 은 keyword 입니다.
- *clause* 는 SQL 문장의 한 부분입니다. 예를 들면, SELECT empno, ename, ... 은

clause입니다.

- *statement* 는 둘 이상의 *clauses*의 조합입니다. 예를 들면, `SELECT * FROM emp` 는 SQL statement 입니다.

실습문제

강사가 여러분을 대신해서 실습한 내용입니다. 명령에 대한 자세한 설명은 다음에 나오는 슬라이드를 통해서 설명 들으실 수 있습니다.

```
SQL> SELECT *
```

```
2 FROM dept;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

```
SQL> SELECT deptno, loc
```

```
2 FROM dept;
```

DEPTNO	LOC
10	NEW YORK
20	DALLAS
30	CHICAGO
40	BOSTON

```
SQL> SELECT deptno
```

```
2 FROM emp;
```

DEPTNO
20
30
30
20

30
30
10
20
10
30
20
30
20
10

14 개의 행이 선택되었습니다.

```
SQL> SELECT DISTINCT deptno  
2 FROM emp;
```

```
DEPTNO  
-----  
10  
20  
30
```

SQL 문장 작성

- **SQL** 문장은 대소문자를 구별하지 않습니다.
- **SQL** 문장은 한 줄 이상일 수 있습니다.
- 키워드는 단축하거나 줄을 나누어 쓸 수 없습니다.
- 절은 대개 줄을 나누어서 씁니다.
- 탭과 들여쓰기(indent)는 읽기 쉽게 하기 위해 사용됩니다.

SQL 문장 작성

아래의 간단한 규칙과 지침에 의해 읽기 쉽고 편집하기 쉬운 문장을 만들 수 있습니다:

- SQL 문장은 지정하지 않았다면 대소문자를 구별하지 않습니다.
- SQL 문장은 한 줄 또는 여러 줄에 입력될 수 있습니다.
- 키워드는 여러 줄에 나누거나 단축될 수 없습니다.
- 절은 보통 읽고 편집하기 쉽게 줄을 나누도록 합니다.
- 탭과 줄 넣기(indent)는 코드를 보다 읽기 쉽게 하기 위해 사용됩니다.
- 일반적으로 키워드는 대문자로 입력됩니다. 다른 모든 단어, 즉 테이블 이름, 열 이름은 소문자로 입력합니다.
- SQL*Plus에서 SQL 문장은 SQL 프롬프트에 입력되며, 이후의 라인은 번호가 붙습니다. 이것을 SQL buffer 라 부릅니다. 오직 한 문장만이 한 번에 버퍼에 있을 수 있습니다.

SQL 문장 실행

- 마지막 절의 끝에 ";"를 넣습니다.
- 버퍼에서 마지막 라인에 슬래시를 넣습니다.
- SQL 프롬프트에 슬래시를 입력합니다.
- SQL 프롬프트에서 SQL*Plus RUN 명령어를 실행합니다.

모든 열 선택

```
SQL> SELECT *  
2 FROM dept;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

모든 행, 열 선택

다음의 SELECT 키워드에 *를 사용하여 테이블의 열 데이터 모두를 디스플레이 할 수 있습니다. 위의 예에서, department 테이블은 3 개의 열을 포함합니다: DEPTNO, DNAME, LOC. 테이블은 4 개의 행을 포함합니다.

또한 SELECT 키워드 이후에 모든 열을 나열하여 테이블의 모든 열을 디스플레이 할 수 있습니다. 예를 들면, 다음 예와 같은 SQL 문장은 DEPT 테이블의 모든 행과 열을 디스플레이합니다:

```
SQL> SELECT      deptno, dname, loc
2 FROM          dept;
```

특정 열 선택

```
SQL> SELECT deptno, loc
2 FROM dept;
```

DEPTNO	LOC
10	NEW YORK
20	DALLAS
30	CHICAGO
40	BOSTON

특정 열과 모든 행 선택

열 이름을 콤마(,)로 구분하여 명시함으로써 테이블의 특정 열을 디스플레이 하는 SQL 문장을 사용할 수 있습니다. 위의 예는 DEPT 테이블의 모든 부서 번호와 위치를 디스플레이 합니다.

SELECT 절에서 원하는 열의 순서대로 결과에 나타나기를 원하면 보고자 하는 열을 순서대로 명시합니다:

```
SQL> SELECT      loc, deptno
```

```
2 FROM dept;
```

LOC	DEPTNO
NEW YORK	10
DALLAS	20
CHICAGO	30
BOSTON	40

디폴트 디스플레이

- 디폴트 데이터 자리맞춤
 - 좌측: 날짜와 문자 데이터
 - 우측: 숫자 데이터
- 디폴트 열 헤딩 디스플레이: 대문자

디폴트 열 헤딩

날짜 열 헤딩과 데이터 뿐만 아니라 문자 열 헤딩과 데이터는 열 폭 내에서 좌측 정렬됩니다. 숫자 헤딩과 데이터는 우측 정렬입니다.

```
SQL> SELECT      ename, hiredate, sal
2 FROM          emp;
```

ENAME	HIREDATE	SAL
-----	-----	-----

KING	17-NOV-81	5000
BLAKE	01-MAY-81	2850
CLARK	09-JUN-81	2450
JONES	02-APR-81	2975
MARTIN	28-SEP-81	1250
ALLEN	20-FEB-81	1600
...		
14 rows selected.		

문자와 날짜 열 헤딩은 절단될 수 있지만 숫자 헤딩은 절단될 수 없습니다. 열 헤딩은 디폴트로 대문자로 나타납니다. **별칭(alias)**으로써 열 헤딩 디스플레이를 변경할 수 있습니다. 열 **별칭(alias)**은 본 과정의 뒤에서 다루도록 합니다.

산술 표현식

산술 연산자를 사용하여 **NUMBER** 와 **DATE** 데이터 간의 표현식을 만듭니다.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

산술 표현식

데이터가 디스플레이 되는 방식을 수정하거나, 계산을 수행하고자 할 때 산술 표현식을 사용할 수 있습니다. 산술 표현식은 열 이름, 숫자 상수 값 그리고 **산술 연산자**를 포함할 수 있습니다.

산술 연산자

위의 슬라이드는 SQL에서 이용할 수 있는 **산술 연산자**입니다. FROM 절을 제외한 SQL 문

장의 절에서 산술 연산자를 사용할 수 있습니다.

산술 연산자 사용

```
SQL> SELECT ename, sal, sal+300  
2 FROM emp;
```

ENAME	SAL	SAL+300
KING	5000	5300
BLAKE	2850	3150
CLARK	2450	2750
JONES	2975	3275
MARTIN	1250	1550
ALLEN	1600	1900
...		
14 rows selected.		

산술 연산자 사용

위의 예는 모든 종업원의 급여를 \$300 증가 시키기 위해 덧셈 연산자를 사용하고 결과에 SAL+300 열을 디스플레이 합니다.

계산된 결과 열 SAL+300 은 EMP 테이블의 새로운 열이 아님을 유의하십시오. 이것은 단지 디스플레이를 위한 것일 뿐입니다. 디폴트로 새로운 열의 이름 즉, sal+300은 생성된 계산식으로부터 유래합니다.

주: SQL*Plus는 산술 연산자 앞뒤의 공백을 무시합니다.

연산자 우선순위

*** / + -**

- 곱하기와 나누기는 더하기와 빼기보다 우선순위가 높습니다.
- 같은 우선순위의 연산자는 좌측에서 우측으로 계산됩니다.
- 괄호는 강제로 계산의 우선순위를 바꾸거나 문장을 명료하게 하기 위해 사용됩니다.

연산자 우선순위

산술 표현식이 하나 이상의 연산자를 포함한다면, 곱하기와 나누기가 먼저 계산됩니다. 표현식 내에 같은 **우선순위**의 연산자가 있다면, 계산은 좌측에서 우측으로 수행됩니다. 먼저 계산되도록 하기 위해서 표현식에 괄호를 사용할 수 있습니다.

연산자 우선순위		
SQL> SELECT ename, sal, 12*sal+100 2 FROM emp;		
ENAME	SAL	12*SAL+100
KING	5000	60100
BLAKE	2850	34300
CLARK	2450	29500
JONES	2975	35800
MARTIN	1250	15100
ALLEN	1600	19300
...		
14 rows selected.		

연산자 우선순위 (계속)

위의 예는 이름, 급여 그리고 종업원의 연봉을 디스플레이 합니다. 월 급여에 12를 곱하고 보너스 \$100을 더하여 연봉을 계산합니다. 곱하기는 더하기 이전에 수행됨을 주의하십시오.

주: 표준 **우선순위**를 재지정하거나 문장의 명료성을 위해서 괄호를 사용하십시오. 예의 경우, 위의 표현식은 결과를 변경시키지 않고 (12*sal)+100으로 작성될 수 있습니다.

괄호 사용

```
SQL> SELECT ename, sal, 12*(sal+100)
2 FROM emp;
```

ENAME	SAL	12*(SAL+100)
KING	5000	61200
BLAKE	2850	35400
CLARK	2450	30600
JONES	2975	36900
MARTIN	1250	16200
...		

14 rows selected.

괄호 사용

연산자가 실행될 순서를 명시하기 위해서 괄호를 사용하여 **우선순위** 규칙을 변경할 수 있습니다.

위의 예는 이름, 급여 그리고 종업원의 연봉을 디스플레이 합니다. 연봉은 월 급여에 보너스 \$100을 더하고 거기에 12를 곱하여 계산합니다. 괄호 때문에 더하기가 곱하기 보다 **우선순위**가 높습니다.

Null 값 정의

- null은 이용할 수 없거나, 지정되지 않았거나, 알 수 없거나 또는 적용할 수 없는 값입니다.
- null은 숫자 0 이나 공백과는 다릅니다.

```
SQL> SELECT ename, job, comm
2 FROM emp;
```

ENAME	JOB	COMM
KING	PRESIDENT	
BLAKE	MANAGER	
...		
TURNER	SALESMAN	0
...		

14 rows selected.

Null 값

행이 특정 열에 대한 데이터 값이 없다면, 값은 null이 됩니다.

null 값은 이용할 수 없거나, 지정되지 않았거나, 알 수 없거나 또는 적용할 수 없는 값입니다. null 값은 Ø이나 공백과는 다릅니다. Ø은 숫자이며 공백은 문자입니다.

열이 NOT NULL로 정의되지 않았거나, 열이 생성될 때 PRIMARY KEY로 정의되지 않았다면, 어떤 데이터형의 열은 null 값을 포함할 수 있습니다.

EMP 테이블의 COMM 열에서 오직 SALESMAN 만이 보너스를 받을 수 있음을 주목하십시오. 다른 종업원은 보너스가 없습니다. null 값은 그러한 사실을 나타냅니다. 판매원 Turner 는 보너스를 받지 않습니다. 그의 보너스는 null이 아니라 Ø임을 주목하십시오.

산술 표현식에서의 Null 값

null 값을 포함하는 산술 표현식은 null로 계산됩니다.

```
SQL> select ename NAME, 12*sal+comm  
2   from emp  
3  WHERE ename='KING';
```

NAME	12*SAL+COMM

KING	

Null 값 (계속)

산술 표현식에 있는 어떤 열 값이 null 이면, 결과는 null 입니다. 예를 들면, Ø으로 나누기를 시도한다면, 에러가 발생합니다. 그러나 null로 숫자 값을 나누면, 결과는 null 또는 unknown 입니다.

위의 예에서, 종업원 KING은 SALESMAN이 아니고 보너스를 받지 않습니다. 산술 표현식에서 COMM 열이 null이기 때문에 결과는 null 입니다.

보다 자세한 정보를 알고자 한다면, 다음을 참조하십시오.

- Oracle Server SQL Reference, Release 7.3,
- Oracle Server SQL Reference, Release 8.0, "Elements of SQL."

열 별칭(alias) 정의

- 열 헤딩 이름을 변경합니다.
- 계산할 때에 유용합니다.
- 열 이름 바로 뒤에 둡니다. 열 이름과 별칭사이에 키워드 '**AS**' 를 넣기도 합니다.
- 공백이나 특수 문자 또는 대문자가 있으면 이중 인용부호(" ")가 필요합니다.

열 별칭

질의의 결과를 디스플레이 할 때, 보통 SQL*Plus 는 열 헤딩으로 선택된 열의 이름을 사용합니다. 이 헤딩은 이해하기가 어려운 경우도 많습니다. 열 **별칭**을 사용하여 열 헤딩을 변경할 수 있습니다.

구분자로 공백을 사용하여 SELECT 목록 안의 열 뒤에 **별칭**을 명시합니다. 디폴트로 열 헤딩은 대문자로 나타납니다. **별칭**이 공백, 특수문자(# 이나 \$ 같은) 또는 대소문자를 포함하면, 이중 인용부호(" ")로 **별칭**을 둘러싸야 합니다.

열 별칭 사용

```
SQL> SELECT ename AS name, sal salary  
2 FROM emp;
```

NAME	SALARY
-----	-----
...	

```
SQL> SELECT ename "Name",  
2 sal*12 "Annual Salary"  
3 FROM emp;
```

Name	Annual Salary
-----	-----
...	

열 별칭 (계속)

첫 번째 예는 모든 종업원의 이름과 월 급여를 디스플레이 합니다. AS 키워드는 열 별칭 이름 앞에 사용됩니다. 질의의 결과는 AS 키워드가 사용되거나 또는 사용되지 않았거나 똑같습니다. 또한 SQL 문장은 열 별칭, 이름 그리고 급여를 소문자로 나타내지만, 반면에 질의의 결과는 열 헤딩을 대문자로 나타냄을 주목하십시오. 이미 언급한 것처럼 열 헤딩은 디폴트로 대문자로 나타납니다.

두 번째 예는 모든 종업원의 이름과 월 급여를 디스플레이 합니다. 연봉(Annual Salary)이 공백을 포함하므로 이중 인용부호에 둘러싸여 있습니다. 결과의 열 헤딩은 열 별칭과 정확히 똑같음을 주목하십시오.

연결 연산자

- 열이나 문자 스트링을 다른 열과 연결합니다.
- 두 개의 수직 바(||) 로 나타냅니다.
- 문자 표현식인 결과 열을 생성합니다.

연결 연산자

연결 연산자(||)를 사용하여 문자 표현식을 생성하기 위해 다른 열, 산술 표현식 또는 상수 값에 열을 연결할 수 있습니다. 연산자의 양쪽에 있는 열은 단일 결과 열을 만들기 위해 조합됩니다.

연결 연산자 사용

```
SQL> SELECT  ename||job AS "Employees"  
2 FROM      emp;
```

```
Employees  
-----  
KINGPRESIDENT  
BLAKEMANAGER  
CLARKMANAGER  
JONESMANAGER  
MARTINSALESMAN  
ALLENSALESMAN  
...  
14 rows selected.
```

연결 연산자 (계속)

예에서, ENAME과 JOB은 **별칭** Employee로 연결되었습니다. 종업원 이름과 종업원 직업이 단일 결과 열을 만들기 위해서 조합되었음을 주목하십시오.

별칭 이름 앞의 AS 키워드는 SELECT 절을 보다 읽기 쉽게 합니다.

리터럴(Literal) 문자 스트링

- **SELECT** 절에 포함된 리터럴은 문자, 표현식 또는 숫자입니다.
- 날짜와 문자 리터럴 값은 단일 인용부호(" ") 안에 있어야 합니다.
- 각각의 문자 스트링은 리턴된 각 행에 대한 결과입니다.

리터럴 문자 스트링

리터럴은 열 이름이나 열 **별칭**이 아닌, SELECT 목록에 포함되어 있는 문자, 표현식 또는 숫자입니다. 그것은 리턴되는 각각의 행에 대해 출력됩니다. 리터럴 스트링은 질의 결과에 포함될 수 있으며, SELECT 목록에서 열과 똑같이 취급됩니다.

날짜와 문자 리터럴은 단일 인용부호(' ') 내에 있어야 합니다; 숫자 리터럴은 그렇게 해서는 안됩니다.

리터럴 문자 스트링 사용

```
SQL>SELECT ename || ' is a ' || job  
2 AS "Employee Details"  
3 FROM emp;
```

```
Employee Details  
-----  
KING is a PRESIDENT  
BLAKE is a MANAGER  
CLARK is a MANAGER  
JONES is a MANAGER  
MARTIN is a SALESMAN  
...  
14 rows selected.
```

리터럴 문자 스트링 (계속)

위의 예는 모든 종업원의 이름과 직업을 디스플레이 합니다. 열은 “Employee Details” 라는 헤딩을 가집니다. SELECT 문장에서 단일 인용부호 사이에 공백이 있음을 주목하십시오.

공백은 결과를 읽기 쉽게 해줍니다.

각각의 종업원에 대한 이름과 급여는 리턴된 행에 의미를 부여 하기 위해서 다음의 예에서, 리터럴로 연결 되었습니다.

```
SQL> SELECT ename ||': '||'1' ||' Month salary = '||sal Monthly  
2 FROM emp;
```

MONTHLY

```
-----  
KING: 1 Month salary = 5000  
BLAKE: 1 Month salary = 2850  
CLARK: 1 Month salary = 2450  
JONES: 1 Month salary = 2975  
MARTIN: 1 Month salary = 1250  
ALLEN: 1 Month salary = 1600  
TURNER: 1 Month salary = 1500  
...  
14 rows selected.
```

중복 행

질의의 디폴트 디스플레이는 중복되는 행을 포함하는 모든 행입니다.

```
SQL> SELECT deptno  
2 FROM emp;
```

DEPTNO
10
30
10
20
...
14 rows selected.

중복 행

특별히 명시하지 않았다면, SQL*Plus 는 중복되는 행을 제거하지 않고 질의의 결과를 디스플레이 합니다. 위의 예는 EMP 테이블로부터 모든 부서번호를 디스플레이 합니다. 부서 번호가 반복되었음을 주목하십시오.

중복 행 제거

SELECT 절에서 **DISTINCT** 키워드를 사용하여 중복되는 행을 제거합니다.

```
SQL> SELECT DISTINCT deptno  
2 FROM emp;
```

DEPTNO
10
20
30

중복 행 (계속)

결과에서 중복되는 행을 제거하기 위해서, SELECT 키워드 바로 뒤에 DISTINCT 키워드를 포함합니다. 위의 예에서 EMP 테이블은 실제로 14 개의 행을 포함하지만 테이블에는 오직 3 개의 유일한 부서 번호만이 있습니다.

DISTINCT 키워드 뒤에 여러 개의 열을 명시할 수 있습니다. DISTINCT 키워드는 선택된 모든 열에 영향을 미치고, 결과는 모든 열의 distinct 한 조합을 나타냅니다.

```
SQL> SELECT      DISTINCT deptno, job
      2 FROM      emp;
```

```
DEPTNO JOB
-----
      10 CLERK
      10 MANAGER
      10 PRESIDENT
      20 ANALYST
...
9 rows selected.
```

제 2 장 데이터 제한과 정렬

목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 질의에 의해 검색되는 행을 제한합니다.
- 질의에 의해 검색되는 행을 정렬합니다.

과정 목표

데이터베이스로부터 데이터를 검색하는 동안에, 디스플레이되는 데이터 행을 축소하거나 디스플레이되는 행의 순서를 명시할 필요가 있을 수 있습니다. 본 과정은 이러한 동작을 수행하기 위해서 사용할 수 있는 SQL 문장을 설명합니다.

Selection을 사용하여 행을 제한

EMP

EMPNO	ENAME	JOB	...	DEPTNO
7839	KING	PRESIDENT		10
7698	BLAKE	MANAGER		30
7782	CLARK	MANAGER		10
7566	JONES	MANAGER		20
...				

"...부서 10의 모든
종업원을 검색"

EMP

EMPNO	ENAME	JOB	...	DEPTNO
7839	KING	PRESIDENT		10
7782	CLARK	MANAGER		10
7934	MILLER	CLERK		10

Selection을 사용하여 행을 제한

위의 예에서, 부서 10의 모든 종업원을 디스플레이 하기를 원한다고 가정합니다. DEPTNO 열에 10의 값을 가진 행의 집합만이 리턴 됩니다. 이 제한 방법은 SQL의 WHERE 절을 사용하는 것입니다.

선택된 행 제한

- **WHERE** 절을 사용하여 리턴되는 행을 제한합니다.

```
SELECT      [DISTINCT] {*, column [alias], ...}  
FROM        table  
[WHERE      condition(s)];
```

- **WHERE** 절은 **FROM** 절 다음에 옵니다.

선택된 행 제한

질의에서 WHERE 절을 사용하여 리턴되는 행을 제한할 수 있습니다. WHERE 절은 수행될 조건을 포함하며, FROM 절 바로 다음에 옵니다.

구문형식에서:

WHERE 조건을 만족하는 행으로 질의를 제한합니다.

condition 열 이름, 표현식, 상수 그리고 **비교 연산자**로 구성됩니다.

WHERE 절은 열, 리터럴 값, 산술 표현식 또는 함수 값을 비교할 수 있습니다. WHERE 절은 다음의 3 가지 요소로 구성됩니다:

- 열 이름
- 비교 연산자
- 열 이름, 상수 또는 값의 목록

WHERE 절 사용

```
SQL> SELECT ename, job, deptno  
2 FROM emp  
3 WHERE job='CLERK';
```

ENAME	JOB	DEPTNO
JAMES	CLERK	30
SMITH	CLERK	20
ADAMS	CLERK	20
MILLER	CLERK	10

WHERE 절 사용

예에서, SELECT 문장은 업무가 CLERK인 모든 종업원의 이름, 업무 그리고 부서 번호를 검색합니다.

업무 CLERK은 EMP 테이블의 job 열과 일치되도록 하기 위해서 대문자로 명시되어야 함을 주의하십시오.

문자 스트링과 날짜

- 문자 스트링과 날짜 값은 단일 인용부호(' ')로 둘러싸여 있습니다.
- 문자 값은 대소문자를 구분하고, 날짜 값은 날짜 형식을 구분합니다.
- 디폴트 날짜 형식은 'DD-MON-YY' 입니다.

```
SQL> SELECT  ename, job, deptno
2 FROM      emp
3 WHERE     ename = 'JAMES';
```

문자 스트링과 날짜

WHERE 절에서 문자 스트링과 날짜는 단일 인용부호('')에 둘러싸여 있어야 합니다. 그러나 숫자 상수는 그래서 안됩니다.

모든 문자 검색은 대소문자를 구분합니다. 다음 예에서, EMP 테이블이 모든 데이터를 대문자로 저장하고 있기 때문에 리턴되는 행이 없습니다.

```
SQL> SELECT  ename, empno, job, deptno
2 FROM      emp
3 WHERE     job='clerk';
```

오라클은 내부 숫자 형식에서 날짜를 세기, 년, 월, 일, 시간, 분 그리고 초로 저장합니다. 디폴트 날짜 출력은 DD-MON-YY 입니다.

주: 디폴트 날짜 형식 변경은 제 3 과에서 다룹니다.
숫자 값은 인용 표시로 둘러싸면 안됩니다.

비교 연산자

연산자	의미
=	같다
>	보다 크다
>=	보다 크거나 같다
<	보다 작다
<=	보다 작거나 같다
<>	같지 않다

비교 연산자

비교 연산자는 조건문에서 사용되며 하나의 표현식을 다른 표현식과 비교합니다. 그들은 다음의 형식으로 WHERE 절에서 사용됩니다:

구문 형식

```
... WHERE expr operator value
```

예

```
... WHERE hiredate='01-JAN-95'  
... WHERE sal>=1500  
... WHERE ename='SMITH'
```


비교 연산자 사용

```
SQL> SELECT ename, sal, comm
2 FROM emp
3 WHERE sal <= comm;
```

ENAME	SAL	COMM
MARTIN	1250	1400

비교 연산자 사용

예에서, SELECT 문장은 EMP 테이블로부터 이름, 급여 그리고 보너스를 검색합니다. 여기서 종업원 급여는 그들의 보너스보다 작거나 같아야 합니다. WHERE 절에서 명확하게 지정된 값이 없음을 주목하십시오. 두 개의 값은 EMP 테이블의 SAL 열과 COMM 열로부터 비교됩니다.

다른 비교 연산자

연산자	의미
BETWEEN ...AND...	두 값의 사이 (포함하는)
IN(list)	어떤 값의 목록과 일치
LIKE	문자 패턴과 일치
IS NULL	null 값

연관된 상세 설명은 없습니다.

BETWEEN 연산자 사용

- 값의 범위에 해당하는 행을 디스플레이 하기 위해서 **BETWEEN** 연산자를 사용합니다.

```
SQL> SELECT  ename, sal
2 FROM      emp
3 WHERE     sal BETWEEN 1000 AND 1500;
```

ENAME	SAL	Lower limit	Higher limit
MARTIN	1250		
TURNER	1500		
WARD	1250		
ADAMS	1100		
MILLER	1300		

BETWEEN 연산자

BETWEEN 연산자를 사용하여 값의 범위에 해당하는 행을 디스플레이 할 수 있습니다. 명시한 범위는 하한 값과 상한 값을 포함합니다.

위의 SELECT 문장은 급여가 \$1000 에서 \$1500 사이에 있는 종업원에 대해서 EMP 테이블로부터 행을 리턴합니다.

주: BETWEEN 연산자로 명시된 값도 포함됩니다. 하한 값을 먼저 명시해야 합니다.

IN 연산자 사용

- 목록에 있는 값들과 비교하기 위해서 **IN** 연산자를 사용합니다.

```
SQL> SELECT  empno, ename, sal, mgr
2 FROM      emp
3 WHERE     mgr IN (7902, 7566, 7788);
```

EMPNO	ENAME	SAL	MGR
7902	FORD	3000	7566
7369	SMITH	800	7902
7788	SCOTT	3000	7566
7876	ADAMS	1100	7788

IN 연산자

명시된 목록에 있는 값에 대해서 테스트 하려면 IN 연산자를 사용합니다.

위의 예는 관리자의 종업원 번호가 7902, 7566, 또는 7788 인 모든 종업원의 종업원 번호, 이름, 급여 그리고 관리자의 종업원 번호를 디스플레이 합니다.

IN 연산자는 어떤 데이터형과도 사용될 수 있습니다. 다음 예는 이름이 WHERE 절의 이름 목록에 포함되는 어떤 종업원에 대해서 EMP 테이블로부터 행을 리턴합니다.

```
SQL> SELECT      empno,  ename,  mgr,  deptno
2 FROM          emp
3 WHERE         ename IN ('FORD' , 'ALLEN');
```

주: 목록에 문자나 날짜가 사용되면, 그들은 단일 인용부호('')로 둘러싸여 있어야 합니다.

LIKE 연산자 사용

- 검색 스트링 값에 대한 와일드카드 검색을 위해서 **LIKE** 연산자를 사용합니다.
- 검색 조건은 리터럴 문자나 숫자를 포함할 수 있습니다.
 - % 는 문자가 없거나 또는 하나 이상을 나타냅니다.
 - _ 는 하나의 문자를 나타냅니다.

```
SQL> SELECT      ename
2 FROM          emp
3 WHERE         ename LIKE 'S%';
```

LIKE 연산자

검색하고자 하는 값을 항상 정확하게 알 수는 없습니다. LIKE 연산자를 사용하여 문자 패턴과 일치하는 행을 선택할 수 있습니다. 문자 패턴 일치 연산을 wildcard 검색이라 부릅니다. 아래 두 개의 기호가 스트링 검색에 사용됩니다.

기호	설명
%	문자가 없거나 하나 이상을 나타냅니다.
_	단일 문자를 나타냅니다.

위의 SELECT 문장은 이름이 “S”로 시작하는 어떤 종업원에 대해서 EMP 테이블로부터 종업원 이름을 리턴합니다. “S”가 대문자임을 주목하십시오. 소문자로 시작되는 “s”는 리턴되지 않을 것입니다.

LIKE 연산자는 BETWEEN 비교에 대한 단축 키로 사용될 수 있습니다. 다음 예는, January 1981 과 December 1981 사이에 입사한 모든 종업원의 이름과 입사일을 디스플레이 합니다.

```
SQL> SELECT  ename, hiredate
2 FROM      emp
3 WHERE     hiredate LIKE '%81';
```

LIKE 연산자 사용

- 패턴 일치 문자를 조합할 수 있습니다.

```
SQL> SELECT  ename
2 FROM      emp
3 WHERE     ename LIKE ' _A%';
```

ENAME
JAMES
WARD

- “%” 나 “_”에 대해서 검색하기 위해서 **ESCAPE** 식별자를 사용할 수 있습니다.

와일드카드 문자 검색

“%” 와 “_” 심볼은 리터럴 문자의 결합하여 사용될 수 있습니다. 위의 예는 이름의 두 번째 문자가 “A”인 모든 종업원의 이름을 디스플레이 합니다.

ESCAPE 옵션

실제로 “%” 와 “_” 문자를 포함하는 열을 검색할 필요가 있을 때는 ESCAPE 옵션을

사용합니다. 이 옵션은 ESCAPE 문자로 어떤 문자를 사용할 지를 명시합니다. 이름이 “A_B”를 포함하는 종업원의 이름을 디스플레이 하려면 다음의 SQL 문장을 사용합니다:

```
SQL> SELECT  ename
2 FROM      emp
3 WHERE     ename LIKE '%A\_B' ESCAPE '\';
```

ESCAPE 옵션은 회피 문자로서 백슬래시(W)를 사용합니다. 회피 문자는 밑줄(_) 앞에 둡니다. 이것은 오라클 서버가 밑줄을 리터럴로 번역하도록 합니다.

IS NULL 연산자 사용

- IS NULL 연산자로 null 값을 테스트 합니다.

```
SQL> SELECT  ename, mgr
2 FROM      emp
3 WHERE     mgr IS NULL;
```

ENAME	MGR
KING	

IS NULL 연산자

IS NULL 연산자는 null 인 값에 대해서 테스트 합니다. null 값은 값이 없거나, 알 수 없거나, 또는 적용할 수 없음을 의미합니다. 그러므로, null 값은 어떤 값과 같거나 또는 다를 수 없으므로 (=)로는 테스트 할 수가 없습니다. 위의 예는 관리자가 없는 모든 종업원의 이름과 관리자를 검색합니다.

예를 들면, 보너스가 없는 모든 종업원에 대해서 이름과 업무 그리고 보너스를 디스플레이 하기 위해서 다음의 SQL 문장을 사용합니다:

```
SQL> SELECT  ename, job, comm
2 FROM      emp
```

```
3      WHERE      comm IS NULL;
```

ENAME	JOB	COMM
-----	-----	-----
KING	PRESIDENT	
BLAKE	MANAGER	
CLARK	MANAGER	
...		

논리 연산자

연산자	의미
AND	양쪽 컴포넌트의 조건이 TRUE이면 TRUE를 리턴합니다.
OR	한쪽 컴포넌트의 조건만 TRUE이면 TRUE를 리턴합니다.
NOT	이후의 조건이 FALSE이면 TRUE를 리턴합니다.

논리 연산자

논리 연산자는 두 개의 조건에 대한 하나의 결과를 만들기 위해서, 또는 하나의 조건의 결과를 바꾸기 위해서 두 개의 컴포넌트 조건의 결과를 조합합니다. SQL에서는 세 개의 연산자를 이용할 수 있습니다:

- AND
- OR
- NOT

지금까지의 모든 예는 WHERE 절에 오직 하나의 조건만을 명시했습니다. AND 와 OR 절을 사용하여 WHERE 절에 여러 개의 조건을 사용할 수 있습니다.

AND 연산자 사용

AND는 양쪽의 조건이 참이어야 **TRUE**를 리턴합니다.

```
SQL> SELECT empno, ename, job, sal
2 FROM emp
3 WHERE sal >= 1100
4 AND job = 'CLERK';
```

EMPNO	ENAME	JOB	SAL
7876	ADAMS	CLERK	1100
7934	MILLER	CLERK	1300

AND 연산자

위의 예에서, 양쪽의 조건은 참이어야 합니다. 그러므로, 업무가 CLERK 이고 급여가 \$1100 이상인 종업원이 선택될 것입니다.

모든 문자 검색은 대소문자를 구별합니다. CLERK 이 대문자가 아니면 리턴되는 행은 없습니다. 문자 스트링은 인용부호로 둘러싸야 합니다.

AND 참 테이블

다음의 테이블은 AND 로 두 개의 표현식을 조합한 결과를 보여 줍니다:

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR 연산자 사용

OR는 한쪽의 조건만 참이면 **TRUE**를 리턴합니다.

```
SQL> SELECT empno, ename, job, sal
2 FROM emp
3 WHERE sal >= 1100
4 OR job = 'CLERK';
```

EMPNO	ENAME	JOB	SAL
7839	KING	PRESIDENT	5000
7698	BLAKE	MANAGER	2850
7782	CLARK	MANAGER	2450
7566	JONES	MANAGER	2975
7654	MARTIN	SALESMAN	1250
...			

14 rows selected.

OR 연산자

위의 예에서, 한쪽의 조건만 참이면 됩니다. 그러므로, 업무가 CLERK 이거나 급여가 \$1100 이상인 종업원이 선택될 것입니다.

OR 참 테이블

다음 테이블은 OR 로 두 개의 표현식을 조합한 결과를 보여 줍니다:

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT 연산자 사용

```
SQL> SELECT ename, job
2 FROM emp
3 WHERE job NOT IN ('CLERK', 'MANAGER', 'ANALYST');
```

ENAME	JOB
KING	PRESIDENT
MARTIN	SALESMAN
ALLEN	SALESMAN
TURNER	SALESMAN
WARD	SALESMAN

NOT 연산자

위의 예는 업무가 CLERK, MANAGER, 또는 ANALYST 가 아닌 모든 종업원의 이름과 업무를 디스플레이 합니다.

NOT 참 테이블

다음의 테이블은 조건에 대해서 NOT 연산자를 적용한 결과를 보여 줍니다:

NOT	TRUE	FALSE	UNKNOWN
	FALSE	TRUE	UNKNOWN

주: NOT 연산자는 BETWEEN, LIKE, 그리고 NULL 같은 다른 SQL 연산자와 함께 사용될 수 있습니다.

```
... WHERE NOT job IN ('CLERK', 'ANALYST')
... WHERE sal NOT BETWEEN 1000 AND 1500
... WHERE ename NOT LIKE '%A%'
... WHERE comm IS NOT NULL
```

우선순위 규칙

우선순위	연산자
1	모든 비교 연산자
2	NOT
3	AND
4	OR

- 괄호를 사용하면 우선순위 규칙은 변경됩니다.

연관된 상세 설명은 없습니다.

우선순위 규칙

```
SQL> SELECT ename, job, sal
  2 FROM emp
  3 WHERE job='SALESMAN'
  4 OR job='PRESIDENT'
  5 AND sal>1500;
```

ENAME	JOB	SAL
KING	PRESIDENT	5000
MARTIN	SALESMAN	1250
ALLEN	SALESMAN	1600
TURNER	SALESMAN	1500
WARD	SALESMAN	1250

AND 연산자의 우선순위 예

위의 예에는 두 개의 조건이 있습니다:

- 첫 번째 조건은 업무가 PRESIDENT 이고 급여가 1500 이상입니다.
- 두 번째 조건은 업무가 SALESMAN 입니다.

그러므로, SELECT 문장은 다음처럼 읽습니다:

“업무가 PRESIDENT 이고 \$1500 이상을 벌거나 또는 업무가 SALESMAN 인 행을 검색합니다.”

우선순위 규칙

우선순위를 강제로 바꾸기 위해서 괄호를 사용합니다

```
SQL> SELECT  ename, job, sal
2 FROM      emp
3 WHERE      (job='SALESMAN '
4 OR         job='PRESIDENT ')
5 AND        sal>1500;
```

ENAME	JOB	SAL
KING	PRESIDENT	5000
ALLEN	SALESMAN	1600

괄호 사용

위의 예에는 두 개의 조건이 있습니다:

- 첫 번째 조건은 업무가 PRESIDENT 이거나 SALESMAN입니다.
- 두 번째 조건은 급여가 1500 이상입니다.

그러므로, SELECT 문장은 다음처럼 읽습니다:

“업무가 PRESIDENT 이거나 SALESMAN 이고 \$1500 이상을 버는 행을 검색합니다.”

ORDER BY 절

- **ORDER BY** 절로 행을 정렬합니다.
 - **ASC**: 오름차순, 디폴트
 - **DESC**: 내림차순
- **ORDER BY** 절은 **SELECT** 문장의 가장 뒤에 옵니다.

```
SQL> SELECT      ename, job, deptno, hiredate
2  FROM          emp
3  ORDER BY hiredate;
```

ENAME	JOB	DEPTNO	HIREDATE
SMITH	CLERK	20	17-DEC-80
ALLEN	SALESMAN	30	20-FEB-81
...			

14 rows selected.

ORDER BY 절

질의 결과에 리턴되는 행의 순서는 정의되지 않습니다. ORDER BY 절은 행을 정렬하는데 사용할 수 있습니다. 사용이 되었다면, ORDER BY 절은 맨 뒤에 두어야 합니다. 정렬을 위한 표현식이나 **별칭**을 명시할 수 있습니다.

구문형식

```
SELECT      expr
FROM        table
[WHERE      condition(s)]
[ORDER BY   {column, expr} [ASC|DESC]];
```

여기서: ORDER BY 검색된 행이 디스플레이 되는 순서를 명시합니다.
 ASC 행의 오름차순 정렬, 디폴트 정렬입니다.
 DESC 행의 내림차순 정렬

ORDER BY 절이 사용되지 않았다면, 정렬 순서가 정의되지 않은 것이며, 오라클 서버는 똑같은 질의를 두 번 실행했을 때 행을 똑같은 순서로 나타내지 않을 수도 있습니다. 특정 순서로 행을 디스플레이 하기 위해서 ORDER BY 절을 사용합니다.

내림차순 정렬

```
SQL> SELECT   ename, job, deptno, hiredate
2  FROM      emp
3  ORDER BY   hiredate DESC;
```

ENAME	JOB	DEPTNO	HIREDATE
ADAMS	CLERK	20	12-JAN-83
SCOTT	ANALYST	20	09-DEC-82
MILLER	CLERK	10	23-JAN-82
JAMES	CLERK	30	03-DEC-81
FORD	ANALYST	20	03-DEC-81
KING	PRESIDENT	10	17-NOV-81
MARTIN	SALESMAN	30	28-SEP-81
...			

14 rows selected.

데이터의 디폴트 정렬

디폴트 정렬은 오름차순입니다:

- 숫자 값은 가장 작은 값이 먼저 디스플레이 됩니다. - 예를 들면, 1-999.
- 날짜 값은 가장 빠른 값이 먼저 디스플레이 됩니다. - 예를 들면, 01-JAN-95 보다 01-JAN-92가 먼저.
- 문자 값은 알파벳 순서로 디스플레이 됩니다. - 예를 들면, A 가 먼저 Z 가 나중에.
- Null 값은 오름차순에서는 제일 나중에 그리고 내림차순에서는 제일 먼저 옵니다.

디폴트 순서 변경

행이 디스플레이 되는 순서를 바꾸기 위해서, ORDER BY 절에서 열 이름 뒤에 DESC 키워드를 명시합니다. 위의 예는 가장 최근에 입사한 종업원으로 결과를 정렬합니다.

열 별칭에 의한 정렬

```
SQL> SELECT empno, ename, sal*12 annsal  
2 FROM emp  
3 ORDER BY annsal;
```

EMPNO	ENAME	ANNSAL
7369	SMITH	9600
7900	JAMES	11400
7876	ADAMS	13200
7654	MARTIN	15000
7521	WARD	15000
7934	MILLER	15600
7844	TURNER	18000

...
14 rows selected.

열 별칭에 의한 정렬

ORDER BY 절에 열 별칭을 사용할 수 있습니다. 위의 예는 연봉으로 데이터를 정렬합니다.

제 3 장 단일행 함수

목 적

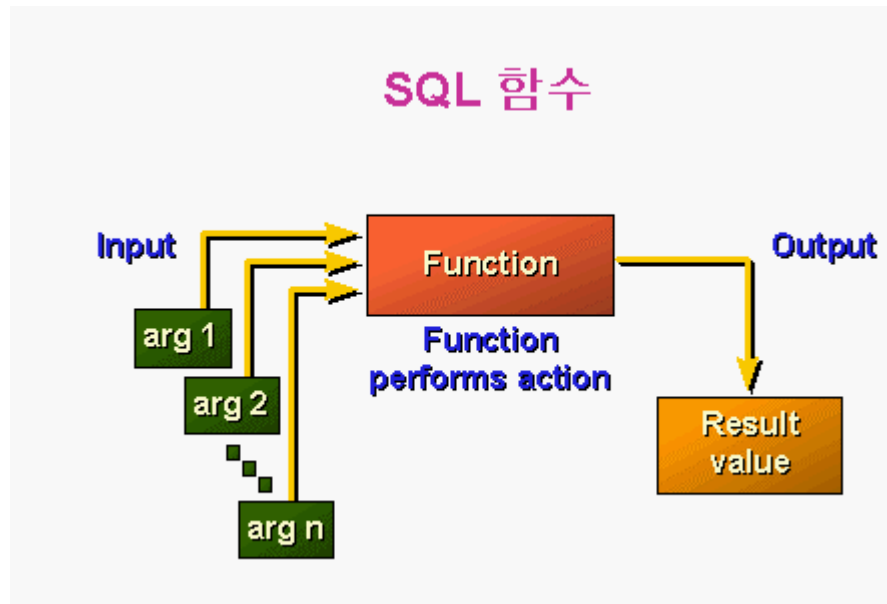
본 과정을 마치면, 다음을 할 수 있어야 합니다.

- **SQL**에서 이용 가능한 다양한 종류의 함수를 기술합니다.
- **SELECT** 문장에서 문자, 숫자 그리고 날짜 함수를 사용합니다.
- 변환 함수를 사용합니다.

과정 목표

함수는 기본적인 질의 블록을 보다 강력하게 만들며 데이터 값을 조작하기 위해 사용됩니다.

이곳은 함수에 대한 두 개의 과정 중 첫 번째 과정입니다. 문자 데이터를 숫자 데이터로 변환하는 것같은, 어떤 데이터 형을 다른 데이터형으로 변환하는 함수 뿐만 아니라 단일 행 문자, 숫자 그리고 날짜 함수에 초점을 맞출 것입니다.



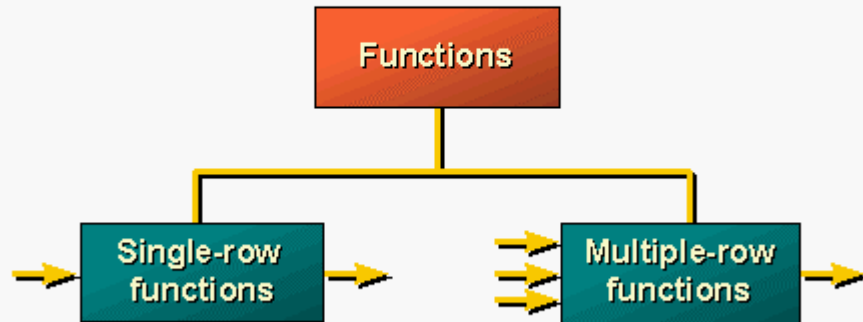
SQL 함수

함수는 SQL 의 아주 강력한 특징이며 다음을 위해서 사용할 수 있습니다:

- 데이터 계산을 수행
- 개별적인 데이터 항목을 수정
- 행의 그룹에 대해 결과를 조작
- 디스플레이를 위한 날짜와 숫자 형식
- 열의 데이터형을 변환

SQL 함수는 **인수**를 받고 결과 값을 내줍니다.

SQL 함수의 두 가지 유형



SQL 함수 (계속)

함수에는 두 가지의 다른 유형이 있습니다:

- 단일 행 함수
- 다중 행 함수

단일 행 함수

이러한 함수는 오직 단일 행에서만 적용가능하고 행별로 하나의 결과를 리턴합니다. 단일 행 함수에는 여러 유형들이 있습니다. 본 과정은 아래에 나열된 것들을 다룹니다:

- 문자
- 숫자
- 날짜
- 변환

다중 행 함수

이러한 함수는 복수의 행을 조작하여 행의 그룹당 하나의 결과를 리턴합니다.

이용 가능한 함수와 구문형식의 완전한 목록에 대해 보다 자세한 정보를 알고자 한다면

- Oracle Server SQL Reference, Release 7.3,
- Oracle Server SQL Reference, Release 8.0 을 참조하십시오.

단일 행 함수

- 데이터 값을 조작합니다.
- 인수(argument)를 받고 하나의 결과를 리턴합니다.
- 리턴될 각각의 행에 적용됩니다.
- 행별로 하나의 결과를 리턴합니다.
- 데이터형을 수정할 수 있습니다.
- 중첩(nested)될 수 있습니다.

```
function_name (column|expression, [arg1, arg2,...])
```

단일 행 함수

단일 행 함수는 데이터 값을 조작하는데 사용됩니다. 하나 이상의 인수(argument)를 받고 질의에 의해 리턴될 각각의 행에 대해 하나의 값을 리턴합니다. 인수(argument)는 다음 중의 하나가 될 수 있습니다:

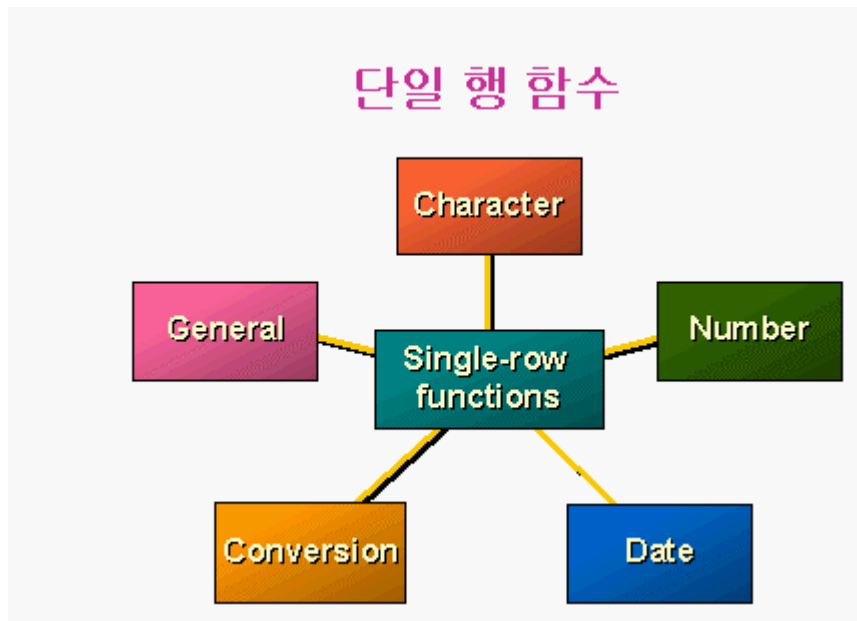
- 사용자가 제공한 상수
- 가변적인 값
- 열 이름
- 표현식

단일 행 함수의 특징

- 질의에서 리턴되는 각각의 행에 대해 수행합니다.
- 행별로 하나의 결과를 리턴합니다.
- 참조 시 사용한 데이터 형과 다른 데이터 형의 데이터 값을 리턴할 수도 있습니다.
- 하나 이상의 인수(argument)를 필요로 합니다.
- SELECT, WHERE, 그리고 ORDER BY 절에서 사용할 수 있습니다. 함수를 중첩할 수 있습니다.

구문형식에서:

function_name	함수 명 입니다.
column	데이터베이스 열 이름 입니다.
expression	어떤 문자 스트링이거나 계산된 표현식입니다.
arg1, arg2	함수에 의해 사용될 수 있는 인수입니다.



단일 행 함수 (계속)

본 과정은 다음의 단일 행 함수를 다룹니다:

- 문자 함수: 문자 입력을 받고 문자와 숫자 값 모두를 리턴할 수 있습니다.
- 숫자 함수: 숫자 입력을 받고 숫자 값을 리턴합니다.
- 날짜 함수: 날짜 데이터형의 값에 대해 수행합니다. 숫자를 리턴하는 MONTHS_BETWEEN 함수를 제외한 모든 날짜 함수는 날짜 데이터형의 값을 리턴합니다.
- 변환 함수: 어떤 데이터 형의 값을 다른 데이터형의 값으로 변환합니다.
- 일반적인 함수
 - NVL 함수
 - DECODE 함수

대소문자 변환 함수

- 문자 스트링에 대해 대소문자를 변환합니다.

함 수	결 과
LOWER('SQL Course')	sql course
UPPER('SQL Course')	SQL COURSE
INITCAP('SQLCourse')	Sql Course

대소문자 변환 함수

LOWER, UPPER, 그리고 INITCAP 은 대소문자 변환 함수입니다.

- LOWER: 대소문자가 혼합되어 있거나 대문자인 스트링을 소문자로 변환합니다.
- UPPER: 대소문자가 혼합되어 있거나 소문자인 스트링을 대문자로 변환합니다.
- INITCAP: 각 단어의 첫번째 문자를 대문자로, 나머지 문자는 소문자로 변환합니다.

```
SQL> SELECT 'The job title for '||INITCAP(ename)||' is '  
2      ||LOWER(job) AS "EMPLOYEE DETAILS"  
3      FROM emp;
```

EMPLOYEE DETAILS

The job title for King is president

The job title for Blake is manager

The job title for Clark is manager

...

14 rows selected.

대소문자 변환 함수 사용

- 직원 **Blake**에 대해서 직원 번호, 이름 그리고 부서번호를 디스플레이 합니다.

```
SQL> SELECT empno, ename, deptno
2 FROM emp
3 WHERE ename = 'blake';
no rows selected
```

```
SQL> SELECT empno, ename, deptno
2 FROM emp
3 WHERE LOWER(ename) = 'blake';
```

EMPNO	ENAME	DEPTNO
7698	BLAKE	30

대소문자 변환 함수 (계속)

위의 예는 종업원 BLAKE의 종업원 번호, 이름 그리고 부서번호를 디스플레이 합니다.

첫 번째 SQL 문장의 WHERE 절은 종업원 이름을 'blake'로 명시합니다. EMP 테이블의 모든 데이터는 대문자로 저장되어 있으므로, 'blake'는 EMP 테이블에서 찾을 수 없어서 "no rows selected"를 리턴합니다.

두 번째 SQL 문장의 WHERE 절은 EMP 테이블의 종업원 이름을 소문자로 변환한 다음에 blake와 비교하기를 명시합니다. 이제 양쪽의 이름이 소문자이므로 일치되는 하나의 행이 선택됩니다. WHERE 절은 똑같은 결과를 얻기 위해서 다음의 방법으로 다시 작성할 수 있습니다 :

```
... WHERE ename = 'BLAKE'
```

결과에서 이름은 데이터베이스에 저장된 대로 나타납니다. 이름에서 첫 번째 문자를 대문자로 디스플레이 하기 위해서 SELECT 문장에서 INITCAP 함수를 사용합니다.

```
SQL> SELECT empno, INITCAP(ename), deptno
2 FROM emp
3 WHERE LOWER(ename) = 'blake';
```

문자 조작 함수

- 문자 스트링 조작

함 수	결 과
CONCAT('Good', 'String')	GoodString
SUBSTR('String',1,3)	Str
LENGTH('String')	6
INSTR('String', 'r')	3
LPAD(sal,10,'*')	*****5000

문자 조작 함수

CONCAT, SUBSTR, LENGTH, INSTR, LPAD 는 본 과정에서 다루는 다섯 가지의 문자 조작 함수입니다.

- CONCAT: 값을 함께 합성합니다. CONCAT는 두 개의 매개변수만 사용할 수 있습니다.
- SUBSTR: 지정된 길이만큼의 스트링을 추출합니다.
- LENGTH: 스트링의 길이를 숫자 값으로 리턴합니다.
- INSTR: 명명된 문자의 위치를 숫자 값으로 리턴합니다.
- LPAD: 문자 값을 우측부터 채웁니다.

주: RPAD 문자 조작 함수는 문자 값을 좌측부터 채웁니다.

문자 조작 함수 사용

```
SQL> SELECT ename, CONCAT (ename, job), LENGTH(ename),
2 INSTR(ename, 'A')
3 FROM emp
4 WHERE SUBSTR(job,1,5) = 'SALES';
```

ENAME	CONCAT (ENAME, JOB)	LENGTH (ENAME)	INSTR(ENAME, 'A')
MARTIN	MARTINSALESMAN	6	2
ALLEN	ALLENSALESMAN	5	1
TURNER	TURNERSALESMAN	6	0
WARD	WARDSALESMAN	4	2

문자 조작 함수 (계속)

위의 예는, 업무가 sales 인 모든 종업원에 대해서 함께 합성된 종업원의 이름과 업무, 종업원 이름의 길이 그리고 종업원 이름에서 문자 A 의 숫자적인 위치를 디스플레이 합니다.

예

이름이 N 으로 끝나는 종업원에 대해서 데이터를 디스플레이 하기 위해서 위의 SQL 문장을 수정합니다.

```
SQL> SELECT      ename, CONCAT(ename, job), LENGTH(ename), INSTR(ename,
'A')
2 FROM          emp
3 WHERE         SUBSTR(ename, -1, 1) = 'N';
```

ENAME	CONCAT (ENAME, JOB)	LENGTH (ENAME)	INSTR (ENAME, 'A')
MARTIN	MARTINSALESMAN	6	2
ALLEN	ALLENSALESMAN	5	1

숫자 함수

- **ROUND:** 명시된 소수점으로 반올림
ROUND(45.926, 2) → 45.93
- **TRUNC:** 명시된 소수점으로 절삭
TRUNC(45.926, 2) → 45.92
- **MOD:** 나누기를 한 후의 나머지
MOD(1600, 300) → 100

숫자 함수

숫자 함수는 숫자 입력을 받고 숫자 값을 리턴합니다.

함 수	목 적
<code>ROUND(column expression, n)</code>	열, 표현식 또는 값을 소수점 n 자리까지 반올림합니다. n 이 생략되면 소수점이 없어집니다. n 이 음수이면 소수점의 왼쪽 자리 수만큼 반올림됩니다.
<code>TRUNC(column expression, n)</code>	열, 표현식 또는 값을 소수점 n 자리까지 절삭합니다. n 이 생략되면 소수점이 없어집니다. n 이 음수이면, 소수점의 왼쪽 자리 수만큼 절삭됩니다.
<code>MOD(m, n)</code>	m 을 n 으로 나눈 나머지를 리턴합니다.

주: 이 목록은 이용 가능한 숫자 함수의 일부분입니다.

숫자 함수에 대해 보다 자세한 정보를 알고자 하다면 다음을 참조하십시오.
- *Oracle Server SQL Reference, Release 7.3 또는 8.0, "Number Functions."*

ROUND 함수 사용

SQL> SELECT	ROUND (45.923,2),	ROUND (45.923,0),
2	ROUND (45.923,-1)	
3 FROM	DUAL;	
<hr/>		
ROUND (45.923,2)	ROUND (45.923,0)	ROUND (45.923,-1)
-----	-----	-----
45.92	46	50

ROUND 함수

ROUND 함수는 열, 표현식 또는 값을 소수점 n 자리로 반올림 합니다. 두 번째 인자가 0 이거나 생략되면, 값은 소수점 위치가 0 으로 반올림 됩니다. 두 번째 인자가 2 이면, 값은 소수점 아래 두 번째 위치로 반올림 됩니다. 반대로, 두 번째 인자가 -2 이면, 값은 소수점 좌측의 두 번째 위치로 반올림 됩니다.

ROUND 함수는 날짜 함수에 함께 사용될 수도 있습니다. 본 과정의 뒤에서 예제를 볼 수 있습니다.

DUAL 은 더미 테이블입니다. 이것에 대한 보다 자세한 것은 뒤에서 다루도록 합니다.

TRUNC 함수 사용

```
SQL> SELECT TRUNC(45.923, 2), TRUNC(45.923),  
2          TRUNC(45.923, -1)  
3 FROM DUAL;
```

TRUNC(45.923, 2)	TRUNC(45.923)	TRUNC(45.923, -1)
45.92	45	40

TRUNC 함수

TRUNC 함수는 열, 표현식 또는 값을 소수점 n 자리로 절삭합니다.

TRUNC 함수는 ROUND 함수와 유사한 **인수**로 수행합니다. 두 번째 인자가 0 이거나 생략되면, 값은 소수점 위치를 0 으로 절삭합니다. 두 번째 **인수**가 2 이면, 값은 두 개의 소수점 아래 위치로 절삭합니다. 반대로 두 번째 인자가 -2 이면, 값은 소수점 좌측부터 두 자리 까지로 절삭합니다.

ROUND 함수처럼 TRUNC 함수도 날짜 함수와 함께 사용될 수 있습니다.

MOD 함수 사용

- 업무가 **salesman**인 모든 종업원에 대해서 보너스에 대한 급여 비율의 나머지를 계산합니다.

```
SQL> SELECT ename, sal, comm, MOD(sal, comm)  
2 FROM emp  
3 WHERE job = 'SALESMAN';
```

ENAME	SAL	COMM	MOD(SAL, COMM)
MARTIN	1250	1400	1250
ALLEN	1600	300	100
TURNER	1500	0	1500
WARD	1250	500	250

MOD 함수

MOD 함수는 value1 을 value2 로 나눈 나머지를 찾습니다. 위의 예는 업무가 salesman 인 모든 종업원에 대해서 보너스에 대한 급여 비율의 나머지를 계산합니다.

날짜 작업

- 오라클은 세기, 년, 월, 일, 시간, 분, 초의 내부 숫자 형식으로 날짜를 저장합니다.
- 디폴트 날짜 형식은 'DD-MON-YY' 입니다.
- **SYSDATE**는 현재의 날짜와 시간을 리턴하는 함수입니다.
- **DUAL** 은 **SYSDATE** 를 보기 위해 사용된 dummy 테이블 입니다.

오라클 날짜 형식

오라클은 세기, 년, 월, 일, 시간, 분, 초의 내부 숫자 형식으로 날짜를 저장합니다. 날짜에 대한 디폴트 디스플레이와 입력 형식은 'DD-MON-YY' 입니다. 적절한 오라클 날짜의 범위는 January 1, 4712 B.C. 와 December 31, 9999 A.D. 사이 입니다.

SYSDATE

SYSDATE 는 현재의 날짜와 시간을 리턴하는 함수입니다. 다른 열 이름을 사용하듯이 SYSDATE 를 사용할 수 있습니다. 예를 들면, 테이블로부터 SYSDATE 를 선택하여 현재 날짜를 디스플레이 할 수 있습니다. DUAL 이라는 더미 테이블로부터 SYSDATE 를 선택하는 것이 관례입니다.

DUAL

DUAL 테이블은 SYS 에 의해 소유되며 모든 사용자가 액세스 할 수 있습니다. DUMMY 라는 하나의 열과 X 값을 가지는 하나의 행을 포함합니다. DUAL 테이블은 오직 한번만 값을 리턴하고자 할 때 유용합니다. 가령 사용자 데이터를 가진 테이블에서 파생되지 않은 상수, 의사열 또는 표현식의 값의 경우입니다.

Example

DUAL 테이블을 사용하여 현재 날짜를 디스플레이 합니다.

```
SQL> SELECT      SYSDATE
      2 FROM      DUAL;
```

날짜 연산

- 날짜에서 숫자를 더하거나 빼어 날짜 결과를 리턴합니다.
- 날짜 사이의 일 수를 알기 위해서 두 개의 날짜를 뺍니다.
- 시간을 24로 나누어서 날짜에 더합니다.

날짜 연산

데이터베이스는 날짜를 숫자로 저장하므로, 더하기와 빼기 같은 산술 연산자를 사용하여 계산을 수행할 수 있습니다. 날짜 뿐 만 아니라 숫자 상수를 더하거나 뺄 수도 있습니다.

다음의 작업을 수행할 수 있습니다:

작업	결과	설명
날짜 + 숫자	날짜	일 수를 날짜에 더합니다.
날짜 - 숫자	날짜	날짜에서 일 수를 뺍니다.
날짜 - 날짜	일수	어떤 날짜에서 다른 날짜를 뺍니다.
날짜 + 숫자/24	날짜	시간을 날짜에 더합니다.

날짜를 가진 산술 연산자 사용

```
SQL> SELECT ename, (SYSDATE-hiredate)/7 WEEKS  
2 FROM emp  
3 WHERE deptno = 10;
```

ENAME	WEEKS
KING	830.93709
CLARK	853.93709
MILLER	821.36566

날짜 연산 (계속)

위의 예는 부서 10에 속하는 모든 종업원에 대해서 이름과 근무한 주의 합계를 디스플레이 합니다. 현재 날짜(SYSDATE)에서 종업원이 입사한 날짜를 빼고 근무한 주의 합계를 구하기 위해서 7로 나눕니다.

주: SYSDATE는 현재 날짜와 시간을 리턴하는 SQL 함수입니다. 결과는 예제와 다를 수 있습니다.

날짜 함수

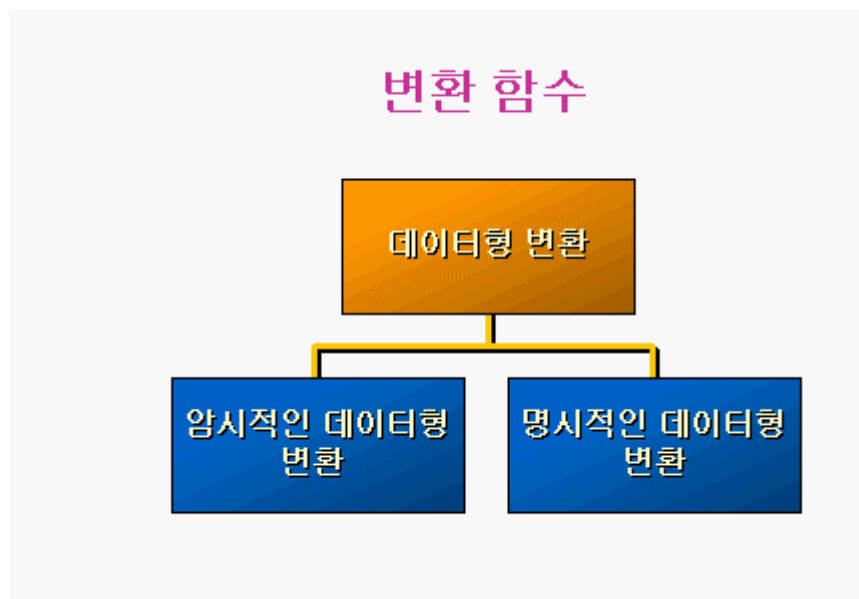
함 수	설 명
MONTHS_BETWEEN	두 날짜 사이의 월 수
ADD_MONTHS	월을 날짜에 더합니다.
NEXT_DAY	명시된 날짜로부터 다음 요일에 대한 날짜를 나타냅니다.
LAST_DAY	월의 마지막 날을 나타냅니다.
ROUND	날짜를 반올림 합니다.
TRUNC	날짜를 절삭합니다.

날짜 함수

날짜 함수는 오라클 날짜에 대해 연산을 합니다. 모든 날짜 함수는 숫자 값을 리턴하는 MONTHS_BETWEEN 을 제외하고는 DATE 데이터형의 값을 리턴합니다.

- MONTHS_BETWEEN(date1, date2): date1 과 date2 사이의 월 수를 리턴합니다. 결과는 음수 또는 양수가 될 수 있습니다. date1 이 date2 보다 이후이면 결과는 양수이고, date1 이 date2 보다 이전이면 결과는 음수입니다. 결과의 비정수 부분은 월의 부분을 나타냅니다.
- ADD_MONTHS(date, n): 월 수 n 을 date 에 더합니다. n 은 정수이어야 하며 음수일 수 있습니다.
- NEXT_DAY(date, 'char'): date 다음의 명시된 요일 ('char')의 날짜를 찾습니다. char는 요일이나 문자 스트링을 나타내는 숫자가 올 수 있습니다.
- LAST_DAY(date): date 를 포함하는 월의 마지막 날짜를 찾습니다.
- ROUND(date[, 'fmt']): 포맷 모델 fmt 에 명시된 단위에 대해 반올림한 date 를 리턴합니다. fmt 가 생략되면, date 를 가장 가까운 날짜로 반올림 됩니다.
- TRUNC(date[, 'fmt']): 포맷 모델 fmt 에 명시된 단위에 대해 절삭한 date 를 리턴합니다. fmt 가 생략되면, date 는 가장 가까운 날짜로 절삭됩니다.

이 목록은 이용 가능한 날짜 함수의 일부분입니다. 포맷 모델(fmt)은 이 장의 뒤에서 다루도록 합니다. 포맷 모델(fmt)의 예는 월 또는 년입니다.



변환 함수

Oracle8 데이터베이스 테이블의 열은 오라클 데이터형은 물론이고 ANSI, DB2 그리고 SQL/DS 데이터형을 사용하여 정의될 수 있습니다. 그러나, 오라클 서버는 내부적으로 그러한 데이터형을 Oracle8 데이터형으로 변환합니다.

오라클 서버는 어떤 일정한 데이터형의 데이터를 사용해야 하는 곳에, 그것과 다른 데이터형의 데이터를 사용할 수 있게 합니다. 이것은 오라클 서버가 자동적으로 데이터형을 변환할 수 있을 때 허용됩니다. 이 데이터형 변환은 오라클 서버에 의해서 암시적으로 행해지거나 또는 사용자에 의해서 명시적으로 행해질 수 있습니다.

암시적인 데이터형 변환은 다음의 두 슬라이드에 설명된 규칙에 따라서 수행됩니다.

명시적인 데이터형 변환은 변환 함수를 사용하여 수행됩니다. 변환 함수는 어떤 데이터형의 값을 다른 데이터형의 값으로 변환합니다.

주: 비록 암시적 데이터형 변환을 이용할 수 있더라도, SQL 문장의 안정성을 위해서 명시적 데이터형 변환을 할 것을 권장합니다.

암시적 데이터형 변환

- 값 할당(assignment) 시, 오라클은 자동으로 다음을 변환할 수 있습니다.

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

암시적 데이터형 변환

값 할당(assignment) 시, 오라클 서버는 다음을 자동으로 변환할 수 있습니다:

- VARCHAR2 또는 CHAR를 NUMBER로
- VARCHAR2 또는 CHAR를 DATE로
- NUMBER를 VARCHAR2로
- DATE를 VARCHAR2로

오라클 서버가 값 할당(assignment) 문장에서 사용된 값의 데이터형을 목표(target) 값의 데이터형으로 변환할 수 있을 경우에 할당(assignment) 문장은 올바르게 수행됩니다.

암시적 데이터형 변환

- 표현식 계산의 경우 오라클은 자동으로 다음을 변환할 수 있습니다.

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE

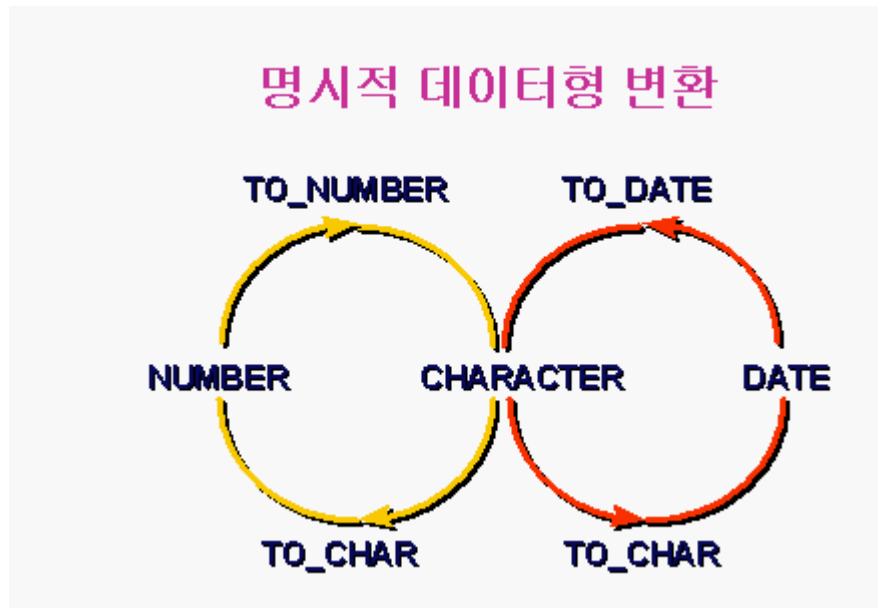
암시적 데이터형 변환

표현식 계산의 경우, 오라클 서버는 다음을 자동으로 변환할 수 있습니다:

- VARCHAR2 또는 CHAR를 NUMBER로
- VARCHAR2 또는 CHAR를 DATE로

일반적으로 오라클 서버는 할당(assignment) 시의 변환에 대한 규칙으로 데이터형 변환이 이루어질 수 없는 곳에서 데이터형 변환이 필요할 때 표현식에 대한 규칙을 사용합니다.

주: CHAR가 NUMBER로의 변환은 오직 문자 스트링이 적절한 숫자를 나타낼 때만 가능합니다. CHAR가 DATE로의 변환은 문자 스트링이 디폴트 형식 DD-MON-YY를 가질 경우에만 성공합니다.



명시적 데이터형 변환

SQL 은 어떤 데이터형 값을 다른 데이터형 값으로 변환하기 위해 아래 세 개의 함수를 제공합니다.

함 수	목 적
<code>TO_CHAR(<i>number</i> <i>date</i>,['<i>fmt</i>'])</code>	숫자나 문자 값을 포맷 모델 <i>fmt</i> 를 사용하여 VARCHAR2 문자 스트링으로 변환합니다
<code>TO_NUMBER(<i>char</i>)</code>	숫자를 포함하는 문자 스트링을 숫자로 변환합니다.
<code>TO_DATE(<i>char</i>,['<i>fmt</i>'])</code>	날짜를 나타내는 문자 스트링을 명시된 <i>fmt</i> 에 따라서 날짜 값으로 변환합니다. (<i>fmt</i> 가 생략 되면, DD-MON-YY 형식입니다.)

주: 이 목록은 이용할 수 있는 변환 함수의 일부분입니다.

변환 함수에 대해 보다 많은 정보를 알고자 한다면 다음을 참조하십시오.

- Oracle Server SQL Reference, Release 7.3 또는 8.0, “Conversion Functions.”

날짜를 가진 TO_CHAR 함수

TO_CHAR(*date*, '*fmt*')

***fmt*:**

- 단일 인용부호로 둘러싸여 있어야 합니다.
- 어떤 타당한 날짜 형식 요소도 포함할 수 있습니다.
- 추가된 공백을 제거하거나 앞부분의 0을 없애기 위해 **fm** 요소를 사용합니다.
- 데이터 값을 콤마로 구분합니다.

특정 형식으로 날짜를 디스플레이

이전의 모든 오라클 날짜 값은 DD-MON-YY 형식으로 디스플레이 되었습니다. TO_CHAR 함수는 이 디폴트 형식의 날짜를 여러분이 명시한 형식으로 변환합니다.

지침서

- 포맷 모델은 단일 인용부호로 둘러 싸여 있어야 하고 대소문자를 구분합니다.
- 포맷 모델은 어떤 타당한 날짜 형식도 포함할 수 있습니다. 포맷 모델의 날짜 값은 콤마로 구분합니다.
- 결과의 일 명과 월 명은 자동적으로 공백이 추가됩니다.
- 추가된 공백을 제거하거나 앞부분의 0을 없애기 위해서 fm 요소를 사용합니다.
- SQL*Plus COLUMN 명령어로 문자 필드 결과의 디스플레이 폭의 크기를 조절할 수 있습니다.
- 결과 열의 길이는 디폴트로 80자 입니다.

```
SQL> SELECT empno, TO_CHAR(hiredate, 'MM/YY') Month_Hired
2 FROM emp
3 WHERE ename = 'BLAKE';
```

날짜 형식 모델 요소

YYYY	년(숫자로 표현됨)
YEAR	년(문자로 표현됨)
MM	월(두자리의 숫자)
MONTH	월(문자로 표현됨)
DY	요일의 세 자리 약어
DAY	요일의 전체 이름

타당한 날짜 형식 요소의 예

요 소	설 명
SCC or CC	세기; BC 날짜에는 ?S 를 붙입니다.
Years in dates YYYY or SYYYY	년; BC 날짜에는 ?S 를 붙입니다.
YYY or YY or Y	년의 마지막 3, 2 또는 1 자리 수
Y,YYY	콤마가 있는 년
IYYY, IYY, IY, I	ISO 표준에 바탕을 둔 4, 3, 2 또는 1 자리 수
SYEAR or YEAR	문자로 표현된 년; BC 날짜에는 ?S 를 붙입니다.
BC or AD	BC/AD 지시자
B.C. or A.D.	. 이 있는 BC/AD 지시자
Q	년의 4 분의 1
MM	두자리 값의 월
MONTH	9 자리를 위해 공백을 추가한 월 이름
MON	세 자리의 약어로 된 월 이름
RM	로마 숫자 월

WW or W	년이나 월의 주
DDD or DD or D	년, 월 또는 주의 일
DAY	9 자리를 위해 공백을 추가한 요일 이름
DY	세 자리 약어로된 요일 이름
J	Julian day; BC 4713 년 12 월 31 일 이후의 요일 수

날짜 형식 모델 요소

- 시간 요소는 날짜의 시간 부분을 형식화 합니다.
HH24:MI:SS AM → 15:45:32 PM
- 문자 스트링에 이중 인용부호를 둘러싸서 문자 스트링을 추가 합니다.
DD "of" MONTH → 12 of OCTOBER
- 숫자의 접미사는 숫자를 문자로 명시 합니다.
ddsph → fourteenth

시간 형식

시간 정보와 리터럴을 디스플레이 하고 숫자를 명시된 숫자로 변경하기 위해서 다음의 테이블에 나열된 형식을 사용합니다.

요 소	설 명
AM or PM	정오 지시자
A.M. or P.M.	. 이 있는 정오 지시자
HH or HH12 or HH24	하루 중 시간 또는 시간(1-12) 또는 시간(0-23)
MI	분 (0-59)
SS	초 (0-59)
SSSSS	자정 이후의 초 (0-86399)

기타 형식

요 소	설 명
/ . ,	사용 문자가 결과에 다시 나타납니다.
"of the"	인용 부호내의 스트링이 결과에 다시 나타납니다.

숫자 디스플레이에 영향을 주는 접미사 명시

요 소	설 명
TH	서수 (예, DDTH for 4TH)
SP	명시한 수 (예, DDSP for FOUR)
SPTH or THSP	명시한 서수 (예, DDSPTH for FOURTH)

날짜를 가진 TO_CHAR 함수 사용

```
SQL> SELECT  ename,
2           TO_CHAR(hiredate, 'fmDD Month YYYY') HIREDATE
3 FROM      emp;
```

```
ENAME      HIREDATE
-----
KING       17 November 1981
BLAKE      1 May 1981
CLARK      9 June 1981
JONES      2 April 1981
MARTIN     28 September 1981
ALLEN     20 February 1981
...
14 rows selected.
```

날짜를 가진 TO_CHAR 함수

위의 SQL 문장은 모든 종업원에 대해서 이름과 입사일을 디스플레이 합니다. 입사일은 “17 November 1981”처럼 나타납니다.

예

날짜 형식이 “Seventh of February 1981 08:00:00 AM”처럼 나타나도록 위의 예를 수정합니다.

```
SQL> SELECT ename,
2         TO_CHAR(hiredate, 'fmDdspth "of" Month YYYY fmHH:MI:SS AM')
3         HIREDATE
4 FROM emp;
```

ENAME	HIREDATE

KING	Seventeenth of November 1981 12:00:00 AM
BLAKE	First of May 1981 12:00:00 AM
...	
14 rows selected.	

월은 명시된 형식 모델(INITCAP)을 따름을 주목하십시오.

숫자를 가진 TO_CHAR 함수

TO_CHAR(*number*, '*fmt*')
• TO_CHAR 함수를 사용하여 숫자 값을 문자로 디스플레이 하기 위해서 위의 형식을 사용 합니다.

9	숫자를 나타냅니다.
0	0이 디스플레이 되도록 합니다.
\$	달러 기호를 나타냅니다.
L	지역 화폐 기호를 사용합니다.
.	소수점을 출력합니다.
,	1000단위 지시자를 출력합니다.

숫자를 가진 TO_CHAR 함수

문자 스트링 같은 숫자 값으로 작업할 때, 그러한 숫자를 TO_CHAR 함수를 사용하여 문자

데이터형으로 변환해야 합니다. TO_CHAR 함수는 NUMBER 데이터형의 값을 VARCHAR2 로 변환합니다. 이 기법은 특히 연결(concatenation) 시에 유용합니다.

숫자 형식 요소

숫자를 문자 데이터형으로 변환한다면, 아래에 나열된 요소를 사용할 수 있습니다:

요 소	설 명	예	결과
9	숫자 위치(9 의 수는 디스플레이 폭을 결정합니다.)	999999	1234
0	앞에 0 을 디스플레이 합니다.	099999	001234
\$	달러 기호	\$999999	\$1234
L	지역 화폐 기호	L999999	FF1234
.	명시한 위치에 소수점	999999.99	1234.00
,	명시한 위치에 콤마	999,999	1,234
MI	우측에 마이너스 기호(음수 값)	999999MI	1234-
PR	음수를 괄호로 묶습니다.	999999PR	<1234>
EEEE	과학적인 부호표기(형식은 4 개의 E 를 명시해야 합니다.)	99.999EEEE	1.234E+03
V	10 을 n 번 곱합니다. (n = V 뒤의 9 의 수)	9999V99	123400
B	0 값을 0 이 아닌 공백으로 디스플레이 합니다.	B9999.99	1234.00

숫자를 가진 TO_CHAR 함수 사용

```
SQL> SELECT TO_CHAR(sal, '$99,999') SALARY
2 FROM emp
3 WHERE ename = 'SCOTT';
```

```
SALARY
-----
$3,000
```

지침서

- 오라클 서버는 형식 모델에 의해 제공되는 자릿수를 초과하는 숫자에 대해서는 파운드 기호 스트링(#)을 디스플레이 합니다.
- 오라클 서버는 저장된 소수 값을 형식 모델에서 제공하는 소수점 자리수로 반올림 합니다.

TO_NUMBER 와 TO_DATE 함수

- **TO_NUMBER** 함수를 사용하여 문자 스트링을 숫자 형식으로 변환합니다.

```
TO_NUMBER(char)
```

- **TO_DATE** 함수를 사용하여 문자 스트링을 날짜 형식으로 변환합니다.

```
TO_DATE(char[, 'fmt'])
```

TO_NUMBER 와 TO_DATE 함수

문자 스트링을 숫자나 날짜 형식으로 변환하기를 원할 수도 있습니다. 이런 일을 수행하기 위해 TO_NUMBER 나 TO_DATE 함수를 사용합니다. 여러분이 선택할 형식 모델은 앞의 형식 모델 요소를 바탕으로 합니다.

예

February 22, 1981 에 입사한 모든 종업원의 이름과 입사일을 디스플레이 합니다.

```
SQL> SELECT ename, hiredate
2 FROM emp
3 WHERE hiredate = TO_DATE('February 22, 1981', 'Month dd, YYYY');
```

ENAME	HIREDATE
-----	-----
WARD	22-FEB-81

NVL 함수

- **null** 값을 실제 값으로 변환합니다.
- 사용될 수 있는 데이터형은 날짜, 문자, 숫자입니다.
- 데이터형은 일치해야 합니다.
 - **NVL(comm,0)**
 - **NVL(hiredate,'01-JAN-97')**
 - **NVL(job,'No Job Yet')**

NVL 함수

null 값을 실제 값으로 변환하기 위해서, NVL 함수를 사용합니다.

구문형식

```
NVL (expr1, expr2)
```

여기서:

expr1 null 을 포함할 수 있는 소스 값이나 표현식입니다.

expr2 null 변환을 위한 목표(target) 값 입니다.

데이터형을 변환하는데 NVL 함수를 사용할 수 있는데, 리턴 값은 항상 expr1 의 데이터형과 같아야 합니다.

다양한 데이터형에 대한 NVL 변환

데이터형	변환 예
------	------

NUMBER	NVL(<i>number_column</i> , 9)
DATE	NVL(<i>date_column</i> , '01-JAN-95')
CHAR or VARCHAR2	NVL(<i>character_column</i> , 'Unavailable')

NVL 함수 사용

```
SQL> SELECT ename, sal, comm, (sal*12)+NVL(comm,0)
2 FROM emp;
```

ENAME	SAL	COMM	(SAL*12)+NVL(COMM,0)
KING	5000		60000
BLAKE	2850		34200
CLARK	2450		29400
JONES	2975		35700
MARTIN	1250	1400	16400
ALLEN	1600	300	19500
...			

14 rows selected.

NVL 함수

모든 종업원의 연봉을 계산하기 위해서 월 급여에 12를 곱하고 거기에 보너스를 더합니다.

```
SQL> SELECT ename, sal, comm, (sal*12)+comm
2 FROM emp;
```

ENAME	JOB	(SAL*12)+COMM

KING	PRESIDENT	
BLAKE	MANAGER	
CLARK	MANAGER	
JONES	MANAGER	
MARTIN	SALESMAN	16400

...

14 rows selected.

연봉은 보너스를 받는 종업원에 대해서만 계산됨을 주목하십시오. 표현식의 열 값이 null 이면 결과는 null 입니다. 모든 종업원에 대한 값을 계산하기 위해서는 산술 연산자를 적용하기 전에 null 값을 숫자로 변환해야 합니다. 슬라이드의 예에서 NVL 함수는 null 값을 0으로 변환하는데 사용됩니다.

DECODE 함수 사용

```
SQL> SELECT job, sal,  
2          DECODE(job, 'ANALYST', SAL*1.1,  
3                  'CLERK',   SAL*1.15,  
4                  'MANAGER', SAL*1.20,  
5                  SAL)  
6          REVISED_SALARY  
7 FROM emp;
```

JOB	SAL	REVISED_SALARY
PRESIDENT	5000	5000
MANAGER	2850	3420
MANAGER	2450	2940
...		

14 rows selected.

DECODE 함수 사용

위의 SQL 문장에서 JOB 의 값이 해독되었습니다. JOB 이 ANALYST 이면, 급여 증가는 10%이고 JOB 이 CLERK 이면, 급여 증가는 15% 입니다. JOB 이 MANAGER 라면, 급여 증가는 20% 입니다. 다른 업무에 대해서는 급여 증가분이 없습니다.

IF-THEN-ELSE 문장으로 똑같이 작성될 수 있습니다:

```
IF job = 'ANALYST'    THEN sal = sal*1.1  
IF job = 'CLERK'     THEN sal = sal*1.15  
IF job = 'MANAGER'   THEN sal = sal*1.20  
ELSE sal = sal
```

중첩 함수

- 단일 행 함수는 여러 레벨에 걸쳐 중첩될 수 있습니다.
- 중첩 함수는 가장 하위 레벨에서 상위 레벨 순으로 계산됩니다.



중첩 함수

단일 행 함수는 어떤 레벨까지도 중첩될 수 있습니다. 중첩 함수는 가장 내부의 레벨로부터 가장 외부의 레벨 순으로 계산됩니다. 다음 페이지의 예를 보십시오.

중첩 함수

```
SQL> SELECT  ename ,
2           NVL (TO_CHAR(mgr) , 'No Manager ')
3 FROM      emp
4 WHERE     mgr IS NULL;
```

ENAME	NVL (TO_CHAR (MGR) , 'NOMANAGER ')
KING	No Manager

중첩 함수 (계속)

위의 예는 관리자가 없는 회사의 대표자를 디스플레이 합니다. SQL 문장의 평가는 다음의

두 단계를 가집니다:

1. 숫자 값을 문자 값으로 변환하기 위해서 내부 함수를 계산합니다.

-Result1 = TO_CHAR(mgr)

2. null 값을 텍스트 스트링으로 대체하기 위해서 외부 함수를 계산합니다.

-NVL(Result1, 'No Manager')

모든 표현식은 주어진 열 **별칭**이 없으므로 열의 헤딩이 됩니다.

예

입사일로부터 6 개월이 지난후의 다음 금요일의 날짜를 디스플레이 합니다. 결과 날짜는 “Friday, March 12th, 1982”처럼 나타나야 합니다. 결과 정렬은 입사일 순입니다.

```
SQL> SELECT      TO_CHAR(NEXT_DAY(ADD_MONTHS
2              (hiredate, 6), 'FRIDAY'),
3              'fmDay, Month ddth, YYYY')
4              "Next 6 Month Review"
5 FROM          emp
6 ORDER BY      hiredate;
```

제 4 장 다중 테이블로부터 데이터 디스플레이

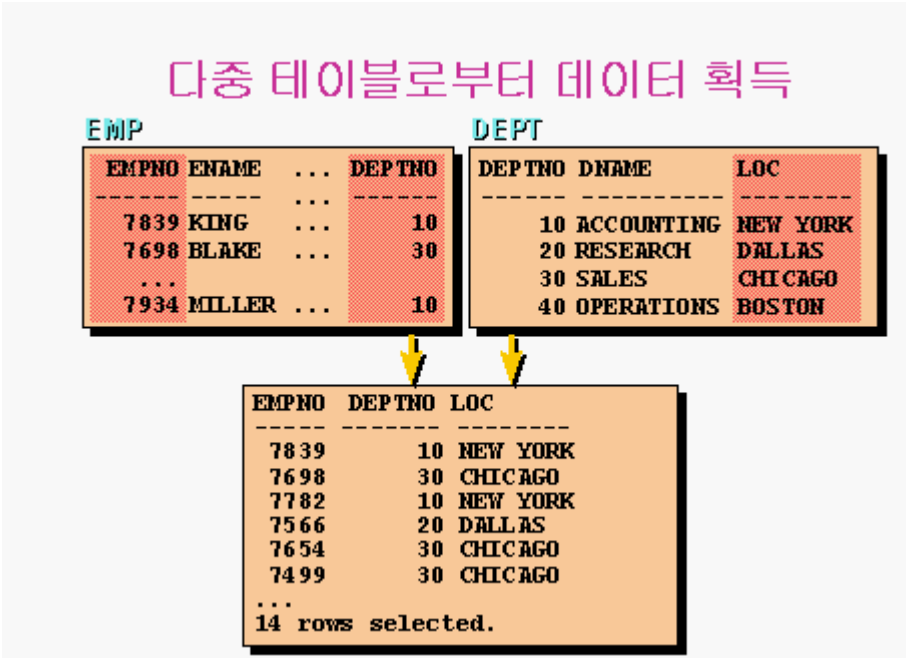
목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- **equality join**과 **nonequality join**을 사용하여 하나 이상의 테이블로부터 데이터를 액세스하는 **SELECT** 문장을 작성합니다.
- **outer join**을 사용하여 일반적인 조인 조건을 만족하지 않는 데이터를 출력합니다.
- 자체적으로 테이블을 조인합니다.

과정 목표

본 과정은 이용 가능한 각기 다른 방법들을 사용하여 하나 이상의 테이블로부터 데이터를 구하는 방법을 다룹니다.



다중 테이블로부터 데이터 획득

때때로 하나 이상의 테이블로부터 데이터를 사용할 필요가 있습니다. 위의 예에서 리포트는 두 개의 테이블로부터 데이터를 디스플레이 합니다.

- EMPNO 는 EMP 테이블에 있습니다.
- DEPTNO 는 EMP 와 DEPT 테이블 양쪽에 있습니다.
- LOC 는 DEPT 테이블에 있습니다.

리포트를 만들기 위해서는 EMP 와 DEPT 테이블을 링크하고 양쪽 테이블로부터 데이터를 액세스 합니다.

조인이란?

- 하나 이상의 테이블로부터 데이터를 질의하기 위해서 조인을 사용합니다.

```
SELECT  table1.column, table2.column
FROM    table1, table2
WHERE   table1.column1 = table2.column2;
```

- **WHERE** 절에 조인 조건을 작성합니다.
- 하나 이상의 테이블에 똑같은 열 이름이 있을 때 열 이름 앞에 테이블 이름을 붙입니다.

조인 정의

데이터베이스에 있는 하나 이상의 테이블로부터 데이터가 필요할 때 join 조건문을 사용합니다. 한 테이블의 행은 관련되는 열에 있는 공통 값에 따라서 즉, primary key와 foreign key 열에 따라서 다른 하나의 테이블 행과 조인 할 수 있습니다. 두 개 이상의 관계형 테이블로부터 데이터를 디스플레이 하기 위해서, WHERE 절에 간단한 조인 조건문을 작성합니다.

구문형식에서:

table.column 데이터를 검색할 테이블과 열을 나타냅니다.

table1.column1 = table2.column2 테이블을 함께 조인(또는 관련)하는 조건문입니다.

지침서

- 테이블을 조인하는 SELECT 문장을 작성할 때, 명확성을 위해서 그리고 데이터베이스 액세스를 향상 시키기 위해서 열 이름 앞에 테이블 명을 붙입니다.
- 똑같은 열 이름이 하나 이상의 테이블에 있으면, 열 이름 앞에 테이블 명이 있어야 합니다.
- n 개의 테이블을 함께 조인하려면, 최소(n-1) 개의 조인 조건문이 필요합니다. 그러므로, 4개의 테이블을 조인하려면 최소한 3개의 조인문이 필요합니다. 이 규칙은 테이블이 연결된(Concatenated) primary key를 가진다면 적용될 수가 없습니다.

SELECT 에 대한 보다 자세한 내용을 알고자 한다면 다음을 참조하십시오.

- *Oracle Server SQL Reference Manual, Release 7.3 또는 8.0*, “SELECT.”

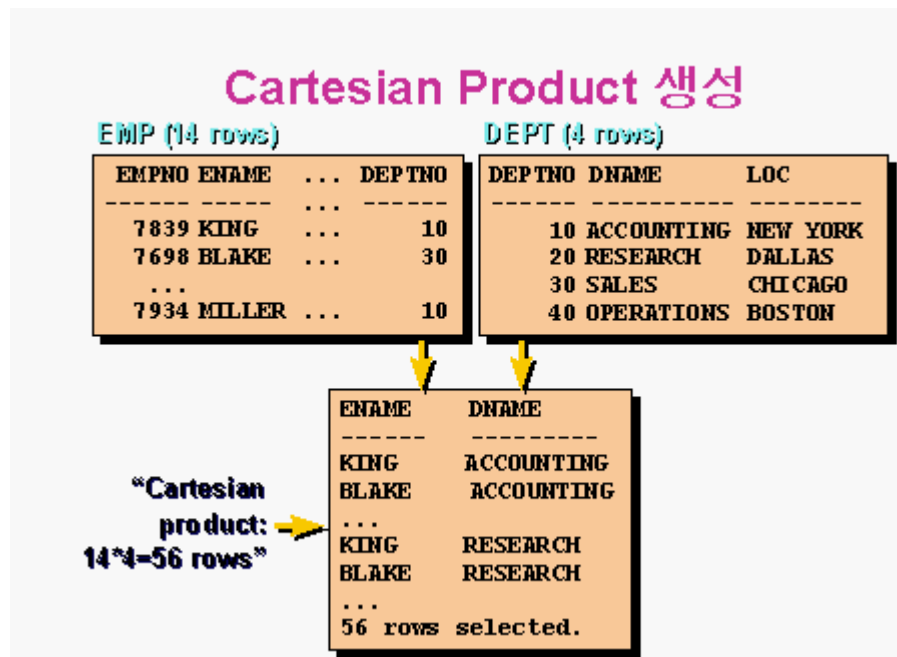
Cartesian Product

- **Cartesian product** 는 다음의 경우에 발생합니다.
 - 조인 조건이 생략된 경우
 - 조인 조건이 잘못된 경우
 - 첫 번째 테이블의 모든 행이 두 번째 테이블의 모든 행과 조인되는 경우
- **Cartesian product**를 피하기 위해서는 항상 **WHERE** 절에 올바른 조인 조건문을 쓰도록 합니다.

Cartesian Product

조인 조건문이 잘못되었거나 완전히 생략되었을 때 모든 행들의 조합인 Cartesian product 라는 결과가 디스플레이 됩니다. 첫 번째 테이블의 모든 행은 두 번째 테이블의 모든 행에 조인됩니다.

Cartesian product 는 많은 수의 행을 생성하는 경향이 있으며, 결과도 거의 유용하지 못합니다. 특별히 모든 테이블로부터 모든 행을 조합할 필요가 없을 경우, 항상 WHERE 절에 올바른 조인 조건문을 적어야 합니다.



Cartesian Product (계속)

Cartesian product 는 조인 조건이 생략될 때 발생합니다. 슬라이드의 예는 EMP 테이블과 DEPT 테이블로부터 종업원 이름과 부서 이름을 디스플레이 합니다. WHERE 절이 명시되지 않았기 때문에 EMP 테이블의 모든 행(14 행)은 DEPT 테이블의 모든 행(4 행)과 조인 되어서 결과에 56 행을 발생합니다.

```
SQL> SELECT  ename, dname
2 FROM      emp, dept;
```

```
ENAME      DNAME
-----
KING        ACCOUNTING
BLAKE       ACCOUNTING
...
KING        RESEARCH
BLAKE       RESEARCH
...
56 rows selected.
```


조인의 유형



조인의 유형

조인 조건에는 두 가지의 중요한 유형이 있습니다:

- Equijoins
- Non-equijoins

추가적인 조인 방법은 다음과 같습니다:

- Outer joins
- Self joins
- Set operators

Equijoin 이란?

EMP			DEPT		
EMPNO	ENAME	DEPTNO	DEPTNO	DNAME	LOC
7839	KING	10	10	ACCOUNTING	NEW YORK
7698	BLAKE	30	30	SALES	CHICAGO
7782	CLARK	10	10	ACCOUNTING	NEW YORK
7566	JONES	20	20	RESEARCH	DALLAS
7654	MARTIN	30	30	SALES	CHICAGO
7499	ALLEN	30	30	SALES	CHICAGO
7844	TURNER	30	30	SALES	CHICAGO
7900	JAMES	30	30	SALES	CHICAGO
7521	WARD	30	30	SALES	CHICAGO
7902	FORD	20	20	RESEARCH	DALLAS
7369	SMITH	20	20	RESEARCH	DALLAS
...			...		
14 rows selected.			14 rows selected.		

Primary key
Foreign key

Equijoin

종업원의 부서 이름을 결정하기 위해서 EMP 테이블의 DEPTNO 열 값과 DEPT 테이블의 DEPTNO 열 값을 비교합니다. EMP 테이블과 DEPT 테이블 사이의 관계성은 equijoin - 즉, 양쪽 테이블의 DEPTNO 열 값이 같아야 합니다. 이 열들은 primary key 와 foreign key 가 됩니다.

주: Equijoins 은 단순 조인 또는 내부 조인이라고도 합니다.

Equijoin 이란?

EMP			DEPT		
EMPNO	ENAME	DEPTNO	DEPTNO	DNAME	LOC
7839	KING	10	10	ACCOUNTING	NEW YORK
7698	BLAKE	30	30	SALES	CHICAGO
7782	CLARK	10	10	ACCOUNTING	NEW YORK
7566	JONES	20	20	RESEARCH	DALLAS
7654	MARTIN	30	30	SALES	CHICAGO
7499	ALLEN	30	30	SALES	CHICAGO
7844	TURNER	30	30	SALES	CHICAGO
7900	JAMES	30	30	SALES	CHICAGO
7521	WARD	30	30	SALES	CHICAGO
7902	FORD	20	20	RESEARCH	DALLAS
7369	SMITH	20	20	RESEARCH	DALLAS
...			...		
14 rows selected.			14 rows selected.		

Foreign key
Primary key

Equijoin 으로 레코드 검색

위의 예에서 :

- SELECT 절은 검색할 열 이름을 명시합니다:
 - EMP 테이블에 있는 열인 종업원 이름, 종업원 번호 그리고 부서번호
 - DEPT 테이블에 있는 열인 부서 번호와 위치
- FROM 절은 데이터베이스가 액세스 해야 하는 두 개의 테이블을 명시합니다:
 - EMP 테이블
 - DEPT 테이블
- WHERE 절은 테이블이 조인되는 형태를 명시합니다:
 - EMP.DEPTNO=DEPT.DEPTNO

DEPTNO 열이 양쪽 테이블에 공통으로 있으므로 모호함을 피하기 위해서 열 이름 앞에 테이블 명을 작성해야 합니다.

AND 연산자를 사용하는 추가적인 검색 조건

EMP			DEPT		
EMPNO	ENAME	DEPTNO	DEPTNO	DNAME	LOC
7839	KING	10	10	ACCOUNTING	NEW YORK
7698	BLAKE	30	30	SALES	CHICAGO
7782	CLARK	10	10	ACCOUNTING	NEW YORK
7566	JONES	20	20	RESEARCH	DALLAS
7654	MARTIN	30	30	SALES	CHICAGO
7499	ALLEN	30	30	SALES	CHICAGO
7844	TURNER	30	30	SALES	CHICAGO
7900	JAMES	30	30	SALES	CHICAGO
7521	WARD	30	30	SALES	CHICAGO
7902	FORD	20	20	RESEARCH	DALLAS
7369	SMITH	20	20	RESEARCH	DALLAS
...			...		
14 rows selected.			14 rows selected.		

추가적인 검색 조건

조인 이외에 WHERE 절에 추가적인 조건을 가질 수 있습니다. 예를 들면, 종업원 King 의 종업원 번호, 이름, 부서번호 그리고 부서 위치를 디스플레이 하기 위해서는 WHERE 절에 추가적인 조건이 필요합니다.

```
SQL> SELECT      empno, ename, emp.deptno, loc
2 FROM          emp, dept
3 WHERE         emp.deptno = dept.deptno
4 AND          INITCAP(ename) = 'King';
```

EMPNO	ENAME	DEPTNO	LOC
7839	KING	10	NEW YORK

테이블 별칭(Alias) 사용

- 테이블 별칭을 사용하여 질의를 간단하게 합니다.

```
SQL> SELECT emp.empno, emp.ename, emp.deptno,
2      dept.deptno, dept.loc
3 FROM emp, dept
4 WHERE emp.deptno=dept.deptno;
```

```
SQL> SELECT e.empno, e.ename, e.deptno,
2      d.deptno, d.loc
3 FROM emp e, dept d
4 WHERE e.deptno=d.deptno;
```

테이블 별칭

테이블 이름으로 열 이름을 제한하는 것은 테이블 명이 길 경우에는 많은 시간을 낭비할 수 있습니다. 테이블 이름 대신에 테이블 **alias** 를 사용할 수도 있습니다. 열 **별칭**이 다른 이름의 열을 주는 것처럼 테이블 **별칭**은 다른 이름의 테이블을 줍니다. 테이블 **별칭**은 SQL 코드를 보다 적게 하도록 도와 줌으로써 메모리를 보다 적게 사용합니다.

예에서, 테이블 **별칭**이 FROM 절에서 어떻게 식별 되는지를 주목하십시오. 테이블 명을 전부 명시하고 공백을 둔 다음 테이블 **별칭**이 명시됩니다. EMP 테이블에 **별칭** E를 주었고, DEPT 테이블에서 **별칭** D를 주었습니다.

지침서

- 테이블 **별칭**은 30자까지 가능하지만 짧을수록 더 좋습니다.
- 테이블 **별칭**이 FROM 절에서 특별한 테이블 명을 위해서 사용된다면, 테이블 **별칭**은 SELECT 문장 전체에서 테이블 명 대신에 사용되어야 합니다.
- 테이블 **별칭**은 의미가 있어야 합니다.
- 테이블 **별칭**은 현재의 SELECT 문장에 대해서만 유효합니다.

두개 이상의 테이블을 조인

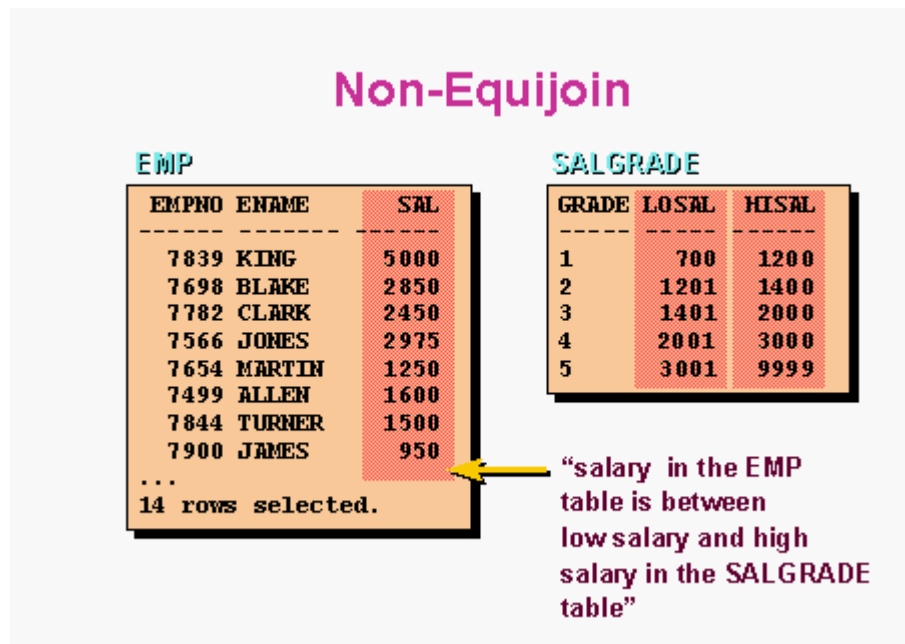
CUSTOMER		ORD		ITEM	
NAME	CUSTID	CUSTID	ORDID	ORDID	ITEMID
JOCKSPORTS	100	101	610	610	3
TKB SPORT SHOP	101	102	611	611	1
VOLLYRITE	102	104	612	612	1
JUST TENNIS	103	106	601	601	1
K+T SPORTS	105	102	602	602	1
SHAPE UP	106	106
WOMENS SPORTS	107	106
...
9 rows selected.		21 rows		64 rows selected.	

추가적인 검색조건

때때로 두개 이상의 테이블을 조인할 필요가 있습니다. 예를 들면, 고객 TKB SPORT SHOP의 이름, 주문처, 항목수, 각 항목의 합계 그리고 각 주문의 합계를 디스플레이 하기 위해서는 CUSTOMER, ORD 그리고 ITEM 테이블을 조인해야 합니다.

```
SQL> SELECT c.name, o.ordid, i.itemid, i.itemtot, o.total
2 FROM customer c, ord o, item i
3 WHERE c.custid = o.custid
4 AND o.ordid = i.ordid
5 AND c.name = 'TKB SPORT SHOP';
```

NAME	ORDID	ITEMID	ITEMTOT	TOTAL
TKB SPORT SHOP	610	3	58	101.4
TKB SPORT SHOP	610	1	35	101.4
TKB SPORT SHOP	610	2	8.4	101.4



Non-Equijoin

EMP 와 SALGRADE 테이블 사이의 관계성은 EMP 테이블의 어떠한 열도 직접적으로 SALGRADE 테이블의 열에 관련되지 않음을 의미하는 non-equijoin 입니다. 두 테이블 사이의 관련성은 EMP 테이블의 SAL 열이 SALGRADE 테이블의 LOSAL 과 HISAL 열 사이에 있다는 것입니다. 관련성은 동등(=)이 아닌 다른 연산자를 사용하여 얻어집니다.

Non-Equijoin으로 레코드 검색

```
SQL> SELECT  e.ename, e.sal, s.grade
2 FROM      emp e, salgrade s
3 WHERE     e.sal
4 BETWEEN   s.losal AND s.hisal;
```

ENAME	SAL	GRADE
JAMES	950	1
SMITH	800	1
ADAMS	1100	1
...		

14 rows selected.

Non-Equijoin (계속)

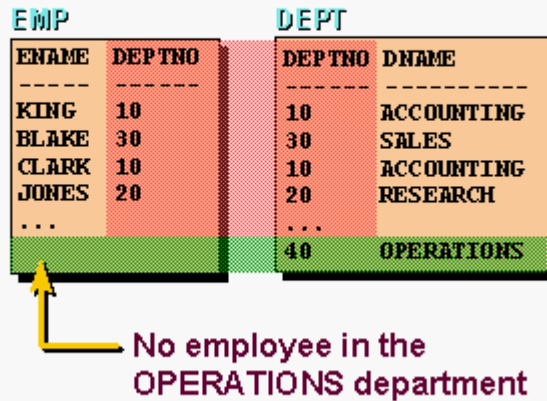
위의 예는 종업원의 급여 등급을 계산하기 위해서 non-equijoin 을 생성합니다. 급여는 하한 값과 상한 값 사이에 있어야 합니다.

이 질의가 실행될 때 모든 종업원들은 정확히 한 번만 나타남을 주목하십시오. 리스트에서 중복되는 종업원은 없습니다. 이에 대한 두 가지의 이유가 있습니다:

- 급여 등급 테이블에서 중복되는 등급을 포함하는 행은 하나도 없습니다. 즉, 종업원의 급여 값은 급여 등급 테이블 중의 어떤 행의 최소 급여와 최대 급여 사이에만 있을 수 있습니다.
- 모든 종업원의 급여는 급여 등급 테이블에 의해 제공된 제한 값 내에 있습니다. 즉, LOSAL 열에 포함된 가장 낮은 값보다 작거나 HISAL 열에 포함된 가장 높은 값보다 많이 버는 종업원은 없습니다.

주: <= 와 >= 같은 다른 연산자를 사용할 수 있지만, BETWEEN 이 가장 간단합니다. BETWEEN 을 사용할 때 하한 값을 먼저 명시하고 상한 값은 나중에 명시함을 명심하십시오. 테이블 **별칭**은 모호성의 가능성 때문이 아니라 성능 이유 때문에 명시되어 있습니다.

Outer Join



Outer Join 으로 직접 일치하는 것이 없는 레코드 리턴

행이 조인 조건을 만족하지 않으면, 행은 질의 결과에 나타나지 않을 것입니다. 예를 들면, EMP 와 DEPT 테이블의 equijoin 조건에서 부서 OPERATIONS 는 해당 부서에 아무도 없기 때문에 나타나지 않습니다.

```
SQL> SELECT e.ename, e.deptno, d.dname
2 FROM emp e, dept d
3 WHERE e.deptno = d.deptno;
```

ENAME	DEPTNO	DNAME
KING	10	ACCOUNTING
BLAKE	30	SALES
CLARK	10	ACCOUNTING
JONES	20	RESEARCH
...		
ALLEN	30	SALES
TURNER	30	SALES
JAMES	30	SALES

...

14 rows selected.

Outer Join

- 조인 조건을 만족하지 않는 행들도 보기 위해서 **outer join**을 사용합니다.
- **Outer join** 연산자는 더하기 기호(+)입니다.

```
SELECT  table.column, table.column
FROM    table1, table2
WHERE   table1.column(+) = table2.column;
```

```
SELECT  table.column, table.column
FROM    table1, table2
WHERE   table1.column = table2.column(+);
```

Outer Join 으로 직접 일치하는 것이 없는 레코드 리턴

조인 조건에 outer join 연산자를 사용하면 빠졌던 행들도 리턴됩니다. 연산자는 괄호로 둘러싸인 더하기 기호(+)이며, 정보가 부족한 조인쪽에 위치합니다. 이 연산자는 한 개 이상의 null 행을 생성하고, 조인 가능한 결함이 없는 테이블의 하나 이상의 행들이 이런 null 행에 조인됩니다.

구문형식에서:

table1.column = table2.column (+)

테이블을 함께 조인(또는 관련)시키는 조건입니다. outer join 심볼입니다. WHERE 절 조건의 양쪽이 아니라 어느 한쪽에 둘 수 있습니다. 일치하는 행이 없는 테이블의 열 이름 뒤에 **outer join** 연산자를 둡니다.

Outer Join 사용

```
SQL> SELECT  e.ename, d.deptno, d.dname
2  FROM      emp e, dept d
3  WHERE     e.deptno(+) = d.deptno
4  ORDER BY e.deptno;
```

ENAME	DEPTNO	DNAME
KING	10	ACCOUNTING
CLARK	10	ACCOUNTING
...		
		40 OPERATIONS
15 rows selected.		

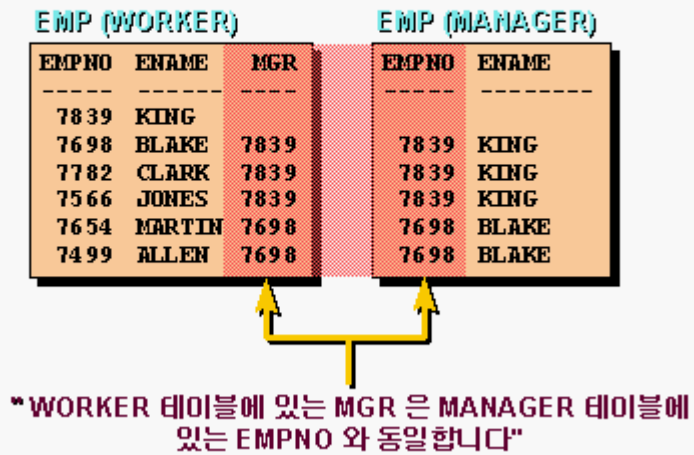
Outer Join 으로 직접 일치하는 것이 없는 레코드 리턴 (계속)

위의 예는 모든 부서의 번호와 이름을 디스플레이 합니다. 종업원이 없는 OPERATIONS 부서도 마찬가지로 디스플레이 됩니다.

Outer Join 제약사항

- outer join 연산자는 정보가 부재하는 쪽의 표현식 한 쪽에만 둡니다.?다른 테이블의 어떠한 열과도 직접적으로 일치하는 것이 없는 한 테이블의 행을 리턴합니다.
- outer join 을 포함하는 조건은 IN 연산자를 사용할 수 없고, OR 연산자에 의해 다른 조건과 연결될 수 없습니다.

Self Join



자체적으로 테이블 조인

때때로 자체적으로 테이블을 조인할 필요가 있습니다. 각 종업원의 관리자 명을 알기 위해서 자체적으로 EMP 테이블을 조인하는게 필요합니다. 예를 들면, Blake 의 관리자 명을 알려면, 다음이 필요합니다:

- ENAME 열을 검사하여 EMP 테이블에서 Blake를 찾습니다.
- MGR 열을 검사하여 Blake 에 대한 관리자 번호를 찾습니다. Blake의 관리자 번호는 7839입니다.
- ENAME 열을 검사하여 EMPNO 가 7839 인 관리자 명을 찾습니다. King의 종업원 번호는 7839 입니다. 그러므로 King 은 Blake의 관리자입니다.

이 처리에서는 테이블을 두 번 검사합니다. 첫 번째는 테이블에서 ENAME 열이 Blake 이고 MGR 값이 7839 인 것을 찾기 위해서 검사합니다. 두 번째는 EMPNO 열이 7839 이고 ENAME 열이 King 인 것을 찾습니다.

자체적으로 테이블 조인

```
SQL> SELECT worker.ename||' works for '||manager.ename  
2 FROM emp worker, emp manager  
3 WHERE worker.mgr = manager.empno;
```

```
WORKER.ENAME||'WORKSFOR'||MANAG  
-----  
BLAKE works for KING  
CLARK works for KING  
JONES works for KING  
MARTIN works for BLAKE  
...  
13 rows selected.
```

자체적으로 테이블 조인 (계속)

위의 예는, 자체적으로 EMP 테이블을 조인합니다. 같은 테이블 EMP에 대해 두 개의 별칭 WORKER와 MANAGER를 사용함으로써 FROM 절에 두 개의 테이블을 사용하는 것과 같습니다.

이 예에서 WHERE 절은 “worker의 관리자 번호가 manager의 종업원 번호와 일치하는 상황”을 의미하는 조인을 포함합니다,

제 5 장 그룹 함수를 사용한 데이터 집계

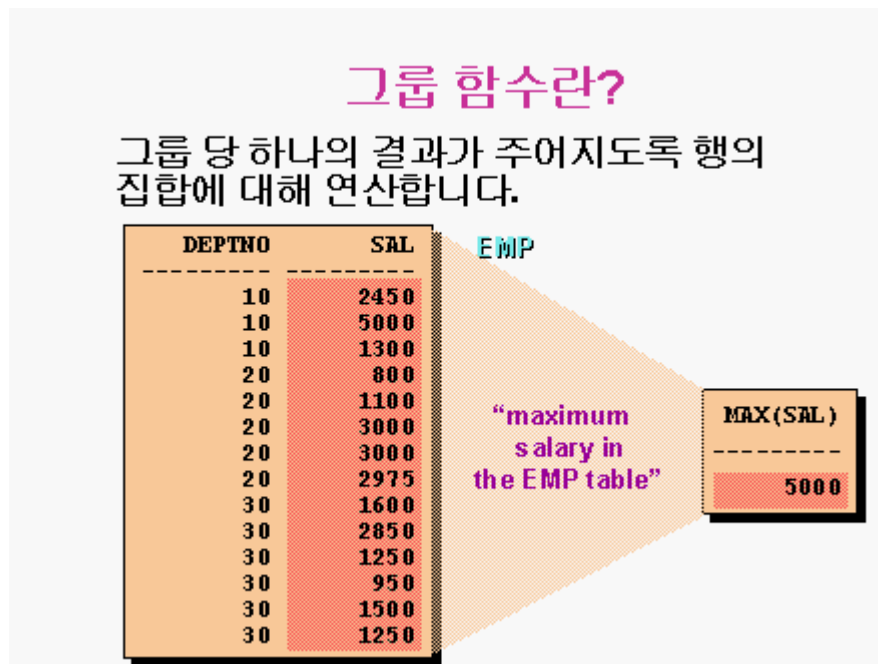
목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 이용 가능한 그룹 함수를 식별합니다.
- 그룹 함수의 사용을 기술합니다.
- **GROUP BY** 절을 사용하여 데이터를 그룹화합니다.
- **HAVING** 절을 사용하여 그룹된 행을 포함하거나 제외합니다.

과정 목표

본 과정은 함수에 대해서 더 배웁니다. 행의 그룹에 대해서 평균과 같은 통계 정보를 얻는데 초점을 둡니다. 한 테이블의 행들을 더 적은 단위의 집합으로 그룹화 시키는 방법과 행의 그룹에 대해서 조건을 명시하는 방법을 토의합니다.



그룹 함수

단일 행 함수와는 달리 그룹 함수는 그룹 당 하나의 결과가 주어지도록 행의 집합에 대해 연산합니다. 이러한 집합은 전체 테이블이거나 특정 그룹일 수 있습니다.

그룹 함수의 유형

- AVG
- COUNT
- MAX
- MIN
- STDDEV
- SUM
- VARIANCE

그룹 함수 (계속)

각각의 함수는 인수(argument)를 받습니다. 다음의 테이블은 구문형식에서 사용할 수 있는 옵션을 보여줍니다.

함 수	설 명
AVG([DISTINCT <u>ALL</u>]n)	null 값을 무시한 n 의 평균 값
COUNT({* [DISTINCT <u>ALL</u>]expr})	행의 수, expr 은 null 값을 제외하고 계산합니다. *를 사용하여 중복되거나 null 인 행들을 포함하여 모든 행을 계산합니다.
MAX([DISTINCT <u>ALL</u>]expr)	Null 값을 무시한 expr 의 최대 값
MIN([DISTINCT <u>ALL</u>]expr)	Null 값을 무시한 expr 의 최소 값
STDDEV([DISTINCT <u>ALL</u>]x)	Null 값을 무시한 n 의 표준편차
SUM([DISTINCT <u>ALL</u>]n)	Null 값을 무시한 n 의 합계
VARIANCE([DISTINCT <u>ALL</u>]x)	Null 값을 무시한 n 의 분산

AVG와 SUM 함수 사용

숫자 데이터에 대해서 **AVG** 과 **SUM** 을 사용할 수 있습니다.

```
SQL> SELECT AVG(sal), MAX(sal),  
2          MIN(sal), SUM(sal)  
3 FROM emp  
4 WHERE job LIKE 'SALES%';
```

AVG (SAL)	MAX (SAL)	MIN (SAL)	SUM (SAL)
1400	1600	1250	5600

그룹 함수

숫자 데이터를 저장할 수 있는 열에 대해서 AVG, SUM, MIN 그리고 MAX 함수를 사용할 수 있습니다. 위의 예는 모든 판매원에 대해서 급여의 평균, 최고액, 최저액 그리고 합계를 디스플레이 합니다.

MIN과 MAX 함수 사용

모든 데이터형에 대해서 **MIN** 과 **MAX** 를 사용할 수 있습니다.

```
SQL> SELECT MIN(hiredate), MAX(hiredate)  
2 FROM emp;
```

MIN (HIRED)	MAX (HIRED)
17-DEC-80	12-JAN-83

그룹 함수 (계속)

모든 데이터형에 대해서 MAX 와 MIN 함수를 사용할 수 있습니다. 위의 예는 가장 먼저

입사한 종업원과 가장 나중에 입사한 종업원을 디스플레이 합니다.

아래의 예는 모든 종업원에 대해서 알파벳순으로 제일 빠른 종업원의 이름과 제일 늦은 종업원을 디스플레이 합니다.

```
SQL> SELECT      MIN(ename), MAX(ename)
      2 FROM      emp;
```

```
MIN( ENAME )  MAX( ENAME )
-----
ADAMS         WARD
```

주: AVG, SUM, VARIANCE 그리고 STDDEV 함수는 숫자 데이터형에서만 사용될 수 있습니다.

COUNT 함수사용

COUNT(*) : 테이블 행의 수를 리턴합니다.

```
SQL> SELECT COUNT(*)
      2 FROM   emp
      3 WHERE  deptno = 30;
```

```
COUNT(*)
-----
        6
```

COUNT 함수

COUNT 함수는 두 가지 형식이 있습니다:

- COUNT(*)
- COUNT(expr)

COUNT(*) 는 중복되는 행과 null 값을 포함하는 행을 포함하여 테이블 행의 수를 리턴합니다.

반대로, COUNT(expr) 는 expr 에 의해 인식된 열에서 null 이 아닌 행의 수를 리턴합니다. 위의 예는 부서 30 의 종업원 수를 디스플레이 합니다.

COUNT 함수 사용

COUNT(expr) : NULL 이 아닌 행의 수를 리턴합니다.

```
SQL> SELECT COUNT(comm)
2 FROM emp
3 WHERE deptno = 30;
```

COUNT (COMM)
4

COUNT 함수 (계속)

위의 예는 보너스를 받는 부서 30 의 종업원 수를 디스플레이 합니다. 부서 30 의 두 명의 종업원이 보너스를 받을 수 없고 COMM 열에 null 값을 포함하므로 행의 전체 수가 4 가 되었다는 것을 주목하십시오.

예

EMP 테이블의 부서 수를 디스플레이 합니다.

SQL> SELECT COUNT(deptno)
2 FROM emp;
COUNT (DEPTNO)

14

EMP 테이블의 중복되지 않는 부서 수를 디스플레이 합니다.

SQL> SELECT	COUNT(DISTINCT (deptno))
2 FROM	emp;
COUNT(DISTINCT(DEPTNO))	

	3

그룹 함수와 Null 값

그룹 함수는 열에 있는 **NULL** 값을 무시합니다.

```
SQL> SELECT AVG(comm)
2 FROM emp;
```

```
AVG(COMM)
-----
550
```

그룹 함수와 Null 값

COUNT (*) 를 제외한 모든 그룹 함수는 열에 있는 null 값을 무시합니다. 위의 예에서 평균은 테이블의 COMM 열에 저장되어 있는 타당한 값의 행에 대해서만 계산됩니다. 평균은 모든 종업원에게 지급되는 보너스를 보너스를 받는 종업원의 수(4)로 나누어 계산합니다.

그룹 함수와 NVL 함수 사용

NVL 함수는 그룹 함수가 **NULL** 값을 포함하는 것을 가능하게 합니다.

```
SQL> SELECT AVG(NVL(comm,0))  
2 FROM emp;
```

```
AVG(NVL(COMM,0))  
-----  
157.14286
```

그룹 함수와 Null 값 (계속)

NVL 함수는 그룹 함수가 null 값을 포함하는 것을 가능하게 합니다. 위의 예에서 평균은 테이블의 COMM 열에 저장된 값이 null 인지 여 상관 없이 모든 행을 계산합니다. 평균은 모든 종업원에게 지급되는 보너스를 전체 종업원의 수(14)로 나누어 계산됩니다.

데이터 그룹 생성

EMP

DEPTNO	SAL		DEPTNO	AVG(SAL)
10	2450		10	2916.6667
10	5000	2916.6667		
10	1300			
20	800		20	2175
20	1100	2175		
20	3000			
20	3000			
20	2975			
30	1600		30	1566.6667
30	2850	1566.6667		
30	1250			
30	950			
30	1500			
30	1250			

“average
salary
in EMP
table
for each
department”

데이터 그룹

지금까지 모든 그룹 함수는 테이블을 하나의 큰 그룹으로 다루었습니다. 여기서는 테이블의 데이터를 보다 작은 그룹으로 나누는 것이 필요합니다. 이것은 GROUP BY 절을 사용하여 수행할 수 있습니다.

데이터 그룹 생성: GROUP BY 절

```
SELECT      column, group_function(column)
FROM        table
[WHERE      condition]
[GROUP BY   group_by_expression]
[ORDER BY   column] ;
```

테이블의 행을 **GROUP BY** 절을 사용하여 보다 작은 그룹으로 나눕니다.

GROUP BY 절

테이블의 행을 그룹으로 나누기 위해서 GROUP BY 절을 사용할 수 있습니다. 그런 다음에 각각의 그룹에 대해서 통계정보를 리턴하기 위해서 그룹 함수를 사용할 수 있습니다.

구문형식에서:

group_by_expression 행을 그룹핑 하기 위한 기준이 될 열 값을 명시합니다.

지침

- SELECT 절에 그룹 함수를 포함한다면, GROUP BY 절에 각각의 열이 명시되지 않으면 각각의 그룹별 결과를 얻을 수 없습니다.
- WHERE 절을 사용하여 행을 그룹으로 나누기 전에 미리 행을 제외시킬 수 있습니다.
- GROUP BY 절에 열을 포함해야 합니다.
- GROUP BY 절에 열 **별칭**을 사용할 수 없습니다.

- 기본적으로, 행들은 GROUP BY 목록에 포함된 열의 오름차순 정렬로 저장됩니다. ORDER BY 절을 사용하여 이것을 변경할 수 있습니다.

GROUP BY 절 사용

SELECT 절에서 그룹 함수 안에 없는 열들은 **GROUP BY** 절에 있어야 합니다.

```
SQL> SELECT deptno, AVG(sal)
2 FROM emp
3 GROUP BY deptno;
```

DEPTNO	AVG(SAL)
10	2916.6667
20	2175
30	1566.6667

GROUP BY 절 (계속)

GROUP BY 절을 사용할 때, 그룹 함수에 포함되지 않는 SELECT 절의 모든 열들은 GROUP BY 절에 포함되도록 합니다. 위의 예는 각각의 종업원에 대해서 부서 번호와 평균 급여를 디스플레이 합니다. 다음은 GROUP BY 절을 포함하는 위의 SELECT 문장을 계산하는 방법입니다.

- SELECT 절은 검색할 열을 명시합니다.
 - EMP 테이블의 부서번호 열
 - GROUP BY 절에 명시된 그룹의 모든 급여 평균
- FROM 절은 데이터베이스가 액세스해야 할 테이블을 명시합니다: EMP 테이블
- WHERE 절은 검색할 행을 명시합니다. WHERE 절이 없기 때문에 디폴트로 모든 행이 검색됩니다.
- GROUP BY 절은 행을 그룹핑 하는 방법을 명시합니다. 행은 부서번호에 의해 그룹화 되어 있으므로 급여 열에 적용된 AVG 함수는 각각의 부서에 대한 평균 급여를 계산할 것입니다.

GROUP BY 절 사용

GROUP BY 절의 열은 **SELECT** 절에 꼭 있을 필요는 없습니다.

```
SQL> SELECT    AVG(sal)
  2 FROM      emp
  3 GROUP BY deptno;
```

AVG(SAL)
2916.6667
2175
1566.6667

GROUP BY 절 (계속)

GROUP BY 열은 SELECT 절에 꼭 있어야 하는 것은 아닙니다. 예를 들면, 위의 SELECT 문장은 디스플레이 하지 않는 각각의 부서에 대해서 평균 급여를 디스플레이 합니다. 그러나 부서번호가 없는 결과는 의미가 없어 보입니다. ORDER BY 절에 그룹 함수를 사용할 수 있습니다.

```
SQL> SELECT    deptno, AVG(sal)
  2 FROM      emp
  3 GROUP BY    deptno
  4 ORDER BY    AVG(sal);
```

DEPTNO	AVG(SAL)
30	1566.6667
20	2175
10	2916.6667

하나 이상의 열로 그룹화

EMP

DEPTNO	JOB	SAL
10	MANAGER	2450
10	PRESIDENT	5000
10	CLERK	1300
20	CLERK	300
20	CLERK	1100
20	ANALYST	3000
20	ANALYST	3000
20	MANAGER	2975
30	SALESMAN	1600
30	MANAGER	2850
30	SALESMAN	1250
30	CLERK	950
30	SALESMAN	1500
30	SALESMAN	1250

"sum salaries in the EMP table for each job, grouped by department"

DEPTNO	JOB	SUM(SAL)
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	6000
20	CLERK	1900
20	MANAGER	2975
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600

그룹내의 그룹

때때로 그룹내의 그룹에 대한 결과를 알아야 할 필요가 있습니다. 위의 예는 각 부서내의 업무별로 급여 합계를 디스플레이 하는 리포트를 보여 줍니다.

EMP 테이블은 먼저 부서번호로 그룹화한 다음에 업무로 그룹화합니다. 예를 들면, 부서 20에 있는 두 개의 Clerk은 함께 그룹화 되었고, 단일 결과(급여 합계)는 그룹내의 모든 판매원에 대해서 산출되었습니다.

다중 열에서 GROUP BY 절 사용

```
SQL> SELECT deptno, job, sum(sal)
2 FROM emp
3 GROUP BY deptno, job;
```

DEPTNO	JOB	SUM(SAL)
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	6000
20	CLERK	1900
...		

9 rows selected.

그룹내의 그룹 (계속)

GROUP BY 절에 하나 이상의 열을 사용하여 그룹과 서브 그룹에 대한 통계 결과를 리턴할 수 있습니다. GROUP BY 절에 있는 열의 순서에 의해서 결과의 디폴트 정렬 순서를 결정할 수 있습니다. 다음은 GROUP BY 절을 포함하는 위의 SELECT 문장이 계산되는 단계를 보여 줍니다.

- SELECT 절은 검색할 열을 명시합니다:
 - EMP 테이블의 부서번호
 - EMP 테이블의 업무GROUP BY 절에 명시된 그룹의 모든 급여를 더합니다.
- FROM 절에 데이터베이스가 액세스할 테이블을 명시합니다: EMP 테이블.
- GROUP BY 절은 행을 그룹화하는 방법을 명시합니다:
 - 먼저, 부서번호로 행을 그룹화 합니다.
 - 두 번째로 부서번호 그룹 내에서 업무로 행을 그룹화 합니다.

SUM 함수는 각각의 부서번호 그룹 내의 모든 업무에 대한 급여 열에 적용됩니다.

그룹 함수를 잘못 사용한 질의

SELECT 절의 그룹 함수가 아닌 모든 열이나 표현식은 **GROUP BY** 절에 있어야 합니다.

```
SQL> SELECT deptno, COUNT(ename)
2 FROM emp;
```

```
SELECT deptno, COUNT(ename)
      *
ERROR at line 1:
ORA-00937: not a single-group group function
```

그룹 함수를 잘못 사용한 질의

같은 SELECT 문장에 개별적인 열(DEPTNO)과 그룹 함수(COUNT)를 혼합해서 사용할 때, 개별적인 열(이 경우에는 DEPTNO)을 명시하는 GROUP BY 절을 포함해야 합니다. GROUP

BY 절이 없으면 “not a single-group group function” 이라는 에러 메시지가 나타나고 ‘*’로 잘못 된 열을 가리킵니다. 위의 예러는 GROUP BY 절을 추가하여 정정할 수 있습니다.

```
SQL> SELECT      deptno, COUNT(ename)
2 FROM          emp
3 GROUP BY      deptno;
```

DEPTNO	COUNT(ENAME)
-----	-----
10	3
20	5
30	6

주: 그룹 함수가 아닌 SELECT 절의 어떤 열이나 표현식은 GROUP BY 절에 있어야 합니다.

그룹 함수를 잘못 사용한 질의

- **WHERE** 절을 사용하여 그룹을 제한할 수 없습니다.
- 그룹을 제한 하기 위해서 **HAVING** 절을 사용합니다.

```
SQL> SELECT      deptno, AVG(sal)
2 FROM          emp
3 WHERE         AVG(sal) > 2000
4 GROUP BY      deptno;
```

```
WHERE AVG(sal) > 2000
*
ERROR at line 3:
ORA-00934: group function is not allowed here
```

그룹 함수를 잘못 사용한 질의 (계속)

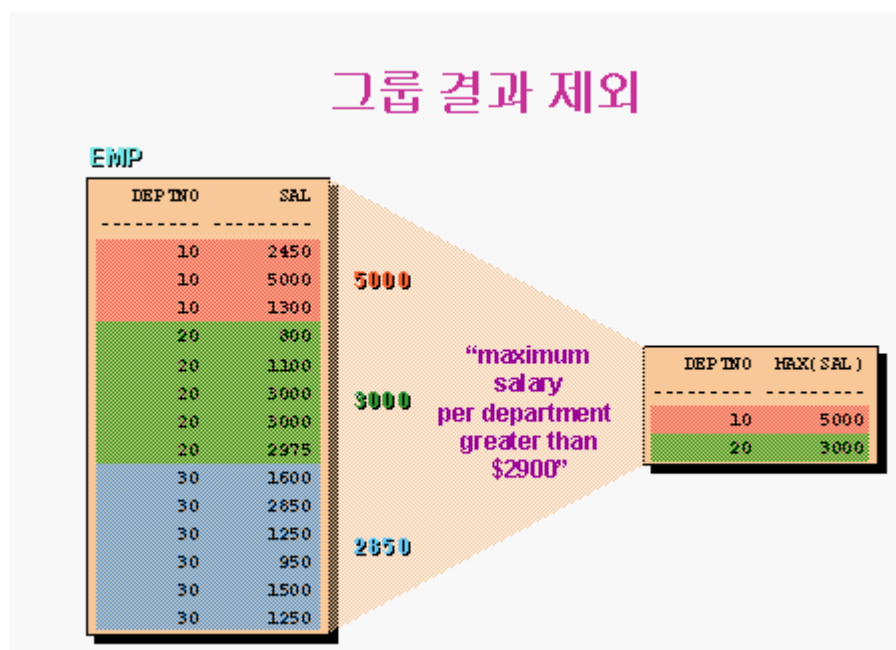
WHERE 절을 사용하여 그룹을 제한할 수 없습니다. 위의 SELECT 문장은 평균 급여가 \$2000 이상인 부서의 평균 급여를 제한하기 위해서 WHERE 절을 사용했기 때문에 에러가

발생합니다.

위의 예러는 HAVING 절을 사용하여 정정할 수 있습니다.

```
SQL> SELECT deptno, AVG(sal)
2 FROM emp
3 GROUP BY deptno
4 HAVING AVG(sal) > 2000;
```

DEPTNO	AVG (SAL)
10	2916.6667
20	2175



그룹 결과 제한

WHERE 절을 사용하여 검색하는 행을 제한하는 것과 똑같은 방법으로 HAVING 절을 사용하여 그룹을 제한합니다. 최대 급여가 \$2900 이상인 부서만을 나타내기 위해서 각 부서의 최대 급여를 알고자 한다면 다음의 두 가지 일을 해야 합니다:

1. 부서번호로 그룹핑하여 각 부서의 평균 급여를 알아냅니다.
2. 최대 급여가 \$2900 이상인 부서들로 그룹을 제한합니다.

그룹 결과 제외: HAVING 절

HAVING 절을 사용하여 그룹을 제한합니다.

- 행을 그룹화 합니다.
- 그룹 함수를 적용합니다.
- **HAVING 절**과 일치하는 그룹을 디스플레이 합니다.

```
SELECT      column, group_function
FROM        table
[WHERE      condition]
[GROUP BY   group_by expression]
[HAVING     group_condition]
[ORDER BY   column] ;
```

HAVING 절

HAVING 절을 사용하여 디스플레이될 그룹을 명시합니다. 그러므로, 그룹 정보를 바탕으로 그룹을 한층 더 제한합니다.

구문형식에서:

group_condition

행의 그룹을 제한하여 명시된 조건이 TRUE 인 그룹들만 리턴되도록 합니다.

오라클 서버는 HAVING 절을 사용할 때 다음의 단계를 수행합니다:

- 행을 그룹화 합니다.
- 그룹 함수를 그룹에 적용합니다.
- HAVING 절에 있는 조건과 일치하는 그룹을 디스플레이 합니다.

HAVING 절은 GROUP BY 절 앞에 올 수 있지만, GROUP BY 절을 먼저 두는 게 더 논리적이므로 권장됩니다. HAVING 절이 SELECT 절에 있는 그룹에 적용되기 전에 그룹은 구성되고 그룹 함수는 계산됩니다.

HAVING 절 사용

```
SQL> SELECT deptno, max(sal)
2 FROM emp
3 GROUP BY deptno
4 HAVING max(sal) > 2900;
```

DEPTNO	MAX(SAL)
10	5000
20	3000

HAVING 절 (계속)

위의 예는 최대 급여가 \$2900 이상인 부서에 대해서 부서번호와 최대 급여를 디스플레이 합니다.

SELECT 절에 그룹 함수를 사용하지 않고 GROUP BY 절만 사용할 수 있습니다.

그룹 함수의 결과를 근거로 행을 제한한다면, HAVING 절 뿐만 아니라 GROUP BY 절을 사용해야 합니다.

다음의 예는 최대 급여가 \$2900 이상인 부서에 대해서 부서번호와 평균 급여를 디스플레이 합니다.

```
SQL> SELECT deptno, AVG(sal)
2 FROM emp
3 GROUP BY deptno
4 HAVING MAX(sal) > 2900;
```

DEPTNO	AVG(SAL)
10	2916.6667
20	2175

HAVING 절 사용

```
SQL> SELECT      job, SUM(sal) PAYROLL
2  FROM          emp
3  WHERE         job NOT LIKE 'SALES%'
4  GROUP BY     job
5  HAVING        SUM(sal) > 5000
6  ORDER BY     SUM(sal) ;
```

JOB	PAYROLL
ANALYST	6000
MANAGER	8275

HAVING 절 (계속)

위의 예는 전체 수입이 \$5000 을 초과하는 각 업무에 대해서 업무와 월 급여 합계를 디스플레이 합니다. 예는 판매원은 제외하고 월 급여 합계로 결과를 정렬합니다.

제 6 장 서브쿼리 (Subquery)

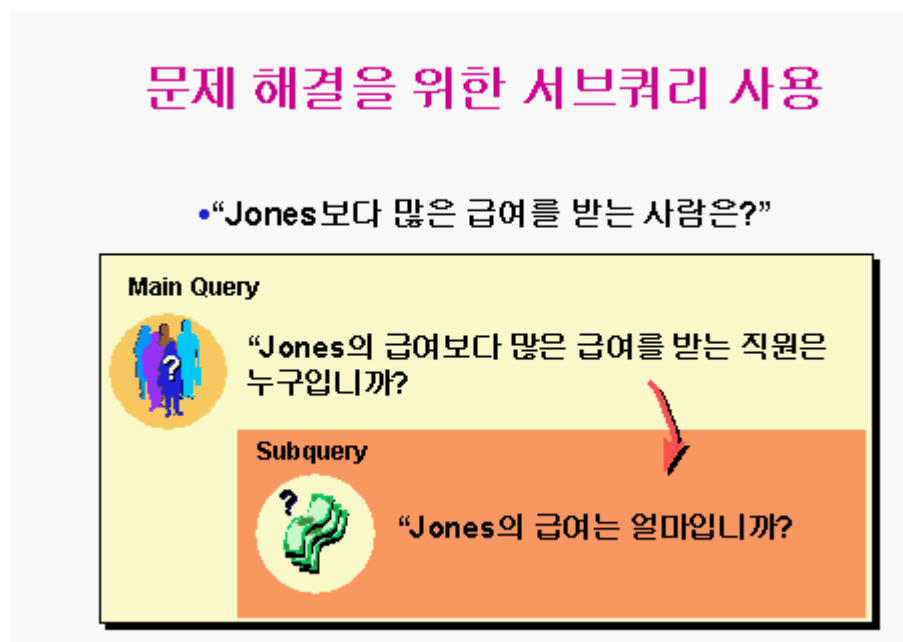
목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 서브쿼리가 해결할 수 있는 문제의 유형을 기술합니다.
- 서브쿼리를 정의합니다.
- 서브쿼리의 유형을 나열합니다.
- 단일 행 서브쿼리와 다중 행 서브쿼리를 작성합니다.

과정 목표

본 과정에서는 SELECT 문장의 보다 진보된 특징에 대해서 배울 것입니다. 알려지지 않은 조건 값을 근거로 값을 구하기 위해 다른 SQL 문장의 WHERE 절에 서브쿼리를 작성합니다. 본 과정은 단일 행 서브쿼리와 다중 행 서브쿼리를 다룹니다.



문제를 해결하기 위한 서브쿼리 사용

Jones 보다 많은 급여를 받는 종업원을 찾기 위한 질의를 작성하고자 합니다.

이 문제를 해결하기 위해서 두개의 질의가 필요합니다. 하나의 질의는 Jones 의 급여를 알기 위한 것이고, 두번째 질의는 그것보다 많은 급여를 받는 종업원을 찾는 것입니다. 하나의 질의를 다른 질의 내부에 두는 방법으로 두개의 질의를 조합하여 이 문제를 해결할 수 있습니다.

내부 질의 또는 서브쿼리는 외부질의 또는 메인 쿼리(main query)에 의해 사용되는 값을 리턴합니다. 서브쿼리의 사용은 두개의 질의를 순차적으로 수행하여 두번째 질의의 검색 값으로 첫번째 질의의 결과를 사용하는 것입니다.

서브쿼리

```
SELECT    select_list
FROM      table
WHERE     expr operator
          (SELECT    select_list
           FROM      table);
```

- 서브쿼리(내부 질의)는 메인 쿼리 이전에 실행합니다.
- 서브쿼리의 결과는 메인 쿼리(외부 질의)에 의해 사용됩니다.

서브쿼리

서브쿼리는 다른 SELECT 문장의 절에 **내장**된 SELECT 문장입니다. 서브쿼리를 사용하여 간 단한 문장을 강력한 문장으로 만들 수 있습니다. 테이블 자체의 데이터에 의존하는 조건으로 테이블의 행을 검색할 필요가 있을 때 서브쿼리는 아주 유용합니다.

다음의 SQL 절에 서브쿼리를 작성할 수 있습니다.

- WHERE 절
- HAVING 절
- FROM 절

구문형식에서:

operator >, = 또는 IN 같은 **비교 연산자**를 포함합니다.

주: **비교 연산자**는 단일 행 연산자(>, =, >=, <, <>, <=)와 다중 행 연산자(IN, ANY, ALL)가 있습니다.

서브쿼리는 종종 중첩된 SELECT, 부속 SELECT 또는 내부 SELECT 문장으로 불리워집니다. 일반적으로 서브쿼리를 먼저 실행하고, 그것의 결과를 메인 쿼리 또는 외부 질의에 대한 질의 조건을 완성하는데 사용합니다.

서브쿼리 사용

```
SQL> SELECT ename  
2 FROM emp  
3 WHERE sal > 2975  
4 (SELECT sal  
5 FROM emp  
6 WHERE empno=7566) ;
```

ENAME

KING
FORD
SCOTT

서브쿼리 사용

슬라이드에서 내부 질의는 종업원 7566의 급여를 결정합니다. 외부 질의는 내부 질의의 결과를 받으며, 이 급여보다 많이 받는 모든 종업원을 디스플레이하기 위해 이 결과를 사용합니다.

서브쿼리 사용 지침

- 서브쿼리는 괄호로 둘러싸야 합니다.
- 서브쿼리는 비교 연산자의 오른쪽에 있어야 합니다.
- 서브쿼리에 **ORDER BY** 절을 포함하지 마십시오.
- 단일 행 서브쿼리에는 단일 행 연산자를 사용하십시오.
- 다중 행 서브쿼리에는 다중 행 연산자를 사용하십시오.

서브쿼리 사용 지침

- 서브쿼리는 괄호로 둘러싸야 합니다.

- 서브쿼리는 **비교 연산자**의 오른쪽에 있어야 합니다.
- 서브쿼리는 ORDER BY 절을 포함할 수 없습니다. SELECT 문장에 대해서는 오직 하나의 ORDER BY 절을 가질 수 있으며, SELECT 문장의 제일 마지막에 있어야 합니다.
- 서브쿼리에서는 두 종류의 **비교 연산자**를 사용합니다. 단일 행 **비교 연산자**와 다중 행 **비교 연산자**.

단일 행 서브쿼리

- 오직 하나의 행만을 리턴합니다.
- 단일 행 비교 연산자를 사용합니다.

연산자	의미
=	같다
>	보다 크다
>=	보다 크거나 같다
<	보다 작다
<=	보다 작거나 같다
<>	같지 않다

단일 행 서브쿼리

단일 행 서브쿼리는 내부 SELECT 문장으로부터 하나의 행을 리턴하는 질의입니다. 이런 유형의 서브쿼리는 단일 행 연산자를 사용합니다. 슬라이드는 단일 행 연산자의 목록을 보여줍니다.

예

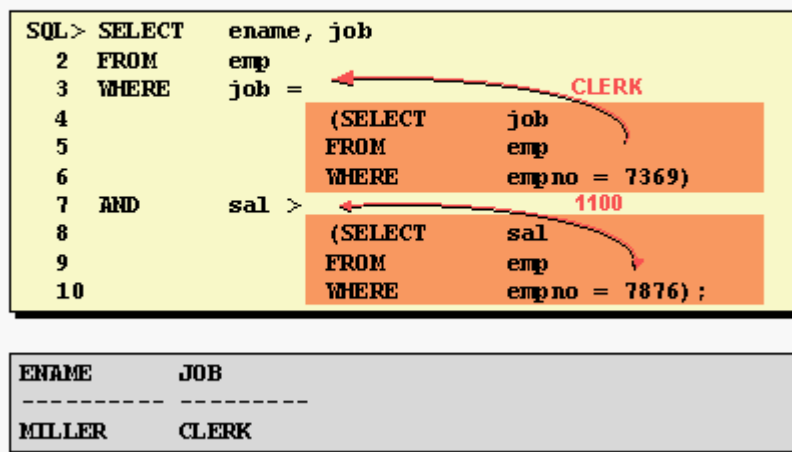
업무가 종업원 7369 과 똑같은 종업원을 디스플레이합니다.

```
SQL> SELECT      ename, job
2 FROM          emp
3 WHERE         job =
4                (SELECT job
```

5	FROM	emp
6	WHERE	empno = 7369);

ENAME	JOB
-----	-----
JAMES	CLERK
SMITH	CLERK
ADAMS	CLERK
MILLER	CLERK

단일 행 서브쿼리 실행



단일 행 서브쿼리 실행

위의 예는 업무가 종업원 7369의 업무와 같고, 급여가 종업원 7876보다 많은 종업원을 디스플레이합니다.

예는 세개의 질의 블록으로 구성됩니다: 한 개의 외부 질의와 두 개의 내부 질의. 내부 질의 블록들이 먼저 실행되고, 질의 결과 CLERK과 1100을 리턴합니다. 그런 다음에 외부 질의 블록이 처리되고 외부 질의의 검색 조건을 완성하기 위해 내부 질의에 의해 리턴된 값을 사용합니다.

내부 질의 두개는 단일 값(CLERK 과 1100)을 리턴하므로 이 SQL 문장을 단일 행 서브쿼리라고 부릅니다.

주: 외부 질의와 내부 질의는 다른 테이블로부터 데이터를 구할 수 있습니다.

서브쿼리를 가진 HAVING 절

- 오라클 서버는 먼저 서브쿼리를 실행합니다.
- 오라클 서버는 결과를 메인 쿼리의 **HAVING** 절에 리턴합니다.

```
SQL> SELECT      deptno, MIN(sal)
2 FROM          emp
3 GROUP BY      deptno
4 HAVING        MIN(sal) >
5              (SELECT MIN(sal)
6 FROM          emp
7 WHERE         deptno = 20);
```

서브쿼리를 가진 HAVING 절

서브쿼리를 WHERE 절 뿐만아니라 HAVING 절에서도 사용할 수 있습니다. 오라클 서버는 서브쿼리를 실행하고 그 결과를 메인 쿼리의 HAVING 절에 리턴합니다.

슬라이드의 SQL 문장은 부서 20 보다 최소 급여가 많은 모든 부서를 디스플레이합니다.

DEPTNO	MIN(SAL)
-----	-----
10	1300
30	950

예

가장 적은 평균 급여를 찾습니다.

```

SQL> SELECT      job, AVG(sal)
2  FROM          emp
3  GROUP BY      job
4  HAVING        AVG(sal) = (SELECT      MIN(AVG(sal))
5                                FROM      EMP
6                                GROUP BY  job);

```

이 문장에서 잘못된 점은?

```

SQL> SELECT empno, ename
2  FROM emp
3  WHERE sal = (SELECT MIN(sal)
4                FROM emp
5                GROUP BY deptno);

```

```

ERROR:
ORA-01427: single-row subquery returns more than
one row
no rows selected

```

서브쿼리 에러

서브쿼리의 일반적인 에러중의 하나는 단일 행 서브쿼리에 대해서 하나 이상의 행이 리턴되는 것입니다.

위의 SQL 문장에서 GROUP BY 절(deptno)을 포함하는 서브쿼리는 각 그룹에 대해 다중 행을 리턴합니다. 이 경우 서브쿼리의 결과는 800, 1300 그리고 950 이 될 것입니다.

외부 질의는 서브쿼리의 결과(800,950,1300)를 받으며 이러한 결과는 WHERE 절에서 사용 됩니다. WHERE 절은 오직 하나의 값을 필요로 하는 단일 행 **비교 연산자**인 동등(=) 연산자를 포함합니다. = 연산자는 서브쿼리로부터 하나이상의 값을 받을 수 없으므로 에러를 발생시킵니다.

이 에러를 정정하려면 = 연산자를 IN 으로 변경하면 됩니다.

다중 행 서브쿼리

- 하나 이상의 행을 리턴합니다.
- 다중 행 비교 연산자를 사용합니다.

연산자	의미
IN	목록의 어떤 값과 같다
ANY	값을 서브쿼리에 의해 리턴된 각각의 값과 비교한다
ALL	값을 서브쿼리에 의해 리턴된 모든 값과 비교한다

다중 행 서브쿼리

하나 이상의 행을 리턴하는 서브쿼리를 다중 행 서브쿼리 라고 부릅니다. 다중 행 서브쿼리는 단일 행 연산자 대신에 다중 행 연산자를 사용합니다. 다중 행 연산자는 하나 이상의 값을 요구 합니다.

```
SQL> SELECT  ename, sal, deptno
2 FROM      emp
3 WHERE     sal IN (SELECT  MIN(sal)
4                      FROM      emp
5                      GROUP BY deptno);
```

예

부서에 속한 종업원의 최소 급여와 동일한 급여를 받는 종업원을 찾습니다. 결과적으로, 부서에서 최소 급여를 받는 종업원 정보에, 그 급여와 동일한 급여를 받는 다른 부서의 종업원의 정보를 가져올 것입니다.

내부 질의가 먼저 실행되어 세개의 결과 800, 950, 1300 을 포함하는 질의 결과를 산출합니다. 그런 다음에 메인 쿼리 블록이 처리되고 검색 조건을 완성시키기 위해서 내부 질의에 의해 리턴된 값을 사용합니다. 사실, 메인 쿼리는 오라클 서버에게는 다음처럼 보입니다.

```
SQL> SELECT  ename, sal, deptno
2 FROM      emp
3 WHERE     sal IN (800, 950, 1300);
```

다중 행 서브쿼리에서 ANY 연산자 사용

```
SQL> SELECT  empno, ename, job
2 FROM      emp
3 WHERE     sal < ANY
4
5
6
7 AND      job <> 'CLERK';
```

(SELECT sal
FROM emp
WHERE job = 'CLERK')

EMPNO	ENAME	JOB
7654	MARTIN	SALESMAN
7521	WARD	SALESMAN

다중 행 서브쿼리(계속)

ANY 연산자(그리고 그것의 synonym SOME 연산자)는 값을 서브쿼리에 의해 리턴된 각각의 값과 비교합니다. 위의 예는 급여가 어떤 clerk 보다 작으면서 clerk 이 아닌 종업원을 디스플레이 합니다. Clerk 의 최대 급여는 \$1300 입니다. SQL 문장은 clerk 이 아니고 \$1300 이하를 버는 모든 종업원을 디스플레이합니다.

<ANY 는 최대값보다 작음을 의미합니다. >ANY 는 최대값보다 많음을 의미합니다. =ANY 는 IN 과 같습니다.

다중 행 서브쿼리에서 ALL 연산자 사용

SQL>	SELECT	empno, ename, job	1688.0007
2	FROM	emp	296
3	WHERE	sal > ALL	2816.0007
4		(SELECT avg(sal)	
5		FROM emp	
6		GROUP BY deptno);	

EMPNO	ENAME	JOB
7839	KING	PRESIDENT
7566	JONES	MANAGER
7902	FORD	ANALYST
7788	SCOTT	ANALYST

다중 행 서브쿼리(계속)

ALL 연산자는 값을 서브쿼리에 의해 리턴된 모든 값과 비교합니다. 위의 예는 급여가 모든 부서의 평균보다 많은 종업원을 디스플레이합니다. 부서의 가장 많은 평균 급여는 \$2916.66 이므로 질의는 \$2916.66 보다 많은 급여를 받는 종업원을 리턴합니다.

>ALL 은 최대값보다 많음을 의미하며 <ALL 은 최대값보다 적음을 의미합니다.

NOT 연산자는 IN, ANY 그리고 ALL 연산자에서도 사용할 수 있습니다.

제 7 장 다중 열 서브 쿼리 (Subquery)

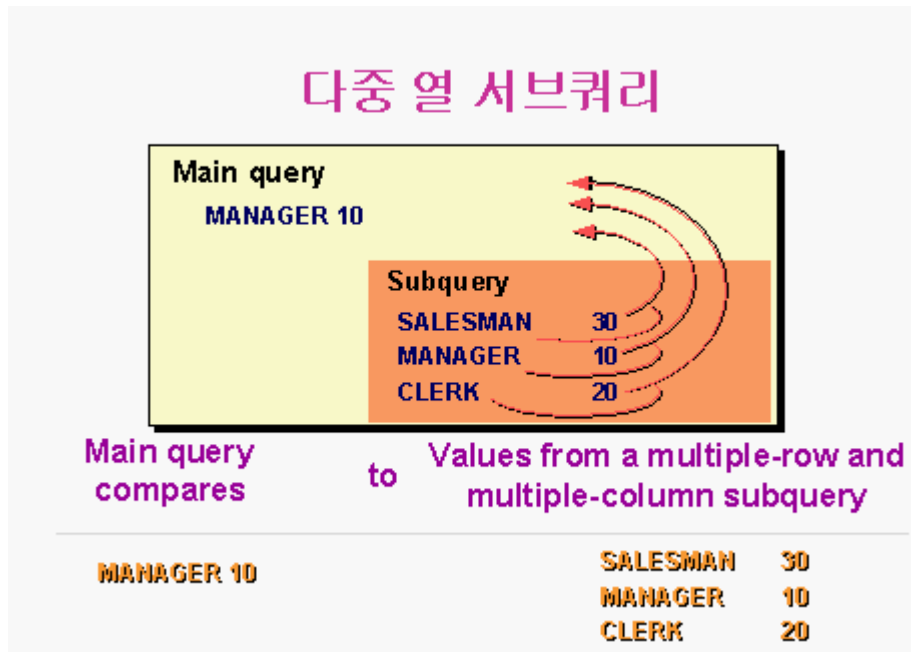
목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 다중 열 서브쿼리를 작성합니다.
- **NULL** 값이 검색되었을 때 서브쿼리의 수행을 기술하고 설명합니다.

과정 목표

본 과정에서는 다중 열 서브쿼리 작성과 SELECT 문장의 FROM 절에서 서브쿼리를 작성하는 방법을 배웁니다.



다중 열 서브쿼리

지금까지는 SELECT 문장의 WHERE 절 또는 HAVING 절에서 오직 하나의 열을 비교하는 단일 행 서브쿼리와 다중 행 서브쿼리를 작성했습니다. 두개 이상의 열을 비교하려고 한다면, **논리 연산자**를 사용하여 혼합 WHERE 절을 작성해야 합니다. 다중 열 서브쿼리는 중복되는 WHERE 절 조건을 단일 WHERE 절로 조합하도록 합니다.

구문형식

```
SELECT column, column, ...
FROM table
WHERE (column, column, ...) IN
      (SELECT column, column, ...
       FROM table
       WHERE condition);
```


다중 열 서브쿼리 사용

- 급여와 보너스가 부서 30에 있는 어떤 직원의 보너스와 급여와 같은 직원의 이름, 부서 번호, 급여 그리고 보너스를 디스플레이합니다.

```
SQL> SELECT  ename, deptno, sal, comm
2  FROM      emp
3  WHERE     (sal, NVL(comm,-1)) IN
4
5              (SELECT sal, NVL(comm,-1)
6              FROM      emp
6              WHERE     deptno = 30);
```

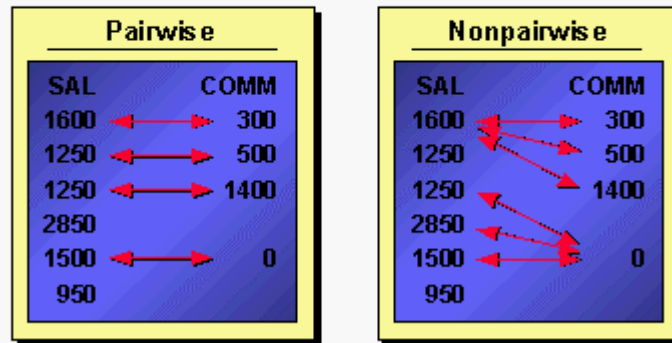
다중 열 서브쿼리 사용

위의 예는 서브쿼리가 하나 이상의 열을 리턴하기 때문에 다중 열 서브쿼리입니다. 이것은 SAL 열과 COMM 열을 비교합니다. 이것은 급여와 보너스가 부서 30에 있는 어떤 종업원의 급여와 보너스에 일치하는 종업원의 이름, 부서 번호, 급여 그리고 보너스를 디스플레이합니다.

위의 SQL 문장의 결과는 다음과 같습니다.

ENAME	DEPTNO	SAL	COMM
-----	-----	-----	-----
JAMES	30	950	
WARD	30	1250	500
MARTIN	30	1250	1400
TURNER	30	1500	0
ALLEN	30	1600	300
BLAKE	30	2850	
6 rows selected.			

열 비교



Pairwise 와 Nonpairwise 비교

다중 열 서브쿼리에서의 열 비교는 pairwise 비교 또는 nonpairwise 비교입니다. 앞 슬라이드의 예에서 pairwise 비교는 WHERE 절에서 실행됩니다. SELECT 문장의 각 후보 행은 부서 30에 있는 종업원의 급여와 보너스에 일치합니다.

nonpairwise 비교(a cross product)를 원한다면 다중 조건을 가지는 WHERE 절을 사용해야 합니다.

Nonpairwise 비교 서브쿼리

- 급여와 보너스가 부서 30에 있는 어떤 직원의 보너스와 급여에 일치하는 직원의 이름, 부서 번호, 급여 그리고 보너스를 디스플레이합니다.

```
SQL> SELECT  ename, deptno, sal, comm
2 FROM      emp
3 WHERE      sal IN
4             (SELECT sal
5              FROM    emp
6              WHERE   deptno = 30)
7 AND
8             NVL(comm,-1) IN
9             (SELECT NVL(comm,-1)
10              FROM    emp
11              WHERE   deptno = 30);
```

Nonpairwise 비교 서브쿼리

위의 예는 열의 nonpairwise 비교입니다. 급여와 보너스가 부서 30에 있는 종업원의 급여와 보너스에 일치하는 종업원의 이름, 부서 번호, 급여 그리고 보너스를 디스플레이합니다.

위의 SQL 문장의 결과는 다음과 같습니다.

ENAME	DEPTNO	SAL	COMM
-----	-----	-----	-----
JAMES	30	950	
BLAKE	30	2850	
TURNER	30	1500	0
ALLEN	30	1600	300
WARD	30	1250	500
MARTIN	30	1250	1400

6 rows selected.

마지막 두 질의의 비교 조건은 다르지만 결과는 같습니다. 결과는 EMP 테이블의 특정 데이터로부터 구해진 것입니다.

Pairwise 서브쿼리

```
SQL> SELECT  ename, deptno, sal, comm
2  FROM      emp
3  WHERE     (sal, NVL(comm,-1)) IN
4             (SELECT sal, NVL(comm,-1)
5                FROM   emp
6                WHERE  deptno = 30);
```

ENAME	DEPTNO	SAL	COMM
-----	-----	-----	-----
JAMES	30	950	
WARD	30	1250	500
MARTIN	30	1250	1400
TURNER	30	1500	0
ALLEN	30	1600	300
BLAKE	30	2850	

6 rows selected.

Pairwise 서브쿼리

pairwise 서브쿼리의 결과는 역시 똑같이 6 개의 행을 리턴합니다.

Nonpairwise 서브쿼리

```
SQL> SELECT  ename,deptno, sal, comm
2 FROM      emp
3 WHERE     sal IN      (SELECT sal
4                        FROM    emp
5                        WHERE   deptno = 30)
6 AND
7          NVL(comm,-1) IN (SELECT NVL(comm,-1)
8                          FROM    emp
9                          WHERE   deptno = 30);
```

ENAME	DEPTNO	SAL	COMM
JAMES	30	950	
BLAKE	30	2850	
TURNER	30	1500	0
ALLEN	30	1600	300
CLARK	10	1500	300
...			

7 rows selected.

Nonpairwise 서브쿼리

nonpairwise 서브쿼리의 결과는 종업원 Clark 을 포함합니다. Clark 의 급여는 Turner 와 같고 보너스는 Allen 과 같습니다.

서브쿼리에서의 NULL 값

```
SQL> SELECT  employee.ename
2 FROM      emp employee
3 WHERE     employee.empno NOT IN
4           (SELECT manager.mgr
5            FROM    emp manager);
no rows selected.
```

서브쿼리의 결과 집합에 Null 리턴

위의 SQL 문장은 어떤 종속하는 직원을 가지지 않는 종업원을 디스플레이 하려고 합니다. 논리적으로, 이 SQL 문장은 8 개의 행을 리턴해야 합니다. 그러나, SQL 문장은 어떤 행도 리턴하 지 않습니다. 내부 질의에 의해 리턴된 값 중의 하나가 null 값이므로 전체 질의는 행이 없다고 리턴합니다. 그 이유는 null 값을 비교하는 모든 조건은 null 이 되기 때문입니다. 그래서 서브쿼리의 결과 집합의 일부가 null 값일 때 NOT IN 연산자를 사용하지 마십시오. NOT IN 연산자는 !=ALL 과 똑같습니다.

서브쿼리 결과 집합의 일부분으로서의 null 값은 IN 연산자를 사용할 때는 문제가 되지 않습니다. IN 연산자는 =ANY 와 똑같습니다. 예를 들면, 종속적인 어떤 직원을 가지는 종업원을 디스플레이 하기 위해서는 다음의 SQL 문장을 사용합니다.

```
SQL> SELECT      employee.ename
2 FROM          emp employee
3 WHERE         employee.empno IN (SELECT manager.mgr
4                                     FROM emp manager);
```

```
ENAME
-----
KING
...
6 rows selected.
```

제 8 장 데이터 조작

목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 각 **DML** 문장을 기술합니다.
- 테이블에 행을 삽입합니다.
- 테이블에 행을 갱신합니다.
- 테이블로부터 행을 삭제합니다.

과정 목표

본 과정에서는 테이블에 행을 삽입하고, 테이블에 있는 기존의 행을 갱신하고 그리고 테이블로부터 기존의 행을 삭제하는 방법을 배웁니다. 또한 COMMIT, [SAVEPOINT](#) 그리고 ROLLBACK 문장으로 트랜잭션을 제어하는 방법을 배웁니다.

데이터 조작용어

- **DML** 문장은 다음의 경우에 실행됩니다.
 - 테이블에 새로운 행을 추가
 - 테이블에 있는 기존의 행을 변경
 - 테이블로부터 기존의 행을 제거

데이터 조작용어

DML([Data manipulation language](#))은 SQL의 핵심 부분입니다. 데이터베이스에 데이터를 추가, 갱신 또는 삭제하고자 한다면 DML 문장을 실행합니다. 작업의 논리적인 단위 형태인 DML 문장의 모음을 트랜잭션이라고 합니다.

은행 데이터베이스를 고려해 봅시다. 은행 고객이 저축성 예금을 당좌 예금으로 전달할 때, 트랜잭션은 세 가지의 분리되는 작업으로 구성됩니다: 저축성 예금 감소, 당좌 예금 증가 그리고 트랜잭션 일지에 트랜잭션을 기록. 오라클 서버는 올바르게 예금을 유지하기 위해 세 가지 SQL 문장 모두가 수행되도록 해야 합니다. 누군가가 실행 시에 트랜잭션의 문장 중의 하나를 막아 버린다면, 그 트랜잭션 내의 다른 문장은 취소되어야 합니다.

테이블에 새로운 행 추가

50	DEVELOPMENT	DETROIT
----	-------------	---------

New row

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

“...새로운 행을 DEPT
테이블에 삽입...”

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
50	DEVELOPMENT	DETROIT

테이블에 새로운 행 추가

위의 그래프는 DEPT 테이블에 새로운 부서를 추가합니다.

INSERT 문장

- **INSERT** 문장을 사용하여 테이블에 새로운 행을 추가합니다.

```
INSERT INTO table [(column [, column...])]  
VALUES (value [, value...]);
```

- 이 구문형식으로는 한번에 오직 하나의 행만이 삽입됩니다.

테이블에 새로운 행 추가(계속)

INSERT 문장을 실행하여 테이블에 새로운 행을 추가할 수 있습니다.

구문형식에서:

table	테이블의 이름입니다.
column	테이블의 열 이름입니다.
value	열에 해당되는 값입니다.

주: VALUES 절을 가지는 이 문장은 테이블에 한번에 오직 하나의 행만을 추가합니다.

새로운 행 삽입

- 각각의 열에 대한 값을 포함하는 새로운 행을 삽입합니다.
- 테이블에 있는 열의 디폴트 순서로 값을 나열합니다.
- **INSERT** 절에서 열을 선택적으로 나열합니다.

```
SQL> INSERT INTO dept (deptno, dname, loc)
      2 VALUES (50, 'DEVELOPMENT', 'DETROIT')
      1 row created.
```

- 문자와 날짜 값은 단일 따옴표 내에 둡니다.

테이블에 새로운 행 추가(계속)

각각의 열에 대한 값을 포함하는 새로운 행을 삽입할 수 있기 때문에 열 목록은 INSERT 절에서 요구되지 않습니다. 그러나, 열 목록을 사용하지 않는다면 값은 테이블에 있는 열의 디폴트 순서로 나열되어야만 합니다.

```
SQL> DESCRIBE dept
```


Name	Null?	Type
-----	-----	-----
DEPTNO	NOT NULL	NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

명료함을 위해서 INSERT 절에서 열 목록을 사용합니다.

문자와 날짜 값은 단일 인용 표시 내에 있어야 합니다. 숫자 값은 단일 인용 표시 내에 두어서는 안됩니다.

Null 값을 가진 새로운 행 추가

- 암시적 방법: 열 목록으로부터 열을 생략합니다.

```
SQL> INSERT INTO dept (deptno, dname)
2 VALUES (60, 'MIS');
1 row created.
```

- 명시적 방법: **NULL** 키워드를 명시합니다.

```
SQL> INSERT INTO dept
2 VALUES (70, 'FINANCE', NULL);
1 row created.
```

NULL 값을 삽입하기 위한 방법

방 법	설 명
암시적	열 목록으로부터 열을 생략합니다.
명시적	VALUES 목록에 NULL 키워드를 명시합니다. VALUES 목록에 공백 스트링(' ')을 명시합니다. 문자 스트링과 날짜 스트링의 경우에만.

SQL*Plus DESCRIBE 명령어로부터 Null? 상태를 검사하여 목표(target) 열이 널 값을 허용하는지 확인하십시오

오라클 서버는 자동적으로 모든 데이터형, 데이터 범위 그리고 데이터 무결성 제약조건을 강요합니다.

특수 값 삽입

SYSDATE 함수는 현재 날짜와 시간을 기록합니다.

```
SQL> INSERT INTO emp (empno, ename, job,
2 mgr, hiredate, sal, comm,
3 deptno)
4 VALUES (7196, 'GREEN', 'SALESMAN',
5 7782, SYSDATE, 2000, NULL,
6 10);
1 row created.
```

SQL 함수를 사용하여 특수 값 삽입

테이블에 특수 값을 입력하기 위해서 의사열(pseudocolumn)을 사용할 수 있습니다.

위의 예는 EMP 테이블에 있는 종업원 Green 에 대한 정보를 기록합니다. HIREDATE 열에 현재 시간과 날짜를 입력합니다. 현재 날짜와 시간을 위해서 SYSDATE 함수를 사용합니다. 또한 테이블에 행을 입력할 때 USER 함수를 사용할 수 있습니다.

테이블에 추가한 내용을 확인

```
SQL> SELECT empno, ename, job, hiredate, comm
2 FROM emp
3 WHERE empno = 7196;
```

EMPNO	ENAME	JOB	HIREDATE	COMM
7196	GREEN	SALESMAN	01-DEC-97	

특정 날짜 값 삽입

- 새로운 종업원을 추가합니다.

```
SQL> INSERT INTO emp
2 VALUES (2296, 'AROMANO', 'SALESMAN', 7782,
3          TO_DATE('FEB 3, 97', 'MON DD, YY'),
4          1300, NULL, 10);
1 row created.
```

- 추가 내용을 확인합니다.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
2296	AROMANO	SALESMAN	7782	03-FEB-97	1300		10

특정 날짜와 시간 값 입력

형식 DD-MON-YY 는 항상 날짜 값을 입력할 때 사용됩니다. 이 형식으로 현재 세기에 대한 디폴트 세기를 다시 호출합니다. 날짜가 시간 정보를 포함하므로 디폴트 시간은 자정(00:00:00)입니다.

날짜를 다른 세기로 입력하기를 원하고, 또 특정 시간을 요구한다면 TO_DATE 함수를 사용합니다.

슬라이드의 예는 EMP 테이블에 종업원 Aromano 에 대한 정보를 기록합니다. HIREDATE 열을 February 3, 1997 로 설정합니다.

주: RR 형식을 설정한다면 아마도 세기는 현재 세기가 아닐 수도 있습니다.

다른 테이블로부터 행 복사

- 서버쿼리로 **INSERT** 문장을 작성합니다.

```
SQL> INSERT INTO managers(id, name, salary, hiredate)
2      SELECT empno, ename, sal, hiredate
3      FROM   emp
4      WHERE  job = 'MANAGER';
3 rows created.
```

- **VALUES** 절을 사용하지 않습니다.
- 서버쿼리의 열 수와 **INSERT** 절의 열 수는 일치합니다.

다른 테이블로부터 행 복사

기존의 테이블로부터 값을 가져와 테이블에 추가하기 위해서 INSERT 문장을 사용할 수 있습니다. VALUES 절에서 서버쿼리를 사용할 수 있습니다.

구문형식

```
INSERT INTO table [ column (, column) ]
        subquery;
```

여기서:	table	테이블의 이름입니다.
	column	테이블에 있는 열의 이름입니다.
	subquery	행을 테이블에 리턴할 서버쿼리입니다.

“SELECT,” 서버쿼리에 대해서 보다 많은 정보를 알고자 한다면, *Oracle Server SQL Reference, Release 7.3* 또는 *8.0* 을 참조하십시오.

INSERT 절의 열 목록에서 열의 개수와 그들의 데이터형은 서버쿼리에 있는 값의 개수와 그들의 데이터형과 일치해야 합니다.

테이블 데이터 변경

EMP

EMPNO	ENAME	JOB	...	DEPTNO
7839	KING	PRESIDENT		10
7698	BLAKE	MANAGER		30
7782	CLARK	MANAGER		10
7566	JONES	MANAGER		20
...				

“...EMP 테이블의
행을 갱신...”

EMP

EMPNO	ENAME	JOB	...	DEPTNO
7839	KING	PRESIDENT		10
7698	BLAKE	MANAGER		30
7782	CLARK	MANAGER		20
7566	JONES	MANAGER		20
...				

테이블 데이터 변경

위의 그래프는 Clark 의 부서 번호를 10 에서 20 으로 변경합니다.

UPDATE 문장

- **UPDATE** 문장으로 기존의 행을 갱신합니다.

```
UPDATE      table
SET         column = value [, column = value]
[WHERE      condition];
```

- 필요하다면 하나 이상의 행을 갱신합니다.

행 갱신

UPDATE 문장을 사용하여 기존의 행을 수정할 수 있습니다.

위의 구문형식에서:

table	테이블의 이름입니다.
column	테이블의 열 이름입니다.
value	열에 대한 관련 값이나 서브쿼리입니다.
condition	갱신할 행을 명시하고, 열 이름, 표현식, 상수, 서브쿼리 그리고 비교 연산자로 구성됩니다.

갱신된 행을 디스플레이하기 위해 테이블을 질의하여 갱신 작업을 확인합니다.

“UPDATE.”에 대해서 보다 많은 정보를 알고자 한다면, *Oracle Server SQL Reference, Release 7.3* 또는 *8.0* 을 참조하십시오.

주: 일반적으로 단일 행을 식별하기 위해서 기본 키(primary key)를 사용합니다. 다른 열을 사용하면 원하지 않는 여러 행이 갱신될 수 있습니다. 예를 들면, 이름으로 EMP 테이블에서 단일 행을 식별하면 똑같은 이름을 가지는 종업원이 여러 명이 있을 수 있기 때문에 모호해집니다.

테이블 행 갱신

- 특정 열이나 행은 **WHERE** 절을 명시함으로써 수정될 수 있습니다.

```
SQL> UPDATE emp
2 SET deptno = 20
3 WHERE empno = 7782;
1 row updated.
```

- WHERE** 절을 생략하면 테이블에 있는 모든 행이 수정됩니다.

```
SQL> UPDATE employee
2 SET deptno = 20;
14 rows updated.
```

행 갱신(계속)

UPDATE 문장은 특정 행(들)을 갱신합니다. WHERE 절이 명시되었다면, 위의 예는 종업원

7782(Clark)를 부서 번호 20 으로 전달합니다.

WHERE 절을 생략하면 테이블의 모든 행이 갱신됩니다.

```
SQL> SELECT  ename, deptno
2 FROM      employee;
```

ENAME	DEPTNO
-----	-----
KING	20
BLAKE	20
CLARK	20
JONES	20
MARTIN	20
ALLEN	20
TURNER	20
...	

14 rows selected.

주: EMPLOYEE 테이블은 EMP 와 똑같은 데이터를 가집니다.

다중 열 서브쿼리로 갱신

종업원 7499의 업무와 부서에 일치하도록
종업원 7698의 업무와 부서를 갱신합니다.

```
SQL> UPDATE emp
2 SET      (job, deptno) =
3          (SELECT job, deptno
4           FROM emp
5           WHERE empno = 7499)
6 WHERE empno = 7698;
1 row updated.
```

다중 열 서브쿼리로 행 갱신

다중 열 서브쿼리는 UPDATE 문장의 SET 절로 구현할 수 있습니다.

구문형식

```
UPDATE      table
SET    (column, column, ...) =
        (SELECT column, column,
          FROM    table
          WHERE   condition)
WHERE condition;
```

테이블로부터 행 제거

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
50	DEVELOPMENT	DETROIT
60	MIS	
...		

“...DEPT 테이블로부터
행을 삭제...”

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
60	MIS	
...		

테이블로부터 행 제거

위의 그래프는 DEPT 테이블(DEPT 테이블에는 정의된 제약조건이 없다고 가정합니다)로부터 DEVELOPMENT 부서를 제거합니다.

DELETE 문장

DELETE 문장을 사용하여 테이블로부터 기존의 행을 제거할 수 있습니다.

```
DELETE [FROM] table  
[WHERE condition];
```

행 삭제

DELETE 문장을 사용하여 기존의 행을 제거합니다.

구문형식에서:

table 테이블의 이름입니다.

condition 삭제된 행을 명시하고, 열 이름, 표현식, 상수, 서브쿼리 그리고 [비교 연산자](#)로 구성됩니다.

“DELETE.”에 대해 보다 자세한 정보를 원한다면 *Oracle Server SQL Reference, Release 8.0* 을 참조하십시오.

테이블로부터 행 삭제

- **WHERE** 절을 명시하여 특정 행이나 행들을 삭제할 수 있습니다.

```
SQL> DELETE FROM      department
      2 WHERE          dname = 'DEVELOPMENT';
1 row deleted.
```

- **WHERE** 절을 생략하면 테이블의 모든 행이 삭제됩니다.

```
SQL> DELETE FROM      department ;
4 rows deleted.
```

행 삭제(계속)

DELETE 문장에 WHERE 절을 명시하여 특정 행(들)을 삭제할 수 있습니다. 위의 예는 DEPARTMENT 테이블로부터 DEVELOPMENT 부서를 삭제합니다. SELECT 문장을 사용하여 삭제된 행을 디스플레이하여 삭제 작업을 확인할 수 있습니다.

```
SQL> SELECT *
      2 FROM      department
      3 WHERE      dname = 'DEVELOPMENT';
no rows selected.
```

예

January 1, 1997 이후에 입사한 모든 종업원을 제거합니다.

```
SQL> DELETE FROM emp
      2 WHERE      hiredate > TO_DATE('01.01.97', 'DD.MM.YY');
1 row deleted.
```

WHERE 절을 생략한다면 테이블에 있는 모든 행이 삭제될 것입니다. 슬라이드의 두번째 예는 WHERE 절을 명시하지 않았기 때문에 DEPARTMENT 테이블의 모든 행을 삭제합니다.

주: DEPARTMENT 테이블은 DEPT 테이블과 똑같은 데이터를 가집니다.

다른 테이블을 근거로 한 행 삭제

다른 테이블의 값을 근거로 테이블로부터 행을 제거하기 위해서 **DELETE** 문장에서 서브쿼리를 사용할 수 있습니다.

```
SQL> DELETE FROM      employee
2  WHERE      deptno =
3              (SELECT  deptno
4                  FROM    dept
5                  WHERE   dname = 'SALES' ) ;
6 rows deleted.
```

다른 테이블을 근거로 하는 행 삭제

다른 테이블의 값을 근거로 테이블로부터 행을 삭제하기 위해 서브쿼리를 사용할 수 있습니다. 위의 예는 부서 30에 있는 모든 종업원을 삭제합니다. 서브쿼리는 SALES 부서에 대해 부서 번호를 알기 위해서 DEPT 테이블을 검색합니다. 그런 다음에 서브쿼리는 이 부서 번호를 근거로 하는 EMPLOYEE 테이블로부터 데이터의 행을 삭제하는 메인 쿼리로 부서 번호를 넘겨줍니다.

행 삭제 : 무결성 제약조건 에러

```
SQL> DELETE FROM dept  
2 WHERE deptno = 10;
```

```
DELETE FROM dept  
*  
ERROR at line 1:  
ORA-02292: integrity constraint (USR.EMP_DEPTNO_FK)  
violated - child record found
```

무결성 제약조건 에러

무결성 제약조건을 위반하도록 레코드를 삭제하고자 한다면 에러가 발생할 것입니다. 위의 예는 DEPT 테이블로부터 부서 번호 10을 삭제하려고 시도하지만, 부서 번호를 EMP 테이블에 있는 **외래 키(foreign key)**가 사용하기 때문에 에러가 발생합니다. 삭제하려고 하는 부모 레코드가 자식 레코드를 가지고 있다면 child record found violation ORA-02292 라는 에러 메시지가 나타납니다.

제 9 장 데이터베이스 트랜잭션

목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

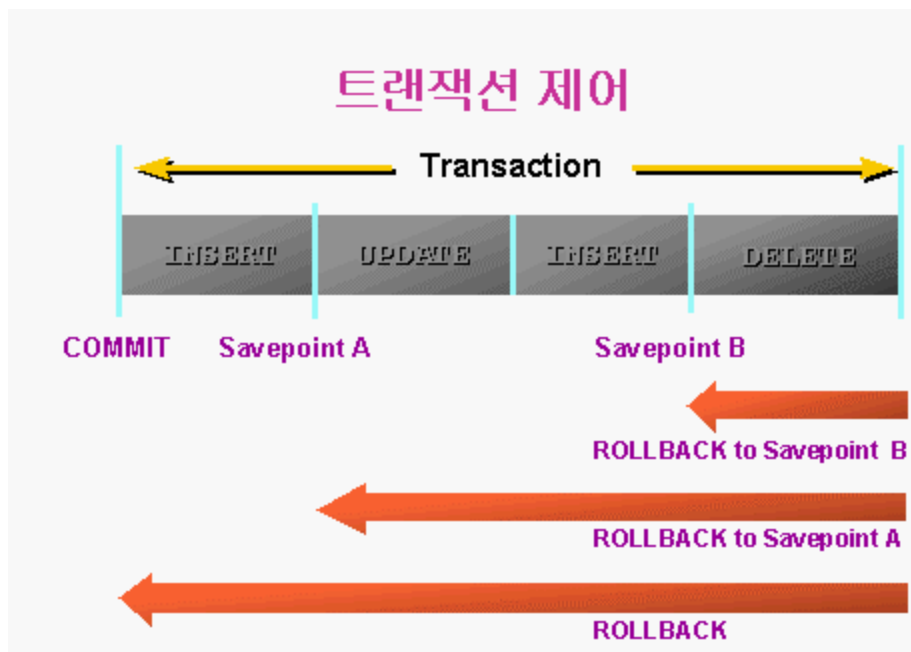
- 트랜잭션을 이해합니다.
- 트랜잭션을 제어합니다.

과정 목표

본 과정에서는 트랜잭션의 개념에 대해서 상세히 알아보고, 트랜잭션을 구성하는 여러 가지 요소들에 대해서 배웁니다. 또한 COMMIT, **SAVEPOINT** 그리고 ROLLBACK 문장으로 트랜잭션을 제어하는 방법을 배웁니다.

COMMIT과 ROLLBACK의 장점

- 데이터 일관성을 제공합니다.
- 데이터를 영구적으로 변경하기 전에 데이터 변경을 미리 보게 합니다.
- 관련된 작업을 논리적으로 그룹화 합니다.



명시적 트랜잭션 제어 문장

COMMIT, **SAVEPOINT** 그리고 ROLLBACK 문장을 사용하여 트랜잭션의 논리를 제어할 수 있습니다.

문장	설명
COMMIT	모든 미결정 데이터를 영구적으로 변경함으로서 현재 트랜잭션을 종료합니다.
SAVEPOINT name	현재 트랜잭션 내에 savepoint 를 표시합니다.
ROLLBACK [TO SAVEPOINT name]	ROLLBACK 은 모든 미결정 데이터 변경을 버림으로써 현재의 트랜잭션을 종료합니다. ROLLBACK TO SAVEPOINT name 은 savepoint 와 모든 연이은 변경을 버립니다.

주: **SAVEPOINT** 는 ANSI 표준 SQL 이 아닙니다.

암시적 트랜잭션 처리

- 자동적인 커밋은 다음의 환경 하에서 발생합니다.
 - **DDL** 문장이 완료시
 - **DCL** 문장이 완료시
 - 명시적인 **COMMIT**이나 **ROLLBACK** 없이 **SQL*Plus**를 그냥 종료할 때
- 자동적인 롤백은 **SQL*Plus**를 비정상적으로 종료하거나 시스템 실패 하에서 발생합니다.

암시적 트랜잭션 처리

상태	상황
자동 커밋	DDL 문장이나 DCL 문장은 문제가 되지 않습니다. 명시적인 COMMIT 이나 ROLLBACK 없이

	SQL*Plus 를 정상적으로 종료
자동 롤백	SQL*Plus 의 비정상적인 종료나 ,또는 시스템 오류

주: SQL*Plus AUTOCOMMIT 명령은 ON 이나 OFF 로 토글될 수 있습니다. ON 이면 각각의 개별적인 DML 문장이 실행되자마자 커밋됩니다.그러므로 변경을 롤백할 수 없습니다. OFF 로 설정하면 COMMIT 은 명시적으로 실행될 수 있습니다. 또한 COMMIT 은 DDL 문장이 완료되거나 SQL*Plus 를 종료할 때 완료됩니다

시스템 실패

트랜잭션이 시스템 실패에 의해 중단될 때 모든 트랜잭션은 자동적으로 롤백됩니다. 이것은 데이터에 대해서 원하지 않는 변경을 막아 주고 테이블을 마지막으로 커밋된 시점으로 상태를 되돌립니다. SQL 은 테이블의 무결성을 보호합니다.

COMMIT이나 ROLLBACK 이전의 데이터 상태

- 데이터의 이전 상태는 복구될 수 있습니다.
- 현재 사용자는 **SELECT** 문장을 사용하여 **DML** 작업의 결과를 검토할 수 있습니다.
- 다른 사용자들은 현재 사용자에게 의한 **DML** 문장의 결과를 볼 수 없습니다.
- 변경된 행들은 잠금상태(*locked*)입니다; 다른 사용자들은 변경된 행들 내의 데이터를 변경할 수 없습니다.

변경을 커밋

트랜잭션 동안에 이루어지는 모든 데이터 변경은 트랜잭션이 커밋되기 전까지는 임시적

입이다.

COMMIT 이후의 데이터 상태

- 데이터베이스에 영구적으로 데이터를 변경합니다.
- 데이터의 이전 상태는 완전히 상실됩니다.
- 모든 사용자가 결과를 볼 수 있습니다.
- 변경된 행들의 잠금상태가 해제됩니다. 그러한 행들은 다른 사용자가 조작하기 위해 이용가능합니다.
- 모든 **savepoint**가 제거됩니다.

COMMIT 이나 ROLLBACK 이전의 데이터 상태

- 데이터 조작 작업은 주로 [데이터베이스 버퍼](#)에 영향을 미치므로 데이터의 이전 상태는 복구 가능합니다.
- 현재 사용자는 테이블을 질의하여 데이터 조작 작업의 결과를 검사할 수 있습니다.
- 다른 사용자는 현재 사용자에 의해 만들어진 데이터 조작 작업의 결과를 볼 수 없습니다.
- 변경된 행은 잠금상태입니다. 다른 사용자들은 변경된 행들 내의 데이터를 변경 할 수 없습니다.

데이터 COMMIT 작업

- 변경을 합니다.

```
SQL> UPDATE emp
2 SET deptno = 10
3 WHERE empno = 7782;
1 row updated.
```

- 변경을 커밋합니다.

```
SQL> COMMIT;
Commit complete.
```

변경 커밋(계속)

위의 예는 EMP 테이블을 갱신하고 종업원 7782(Clarke)에 대한 부서 번호를 10 으로 설정합니다. 그런 다음에 COMMIT 문장을 수행하여 영구적으로 변경을 합니다.

예

최소한 한 명의 종업원이 있는 새로운 ADVERTISING 부서를 생성합니다. 데이터 변경을 영구적으로 만듭니다.

```
SQL> INSERT INTO department(deptno, dname, loc)
2 VALUES (50, 'ADVERTISING', 'MIAMI');
1 row created.
```

```
SQL> UPDATE employee
2 SET deptno = 50
3 WHERE empno = 7876;
1 row updated.
```

```
SQL> COMMIT;
Commit complete.
```

ROLLBACK 이후의 데이터 상태

- 롤백 문장을 사용하여 모든 결정되지 않은 변경들을 버립니다.
- 데이터 변경이 취소됩니다.
- 데이터의 이전 상태로 복구됩니다.
- 변경된 행들의 잠금상태가 해제됩니다.

```
SQL> DELETE FROM employee;  
14 rows deleted.  
SQL> ROLLBACK;  
Rollback complete.
```

변경 롤백

ROLLBACK 문장을 사용하여 모든 미결정 변경을 버립니다.

다음은 ROLLBACK 입니다.

- 데이터 변경은 취소됩니다.
- 데이터의 이전 상태로 복구됩니다.
- 변경된 행의 잠금이 해제됩니다.

예

TEST 테이블의 레코드를 제거하려고 하면 뜻하지 않게 테이블을 지워 버릴 수 있습니다. 실수를 정정한 다음에 올바른 문장을 다시 실행하여 데이터를 영구적으로 변경합니다.

```
SQL> DELETE FROM test;  
25,000 rows deleted.  
SQL> ROLLBACK;  
Rollback complete.  
SQL> DELETE FROM test  
2 WHERE id = 100;  
1 row deleted.  
SQL> SELECT *  
2 FROM test
```

```
3 WHERE id = 100;
No rows selected.
SQL> COMMIT;
Commit complete.
```

표시자(Marker)로 변경을 롤백

- **SAVEPOINT** 문장을 사용하여 현재 트랜잭션 내에 표시자(marker)를 생성합니다.
- **ROLLBACK TO SAVEPOINT** 문장을 사용하여 표시자(marker)까지로 롤백합니다.

```
SQL> UPDATE...
SQL> SAVEPOINT update_done;
Savepoint created.
SQL> INSERT...
SQL> ROLLBACK TO update_done;
Rollback complete.
```

Savepoint로 변경을 롤백

SAVEPOINT 문장을 사용하여 현재 트랜잭션 내에 표시자(marker)를 생성할 수 있습니다. 그러므로 트랜잭션은 보다 작은 영역으로 나눌 수 있습니다. 그런 다음에 ROLLBACK TO SAVEPOINT 문장을 사용하여 해당 표시자까지로 미결정의 변경들을 버립니다.

이전의 savepoint와 똑같은 이름으로 두번째 savepoint를 생성한다면 이전의 savepoint는 삭제 됩니다.

문장 레벨 롤백

- 실행동안에 단일의 **DML** 문장이 실패하면, 단지 그 문장만을 롤백합니다.
- 오라클 서버는 암시적인 **savepoint**를 구현합니다.
- 모든 다른 변경들은 보유됩니다.
- 사용자는 **COMMIT**이나 **ROLLBACK** 문장을 실행하여 명시적으로 트랜잭션을 종료합니다.

문장 레벨 롤백

문장 실행 에러가 발견되면 트랜잭션의 일부는 암시적 롤백으로 버려질 수 있습니다. 단일 DML 문장이 트랜잭션 실행 동안에 실패한다면 그 문장에 의한 변경은 문장 레벨 롤백에 의해 취소되지만, 트랜잭션에 있는 이전의 DML 문장에 의해 만들어진 변경은 버려지지 않을 것입니다. 그 문장들은 사용자에게 의해서 명시적으로 커밋이나 롤백될 수 있습니다.

오라클은 모든 데이터 정의어(DDL) 문장에서 암시적 COMMIT 을 수행합니다. 그래서, 비록 DDL 문장이 성공적으로 실행되지 않았더라도 서버가 커밋을 실행했기 때문에 이전의 문장으로 롤백할 수 없습니다.

COMMIT 이나 ROLLBACK 문장을 실행하여 트랜잭션을 명시적으로 종료합니다.

읽기 일관성

- 읽기 일관성은 항상 데이터의 검색이 일관되게 보증합니다.
- 어떤 사용자에게 의해 행해진 변경은 다른 사용자에게 의해 행해진 변경과 충돌하지 않습니다.
- 데이터를 똑같이 보증합니다.
 - **Reader**는 **writer**에 대해서 기다리지 않습니다.
 - **Writer**는 **reader**에 대해서 기다리지 않습니다.

읽기 일관성

- 데이터베이스 사용자가 데이터베이스를 액세스하는 두가지 유형이 있습니다.
- 읽기 연산자 (SELECT 문장)
- 쓰기 연산자 (INSERT, UPDATE, DELETE 문장)

읽기 일관성이 필요한데 이것은 다음을 발생시킵니다.

- 데이터베이스 reader와 writer는 일관되게 데이터를 보게됩니다.
- Reader는 변경 중인 데이터는 보지를 않습니다.
- Writer는 데이터베이스에 대해서 일관되게 수행된 변경을 보증합니다.
- 하나의 writer에 의해 만들어진 변경은 다른 writer가 만드는 변경과 분열이나 충돌하지 않습니다.

읽기 일관성의 목적은 각각의 사용자가 DML 작업을 시작하기 전에, 마지막 커밋 시에 저장된 데이터를 알 수 있도록 보증하는 것입니다.

잠금 (Locking)

오라클 잠금:

- 동시에 일어나는(**concurrent**) 트랜잭션 사이의 상호작용이 파괴되지 않도록 막아 줍니다.
- 사용자 액션을 요구하지 않습니다.
- 자동적으로 가장 낮은 레벨의 제약조건을 사용합니다.
- 트랜잭션이 지속되도록 합니다.

잠금(Locking)이란?

Lock 은 사용자 객체(테이블이나 행)나 사용자에게 보이지 않는 시스템 객체(공유 데이터 구조 및 데이터 사전 행) 등의 똑같은 자원을 액세스하는 트랜잭션들 사이의 유해한 상호작용 을 막아 줍니다.

오라클 데이터 잠금(Locking) 방법

오라클 데이터베이스에서 Locking 은 완전히 자동적이며 사용자 액션을 요구하지 않습니다. 암시적 locking 은 SELECT 를 제외한 모든 SQL 에 대해서 발생합니다. 오라클 디폴트 locking 기법은 자동적으로 제약사항의 가장 낮은 레벨을 사용하여, 가장 높은 단계의 동시성을 제공하며 또한 데이터 무결성을 최대한 제공합니다. 오라클은 또한 사용자가 데이터를 수동 으로 잠그는(lock) 것을 허용합니다.

잠금(Locking) 모드

오라클은 다중사용자 데이터베이스에서 두 가지의 locking 모드를 사용합니다.

Lock 모드	설명
exclusive	자원이 공유되는 것을 막아줍니다. 자원을 배타적으로 lock 하는 첫번째 트랜잭션은 배타적 잠금이 해제되기 전까지는 자원을 변경할 수 있는 유일한 트랜잭션입니다.

share lock	<p>자원이 공유되도록 허용합니다.</p> <p>데이터를 읽는 다중 사용자는 데이터를 공유하고, writer(배타적 잠금이 필요한)에 의해 동시에 액세스 되는 것을 막기 위해서 공유 잠금을 유지합니다. 똑같은 자원상에서 여러 개의 트랜잭션은 공유 잠금을 요구할 수 있습니다.</p>
------------	---

제 10 장 테이블 생성과 관리

목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 주된 데이터베이스 객체를 기술합니다.
- 테이블을 생성합니다.
- 열 정의를 명시할 때, 사용 가능한 데이터 형을 기술합니다.
- 테이블 정의를 변경합니다.
- 테이블을 삭제, 이름 변경, 자르기합니다.

과정 목표

본 과정에서는 주된 데이터베이스 객체와 서로에 대한 그들의 관련성에 관하여 배웁니다. 또한 테이블을 생성, 변경 그리고 삭제하는 방법을 배웁니다.

데이터베이스 객체

객체	설명
Table	행과 열로 구성된 기본적인 저장매체의 단위
View	하나 이상의 테이블로부터 데이터의 부분집합을 논리적으로 표현
Sequence	기본 키 값을 발생
Index	질의 성능을 향상
Synonym	객체에 대체 이름을 부여

데이터베이스 객체

오라클 데이터베이스는 여러 개의 데이터 구조를 포함할 수 있습니다. 데이터베이스 설계에서 작성된 각각의 구조는 데이터베이스 개발 단계 동안에 생성할 수 있습니다.

- Table: 데이터를 저장합니다.
- View: 하나 이상의 테이블로부터의 데이터 부분집합입니다.
- Sequence: 기본 키 값을 발생합니다.
- Index: 어떤 질의의 성능을 향상시킵니다.
- Synonym: 객체에 대체 이름을 부여합니다.

Oracle8 테이블 구조

- 비록 사용자가 데이터베이스를 사용하는 동안이라도 언제든지 테이블을 생성할 수 있습니다.
- 어떤 테이블의 크기를 명시할 필요는 없습니다. 크기는 최대로 전체 데이터베이스에 할당된 공간만큼 정의됩니다. 그러나, 얼마나 많은 테이블 공간이 초과 사용될 지를 평가하는 것은 중요합니다.
- 테이블 구조는 온라인으로 수정할 수 있습니다.

주: 보다 많은 데이터베이스 객체를 이용할 수 있지만 본 과정에서는 다루지 않습니다.

이름 지정 규칙

테이블과 열의 이름 :

- 문자로 시작해야 합니다.
- 문자 길이는 1-30이어야 합니다.
- 오직 **A-Z, a-z, 0-9, _, \$** 그리고 **#**만을 포함할 수 있습니다.
- 동일한 사용자가 소유한 객체의 이름은 중복되어서는 안됩니다.
- 오라클 서버 예약어는 안됩니다.

이름 지정 규칙

어떤 오라클 데이터베이스 객체 이름을 지정하는 표준 규칙에 따라서 데이터베이스 테이블과 열의 이름을 정합니다.

- 테이블 이름과 열 이름은 문자로 시작해야 하며 1-30 문자 길이를 가질 수 있습니다.
이름은 오직 A-Z, a-z, 0-9, _ (underscore), \$ 그리고 # (유효한 문자이지만 권장되지는 않습니다.) 문자만을 포함해야 합니다.
- 이름은 오라클 서버 사용자에 의해 소유된 다른 객체의 이름과 중복되어서는 안됩니다.
- 이름은 오라클 예약어이어서는 안됩니다.

이름 지정 지침

- 테이블이나 다른 데이터베이스 객체에 대한 서술적인 이름을 사용합니다.
- 다른 테이블에도 일관되게 똑같은 **엔티티** 이름을 지정합니다. 예를 들면, EMP 테이블과 DEPT 테이블에서 부서 번호 열은 DEPTNO라고 명명합니다.

주: 이름은 대소문자를 구분하지 않습니다. 예를 들면, EMP 는 eMP 또는 Emp 와 똑같이 취급 합니다.

“Object Names and Qualifiers.”에 대하여 보다 많은 정보를 알고자 한다면, *Oracle Server SQL Reference, Release 8.0* 을 참조하십시오.

CREATE TABLE 문장

- 다음을 가져야 합니다.
 - **CREATE TABLE** 권한
 - 저장 영역

```
CREATE TABLE [schema.] table  
(column datatype [DEFAULT expr] ,...);
```

- 다음을 명시해야 합니다.
 - 테이블 이름
 - 열 이름, 열 데이터형 및 열 크기

CREATE TABLE 문장

SQL CREATE TABLE 문장을 실행하여 데이터를 저장하기 위한 테이블을 생성합니다. 이 문장은 데이터 정의어(DDL) 문장의 하나인데, 이후의 여러 장에 걸쳐서 다루어집니다. DDL 문장은 Oracle8 데이터베이스 구조를 생성, 수정 또는 삭제하는데 사용되는 SQL 문장의 부분 집합입니다. 이러한 문장은 데이터베이스에 즉각 영향을 미치며, 데이터베이스 사전에 정보를 기록합니다.

테이블을 생성하기 위해서 사용자는 CREATE TABLE 권한과 객체를 생성하기 위한 저장 영역을 가져야 합니다. 데이터베이스 관리자는 다음 과정에서 다룰 데이터 제어어(DCL)를 사용하는데, 이것은 사용자에게 권한을 부여합니다.

구문형식에서:

<i>schema</i>	소유자의 이름과 똑같습니다.
<i>table</i>	테이블의 이름입니다.
DEFAULT <i>expr</i>	INSERT 문장에서 값을 생략할 경우, 디폴트 값을 명시합니다.
<i>column</i>	열의 이름입니다.
<i>datatype</i>	열의 데이터형과 길이입니다.

“CREATE TABLE.”에 대해서 보다 많은 정보를 알고자 한다면, *Oracle Server SQL Reference, Release 7.3* 또는 *8.0* 을 참조하십시오.

테이블 생성

- 테이블 생성

```
SQL> CREATE TABLE dept
2      (deptno NUMBER(2),
3        dname  VARCHAR2(14),
4        loc    VARCHAR2(13));
Table created.
```

- 테이블 생성 확인

```
SQL> DESCRIBE dept
```

Name	Null?	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

테이블 생성

위의 예는 DEPTNO, DNAME 그리고 LOC 이라는 세 개의 열을 생성합니다. DESCRIBE 명령어를 사용하여 테이블의 생성을 확인합니다.

DDL 문장으로 테이블을 생성하므로 이 문장을 실행되면 자동으로 커밋됩니다.

데이터 사전 질의

- 사용자가 소유한 테이블을 기술합니다.

```
SQL> SELECT *  
2 FROM user_tables;
```

- 사용자가 소유한 별개의 다른 객체 유형을 보여 줍니다.

```
SQL> SELECT DISTINCT object_type  
2 FROM user_objects;
```

- 사용자가 소유한 테이블, 뷰, 동의어 그리고 시퀀스를 보여줍니다.

```
SQL> SELECT *  
2 FROM user_catalog;
```

데이터 사전 질의

여러분이 소유한 다양한 데이터베이스 객체를 보기 위해서 데이터 사전 테이블을 질의할 수 있습니다. 다음이 종종 사용되는 데이터 사전 테이블입니다.

- USER_TABLES
- USER_OBJECTS
- USER_CATALOG

주: USER_CATALOG 는 CAT 과 동의어입니다. 이 동의어를 SQL 에서 USER_CATALOG 대신에 사용할 수 있습니다.

```
SQL> SELECT *  
2 FROM CAT;
```

데이터형

데이터형	설명
VARCHAR2(size)	가변 길이 문자 데이터
CHAR(size)	고정 길이 문자 데이터
NUMBER(p,s)	가변 길이 숫자 데이터
DATE	날짜와 시간 값
LONG	2기가바이트까지의 가변 길이 문자 데이터
CLOB	4 기가바이트까지의 단일 바이트 문자 데이터
RAW, LONG RAW	원시 이진 데이터
BLOB	4 기가바이트까지의 이진 데이터
BFILE	4기가 바이트까지의 외부 파일에 저장된 이진 데이터

데이터형(음성 설명은 없습니다.)

데이터형	설명
VARCHAR2(size)	가변 길이 문자 데이터 (최대 크기는 명시해야 하며, 최소 크기는 1, 최대 크기는 4000 입니다.)
CHAR(size)	size Byte 길이의 고정 길이 문자 데이터 (디폴트이며 최소 크기는 1, 최대 크기는 2000 입니다.)
NUMBER(p,s)	정밀도 p 와 스케일 s 를 가지는 숫자; 정밀도는 십진 자리수의 최대 개수이며, 스케일은 소수점 오른쪽의 자리수입니다. (정밀도는 1 에서 38 까지의 범위이며 스케일은 -84 에서 127 까지의 범위일 수 있습니다.)
DATE	January 1, 4712 B.C. 와 December 31, 9999 A.D.사이의 날짜와 시간 값입니다.
LONG	2 기가바이트까지의 가변 길이 문자 데이터입니다.
CLOB	4 기가바이트까지의 단일 바이트 문자 데이터입니다.
RAW(size)	size 길이의 원시 이진 데이터입니다. 최대 크기는 2000 입니다. (최대 크기는 명시해야 합니다.)

LONG RAW	2 기가바이트까지의 가변 길이 원시 이진 데이터입니다.
BLOB	4 기가바이트까지의 이진 데이터입니다.
BFILE	4 기가바이트까지의 외부 파일에 저장된 이진 데이터입니다.

DEFAULT 옵션

- 삽입 시에 열에 대한 디폴트 값을 명시합니다.

```
... hiredate DATE DEFAULT SYSDATE, ...
```

- 가능한 값은 리터럴 값, 표현식 또는 **SQL** 함수입니다.
- 불가능한 값은 다른 열의 이름이나 의사열입니다.
- 디폴트 데이터형은 열의 데이터형과 일치해야 합니다.

DEFAULT 옵션

열은 DEFAULT 옵션을 사용하여 디폴트 값을 부여할 수 있습니다. 이 옵션은 열에 대한 값없이 어떤 행을 입력할 경우 null 값이 열에 입력되지 않도록 해 줍니다. 디폴트 값은 리터럴, 표현식 또는 SYSDATE 나 USER 같은 SQL 함수일 수 있지만, 다른 열의 이름이나 NEXTVAL 이나 CURRVAL 같은 **의사 열**은 안됩니다. 디폴트 표현식은 열의 데이터형과 일치해야 합니다.

서브쿼리를 사용한 테이블 생성

- **CREATE TABLE** 문장과 **AS subquery** 옵션을 조합하여 테이블을 생성하고 행을 삽입합니다.

```
CREATE TABLE table  
            [column(, column...)]  
AS subquery;
```

- 서브쿼리 열의 개수와 명시된 열의 개수를 일치시킵니다.
- 열 이름과 디폴트 값을 가진 열을 정의합니다.

다른 테이블에 있는 행으로부터 테이블 생성

테이블을 생성하는 두번째 방법은 테이블을 생성하고 서브쿼리로부터 리턴된 행을 삽입하기 위해서 AS subquery 절을 적용하는 것입니다.

구문형식에서:

table 테이블의 이름입니다.

column 열 이름, 디폴트 값 그리고 무결성 제약조건입니다.

subquery 새로운 테이블에 삽입할 행의 집합을 정의한 SELECT 문장입니다.

지침

- 테이블은 명시된 열 이름으로 생성되고 그리고 SQL 문장에 의해 리턴된 행들이 테이블에 삽입됩니다.
- 열 정의는 오직 열 이름과 디폴트 값만을 포함할 수 있습니다.
- 열이 명시되었다면, 열의 수는 서브쿼리 SELECT 목록에 있는 열의 수와 같아야 합니다.
- 열이 명시되지 않았다면, 테이블의 열 이름은 서브쿼리의 열 이름과 같습니다.

서브쿼리를 사용한 테이블 생성

```
SQL> CREATE TABLE dept30
2 AS
3 SELECT empno, ename, sal*12 ANNSAL, hiredate
4 FROM emp
5 WHERE deptno = 30;
Table created.
```

```
SQL> DESCRIBE dept30
```

Name	Null?	Type
EMPNO	NOT NULL	NUMBER(4)
ENAME		VARCHAR2(10)
ANNSAL		NUMBER
HIREDATE		DATE

다른 테이블에 있는 행으로부터 테이블 생성(계속)

위의 예는 부서 30 에서 근무하는 모든 종업원에 대한 상세정보를 포함하는 DEPT30 이라는 테이블을 생성합니다. DEPT30 이라는 테이블은 EMP 에서 유래합니다.

SQL*Plus DESCRIBE 명령을 사용하여 데이터베이스 테이블의 존재를 검증하며 열 정의를 검사합니다.

표현식을 선택할 때 열 **별칭**을 부여합니다.

ALTER TABLE 문장

다음의 경우에 **ALTER TABLE** 문장을 사용합니다.

- 새로운 열 추가
- 기존 열 변경
- 새로운 열에 대한 디폴트 값 정의

```
ALTER TABLE table
ADD          (column datatype [DEFAULT expr]
             [, column datatype]...);
```

```
ALTER TABLE table
MODIFY       (column datatype [DEFAULT expr]
             [, column datatype]...);
```

ALTER TABLE 문장

테이블을 생성한 이후에 열이 생략되었거나 열 정의를 변경할 필요가 있기 때문에 테이블 구조를 변경할 필요가 있을 수 있습니다. ALTER TABLE 문장을 사용하여 이것을 수행할 수 있습니다.

ALTER TABLE 문장에 ADD 절을 사용하여 테이블에 열을 추가할 수 있습니다.

구문형식에서:

<i>table</i>	테이블의 이름입니다.
<i>column</i>	새로운 열의 이름입니다.
<i>datatype</i>	새로운 열의 데이터형과 길이입니다.
DEFAULT <i>expr</i>	새로운 열에 대한 디폴트 값을 명시합니다.

ALTER TABLE 문장에 MODIFY 절을 사용하여 테이블에 있는 기존의 열을 변경할 수 있습니다.

주: 슬라이드는 ALTER TABLE 에 대한 단축된 구문형식을 보여줍니다. ALTER TABLE 은 추후의 과정에서 다룹니다.

열 추가

- **ADD** 절을 사용하여 열을 추가합니다.

```
SQL> ALTER TABLE dept30  
2 ADD (job VARCHAR2(9));  
Table altered.
```

- 새로운 열이 마지막 열이 됩니다.

EMPNO	ENAME	ANNSAL	HIREDATE	JOB
7698	BLAKE	34200	01-MAY-81	
7654	MARTIN	15000	28-SEP-81	
7499	ALLEN	19200	20-FEB-81	
7844	TURNER	18000	08-SEP-81	
...				

6 rows selected.

열 추가에 대한 지침

- 열을 추가하거나 수정할 수 있지만, 테이블에서 열을 삭제할 수는 없습니다.
- 열이 어디에 나타날지를 명시할 수 없습니다. 새로운 열은 마지막 열이 됩니다.

위의 예는 DEPT30 테이블에 JOB 열을 추가합니다. JOB 열은 테이블에서 마지막 열이 됩니다.

주: 열을 추가할 때 테이블이 이미 어떤 행을 포함하고 있다면, 새로운 열은 이미 존재하는 모든 행에 대해서 null로 초기화합니다.

열 수정

- 열의 데이터형, 크기 그리고 디폴트 값을 변경할 수 있습니다.

```
ALTER TABLE dept30
MODIFY (ename VARCHAR2(15));
Table altered.
```

- 디폴트 값을 변경하는 것은 오직 테이블에 가해지는 변경 이후의 삽입에만 영향을 미칩니다.

열 수정

ALTER TABLE 문장에 MODIFY 절을 사용하여 열 정의를 수정할 수 있습니다. 열 수정은 열의 데이터형, 크기 그리고 디폴트 값 변경을 할 수 있습니다.

지침

- 숫자 열의 **정밀도**나 폭을 증가시킵니다.
- 열이 오직 null 값만을 포함하거나 테이블에 행이 없으면 열의 폭을 감소시킵니다.
- 열이 null 값을 포함하면 데이터형을 변경시킵니다.
- 열이 null 값을 포함하거나 크기를 변경하지 않으면 CHAR 열을 VARCHAR2 데이터형으로 변경하거나
- VARCHAR2 열을 CHAR 데이터형으로 변경합니다.
- 열의 디폴트 값을 변경시키는 것은 오직 테이블에 가해지는 이후의 삽입에만 영향을 미칩니다.

테이블 삭제

- 테이블의 모든 데이터와 구조가 삭제됩니다.
- 어떤 결정되지 않은 트랜잭션이 커밋됩니다.
- 모든 인덱스가 삭제됩니다.
- 이 문장은 롤백할 수 없습니다.

```
SQL> DROP TABLE dept30;  
Table dropped.
```

테이블 삭제

DROP TABLE 문장은 Oracle8 테이블의 정의를 삭제합니다. 테이블을 삭제할 때 데이터베이스는 테이블에 있는 모든 데이터와 그와 [연관](#)된 모든 인덱스를 상실합니다.

구문형식

```
DROP TABLE table;
```

여기서: table 테이블의 이름입니다.

지침서

- 테이블로부터 모든 데이터를 삭제합니다.
- 뷰나 동의어는 남지만 무효가 됩니다.
- 어떤 결정되지 않은 트랜잭션을 커밋합니다.
- 오직 테이블의 생성자나 DROP ANY TABLE 권한을 가진 사용자만이 테이블을 삭제할 수 있습니다.

일단 실행된 DROP TABLE 문장은 복구할 수 없습니다. 오라클 서버는 DROP TABLE 문장을 실행할 때 삭제에 대한 질문을 하지 않습니다. 여러분이 해당 테이블을 소유하거나

high-level 권한을 가지고 있다면, 테이블은 즉시 삭제될 것입니다. 모든 DDL 문장이 커밋될 것이므로 트랜잭션을 영구적으로 만듭니다.

객체 이름 변경

- 테이블 이름, 뷰, 시퀀스 또는 동의어를 변경하기 위해, **RENAME** 문장을 실행합니다.

```
SQL> RENAME dept TO department;
Table renamed.
```

- 객체의 소유자이어야 합니다.

테이블 이름 변경

추가적인 DDL 문장은 테이블, 뷰, 시퀀스 또는 동의어에 사용되는 RENAME 문장을 포함합니다.

구문형식

```
RENAME      old_name  TO  new_name;
```

여기서: old_name 예전의 테이블 이름, 뷰, 시퀀스 또는
동의어입니다.

`new_name` 테이블, 뷰, 시퀀스 또는 동의어의 새로운 이름입니다.

여러분은 이름 변경하는 객체의 소유자이어야 합니다.

테이블 삭제

- **TRUNCATE TABLE 문장:**

- 테이블의 모든 행을 삭제합니다.
- 해당 테이블에 사용된 기억 공간을 해제합니다.

```
SQL> TRUNCATE TABLE department;  
Table truncated.
```

- TRUNCATE를 사용하여 삭제한 행을 롤백 할 수 없습니다.
- 대안적으로, DELETE 문장을 사용하여 행을 삭제합니다.

테이블 삭제

테이블로부터 모든 행을 삭제하거나 해당 테이블의 저장 공간을 해제하기 위한 다른 DDL 문장은 TRUNCATE TABLE 문장입니다. TRUNCATE TABLE 문장을 사용할 때 삭제한 행을 롤백할 수 없습니다.

구문형식

```
TRUNCATE TABLE table;
```

여기서: table 테이블의 이름입니다.

여러분은 테이블의 소유자이거나 테이블을 삭제할 수 있는 DELETE ANY TABLE 시스템 권한을 가져야 합니다.

DELETE 문장은 테이블의 모든 행을 삭제할 수 있지만, 저장 공간을 해제할 수는 없습니다.

테이블에 주석문 추가

- **COMMENT** 문장을 사용하여 테이블이나 열에 주석문을 추가할 수 있습니다.

```
SQL> COMMENT ON TABLE emp  
2 IS 'Employee Information';  
Comment created.
```

- 주석문은 데이터 사전 뷰를 통하여 볼 수 있습니다.
 - ALL_COL_COMMENTS
 - USER_COL_COMMENTS
 - ALL_TAB_COMMENTS
 - USER_TAB_COMMENTS

테이블에 주석문 추가

COMMENT 문장을 사용하여 열, 테이블, 뷰 또는 스냅샷에 대하여 2000 byte 까지 주석문을 추가할 수 있습니다. 주석문은 데이터 사전에 저장되며 COMMENTS 열에서 다음의 데이터 사전중의 하나로 볼 수 있습니다.

- ALL_COL_COMMENTS
- USER_COL_COMMENTS
- ALL_TAB_COMMENTS
- USER_TAB_COMMENTS

구문형식

```
COMMENT ON TABLE table | COLUMN table.column  
IS 'text';
```

여기서: table 테이블의 이름입니다.
 column 테이블에 있는 열의 이름입니다.
 text 주석 문장입니다.

공백 스트링('')을 설정하여 데이터베이스로부터의 주석문을 삭제할 수 있습니다.

```
SQL> COMMENT ON TABLE emp IS ' ';
```

제 11 장 제약조건포함

목적

본 과정을 마치면 , 다음을 할 수 있어야 합니다.

- 제약조건을 기술합니다.
- 제약조건을 생성하고 유지합니다.

과정 목적

본 과정에서는 무결성 제약조건을 포함하는 업무 규칙을 수행하는 법을 배웁니다.

제약조건이란?

- 제약조건은 테이블 레벨에서 규칙을 적용합니다.
- 제약조건은 종속성이 존재할 경우 테이블의 삭제를 방지합니다.
- 다음의 제약조건 유형이 오라클에서 유효합니다.
 - **NOT NULL**
 - **UNIQUE Key**
 - **PRIMARY Key**
 - **FOREIGN Key**
 - **CHECK**

제약조건

오라클 서버는 부적합한 데이터가 테이블에 삽입되는 것을 방지하기 위해 제약조건(Constraint)을 사용합니다.

- 그 테이블에서 행이 삽입되거나, 갱신되거나, 삭제될 때마다 테이블에서 규칙을 적용합니다.
- 다른 테이블에 종속성이 있다면 테이블의 제거를 방지합니다.
- Developer/2000 같은 오라클 툴들에 대한 규칙을 제공합니다.

데이터 무결성 제약조건

제약조건	기 술
NOT NULL	이열은 null 값을 포함하지 않음을 지정합니다.
UNIQUE Key	테이블의 모든 행에 대해 유일해야 하는 값을 가진 열 또는 열의 조합을 지정합니다.
PRIMARY KEY	유일하게 테이블의 각 행을 식별합니다
FOREIGN KEY	열과 참조된 테이블의 열 사이의 외래키 관계를 적용하고 설정합니다.
CHECK	참이어야 하는 조건을 지정합니다.

자세한 내용은 *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “**CONSTRAINT Clause.**”를 참조하십시오.

제약조건 지침

- 사용자가 제약조건을 명명하거나 오라클 서버는 **SYS_Cn** 포맷을 사용하여 이름을 생성합니다.
- 제약조건 생성 시점:
 - 테이블이 생성되는 시간에
 - 테이블이 생성된 후에
- 열 또는 테이블 레벨에서 제약조건을 정의합니다.
- 데이터 사전에서 제약조건을 봅니다.

제약조건 지침

모든 제약조건은 데이터 사전에 저장됩니다. 제약조건은 의미있는 이름을 부여했다면 참조가 쉽습니다. 제약조건 이름은 표준 객체 이름 규칙을 수행해야 합니다. 제약조건을 명명하지 않는다면, 오라클은 유일한 제약조건 이름을 생성하기 위해, SYS_Cn 형식으로 이름을 생성 합니다.

제약조건은 테이블 생성 시나 테이블이 생성된 후에 정의될 수 있습니다.

USER_CONSTRAINTS 데이터 사전 뷰를 검색함으로써 지정 테이블에 대해 정의된 제약조건을 볼 수 있습니다.

제약조건 정의

```
CREATE TABLE [schema.] table
(column datatype [DEFAULT expr]
[column_constraint],
...
[table_constraint]);
```

```
CREATE TABLE emp(
empno NUMBER(4),
ename VARCHAR2(10),
...
deptno NUMBER(7,2) NOT NULL,
CONSTRAINT emp_empno_pk
PRIMARY Key (EMPNO));
```

제약조건 정의

슬라이드는 테이블 생성 중에 제약조건을 정의하는 구문을 제공합니다.

구문에서:

schema	소유자명과 동일합니다.
table	테이블의 이름입니다.
DEFAULT expr	값이 INSERT 문장에서 생략된다면 디폴트 값을 지정합니다.
column	열의 이름입니다.
datatype	열의 데이터 유형과 길이입니다.
column_constraint	열 정의의 일부로서 무결성 제약조건입니다.
table_constraint	테이블 정의의 일부로서 무결성 제약조건입니다.

자세한 내용은 *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “CREATE TABLE” 을 참조하십시오.

제약조건 정의

- 열 레벨 제약조건

```
column [CONSTRAINT constraint_name] constraint_type,
```

- 테이블 레벨 제약조건

```
column,...  
[CONSTRAINT constraint_name] constraint_type  
(column, ...),
```

제약조건 정의 (계속)

제약조건은 대개 테이블과 동시에 생성됩니다. 제약조건은 테이블의 생성 후에 테이블로 추가될 수 있고 또한 일시적으로 불가능해 질 수 있습니다.

제약조건레벨	기 술
열	열별로 정의. 무결성 제약조건의 어떤 유형도 정의 가능.
테이블	하나 이상의 열을 참조하고, 테이블의 열 정의와는 개별적으로 정의. NOT NULL 을 제외한 임의의 제약조건 정의 가능.

구문에서

constraint_name	제약조건의 이름입니다.
constraint_type	제약조건의 유형입니다.

NOT NULL 제약조건

열에 대해 **null** 값이 허용되지 않도록 보장합니다.

EMP

EMPNO	ENAME	JOB	...	COMM	DEPTNO
7839	KING	PRESIDENT			10
7698	BLAKE	MANAGER			30
7782	CLARK	MANAGER			10
7566	JONES	MANAGER			20
...					

NOT NULL constraint
(no row may contain
a null value for
this column)

Absence of NOT NULL
constraint
(any row can contain
null for this column)

NOT NULL constraint

(any row can contain
null for this column)

NOT NULL 제약조건

NOT NULL 제약조건은 열에서 null 값이 허용되지 않도록 보증합니다. NOT NULL 제약조건이 없는 열은 디폴트로 null 값을 포함할 수 있습니다.

NOT NULL 제약조건

열 레벨에서만 정의됩니다.

```
SQL> CREATE TABLE emp(  
2     empno    NUMBER(4),  
3     ename    VARCHAR2(10) NOT NULL,  
4     job      VARCHAR2(9),  
5     mgr      NUMBER(4),  
6     hiredate DATE,  
7     sal      NUMBER(7,2),  
8     comm     NUMBER(7,2),  
9     deptno   NUMBER(7,2) NOT NULL);
```

NOT NULL 제약조건(계속)

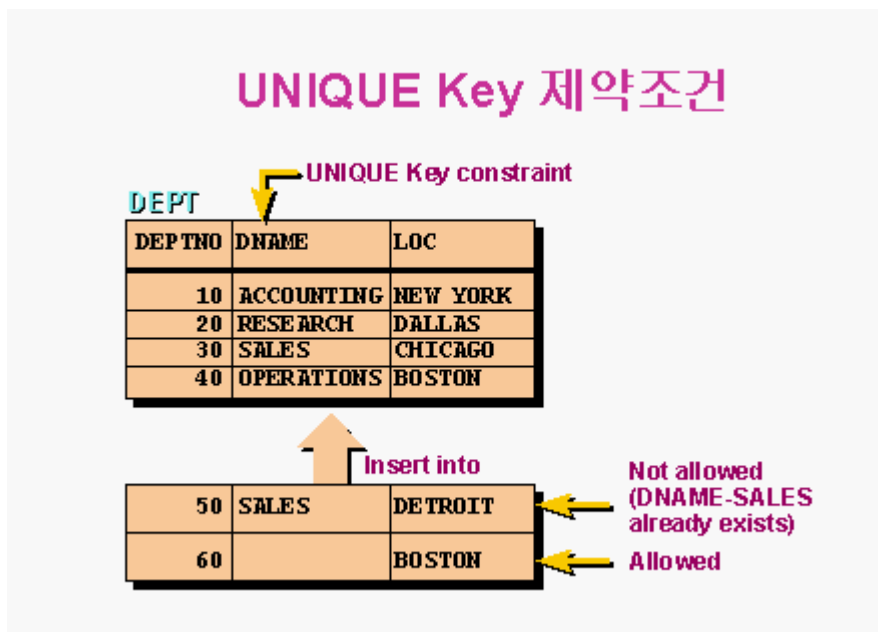
NOT NULL 제약조건은 테이블 레벨이 아닌 열 레벨에서만 지정될 수 있습니다. 위의 예는

EMP 테이블의 ENAME 과 DEPTNO 열에 대해 NOT NULL 제약조건을 적용합니다. 이 제약조건은 이름이 명시되지 않았기 때문에, 오라클 서버는 이름을 생성합니다.

제약조건 지정 중에 제약조건의 이름을 지정할 수 있습니다.

```
... deptno NUMBER(7,2)
      CONSTRAINT emp_deptno_nn NOT NULL...
```

주: 본 과정의 예에서 기술된 모든 제약조건은 본 과정에서 제공된 견본 테이블에 나타나지 않을 수도 있습니다. 원한다면 이 제약조건은 테이블에 추가될 수 있습니다.



UNIQUE Key 제약조건

UNIQUE Key 무결성 제약조건은 열 또는 열의 집합의 모든 값들이 유일해야 함을 요구합니다. 즉, 지정된 열 또는 열의 집합에서 중복 값을 가지는 테이블의 두 행은 없습니다. UNIQUE Key 제약조건 정의에 포함된 열(열의 집합)은 unique Key 라고 부릅니다. UNIQUE Key 가 하나 이상의 열을 포함한다면, 열 그룹은 composite unique Key 라고 부릅니다.

UNIQUE Key 제약조건은 동일 열에 대해 NOT NULL 제약조건을 정의하지 않는다면 null의 출력을 허용합니다. 사실, 행은 null 이 어느 것과도 동일하게 여겨지지 않기 때문에 NOT NULL 제약조건이 없으면 열에 대해 null 을 포함할 수 있습니다. 열(또는 조합 UNIQUE

키의 모든 열)에서 null 은 항상 UNIQUE Key 제약조건을 만족합니다.

주: 하나 이상의 열에서 UNIQUE 제약조건에 대한 검색 메커니즘 때문에, 부분적으로 null 인 조합 UNIQUE Key 제약조건의 null 이 아닌 열에서 동일한 값을 가질 수 없습니다

UNIQUE Key 제약조건

테이블 레벨 또는 열 레벨에서 정의 됩니다.

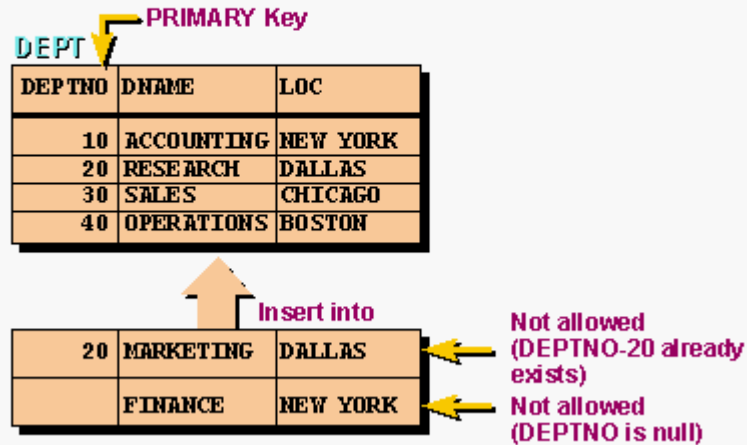
```
SQL> CREATE TABLE dept(  
2     deptno    NUMBER(2),  
3     dname     VARCHAR2(14),  
4     loc       VARCHAR2(13),  
5     CONSTRAINT dept_dname_uk UNIQUE(dname));
```

UNIQUE Key 제약조건(계속)

UNIQUE Key 제약조건은 열 또는 테이블 레벨에서 정의될 수 있습니다. 조합 유일 키는 테이블 레벨 정의를 사용하여 생성됩니다. 슬라이드의 예는 DEPT 테이블의 DNAME 열에 대해 UNIQUE Key 제약조건을 적용합니다. 제약조건의 이름은 DEPT_DNAME_UK 입니다.

주: 오라클 서버는 유일 키에서 유일 인덱스를 명시적으로 생성하여 UNIQUE Key 제약조건을 적용합니다.

PRIMARY Key 제약조건



PRIMARY Key 제약조건

PRIMARY Key 제약조건은 테이블에 대한 기본 키를 생성합니다. 하나의 기본 키만이 각 테이블에 대해 생성될 수 있습니다. PRIMARY Key 제약조건은 테이블에서 각행을 유일하게 식별하는 열 또는 열의 집합입니다. 이 제약조건은 열 또는 열의 집합의 유일성을 요구하고 null 값을 포함할 수 없음을 보증합니다.

PRIMARY Key 제약조건

테이블 레벨 또는 열 레벨에서 정의됩니다.

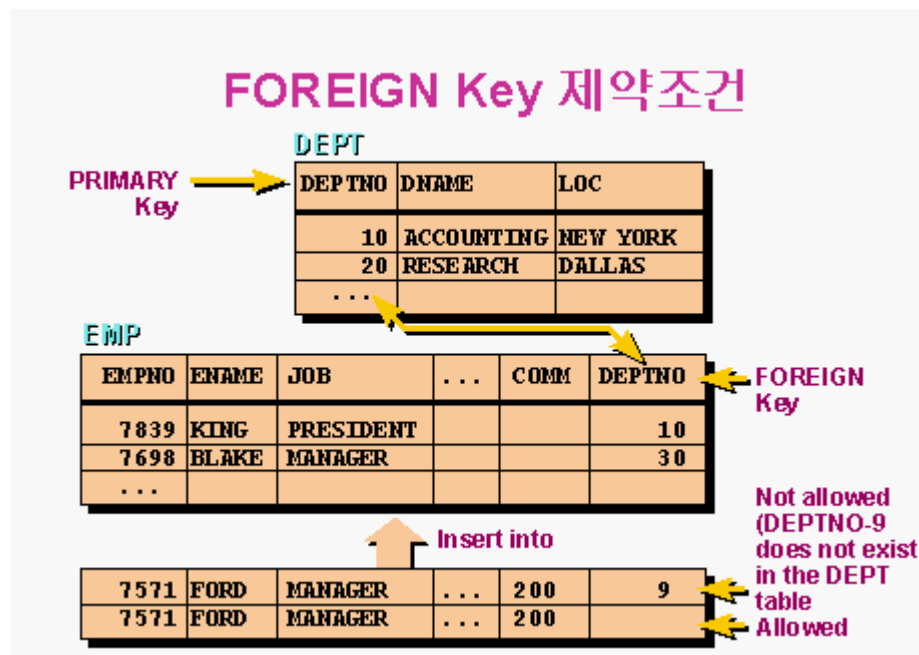
```
SQL> CREATE TABLE dept(
2     deptno    NUMBER(2),
3     dname     VARCHAR2(14),
4     loc       VARCHAR2(13),
5     CONSTRAINT dept_dname_uk UNIQUE (dname),
6     CONSTRAINT dept_deptno_pk PRIMARY Key(deptno));
```

PRIMARY Key 제약조건(계속)

PRIMARY Key 제약조건은 열 레벨 또는 테이블 레벨에서 정의될 수 있습니다. 조합 PRIMARY Key 는 테이블 정의 레벨을 사용하여 생성됩니다.

슬라이드의 예는 DEPT 테이블의 DEPTNO 열에서 PRIMARY Key 제약조건을 정의합니다. 제약조건 이름은 DEPT_DEPTNO_PK 입니다.

주: UNIQUE 인덱스는 자동으로 PRIMARY Key 열에 대해 생성됩니다



FOREIGN Key 제약조건

FOREIGN Key, 또는 참조 무결성 제약조건은 열 또는 열의 집합을 foreign key 로 지정하여 동일 테이블 또는 다른 테이블 간의 기본 키 또는 유일 키 사이의 관계를 설정합니다.

슬라이드의 예에서 DEPTNO 는 EMP 테이블(종속 또는 자식 테이블)에서 외래 키로서 정의되었습니다. 이것은 DEPT 테이블(참조 또는 부모 테이블)의 DEPTNO 열을 참조합니다.

외래 키 값은 부모 테이블에서 존재하는 값과 일치해야 하거나 NULL 이 되어야 합니다.

외래 키는 데이터 값을 기초로 하며 순전히 논리적이거나 물리적이거나 포인터가 아닙니다.

FOREIGN Key 제약조건

테이블 레벨 또는 열 레벨에서 정의됩니다.

```
SQL> CREATE TABLE emp(  
2     empno    NUMBER(4),  
3     ename    VARCHAR2(10) NOT NULL,  
4     job      VARCHAR2(9),  
5     mgr      NUMBER(4),  
6     hiredate DATE,  
7     sal      NUMBER(7,2),  
8     comm     NUMBER(7,2),  
9     deptno   NUMBER(7,2) NOT NULL,  
10    CONSTRAINT emp_deptno_fk FOREIGN Key (deptno)  
11    REFERENCES dept (deptno));
```

FOREIGN Key 제약조건(계속)

FOREIGN Key 제약조건은 열 또는 테이블 제약조건 레벨에서 정의될 수 있습니다. 조합 외래 키는 테이블 레벨 정의를 사용하여 생성됩니다. 슬라이드의 예는 EMP 테이블의 DEPTNO 열에서 FOREIGN Key 제약조건을 정의합니다. 제약조건 이름은 EMP_DEPTNO_FK 입니다

FOREIGN Key 제약조건 키워드

- **FOREIGN Key**

테이블 제약조건 레벨에서 자식 테이블 열을 정의합니다.

- **REFERENCES**

부모 테이블에서 테이블과 열을 식별합니다.

- **ON DELETE CASCADE**

자식 테이블에서 행의 종속을 삭제하고 부모 테이블에서 삭제를 허용합니다.

FOREIGN Key 제약조건 (계속)

외래 키는 자식 테이블에서 정의되고, 참조된 열을 포함하는 테이블은 부모 테이블입니다.

외래 키는 다음의 키워드 결합을 사용하여 정의됩니다.

- FOREIGN Key 는 테이블 제약조건 레벨에서 자식 테이블 열을 정의하기 위해 사용됩니다.
- REFERENCES 는 부모 테이블에서의 열과 테이블을 식별합니다.
- ON DELETE CASCADE 는 부모 테이블에서 행이 제거될 때 자식 테이블에 있는 종속적인 행도 자동적으로 제거된다는 것을 나타냅니다

ON DELETE CASCADE 옵션이 없다면, 부모 테이블에 있는 행을 자식 테이블에서 참조할 경우에 삭제될 수 없습니다.

CHECK 제약조건

- 각행을 만족해야 하는 조건을 정의합니다.
- 표현식은 허용되지 않습니다:
 - CURRVAL, NEXTVAL, LEVEL, ROWNUM 에 대한 참조
 - SYSDATE, UID, USER, USERENV 함수에 대한 호출
 - 다른 행에 있는 다른 값을 참조하는 질의

```
..., deptno NUMBER(2),  
CONSTRAINT emp_deptno_ck  
CHECK (DEPTNO BETWEEN 10 AND 99),...
```

CHECK 제약조건

CHECK 제약조건은 각 행이 만족해야 하는 조건을 정의합니다. 조건은 다음의 경우만 제외하고는 질의 조건과 동일한 구성을 사용할 수 있습니다.

- CURRVAL, NEXTVAL, LEVEL, ROWNUM pseudocolumn 에 대한 참조
- SYSDATE, UID, USER, USERENV 함수에 대한 호출

- 다른 행의 다른 값을 참조하는 질의

단일 열은 복수 CHECK 제약조건을 가질 수 있습니다. 열에서 정의할 수 있는 CHECK 제약조건의 수에 대한 한계는 없습니다. CHECK 제약조건은 열 레벨 또는 테이블 레벨에서 정의될 수 있습니다.

제약조건 추가

```
ALTER TABLE table
ADD [CONSTRAINT constraint] type (column);
```

- 제약조건의 추가 또는 삭제는 가능, 수정은 불가능
- 제약조건의 활성화 또는 비활성화
- **MODIFY** 절을 사용하여 **NOT NULL** 제약조건을 추가합니다.

제약조건 추가

ADD 절을 가지는 ALTER TABLE 문장을 사용하여 기존의 테이블에 대한 제약조건을 추가할 수 있습니다.

구문형식에서:

<i>table</i>	테이블의 이름입니다.
<i>constraint</i>	제약조건의 이름입니다.
<i>type</i>	제약조건 유형입니다.
<i>column</i>	제약조건에 의해 영향 받은 열의 이름입니다.

제약조건 이름 구문형식은 비록 권장되지만 선택적입니다. 제약조건을 명명하지 않는다면, 시스템이 제약조건 이름을 생성합니다.

지침

- 제약조건의 구조를 수정할 수는 없지만 제약조건을 추가, 삭제, 활성화 , 비활성화 할 수 있습니다.
- ALTER TABLE 문장의 MODIFY 절을 사용하여 기존의 열에 대해 NOT NULL 제약조건을 추가할 수 있습니다.

주: 데이터는 열이 추가되는 시점에서 기존의 열에 대해 명시될 수 없기 때문에 테이블에 행이 하나도 없을 경우에만 NOT NULL 열을 정의할 수 있습니다.

제약조건 추가

EMP테이블 안에 유효한 직원으로 이미 존재해야 하는 관리자를 나타내는 **EMP** 테이블에 대해서 **FOREIGN Key**제약조건을 추가할 수 있습니다.

```
SQL> ALTER TABLE      emp
2  ADD CONSTRAINT emp_mgr_fk
3      FOREIGN Key(mgr) REFERENCES emp(empno) ;
Table altered.
```

제약조건 추가 (계속)

위의 예는 EMP 테이블에서 **FOREIGN Key** 를 생성합니다. 제약조건은 EMP 테이블에서 유효한 종업원으로 존재하는 관리자를 확인합니다.

제약조건 삭제

- **EMP**테이블에서 관리자 제약조건을 제거합니다.

```
SQL> ALTER TABLE      emp
2  DROP CONSTRAINT      emp_mgr_fk;
Table altered.
```

- **DEPT** 테이블에서 **PRIMARY Key** 제약조건을 제거하고 **EMP.DEPTNO** 열에서 관련 **FOREIGN Key** 제약조건을 삭제합니다.

```
SQL> ALTER TABLE      dept
2  DROP PRIMARY Key CASCADE;
Table altered.
```

제약조건 삭제

제약조건을 삭제하기 위해서, `USER_CONSTRAINTS` 와 `USER_CONS_COLUMNS` 데이터 사전 뷰에서 제약조건 이름을 식별할 수 있습니다. 이때 `DROP` 절과 `ALTER TABLE` 문장을 사용합니다. `DROP` 의 `CASCADE` 옵션은 모든 종속적인 제약조건이 삭제되게 합니다.

구문형식

```
ALTER TABLE    table
DROP  PRIMARY Key | UNIQUE (column) |
CONSTRAINT      constraint [CASCADE];
```

여기서: `table` 테이블의 이름입니다.
 `column` 제약조건에 의해 영향 받은 열의 이름입니다.
 `constraint` 제약조건의 이름입니다.

무결성 제약조건을 삭제할 때, 그 제약조건은 더 이상 오라클 서버에 의해 적용되지 않으며, 데이터 사전에서 확인할 수 없습니다.

제약조건 사용불가

- 무결성 제약조건을 비활성화 하기 위해 **ALTER TABLE**문장의 **DISABLE** 절을 실행합니다.
- 종속적인 무결성 제약조건을 비활성화 하기 위해서 **CASCADE** 옵션을 적용합니다.

```
SQL> ALTER TABLE emp  
2 DISABLE CONSTRAINT emp_empno_pk CASCADE;  
Table altered.
```

제약조건 사용불가

DISABLE 절을 가진 ALTER TABLE 문장을 사용하여 삭제 또는 재생성 없이 제약조건을 비활성화할 수 있습니다.

구문형식

```
ALTER TABLE table  
DISABLE CONSTRAINT constraint [CASCADE];
```

여기서: table 테이블의 이름입니다.
 constraint 제약조건의 이름입니다.

지침서

CREATE TABLE 문장과 ALTER TABLE 문장으로 DIABLE 절을 사용할 수 있습니다.
CASCADE 절은 종속적인 무결성 제약조건을 비활성화 합니다.

제약조건 사용가능

- **ENABLE** 절을 사용하여 테이블 정의에서 현재 비활성화된 무결성 제약조건을 활성화 합니다.

```
SQL> ALTER TABLE      emp
      2  ENABLE CONSTRAINT  emp_empno_pk;
Table altered.
```

- **UNIQUE** 키 또는 **PRIMARY Key**가 활성화되면 **UNIQUE** 또는 **PRIMARY Key** 인덱스는 자동으로 생성됩니다.

제약조건 사용가능

ENABLE 절을 가진 ALTER TABLE 문장과 사용하여 삭제 또는 재생성 없이 제약조건을 활성화할 수 있습니다.

구문형식

```
ALTER TABLE  table
ENABLE  CONSTRAINT constraint;
```

여기서: table 테이블의 이름입니다.
 constraint 제약조건의 이름입니다.

지침서

- 제약조건이 활성화 된다면, 그 제약조건은 테이블의 모든 데이터에 대해 적용됩니다. 테이블의 모든 데이터는 제약조건과 일치해야 합니다.
- UNIQUE Key 또는 PRIMARY Key 제약조건이 활성화 된다면, UNIQUE 또는 PRIMARY Key 인덱스는 자동으로 생성됩니다.
- CREATE TABLE 문장과 ALTER TABLE 문장으로 ENABLE 절을 사용할 수 있습니다.

제약조건 보기

모든 제약조건 정의와 이름을 보기 위해
USER_CONSTRAINTS 테이블을 질의합니다.

```
SQL> SELECT constraint_name, constraint_type,  
2         search_condition  
3 FROM   user_constraints  
4 WHERE  table_name = 'EMP';
```

CONSTRAINT_NAME	C SEARCH_CONDITION
SYS_C00674	C EMPNO IS NOT NULL
SYS_C00675	C DEPTNO IS NOT NULL
EMP_EMPNO_PK	P
...	

제약조건 보기

테이블을 생성한 후에, DESCRIBE 명령어를 생성하여 그것의 존재를 확인할 수 있습니다.
검증할 수 있는 제약조건은 NOT NULL 제약조건입니다. 테이블에서 모든 제약조건을 보기
위해서는 USER_CONSTRAINTS 테이블을 질의합니다.

위의 예는 EMP 테이블에서 모든 제약조건을 디스플레이합니다.

주: 테이블 소유자가 이름 붙이지 않은 제약조건은 시스템이 이름을 부여합니다. 제약조건
유형에서, C 는 CHECK 를 담당하고, P 는 PRIMARY Key 를 담당하고, R 은 참조 무결성을
담당하며, U 는 UNIQUE 키를 담당합니다. NULL 제약조건은 실제로는 CHECK 제약조건
임을 명심하십시오.

제약조건과 연관된 열 보기

USER_CONS_COLUMNS 뷰에서 제약조건 이름과 관련된 열을 봅니다.

```
SQL> SELECT constraint_name, column_name  
2 FROM user_cons_columns  
3 WHERE table_name = 'EMP';
```

CONSTRAINT_NAME	COLUMN_NAME
EMP_DEPTNO_FK	DEPTNO
EMP_EMPNO_PK	EMPNO
EMP_MGR_FK	MGR
SYS_C00674	EMPNO
SYS_C00675	DEPTNO

제약조건 보기 (계속)

USER_CONS_COLUMNS 데이터 사전 뷰를 질의함으로써 제약조건에 관련된 열의 이름을 볼 수 있습니다. 이 뷰는 특별히 시스템 지정 이름을 사용하는 제약조건에 유용합니다.

제 12 장 뷰 생성

목 적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 뷰 설명
- 뷰 생성
- 뷰를 통한 데이터 검색
- 뷰의 정의 변경
- 뷰를 통한 데이터 삽입, 갱신, 삭제
- 뷰 삭제

과정 목적

본 과정에서는 뷰 생성과 사용에 대해 배웁니다. 또한, 뷰에 대한 정보를 읽어 들이기 위해 관련 데이터 사전 객체를 질의하는 것을 배우게 됩니다.

데이터베이스 객체

객체	설 명
Table	행과 열로 구성된 기본적인 저장 매체의 단위
View	하나 이상의 테이블로부터 데이터의 부분집합을 논리적으로 표현
Sequence	기본 키 값을 발생
Index	어떤 질의의 성능을 향상
Synonym	객체에 대체 이름을 부여

뷰란?

EMP Table

EMPNO	ENAME	JOB	MANAGER	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10
7782	CLARK	MANAGER	7839 09-NOV-81	2800	1400		10
7934	MILLER	CLERK	7782 23-JAN-82	1300	300		10
7566	JONES	MANAGER	7839 02-MAR-81	2900			20
7698	TURNER	SALESMAN	7566 08-SEP-81	1500			30
7900	JAMES	CLERK	7698 03-DEC-81	950			30
7521	WARD	SALESMAN	7698 22-FEB-81	1250	500		30

EMPVU10 View

EMPNO	ENAME	JOB
7839	KING	PRESIDENT
7782	CLARK	MANAGER
7934	MILLER	CLERK

뷰란?

테이블의 뷰를 생성함으로써 데이터 조합 또는 논리적 부분집합을 나타낼 수 있습니다.

뷰는 테이블 또는 다른 뷰를 기초로 하는 논리적 테이블입니다. 뷰는 그 자체로서 소유하는 데이터는 없지만 창문처럼 창문을 통해 어떤 데이터를 보거나 변경할 수 있습니다.

뷰의 사용 목적은?

- 데이터베이스 액세스를 제한하기 위해
- 복잡한 질의를 쉽게 만들기 위해
- 데이터의 독립성을 허용하기 위해
- 동일한 데이터의 다른 뷰를 나타내기 위해

뷰의 장점

- 뷰는 데이터베이스의 선택적인 부분을 디스플레이 할 수 있기 때문에 데이터베이스에 대한 액세스를 제한할 수 있습니다.
- 사용자가 복잡한 질의로부터 결과를 검색하기 위한 단순한 질의를 만들도록 합니다. 예를 들면, 뷰는 조인 문장을 작성하는 방법을 알 필요 없이 다중 테이블에서 정보를 질의 할 수 있도록 해 줍니다.
- ad hoc 사용자와 어플리케이션 프로그램에 대한 데이터 독립성을 제공합니다. 하나의 뷰는 여러 개의 테이블로부터 데이터를 검색하는데 사용될 수 있습니다.
- 특정 조건에 따라서 데이터를 액세스하는 사용자 그룹을 제공합니다.

보다 자세한 정보를 알고자 한다면, 다음을 참조하십시오. *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “CREATE VIEW.”

단순 뷰와 복합 뷰

특징	단순 뷰	복합 뷰
테이블 수	하나	하나 이상
함수 포함	없음	있음
데이터 그룹 포함	없음	있음
뷰를 통한 DML	있음	없음

단순 뷰와 복합 뷰

뷰는 2 가지 종류 즉, 단순 뷰와 복합 뷰가 있습니다. 근본적인 차이점은 DML(삽입, 갱신, 삭제) 작업에 관련되어 있습니다.

- 단순 뷰:
 - 오직 하나의 테이블에서만 데이터가 유래
 - 데이터 그룹 또는 함수를 포함하지 않음
 - 뷰를 통한 DML 수행 가능
- 복합 뷰:
 - 다중 테이블에서 데이터 유래
 - 데이터 그룹 또는 함수를 포함
 - 뷰를 통한 DML을 항상 허용하지 않음

뷰 생성

- **CREATE VIEW** 문장 내에 서브쿼리를 내장합니다.

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
[(alias[, alias]...)]
AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY]
```

- 서브쿼리는 복합 **SELECT** 구문을 포함할 수 있습니다.
- 서브쿼리는 **ORDER BY** 절을 포함할 수 없습니다.

뷰 생성

CREATE VIEW 문장에서 서브쿼리를 **내장**하여 뷰를 생성할 수 있습니다.

구문에서:

OR REPLACE	이미 존재한다면 뷰를 다시 생성합니다.
FORCE	기본 테이블의 존재 여부에 관계없이 뷰를 생성합니다.
NOFORCE	기본 테이블이 존재할 경우에만 뷰를 생성합니다.
view	뷰의 이름입니다.
alias	뷰 질의에 의해 선택된 표현식에 대한 이름을 지정합니다. 별칭 의 수는 뷰에 의해 선택된 표현식의 수와 일치해야 합니다.
subquery	SELECT 문장을 완성합니다. SELECT 리스트에서 열에 대한 별칭 을 사용할 수 있습니다.
WITH CHECK OPTION	단지 뷰에 의해 액세스 가능한 행만이 삽입, 갱신될 수 있음을 명시합니다.
constraint	CHECK OPTION 제약 조건에 대해 지정된 이름입니다.
WITH READ ONLY	뷰에서 수행될 수 있는 DML 작업이 하나도 없음을

확실히 합니다.

뷰 생성

- 부서 10의 직원의 세부사항을 포함하는 **EMPVU10** 뷰를 생성합니다.

```
SQL> CREATE VIEW empvu10
2 AS SELECT empno, ename, job
3 FROM emp
4 WHERE deptno = 10;
View created.
```

- SQL*Plus DESCRIBE** 명령어를 사용하여 뷰의 구조를 기술합니다.

```
SQL> DESCRIBE empvu10
```

뷰 생성 (계속)

위의 예는 부서 10의 모든 종업원에 대한 종업원 번호, 이름, 업무명을 포함하는 뷰를 생성합니다.

SQL*Plus DESCRIBE 명령어를 사용하여 뷰의 구조를 디스플레이 할 수 있습니다.

Name	Null?	Type
-----	-----	-----
EMPNO	NOT NULL	NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(9)

뷰 생성 지침

- 뷰를 정의하는 서브쿼리는 조인, 그룹, 서브쿼리를 포함하는 복잡한 SELECT 구문을 포함할 수 있습니다.
- 뷰를 정의하는 서브쿼리는 ORDER BY 절을 포함할 수 없습니다. ORDER BY 절은 뷰에서 데이터를 읽을 때 지정됩니다.

- CHECK OPTION으로 생성된 뷰에 대한 제약조건 이름을 지정하지 않는다면, 시스템은 SYS_Cn 포맷으로 디폴트 명을 지정합니다.
- 이전에 부여된 객체 권한을 삭제하거나, 뷰를 재생성 또는 재부여 하지 않고 뷰의 정의를 변경하기 위해 OR REPLACE 옵션을 사용할 수 있습니다.

뷰 생성

- 서브쿼리에서 열 별칭을 사용하여 뷰를 생성합니다.

```
SQL> CREATE VIEW salvu30
  2 AS SELECT empno EMPLOYEE_NUMBER, ename NAME,
  3          sal SALARY
  4 FROM emp
  5 WHERE deptno = 30;
View created.
```

- 주어진 별칭 이름으로 이 뷰에서 열을 선택합니다.

뷰 생성 (계속)

서브쿼리에서 열 **별칭**을 포함함으로써 열 이름을 제어할 수 있습니다.

위의 예는 부서 30에 대해 EMPLOYEE_NUMBER **별칭**을 갖는 종업원 번호, NAME **별칭**을 갖는 이름, SALARY **별칭**을 갖는 급여를 포함하는 뷰를 생성합니다.

또는, CREATE VIEW 절에서 열 **별칭**을 포함하여 열 이름을 제어할 수 있습니다.

뷰에서 데이터 검색

```
SQL> SELECT *
      2 FROM empvu30;
```

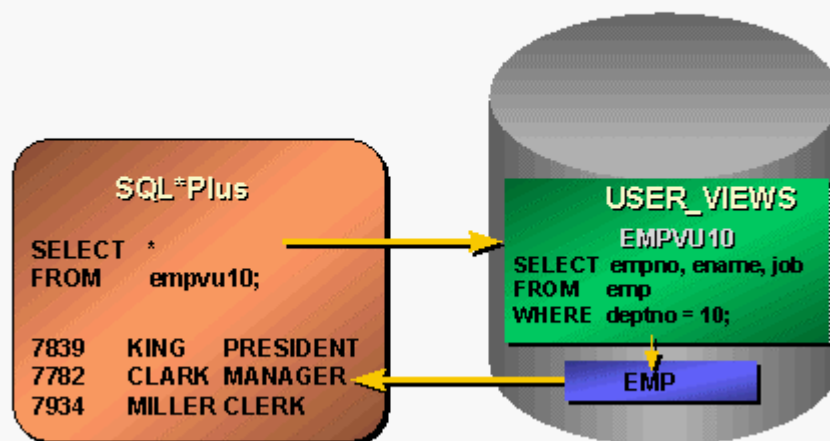
EMPLOYEE_	NUMBER	NAME	SALARY
7698	BLAKE		2850
7654	MARTIN		1250
7499	ALLEN		1600
7844	TURNER		1500
7900	JAMES		950
7521	WARD		1250

6 rows selected.

뷰에서 데이터 검색

임의의 테이블에서 할 수 있는 것처럼 뷰로부터 데이터를 검색할 수 있습니다. 전체 뷰의 내용을 디스플레이 하거나, 단지 특정 행과 열만을 볼 수 있습니다.

뷰 질의



데이터 사전의 뷰

일단 뷰가 생성되면, 뷰의 이름과 뷰 정의를 보기 위해 USER_VIEWS 라는 데이터 사전

테이블을 정의할 수 있습니다. 뷰를 만드는 SELECT 문장의 텍스트는 LONG 열에 저장됩니다.

데이터 액세스 뷰

뷰를 사용하여 데이터를 액세스 할 때 오라클 서버는 다음 작업을 수행합니다:

1. USER_VIEWS 데이터 사전 테이블에서 뷰 정의를 검색합니다.
2. 뷰 기반 테이블에 대한 액세스 권한을 확인합니다.
3. 뷰 질의를 기본 테이블 또는 테이블들에서의 동등한 작업으로 전환합니다. 달리 말하면, 데이터는 기본 테이블에서 검색되거나, 기본 테이블의 데이터에 갱신을 합니다.

뷰 수정

- **CREATE OR REPLACE VIEW** 절을 사용하여 **EMPVU10** 뷰를 수정합니다.

```
SQL> CREATE OR REPLACE VIEW empvu10
2   (employee_number, employee_name, job_title)
3   AS SELECT empno, ename, job
4   FROM emp
5   WHERE deptno = 10;
View created.
```

- **CREATE VIEW** 절에서 열 별칭들은 서브쿼리에서의 열과 동일한 순서로 나열됩니다.

뷰 수정

OR REPLACE 옵션은 비록 이 이름이 이미 존재할지라도 뷰가 생성될 수 있도록 해주므로, 그 소유자에 대한 오래된 뷰 버전을 대체합니다.

주: CREATE VIEW 절에서 열 별칭을 지정할 때, 별칭은 서브쿼리의 열과 동일한 명령으로 나열됨을 명심하십시오

복합 뷰 생성

- 두 테이블로부터 값을 디스플레이 하는 그룹 함수를 포함하는 복잡한 뷰를 생성합니다.

```
SQL> CREATE VIEW      dept_sum_vu
2      (name, minsal, maxsal, avgsal)
3  AS SELECT d.dname, MIN(e.sal), MAX(e.sal),
4      AVG(e.sal)
5  FROM   emp e, dept d
6  WHERE  e.deptno = d.deptno
7  GROUP BY d.dname;
View created.
```

복합 뷰 생성

위의 예는 부서명, 최소 급여, 최대 급여, 부서의 평균 급여의 복잡한 뷰를 생성합니다. 다른 이름이 뷰에 대해 지정됐음을 명심하십시오. 뷰의 어떤 열이 함수나 표현식에서 유래되었다면 **별칭**은 필수적입니다.

SQL*Plus DESCRIBE 명령어를 사용하여 뷰의 구조를 볼 수 있습니다. SELECT 문장을 생성하여 뷰의 내용을 디스플레이 합니다.

```
SQL> SELECT  *
2  FROM      dept_sum_vu;
```

NAME	MINSAL	MAXSAL	AVGSAL
ACCOUNTING	1300	5000	2916.6667
RESEARCH	800	3000	2175
SALES	950	2850	1566.6667

뷰에서 DML 연산 수행 규칙

- 단순 뷰에서 **DML** 연산을 수행할 수 있습니다.
- 뷰가 다음을 포함한다면 행을 제거할 수 없습니다.
 - 그룹 함수
 - **GROUP BY** 절
 - **DISTINCT** 키워드

뷰에서 DML 연산 수행

연산이 어떤 특정 규칙을 따른다면 뷰를 통해 데이터에 DML 작업을 수행할 수 있습니다.
다음 중 어느 것을 포함하지 않는다면 뷰에서 행을 제거할 수 있습니다:

- 그룹 함수
- GROUP BY 절
- DISTINCT 키워드

뷰에서 DML 연산 수행 규칙

- 다음을 포함한다면 뷰에서 데이터를 수정할 수 없습니다.
 - 이전 슬라이드에서 언급된 임의의 조건
 - 표현식으로 정의된 열
 - **ROWNUM** 의사열
- 다음을 포함한다면 뷰에 데이터를 추가할 수 없습니다.
 - 뷰가 이전 슬라이드 또는 위에 언급된 임의의 조건을 포함할 때
 - 뷰에 의해 선택되지 않은 **NOT NULL** 열이 기본 테이블에 있을 때

뷰에서 DML 연산 수행 규칙 (계속)

이전 슬라이드와 다음 중 일부에서 언급된 임의의 조건을 포함하지 않는다면 뷰에서 데이터를 수정할 수 있습니다:

- 표현식으로 정의된 열 - 예를 들면, SALARY * 12
- ROWNUM 의사열

위의 임의의 조건을 포함하지 않고 디폴트 값없는 NOT NULL 열을 포함하지 않는다면, 뷰를 통해 데이터를 추가할 수 있습니다. 뷰를 통해 기본 테이블로 직접적으로 값을 추가할 수 있음을 명심하십시오.

보다 자세한 정보를 알고자 한다면, 다음을 참조하십시오.

Oracle8 Server SQL Reference, Release 7.3 또는 8.0, "CREATE VIEW."

WITH CHECK OPTION 절 사용

- 뷰에 대한 DML 연산이 뷰의 조건을 만족할 때만 수행되도록 합니다.

```
SQL> CREATE OR REPLACE VIEW empvu20
 2 AS SELECT *
 3 FROM emp
 4 WHERE deptno = 20
 5 WITH CHECK OPTION CONSTRAINT empvu20_ck;
View created.
```

- 뷰에서 부서번호를 변경하려고 하면 WITH CHECK OPTION 제약조건에 위배되기 때문에 실패하게 됩니다.

WITH CHECK OPTION 절 사용

뷰를 통해 참조 무결성 체크를 수행하는 것이 가능합니다. 또한 데이터베이스 레벨에서 제약 조건을 적용할 수 있습니다. 뷰는 데이터 무결성을 보호하기 위해 사용될 수 있지만, 사용은 매우 제한됩니다.

뷰를 통해 수행되는 INSERT 와 UPDATE 는 WITH CHECK OPTION 절이 있으면 뷰를 가지고 검색할 수 없는 행 생성을 허용하지 않음을 명시합니다. 그러므로 삽입되거나 갱신되는 데이터에 대해서 무결성 제약조건과 데이터 검증 체크를 허용합니다.

뷰가 선택하지 않은 행에 대해 DML 작업을 수행하려고 하면, 지정된 제약조건 명과 함께 에러가 디스플레이 됩니다.

```
SQL> UPDATE empvu20
      2 SET      deptno = 10
      3 WHERE empno = 7788;
update empvu20
      *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

주: 부서번호가 10으로 변경된다면 뷰는 더 이상 그 종업원들을 볼 수 없기 때문에 아무 행도 갱신되지 않습니다. 그러므로, WITH CHECK OPTION 절로 뷰는 부서 20 종업원만 볼 수 있고, 이 종업원들에 대한 부서번호가 뷰를 통해 변경되는 것을 허용하지 않습니다.

DML 연산 부정

- 뷰의 정의에 **WITH READ ONLY** 옵션을 추가하여 **DML** 연산이 수행될 수 없게 합니다.

```
SQL> CREATE OR REPLACE VIEW empvu10
      2 (employee_number, employee_name, job_title)
      3 AS SELECT empno, ename, job
      4 FROM emp
      5 WHERE deptno = 10
      6 WITH READ ONLY;
View created.
```

- 뷰의 임의의 행에서 **DML**을 수행하려고 하면 오라클 서버 에러 **ORA-01752**가 발생합니다.

DML 연산 부정

WITH READ ONLY 옵션으로 뷰를 생성하여 뷰에서 DML 연산이 수행될 수 없게 합니다.

위의 예는 뷰에서 임의의 DML 연산을 하지 못하도록 EMPVU10 뷰를 수정합니다.

뷰에서 행을 제어하려고 하면 에러가 발생합니다.

```
SQL> DELETE FROM empvu10
      2 WHERE employee_number = 7782;
DELETE FROM empvu10
      *
```

ERROR at line 1:
ORA-01752:Cannot delete from view without exactly one key-preserved table

뷰 제거

- 뷰는 데이터베이스에서 기본 테이블을 기반으로 하기 때문에 데이터 손실 없이 뷰를 삭제합니다.

```
DROP VIEW view;
```

```
SQL> DROP VIEW empvu10;
View dropped.
```

뷰 제거

뷰를 제거하기 위해 DROP VIEW 문장을 사용합니다. 이 문장은 데이터베이스에서 뷰 정의를 제거합니다. 뷰 삭제는 뷰가 만들어진 기본 테이블에는 영향을 미치지 않습니다. 그 뷰에 기초하여 만들어진 뷰 또는 다른 어플리케이션은 무효화 됩니다. 생성자 또는 DROP ANY VIEW 권한을 가진 사람만 뷰를 제거할 수 있습니다.

구문형식에서:

view

뷰의 이름입니다.

제 13 장 시퀀스

목적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 일부 데이터베이스 객체와 그것의 사용을 기술합니다.
- 시퀀스를 생성, 유지, 사용합니다.

과정목적

본 과정에서는 일반적으로 사용되는 다른 데이터베이스 객체 중 일부를 생성하고 유지하는 방법을 배우게 됩니다. 이 객체는 시퀀스 , 인덱스, 동의어를 포함합니다

데이터베이스 객체

객체	설명
Table	행과 열로 구성된 기본적인 저장 매체의 단위
View	하나 이상의 테이블로부터 데이터의 부분집합을 논리적으로 표현
Sequence	기본 키 값을 발생
Index	어떤 질의의 성능을 향상
Synonym	객체에 대체 이름을 부여

데이터베이스 객체

많은 어플리케이션들은 유일한 숫자 값을 기본 키 값으로 사용합니다. 이 요구사항을 처리하기 위해 코드를 어플리케이션에서 구축하거나 유일한 번호를 생성하기 위해 시퀀스를 사용할 수 있습니다.

일부 질의의 성능을 향상시키길 원한다면, 인덱스의 생성을 고려해야 합니다. 또한 열 또는 열집합에서 유일성을 강화시키기 위해 인덱스를 사용할 수 있습니다. 동의어를 사용하여 객체를 위한 대체명을 제공할 수 있습니다.

시퀀스란?

- 자동적으로 유일번호를 생성합니다.
- 공유 가능한 객체
- 주로 기본 키 값을 생성하기 위해 사용됩니다.
- 어플리케이션 코드를 대체합니다.
- 메모리에 캐쉬되면 시퀀스 값을 액세스 하는 효율성을 향상시킵니다.

시퀀스란?

시퀀스는 테이블의 행에 대한 시퀀스 번호를 자동적으로 생성하기 위해 사용될 수 있습니다. 시퀀스는 사용자가 생성한 데이터베이스 객체이며 여러 사용자가 공유할 수 있습니다. 시퀀스에 대한 전형적인 사용은 각 행에 대해 유일해야 하는 기본 키 값을 생성하기 위해서입니다. 시퀀스는 내부 Oracle8 루틴에 의해 발생되고 증가(또는 감소)됩니다.

시퀀스 번호는 테이블과 관계없이 생성되고 저장됩니다. 그러므로 동일한 시퀀스는 다중 테이블에 대해 사용될 수 있습니다.

CREATE SEQUENCE 문장

- 시퀀스 번호를 자동적으로 생성하기 위해 시퀀스를 정의합니다.

```
CREATE SEQUENCE sequence
    [INCREMENT BY n]
    [START WITH n]
    [{MAXVALUE n | NOMAXVALUE}]
    [{MINVALUE n | NOMINVALUE}]
    [{CYCLE | NOCYCLE}]
    [{CACHE n | NOCACHE}];
```

시퀀스 생성

CREATE SEQUENCE 문장을 사용하여 절차적인 번호를 자동적으로 생성합니다.

구문에서:

sequence	시퀀스의 이름입니다.
INCREMENT BY <i>n</i>	정수 값인 <i>n</i> 으로 시퀀스 번호 사이의 간격을 지정합니다. 이 절이 생략되면 시퀀스는 1 씩 증가합니다.
START WITH <i>n</i>	생성하기 위해 첫번째 시퀀스를 지정합니다. 이 절이 생략되면, 시퀀스는 1 로 시작됩니다.
MAXVALUE <i>n</i>	시퀀스를 생성할 수 있는 최대값을 지정합니다.
NOMAXVALUE	오름차순용 10^{27} 최대값과 내림차순용 -1 의 최소값을 지정합니다. 이것은 디폴트 옵션입니다.
MINVALUE <i>n</i>	최소 시퀀스 값을 지정합니다.
NOMINVALUE	오름차순용 1 과 내림차순용 $-(10^{26})$ 의 최소값을 지정합니다. 이것은 디폴트 옵션입니다.
CYCLE NOCYCLE	최대 또는 최소값에 도달한 후에 계속 값을 생성할 지의 여부를 지정합니다. NOCYCLE 이 디폴트 옵션입니다.
CACHE <i>n</i>	얼마나 많은 값이 메모리에 오라클 서버가 미리 할당하고

NOCACHE

유지하는가를 지정합니다. 디폴트로 오라클 서버는 20 을 캐쉬합니다.

주: NOMINVALUE, NOMAXVALUE 에서의 "오름차순"과 "내림차순"은 INCREMENT BY n 에서 n 이 양수(오름차순), 혹은 음수(내림차순)일 경우를 나타냅니다.

시퀀스 생성

- DEPT 테이블의 기본 키에 대해 사용되는 DEPT_DEPTNO 시퀀스를 생성합니다.
- CYCLE 옵션을 사용해서는 안됩니다.

```
SQL> CREATE SEQUENCE dept_deptno
2      INCREMENT BY 1
3      START WITH 91
4      MAXVALUE 100
5      NOCACHE
6      NOCYCLE;
Sequence created.
```

시퀀스작성(계속)

위 예에서 DEPT 테이블의 DEPTNO 에 대해 사용되는 DEPT_DEPTNO 라는 시퀀스를 생성합니다. 시퀀스는 91 에서 시작하며, 캐쉬를 허용치 않으며, 사이클을 허용치 않습니다.

시퀀스 사이클보다 더 빨리 오래된 행을 지우는 신뢰할 만한 메커니즘을 가지고 있지 않은 경우에, 기본 키 값을 생성하기 위해 시퀀스가 사용된다면 CYCLE 옵션을 사용해서는 안됩니다.

자세한 내용은 *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, "CREATE SEQUENCE."를 참조하십시오.

시퀀스 확인

- **USER_SEQUENCES** 데이터 사전 뷰에서 시퀀스 값을 검사합니다.

```
SQL> SELECT  sequence_name, min_value, max_value,
2            increment_by, last_number
3 FROM      user_sequences;
```

- **LAST_NUMBER** 열은 다음 이용 가능한 시퀀스 번호를 디스플레이 합니다.

시퀀스 확인

한번 시퀀스를 생성했으면, 그것은 데이터 사전에 문서화됩니다. 시퀀스가 데이터베이스 객체가 된 이후에 USER_OBJECTS 데이터 사전 테이블에서 식별할 수 있습니다. 또한 데이터 사전의 USER_SEQUENCES 테이블을 검색함으로써 시퀀스의 설정을 확인할 수 있습니다.

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
CUSTID	1	1.000E+27	1	109
DEPT_DEPTNO	1	100	1	91
ORDID	1	1.000E+27	1	622
PRODID	1	1.000E+27	1	200381

NEXTVAL과 CURRVAL 의사열

- **NEXTVAL**은 다음 사용 가능한 시퀀스 값을 리턴합니다.
시퀀스가 참조될 때마다, 다른 사용자에게 조차도 유일 값을 반환합니다.
- **CURRVAL**은 현재 시퀀스 값을 리턴합니다.
CURRVAL이 참조되기 전에 **NEXTVAL**이 먼저 이용되어야 합니다.

시퀀스 사용

일단 시퀀스를 작성했다면, 테이블에 사용할 절차적인 번호를 생성하기 위해 시퀀스를 사용할 수 있습니다. NEXTVAL 과 CURRVAL 의사열을 사용하여 시퀀스 값을 참조하십시오.

NEXTVAL 과 CURRVAL 의사열

NEXTVAL pseudocolumn 은 지정된 시퀀스에서 성공적으로 시퀀스 번호를 빼내기 위해 사용됩니다. 시퀀스 명으로써 NEXTVAL 을 참조합니다. sequence.NEXTVAL 을 참조할 때는 새 시퀀스 번호가 생성되고 현재 시퀀스 번호는 CURRVAL 에 위치하게 됩니다.

CURRVAL pseudocolumn 은 막 생성된 현재 사용자의 시퀀스 번호를 참조하기 위해 사용됩니다. NEXTVAL 은 CURRVAL 이 참조될 수 있기 전에 현재 사용자의 세션에서 시퀀스 번호를 생성하기 위해 사용되어야 합니다. sequence.CURRVAL 이 참조되면 사용자의 프로세스에 사용된 최후 시퀀스 값이 디스플레이 됩니다.

NEXTVAL 과 CURRVAL 사용 규칙

다음에서 NEXTVAL 과 CURRVAL 을 사용할 수 있습니다.

- 서브쿼리의 일부가 아닌 SELECT 문장의 SELECT 리스트
- INSERT 문장에서 서브쿼리 SELECT 리스트
- INSERT 문장의 VALUES 절

- UPDATE 문장의 SET 절

다음에서 NEXTVAL 과 CURRVAL 을 사용할 수 없습니다.

- 뷰의 SELECT 리스트
- DISTINCT 키워드를 이용한 SELECT 문장
- GROUP BY, HAVING, 또는 ORDER BY를 이용한 SELECT 문장
- SELECT, DELETE, 또는 UPDATE 문장에서의 서브쿼리
- CREATE TABLE 또는 ALTER TABLE 명령문의 DEFAULT 표현식

자세한 내용을 보시려면, *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, "Pseudocolumns" 섹션과 "CREATE SEQUENCE."를 참조하십시오.

시퀀스 사용

- 샌디에고에 “**MARKETING**”이라는 새로운 부서를 삽입합니다.

```
SQL> INSERT INTO dept(deptno, dname, loc)
2 VALUES (dept deptno.NEXTVAL,
3 'MARKETING', 'SAN DIEGO');
1 row created.
```

- **DEPT_DEPTNO** 시퀀스에 대한 현재 값을 봅니다.

```
SQL> SELECT dept deptno.CURRVAL
2 FROM dual;
```

시퀀스 사용

슬라이드 예에서 DEPT 테이블에 새 부서를 삽입합니다. 새 부서번호를 생성하기 위해 DEPT_DEPTNO 시퀀스를 사용합니다.

```
SQL> SELECT dept_deptno.CURRVAL
2 FROM dual;
```

CURRVAL

지금 새부서의 스태프가 되는 종업원을 고용하기를 원한다고 가정하십시오. 모든 새 종업원에 대해 반복적으로 실행될 수 있는 INSERT 문장은 아래 코드를 포함할 수 있습니다.

```
SQL> INSERT INTO emp ...
      2 VALUES (emp_empno.NEXTVAL, dept_deptno.CURRVAL, ...
```

주: 위의 예는 시퀀스 EMP_EMPNO 가 새 종업원 번호 생성을 이미 완료했음을 가정합니다.

시퀀스 사용

- 메모리에서 시퀀스 값을 캐쉬하면 이 값에 대해 더 빠른 액세스를 할 수 있습니다.
- 시퀀스 값에서 간격(gap)은 아래 상황에서 발생할 수 있습니다.
 - rollback 발생
 - system 실패(crash)
 - 시퀀스가 다른 테이블에서 사용될 때
- **USER_SEQUENCES** 테이블을 질의하여 **NOCACHE**로 생성된 때에 한해서 다음 사용 가능한 시퀀스를 봅니다.

시퀀스 값 캐쉬

시퀀스 값에 대해 보다 빠른 액세스를 허용하기 위해 메모리에 시퀀스를 캐쉬합니다. 캐쉬는 시퀀스를 처음에 참조할 때 형성됩니다. 다음 시퀀스 값에 대한 각 요구는 캐쉬된 시퀀스에서 읽어들이입니다. 마지막 시퀀스가 사용된 후에 시퀀스에 요구하면 캐쉬된 시퀀스를 메모리에 갖다 놓습니다.

시퀀스에서 간격의 경계

시퀀스 생성자는 간격(gap)없이 절차적인 번호를 발생하는데, 이 작업은 커밋 또는 롤백과

관계없이 발생합니다. 그러므로, 시퀀스를 포함한 문장을 롤백한다면, 번호는 손실됩니다.
시퀀스에서 간격(gap)의 또다른 원인이 될 수 있는 경우는 시스템 실패(crash)입니다.
메모리에서 시퀀스 값을 캐쉬한다면, 시스템 실패(crash)일 때 이 값들은 손실됩니다.

시퀀스가 직접적으로 테이블에 고정되어 있지 않기 때문에 동일 시퀀스가 다중 테이블에서
사용되어 질 수 있습니다. 이 상황이 발생한다면, 각 테이블은 순차적인 번호에 간격이
생길 수 있습니다.

증가 없이 다음 사용 가능 시퀀스 값 뷰

USER_SEQUENCES 테이블을 질의하여 NOCACHE 로 시퀀스가 생성된 경우에만, 증가
없이 다음 사용 가능 값을 볼 수 있습니다.

시퀀스 수정

- 증가값, 최대값, 최소값, 사이클 옵션 또는 캐쉬 옵션을 변경합니다.

```
SQL> ALTER SEQUENCE dept deptno  
2      INCREMENT BY 1  
3      MAXVALUE 999999  
4      NOCACHE  
5      NOCYCLE;  
Sequence altered.
```

시퀀스 수정

시퀀스에 대해 MAXVALUE 한계에 도달한다면, 더 이상 시퀀스 값을 할당 받을 수 없을
것이고, MAXVALUE 를 초과하는 시퀀스를 나타내는 에러를 받을 것입니다. 시퀀스를 계속
사용하기 위해, ALTER SEQUENCE 문장을 사용하여 수정할 수 있습니다.

구문

```
ALTER SEQUENCE sequence
```



```
[ INCREMENT BY n]
[ {MAXVALUE n | NOMAXVALUE} ]
[ {MINVALUE n | NOMINVALUE} ]
[ {CYCLE | NOCYCLE} ]
[ {CACHE n | NOCACHE} ] ;
```

여기서: sequence 시퀀스 생성자의 이름입니다.

보다 자세한 내용은 *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “ALTER SEQUENCE.”를 참조하십시오.

시퀀스 제거

- **DROP SEQUENCE** 문장을 사용하여 데이터 사전에서 시퀀스를 제거합니다.
- 한번 제거되었다면 시퀀스는 더 이상 참조될 수 없습니다.

```
SQL> DROP SEQUENCE dept_deptno;
Sequence dropped.
```

시퀀스 제거

데이터 사전에서 시퀀스를 제거하기 위해, DROP SEQUENCE 문장을 사용합니다. 시퀀스를 제거하기 위해 시퀀스의 소유자이거나 DROP ANY SEQUENCE 권한을 가져야 합니다.

구문

```
DROP SEQUENCE sequence;
```

여기서: sequence 시퀀스 생성자의 이름입니다.

자세한 내용은 *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “DROP SEQUENCE.”를 참조하십시오.

시퀀스 수정 지침

- 시퀀스에 대한 **ALTER** 권한을 가지거나 소유자여야 합니다.
- 이후의 시퀀스 번호만 영향을 받습니다.
- 시퀀스는 다른 번호에서 시퀀스를 다시 시작하기 위해서는 제거하고 다시 생성되어야 합니다.
- 유효한 검사를 수행합니다.

지침(음성 설명은 없습니다)

- 시퀀스를 수정하기 위해서는 시퀀스에 대한 ALTER 권한을 가지거나 소유자이어야 합니다.
- 이후 시퀀스 번호만이 ALTER SEQUENCE 문장에 의해 영향을 받습니다.
- START WITH 옵션은 ALTER SEQUENCE를 사용하여 변경될 수 없습니다. 시퀀스는 다른 번호에서 시퀀스를 다시 시작하기 위해서는 제거하고 다시 생성되어야 합니다.
- 유효성 검사를 수행합니다. 예를 들면, 새 MAXVALUE는 현재 시퀀스 번호보다 작게 될 수 없습니다.

```
SQL> ALTER SEQUENCE dept_deptno
```

```
2      INCREMENT BY 1
```

```
3      MAXVALUE 90
```

```
4      NOCACHE
```

```
5      NOCYCLE;
```

```
ALTER SEQUENCE dept_deptno
```

```
*
```

```
ERROR at line 1:
```

```
ORA-04009: MAXVALUE cannot be made to be less than the current value
```

제 14 장 인덱스

목적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 일부 데이터베이스 객체와 그것의 사용을 기술합니다.
- 인덱스를 생성, 유지합니다.

과정목적

본 과정에서는 일반적으로 사용되는 다른 데이터베이스 객체 중 일부를 생성하고 유지하는 방법을 배우게 됩니다. 이 객체는 시퀀스 , 인덱스, 동의어를 포함합니다.

데이터베이스 객체

객체	설명
Table	행과 열로 구성된 기본적인 저장 매체의 단위
View	하나 이상의 테이블로부터 데이터의 부분집합을 논리적으로 표현
Sequence	기본 키 값을 발생
Index	어떤 질의의 성능을 향상
Synonym	객체에 대체 이름을 부여

데이터베이스 객체

많은 어플리케이션들은 유일한 숫자 값을 기본 키 값으로 사용합니다. 이 요구사항을 처리하기 위해 코드를 어플리케이션에서 구축하거나 유일한 번호를 생성하기 위해 시퀀스를 사용할 수 있습니다.

일부 질의의 성능을 향상시키길 원한다면, 인덱스의 생성을 고려해야 합니다. 또한 열 또는 열집합에서 유일성을 강화시키기 위해 인덱스를 사용할 수 있습니다. 동의어를 사용하여 객체를 위한 대체명을 제공할 수 있습니다.

인덱스란?

- 스키마 객체
- 포인터를 사용하여 행의 검색을 촉진하기 위해 오라클 서버가 사용합니다.
- 빠르게 데이터를 찾기 위해 빠른 경로 액세스 방법을 사용하여 디스크 I/O를 경감시킵니다.
- 인덱스하는 테이블에 대해 독립적입니다.
- 오라클 서버에 의해 자동적으로 사용되고 유지됩니다.

인덱스란?

오라클 서버 인덱스는 포인터를 사용하여 행의 검색을 촉진할 수 있는 스키마 객체입니다. 인덱스는 명시적으로 또는 자동적으로 생성될 수 있습니다. 열에서 인덱스를 가지고 있지 않다면, 이때는 전체 테이블 스캔(scan)이 발생할 것 입니다.

인덱스는 테이블의 행에 대해 직접적이고 빠른 액세스를 제공합니다. 이것의 목적은 빠르게 데이터를 찾기 위해 인덱스 된 경로를 사용하여 디스크 I/O의 필요성을 경감시키는 것 입니다. 인덱스는 오라클 서버에 의해 자동적으로 사용되고 유지됩니다.

한번 인덱스가 생성되면, 사용자에게 요구되는 직접적인 작업은 없습니다. 인덱스는 그들이 인덱스한 테이블에 대해 논리적으로 물리적으로 독립적입니다. 즉 인덱스는 어느 때에나 생성되거나 제거될 수 있고 기본 테이블 또는 다른 인덱스에 영향이 없음을 의미합니다.

주: 테이블을 제거했을 때, 해당 인덱스도 역시 제거됩니다.

자세한 내용은 *Oracle Server Concepts Manual, Release 7.3* 또는 *8.0*, “Schema Objects” section, “Indexes” topic 을 참조하십시오.

인덱스를 생성하는 방법은?

- 자동으로
 - 유일 인덱스는 테이블 정의의 **PRIMARY KEY** 또는 **UNIQUE key** 제약조건을 정의할 때 자동으로 생성됩니다.
- 수동으로
 - 사용자는 행에 대한 액세스 시간을 항상 시키기 위해 열에서 유일하지 않은 인덱스를 생성할 수 있습니다.

인덱스를 생성하는 방법은?

2 가지 타입의 인덱스가 생성될 수 있습니다. 한 타입은 유일한 인덱스입니다. 오라클 서버는 PRIMARY Key 또는 UNIQUE Key 제약조건을 가지기 위해 테이블에서 열을 정의할 때 자동으로 이 인덱스를 생성합니다.

사용자가 생성할 수 있는 다른 타입의 인덱스는 유일하지 않은 인덱스입니다. 예를 들면, 검색 속도를 향상시키기 위해 질의 안의 조인을 위한 **FOREIGN KEY** 열 인덱스를 생성할 수 있습니다.

인덱스 생성

- 하나 이상의 열의 인덱스를 작성합니다.

```
CREATE INDEX index  
ON table (column[, column]...);
```

- **EMP** 테이블의 **ENAME** 열의 질의 액세스 속도를 향상시킵니다.

```
SQL> CREATE INDEX    emp_ename_idx  
2  ON                emp (ename);  
Index created.
```

인덱스 생성

CREATE INDEX 문장을 생성함으로써 하나 이상의 열의 인덱스를 생성합니다.

구문에서:

index	인덱스의 이름입니다.
table	테이블의 이름입니다.
column	인덱스 되기 위한 테이블의 열 이름입니다.

자세한 내용을 보시려면

Oracle Server SQL Reference, Release 7.3 또는 8.0, “CREATE INDEX.”를 참조하십시오.

많은 것이 항상 더 좋은 것은 아니다.

테이블의 많은 인덱스가 질의의 스피드 향상을 꼭 의미하는 것은 아닙니다. 인덱스를 가지고 있는 테이블에 대한 각 DML 작업은 인덱스도 갱신되어야 함을 의미합니다. 많은 인덱스가 테이블과 관련되어 있으면, 오라클 서버는 DML 후에 모든 인덱스를 갱신시키기 위해 더 많은 노력이 필요하게 됩니다.

언제 인덱스를 생성하는가?

- 열은 WHERE 절 또는 조인 조건에서 자주 사용됩니다.

- 열은 광범위한 값을 포함합니다.
- 열은 많은 수의 null 값을 포함합니다.
- 둘 또는 이상의 열은 WHERE 절 또는 조인 조건에서 자주 함께 사용됩니다.
- 테이블은 대형이고 대부분의 질의들은 행의 2-4%보다 적게 읽어들이는 것으로 예상됩니다.

유일성을 강행하기를 원한다면, 테이블 정의에 유일한 제약조건을 정의해야 함을 명심하십시오. 이때 유일한 인덱스는 자동으로 생성됩니다.

인덱스 생성 지침

아래 경우라면 인덱스를 생성하지 마십시오.

- 테이블이 작다.
- 열이 질의의 조건으로 자주 사용되지 않는다.
- 대부분의 질의들은 행의 **2-4%**이상을 읽어들이는 것으로 예상된다.
- 테이블이 자주 갱신된다.

언제 인덱스를 생성해서는 안되는가

- 테이블이 작다.
- 열이 질의의 조건으로 자주 사용되지 않는다.
- 대부분의 질의들은 행의 2-4%이상을 읽어들이는 것으로 예상된다.
- 테이블은 자주 갱신됩니다. 테이블에 하나 이상 인덱스를 가지고 있다면 테이블을 액세스하는 DML 문장은 인덱스의 유지 때문에 상대적으로 더 많은 시간이 걸리게 됩니다.

인덱스 확인

- **USER_INDEXES** 데이터 사전 뷰는 인덱스의 이름과 그것의 유일성을 디스플레이합니다.
- **USER_IND_COLUMNS** 뷰는 인덱스명, 테이블명, 열명을 디스플레이합니다.

```
SQL> SELECT  ic.index_name, ic.column_name,
2           ix.uniqueness
3 FROM      user_indexes ix, user_ind_columns ic
4 WHERE     ic.index_name = ix.index_name
5 AND       ic.table_name = 'EMP';
```

인덱스 확인

USER_INDEXES 데이터 사전 뷰에서 인덱스의 존재를 확인합니다.

또한 USER_IND_COLUMNS 뷰를 질의함으로써 인덱스에 관계된 열을 확인합니다. 위의 예는 EMP 테이블에서 이전에 생성된 인덱스, 관련 이름, 유일성 모두를 디스플레이합니다

INDEX_NAME	COLUMN_NAME	COL_POS	UNIQUENES
-----	-----	-----	-----
EMP_EMPNO_PK	EMPNO	1	UNIQUE
EMP_ENAME_IDX	ENAME	1	NONUNIQUE

주: 위의 출력은 포맷되었습니다.

인덱스 제거

- 데이터 사전에서 인덱스를 제거합니다.

```
SQL> DROP INDEX index;
```

- 데이터 사전에서 **EMP_ENAME_IDX** 인덱스를 제거합니다.

```
SQL> DROP INDEX emp_ename_idx;  
Index dropped.
```

- 인덱스를 제거하기 위해, 인덱스의 소유자이거나 **DROP ANY INDEX** 권한을 가지고 있어야 합니다.

인덱스 제거

인덱스를 수정할 수 없습니다. 인덱스를 변경하기 위해서는, 그것을 제거하고 다시 작성해야 합니다. DROP INDEX 문장을 생성하여 데이터 사전에서 인덱스 정의를 제거합니다. 인덱스를 제거하기 위해서는 인덱스의 소유자이거나 DROP ANY INDEX 권한을 가지고 있어야 합니다.

구문에서

index

인덱스의 이름입니다.

인덱스 제거

- 데이터 사전에서 인덱스를 제거합니다.

```
SQL> DROP INDEX index;
```

- 데이터 사전에서 **EMP_ENAME_IDX** 인덱스를 제거합니다.

```
SQL> DROP INDEX emp_ename_idx;  
Index dropped.
```

- 인덱스를 제거하기 위해, 인덱스의 소유자이거나 **DROP ANY INDEX** 권한을 가지고 있어야 합니다.

인덱스 제거

인덱스를 수정할 수 없습니다. 인덱스를 변경하기 위해서는, 그것을 제거하고 다시 작성해야 합니다. DROP INDEX 문장을 생성하여 데이터 사전에서 인덱스 정의를 제거합니다. 인덱스를 제거하기 위해서는 인덱스의 소유자이거나 DROP ANY INDEX 권한을 가지고 있어야 합니다.

구문에서

index

인덱스의 이름입니다.

제 15 장 동의어

목적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 일부 데이터베이스 객체와 그것의 사용을 기술합니다.
- 개별(**private**)과 공용(**public**)의 동의어를 생성합니다.

과정목적

본 과정에서는 일반적으로 사용되는 다른 데이터베이스 객체 중 일부를 생성하고 유지하는 법을 배우게 됩니다. 이 객체는 시퀀스 , 인덱스, 동의어를 포함합니다.

동의어

- 동의어 (객체의 다른 이름)를 생성하여 객체에 대한 액세스를 단순화합니다.
- 다른 사용자가 소유한 테이블을 참조합니다.
- 객체이름의 길이를 단축합니다.

```
CREATE [PUBLIC] SYNONYM synonym  
FOR object;
```

객체에 대한 동의어 작성

다른 사용자가 소유한 테이블을 참조하기 위해서는, 테이블을 생성한 사용자 이름 뒤에 점을 찍고 테이블 이름을 써야 합니다. 그러나, 동의어를 생성하면 객체의 이름 앞에 스키마 이름을 명시할 필요가 없으며, 또한 테이블, 뷰, 시퀀스, 프로시저 등, 다른 객체에 대한 또 다른 이름을 제공하게 됩니다. 이 방법은 뷰처럼 긴 이름을 가진 객체한테 특히 유용하게 사용할 수 있습니다.

구문에서

PUBLIC	모든 사용자에게 대해 액세스 가능한 동의어를 생성합니다.
synonym	생성되어야 할 동의어의 이름입니다.
object	생성된 동의어에 대한 객체를 식별합니다.

지침

- 객체는, 패키지에 포함될 수 없습니다.
- 개별(private) 동의어 이름은 동일사용자가 소유한 모든 다른 객체의 이름과 달라야 합니다.

자세한 내용을 보시려면, *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “**CREATE SYNONYM.**”를 참조하십시오

객체에 대한 동의어 생성 (계속)

위의 예는 더 빠른 조회를 위해 DEPT_SUM_VU 뷰에 대한 동의어를 생성합니다. DBA는 모든 사용자에게 대해 액세스하기 쉬운 공용(public) 동의어를 생성할 수 있습니다. 아래 예는 앨리스의 DEPT 테이블에 대해 DEPT 라는 공용(public) 동의어를 생성한 것입니다.

```
SQL> CREATE PUBLIC SYNONYM dept
2 FOR          alice.dept;
Synonym created.
```

동의어 제거

동의어를 제거하기 위해, DROP SYNONYM 문장을 사용합니다. DBA 만 공용(public) 동의어를 제거할 수 있습니다.

```
SQL> DROP SYNONYM dept;  
Synonym dropped.
```

자세한 내용을 보시려면, *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “DROP SYNONYM.”를 참조하십시오.

동의어 생성과 제거

- DEPT_SUM_VU 뷰에 대해 단축명을 생성합니다.

```
SQL> CREATE SYNONYM d_sum  
2 FOR dept_sum_vu;  
Synonym Created.
```

- 동의어를 제거합니다.

```
SQL> DROP SYNONYM d_sum;  
Synonym dropped.
```

객체에 대한 동의어 생성 (계속)

위의 예는 더 빠른 조회를 위해 DEPT_SUM_VU 뷰에 대한 동의어를 생성합니다. DBA 는 모든 사용자에게 대해 액세스하기 쉬운 공용(public) 동의어를 생성할 수 있습니다. 아래 예는 앨리스의 DEPT 테이블에 대해 DEPT 라는 공용(public) 동의어를 생성한 것입니다.

```
SQL> CREATE PUBLIC SYNONYM dept  
2 FOR alice.dept;  
Synonym created.
```

동의어 제거

동의어를 제거하기 위해, DROP SYNONYM 문장을 사용합니다. DBA 만 공용(public) 동의어를 제거할 수 있습니다.

```
SQL> DROP SYNONYM dept;  
Synonym dropped.
```

자세한 내용을 보시려면, *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “**DROP SYNONYM.**”를 참조하십시오.

제 16 장 사용자 접근 제어

목적

본 과정을 마치면, 다음을 할 수 있어야 합니다.

- 사용자 생성
- 보안 모델에 대한 쉬운 셋업과 롤 생성
- 객체 권한을 **GRANT, REVOKE**

과정 목적

본 과정에서는, 특정 객체에 대한 데이터베이스 액세스를 제어하는 방법과 각각 다른 수준의 액세스 권한을 갖는 새로운 사용자를 추가하는 법을 배웁니다.

권한

권한은 특정 SQL 문장을 실행하기 위한 권한입니다. 데이터베이스 관리자는 데이터베이스와 그 객체에 대한 액세스를 사용자에게 부여하는 능력을 가진 상급 사용자입니다. 사용자는 데이터베이스에 액세스하기 위해 `system privilege` 가 필요하고 데이터베이스에서 객체의 내용을 조작하기 위해 `object privilege` 가 필요합니다. 사용자는 관련 권한들의 이름있는 그룹인 `role` 이나 다른 사용자에게 추가적으로 권한을 부여하기 위해 권한을 가질 수 있습니다.

스키마

schema 는 테이블 , 뷰, 시퀀스 같은 객체의 모음입니다. 스키마는 데이터베이스 사용자에게 의해 소유되고 사용자와 동일이름을 가집니다.

자세한 내용을 보시려면, *Oracle Server Application Developer’s Guide, Release 7.3* 또는 *8.0*, “Establishing a Security Policy” section, and *Oracle Server Concepts Manual, Release 7.3* 또는 *8.0*, “Database Security” topic.을 참조하십시오

시스템 권한

- 80개 이상의 권한이 있습니다.
- DBA 는 상급의 시스템 권한을 가집니다.
 - 새로운 사용자 생성
 - 사용자 제거
 - 테이블 제거
 - 테이블 백업

시스템 권한

사용자와 롤에 대해 80 개 이상의 시스템 권한이 사용 가능합니다. 시스템 권한은 대개 데이터베이스 관리자에 의해 제공됩니다.

주된 DBA 권한

시스템 권한	승인 작업
CREATE USER	다른 오라클 사용자 생성을 피수여자에게 허용(DBA 역할을 위해 필요한 권한)
DROP USER	다른 사용자 제거
DROP ANY TABLE	임의의 스키마에서 테이블 제거
BACKUP ANY TABLE	export 유틸리티로 임의의 스키마에서 임의의 테이블 백업

사용자 생성

- DBA는 **CREATE USER** 문장을 사용하여 사용자를 생성합니다.

```
CREATE USER    user  
IDENTIFIED BY password;
```

```
SQL> CREATE USER scott  
2 IDENTIFIED BY tiger;  
User created.
```

사용자 생성

DBA는 CREATE USER 문장을 실행하여 사용자를 생성합니다. 사용자는 이때에는 어떤 권한도 가지지 않습니다. DBA는 이때 그 사용자에게 여러 권한을 부여합니다. 이 권한은 데이터베이스 수준에서 사용자가 할 수 있는 것이 무엇인가를 결정합니다. 슬라이드는 사용자 생성을 위해 요약된 구문을 제공합니다.

구문에서

user	생성되어야 하는 사용자명입니다.
password	이 비밀번호로 로그인해야 함을 지정합니다.

자세한 내용을 보시려면, *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “GRANT”(System Privileges and Roles) and “CREATE USER.”를 참조하십시오.

사용자 시스템 권한

- 일단 사용자가 생성되었다면, **DBA**는 사용자에게 대한 특정 시스템 권한을 부여할 수 있습니다.

```
GRANT privilege [, privilege...]  
TO user [, user...];
```

- 어플리케이션 개발자는 다음 시스템 권한을 가질 수 있습니다.
 - **CREATE SESSION**
 - **CREATE TABLE**
 - **CREATE SEQUENCE**
 - **CREATE VIEW**
 - **CREATE PROCEDURE**

주된 사용자 권한

DBA 가 사용자를 생성했으므로, DBA 는 그 사용자에게 대해 권한을 할당할 수 있습니다.

시스템 권한	승인 작업
CREATE SESSION	데이터베이스로 접속
CREATE TABLE	사용자의 스키마에서 테이블 생성
CREATE SEQUENCE	사용자의 스키마에서 시퀀스 생성
CREATE VIEW	사용자의 스키마에서 뷰 생성
CREATE PROCEDURE	사용자의 스키마에서 내장 된 프로시저, 함수, 패키지 생성

구문에서

privilege

허가되는 시스템 권한입니다.

user

사용자명입니다.

시스템 권한 부여

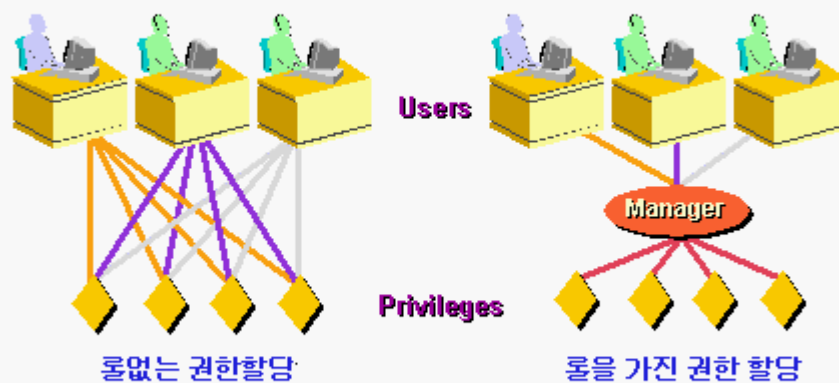
- **DBA**는 사용자에게 특정 시스템 권한을 부여할 수 있습니다.

```
SQL> GRANT create table, create sequence, create view  
2 TO scott;  
Grant succeeded.
```

시스템 권한 부여

DBA 는 사용자에게 시스템 권한을 부여하기 위해 GRANT 문장을 사용합니다. 일단 사용자가 권한을 부여 받았으면, 사용자는 즉시 그 권한을 사용할 수 있습니다. 위의 예에서 사용자 Scott 은 테이블, 시퀀스, 뷰를 생성하기 위해 권한을 할당 받았습니다.

롤(role)이란 ?



롤(role)이란 ?

롤은 사용자에게 부여될 수 있는 관련 권한의 그룹입니다. 롤은 권한을 부여하고 철회하는 것을 수행하고 유지하기 쉽게 합니다. 한 사용자가 여러 롤을 액세스 할 수 있고 다른 여러 사용자에게 동일한 롤을 지정할 수 있습니다.

롤 생성과 할당

우선 DBA 는 롤을 생성해야 합니다. 이때 DBA 는 롤에게 권한을 할당하고 롤을 사용자들에게 할당할 수 있습니다.

구문

```
CREATE ROLE role;
```

여기서: role 생성되어야 할 롤의 이름입니다.

롤이 생성되었으므로, DBA 는 롤에 대한 권한 뿐만 아니라 롤을 사용자에게 할당하기 위한 GRANT 문장을 사용할 수 있습니다.

롤의 생성과 권한 부여

```
SQL> CREATE ROLE manager;  
Role created.
```

```
SQL> GRANT create table, create view  
2 to manager;  
Grant succeeded.
```

```
SQL> GRANT manager to BLAKE, CLARK;  
Grant succeeded.
```

롤 생성

위의 예는 롤(관리자)을 생성하고 그 롤이 테이블과 뷰를 생성하는 권한을 갖도록 해

줍니다. 그리고 관리자 롤을 Blake 와 Clark 에게 부여합니다. 이제 Blake 와 Clark 은 테이블과 뷰를 생성할 수 있습니다.

비밀번호 변경

- 사용자 계정이 생성되었을 때, 각 사용자는 **DBA**가 초기화한 비밀번호를 갖게 됩니다.
- 사용자는 **ALTER USER** 문장을 사용하여 자신의 비밀번호를 수정할 수 있습니다.

```
SQL> ALTER USER scott  
2 IDENTIFIED BY lion;  
User altered.
```

비밀번호 변경

모든 사용자는 사용자가 생성될 때 DBA 에 의해 초기화된 비밀번호를 가집니다. ALTER USER 문장을 사용하여 비밀번호를 변경할 수 있습니다.

구문

ALTER USER user IDENTIFIED BY password;

여기서: user 사용자명입니다.
 password 새 비밀번호를 지정합니다.

이 문장이 비밀번호 변경을 위해 사용될 수 있는데, 많은 다른 옵션이 있습니다. 임의의 다른 옵션 변경을 위해 ALTER USER 권한을 가져야 합니다.

자세한 내용은, *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “ALTER USER.”를 참조하십시오.

객체 권한

Object Privilege	Table	View	Sequence	Procedure
ALTER	✓		✓	
DELETE	✓	✓		
EXECUTE				✓
INDEX	✓			
INSERT	✓	✓		
REFERENCES	✓			
SELECT	✓	✓	✓	
UPDATE	✓	✓		

객체 권한

object privilege 는 특정 테이블, 뷰, 시퀀스, 프로시저에서 특정 작업을 수행하기 위한 권한 또는 권리입니다. 각각의 객체는 부여 가능한 특정 권한 집합을 가지고 있습니다. 상기 테이블은 다양한 객체에 대한 권한을 리스트합니다. 시퀀스에 대해 SELECT 와 ALTER 권한만이 적용됨을 명심하십시오. UPDATE, REFERENCES, 그리고 INSERT 는 갱신 가능한 열을 따로 지정함으로써 제한될 수 있습니다. SELECT 는 몇몇 열으로써 뷰를 생성하고 뷰에 대한 SELECT 권한을 부여함으로써 제한될 수 있습니다. 동의어에 대한 부여는 동의어에 의해 참조되는 기본 테이블에 대한 부여로 전환됩니다.

객체 권한

- 객체 권한은 객체마다 다양합니다.
- 소유자는 객체에 대한 모든 권한을 가집니다.
- 소유자는 사용자 객체에 대한 특정 권한을 제공할 수 있습니다.

```
GRANT      object_priv [(columns)]  
ON         object  
TO         {user|role|PUBLIC}  
[WITH GRANT OPTION];
```

객체 권한 부여

스키마 객체의 타입에 따라 이용 가능한 객체 권한이 다릅니다. 사용자는 자동적으로 자신의 스키마에 포함된 스키마 객체에 대한 모든 객체 권한을 가집니다. 사용자는 임의의 다른 사용자 또는 롤에 대해 자신이 소유한 모든 스키마 객체에 대한 가능한 모든 객체 권한을 부여할 수 있습니다. 부여가 GRANT OPTION 을 포함한다면, 부여자는 또 다른 사용자에게 객체 권한을 연이어 더 부여할 수 있으며 그렇지 않으면, 부여자는 권한을 사용할 수 있지만 다른 사용자에게 대해서는 부여할 수 없습니다.

구문에서:

object_priv	부여받기 위한 객체 권한입니다.
ALL	모든 객체 권한.
columns	권한이 부여되는 테이블 또는 뷰로부터 열을 지정합니다.
ON object	권한이 부여되는 객체입니다.
TO	권한이 부여될 사용자를 식별합니다.
PUBLIC	모든 사용자에게 대해 객체 권한을 부여합니다.
WITH GRANT OPTION	부여자가 다른 사용자와 롤에 대한 객체 권한을 부여할 수 있도록 허용합니다.

객체 권한 부여

- **EMP** 테이블에 대한 질의 권한 부여

```
SQL> GRANT  select
  2  ON      emp
  3  TO      sue, rich;
Grant succeeded.
```

- 사용자와 롤에게 지정 열을 갱신하기 위한 권한 부여

```
SQL> GRANT  update (dname, loc)
  2  ON      dept
  3  TO      scott, manager;
Grant succeeded.
```

지침

- 객체에 대한 권한을 부여하기 위해, 객체는 자신의 스키마에 존재해야 하거나 WITH GRANT OPTION 으로 객체 권한을 부여 받아야 합니다.
- 객체 소유자는 데이터베이스의 어떤 다른 사용자 또는 롤에게 객체에 대한 모든 가능한 객체 권한을 부여할 수 있습니다.
- 객체의 소유자는 자동적으로 그 객체에 대한 모든 객체 권한을 얻습니다.

위의 처음 예에서 EMP 테이블을 질의하기 위해 사용자 Sue 와 Rich 에게 권한을 부여합니다. 두 번째 예는 관리자 롤과 Scott 에게 DEPT 테이블의 지정 열에 대한 UPDATE 권한을 부여합니다.

주: DBA 는 일반적으로 시스템 권한을 할당합니다. 객체를 소유한 모든 사용자는 객체 권한 을 부여할 수 있습니다.

WITH GRANT OPTION과 PUBLIC 키워드 사용

- 권한을 전달하기 위한 사용자 권한 제공

```
SQL> GRANT    select, insert
2  ON        dept
3  TO        scott
4  WITH GRANT OPTION;
Grant succeeded.
```

- 모든 사용자가 Alice의 DEPT 테이블의 데이터를 질의하도록 허용

```
SQL> GRANT    select
2  ON        alice.dept
3  TO        PUBLIC;
Grant succeeded.
```

WITH GRANT OPTION 키워드

WITH GRANT OPTION 으로 부여 받은 권한은 부여자에 의해 다른 사용자와 롤에게 다시 부여될 수 있습니다. WITH GRANT OPTION 으로 테이블을 질의할 수 있고 테이블에 행을 추가할 수 있도록 해 줍니다.

Scott 은 이 권한을 다른 사용자에게 부여할 수 있습니다.

PUBLIC 키워드

테이블의 소유자는 PUBLIC 키워드를 사용하여 모든 사용자에게 액세스 권한을 부여할 수 있습니다. 위의 예는 Alice 의 DEPT 테이블에 대해 모든 사용자가 데이터를 질의할 수 있도록 해 줍니다.

부여된 권한 확인

데이터 사전 테이블	설명
ROLE_SYS_PRIVS	롤에게 부여된 시스템 권한
ROLE_TAB_PRIVS	롤에게 부여된 테이블 권한
USER_ROLE_PRIVS	사용자에 의해 액세스 가능한 롤
USER_TAB_PRIVS_MADE	해당 사용자 객체에 부여된 객체 권한
USER_TAB_PRIVS_RECD	사용자에게 부여된 객체 권한
USER_COL_PRIVS_MADE	해당 사용자 객체의 특정 열에 부여된 객체 권한
USER_COL_PRIVS_RECD	특정 열에 대해 사용자에게 부여된 객체 권한

부여된 권한 확인

권한이 없는 작업을 수행코자 한다면 - 예를 들면, DELETE 권한을 가지고 있지 않은 테이블에서 행을 삭제- 오라클 서버는 수행하는 작업을 허용하지 않습니다.

오라클 서버 오류 메시지 “테이블 또는 뷰는 존재하지 않는다”를 받는다면, 아래 둘 중 하나를 행한 것입니다.

- 존재하지 않는 테이블이나 뷰를 명시했거나,
- 해당 권한을 가지고 있지 않는 뷰 또는 테이블에 대한 작업을 수행하려고 한 것입니다.

가지고 있는 권한을 보기 위해 데이터 사전을 액세스 할 수 있습니다.

슬라이드의 테이블은 다양한 데이터 사전 테이블입니다.

객체 권한 철회 방법

- 다른 사용자에게 대해 부여된 권한을 철회하기 위해 **REVOKE** 문장을 사용합니다.
- **WITH GRANT OPTION**을 통해 다른 사용자에게 부여된 권한도 같이 철회될 것입니다.

```
REVOKE {privilege [, privilege...]|ALL}
ON      object
FROM    {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];
```

객체 권한 철회

REVOKE 문장은 다른 사용자에게 부여된 권한을 제거합니다. REVOKE 문장을 사용할 때 , 명시한 권한은 명시한 사용자에서와 이미 부여 받았던 다른 사용자에서 철회됩니다.

구문에서

CASCADE REFERENCES 권한을 사용하여 만들어진 객체에 대한 [참조 무결성](#) CONSTRAINTS 제약조건을 제거하기 위해 필요합니다.

자세한 내용은, *Oracle Server SQL Reference, Release 7.3* 또는 *8.0*, “REVOKE.”를 참조하십시오.

객체 권한 철회

- 사용자 **Alice**로서, **DEPT** 테이블에 대해 사용자 **Scott**에게 주어진 **SELECT**와 **INSERT** 권한을 철회합니다.

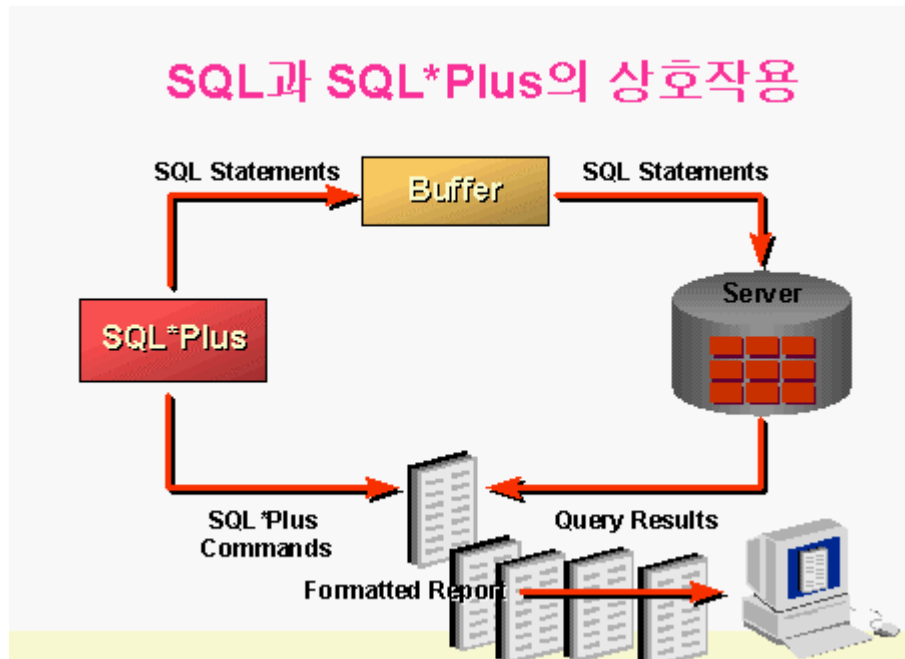
```
SQL> REVOKE select, insert
      2 ON dept
      3 FROM scott;
Revoke succeeded.
```

객체 권한 철회 (계속)

위의 예는 DEPT 테이블에 대해 사용자 Scott에게 주어진 SELECT와 INSERT 권한을 철회합니다.

주: 사용자가 WITH GRANT OPTION으로 권한을 부여 받았다면, 그 사용자는 WITH GRANT OPTION으로 권한을 부여해 줄 수 있고, 그래서 수여자 간의 긴 체인이 가능하지만, 원형 부여는 허용되지 않습니다. 소유자가 다른 사용자에게 권한을 부여한 사용자에서 권한을 철회한다면 REVOKE는 부여된 모든 권한을 연이어 철회합니다. 예를 들면 사용자 A가 WITH GRANT OPTION을 포함하여 사용자 B에 대해 테이블에 SELECT 권한을 부여한다면, 사용자 B는 사용자 C에게 SELECT 권한을 WITH GRANT OPTION으로 부여할 수 있고 사용자 C는 이때 사용자 D에 대해 SELECT 권한을 부여할 수 있습니다. 사용자 A가 사용자 B에게서 권한을 철회한다면, 이때 또한 사용자 C와 D에게서도 권한을 철회하게 됩니다.

제 17 장 SQL*Plus 명령



SQL 과 SQL*Plus

SQL 은 어떤 툴이나 어플리케이션으로부터 오라클 서버와 통신하기 위한 명령어 입니다. 오라클 SQL 은 많은 확장자를 포함합니다. SQL 문장을 입력할 때, SQL buffer 라는 메모리의 한 부분에 저장되며 새로운 문장을 입력할 때까지 유지됩니다.

SQL*Plus 는, SQL 문장의 실행을 위해 SQL 문장을 인식하고 오라클 서버에 SQL 문장을 보내는 오라클 툴이며, 고유의 명령어도 가집니다.

SQL 의 특징

- 프로그래밍 경험이 적거나 전혀 없는 사용자들도 사용할 수 있습니다.
- 비 절차적 언어입니다.
- 시스템을 생성하고 유지 관리하기 위해 요구되는 시간을 절약해 줍니다.
- 영어와 비슷한 언어입니다.

SQL*Plus 의 특징

- 파일로부터 SQL 입력을 받습니다.
- SQL 문장을 수정하기 위한 라인 편집을 제공합니다.
- 환경적인 설정을 제어합니다.
- 질의 결과를 기본적인 리포트 형태로 포맷됩니다.
- 로컬과 원격 데이터베이스를 액세스 합니다.

SQL 과 SQL*Plus (계속)

다음 테이블은 SQL 과 SQL*Plus 를 비교합니다:

SQL	SQL*Plus
데이터를 액세스 하기 위해 오라클 서버와 통신합니다	SQL 문장을 인식하고 그들을 서버로 전송합니다.
ANSI 표준 SQL 을 기본으로 합니다.	SQL 문장을 실행시키기 위한 오라클 소유의 인터페이스 입니다.
데이터베이스의 데이터와 테이블을 조작합니다.	데이터베이스의 값을 조작할 수 없습니다.
하나 이상의 라인이 SQL 버퍼에 입력 됩니다.	한번에 한 라인씩 입력합니다; SQL 버퍼에 저장되지 않습니다.
연속 문자가 없습니다.	명령어가 한 라인보다 길어지면 연속 문자인 대쉬(-)를 이용합니다.
단축될 수 없습니다.	단축될 수 있습니다.
명령을 즉시 실행하기 위해 종료 문자를 사용합니다.	종료 문자가 필요 없습니다. 명령어는 즉시 실행됩니다.
어떤 포매팅을 수행하기 위해 함수를 사용합니다.	데이터를 포맷하기 위해 명령어를 사용합니다.

SQL*Plus의 개요

- **SQL*Plus**로 로그인 합니다.
- 테이블 구조를 기술합니다.
- **SQL** 문장을 편집합니다.
- **SQL*Plus**로부터 **SQL**을 실행합니다.
- **SQL** 문장을 파일로 저장하고 **SQL** 문장을 파일에 추가합니다.
- 저장된 파일을 실행합니다.
- 편집을 하기 위해서 파일로부터 버퍼로 명령어를 로드합니다.

SQL*Plus

SQL*Plus 는 다음을 할 수 있는 환경입니다:

- 데이터베이스로부터 데이터를 검색, 수정, 추가 그리고 삭제하기 위해 SQL 문장을 실행합니다.
- 질의 결과를 포맷, 계산 수행, 저장 그리고 리포트 형식으로 출력합니다.
- 앞으로도 계속 사용할 수 있도록 SQL 문장을 스크립트 파일로 만듭니다.

SQL*Plus 명령어는 다음의 주요 범주들로 나눌 수 있습니다:

범주	목적
Environment	한 세션에 대한 SQL 문장의 일반적인 행위에 영향을 미칩니다.
Format	질의 결과를 포맷합니다.
File manipulation	스크립트 파일을 저장, 로드 그리고 실행합니다
Execution	SQL 버퍼로부터 Oracle8 서버로 SQL 문장을 전송합니다.
Edit	버퍼의 SQL 문장을 수정합니다.
Interaction	변수를 생성하여 SQL 문장으로 전달하고 변수 값을 출력하고 그리고 스크린으로 메시지를 출력하게 합니다.

Miscellaneous	데이터베이스에 연결하기 위한 다양한 명령어를 가지며, SQL*Plus 환경을 조작하고 열 정의를 디스플레이 합니다.
---------------	--

테이블 구조 디스플레이

```
SQL> DESCRIBE dept
```

Name	Null?	Type
DEPTNO	NOT NULL	NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

테이블 구조 디스플레이 (계속)

위의 예는 DEPT 테이블의 구조에 관한 정보를 디스플레이 합니다.

결과에서:

Null? 열이 데이터를 포함해야 하는지의 여부를 가리킵니다; NOT NULL 은 열이 데이터를 포함해야 함을 의미합니다.

Type 열의 데이터 형을 디스플레이 합니다.

데이터 형은 다음과 같습니다:

데이터 형	설명
NUMBER(p,s)	Number 값은 십진수 최대 자리 수 p, 소수점 우측에 나타날 수 s 를 가지고 있습니다.
VARCHAR2(s)	최대 크기 s 의 변동 길이 문자 값
DATE	January 1, 4712 B.C. 와 December 31, 9999 A.D. 사이의 날짜와

	시간 값
CHAR(s)	크기 s 의 고정 길이 문자 값

SQL*Plus 편집 명령어

- **A[PPEND] *text***
- **C[HANGE] / *old* / *new***
- **C[HANGE] / *text* /**
- **CL[EAR] BUFF[ER]**
- **DEL**
- **DEL *n***
- **DEL *m n***

SQL*Plus 편집 명령어

SQL*Plus 명령어는 한 번에 한 라인이 입력되며 SQL 버퍼에 저장되지 않습니다.

명령어	설명
A[PPEND] <i>text</i>	텍스트를 현재 라인의 끝에 추가합니다.
C[HANGE] / <i>old</i> / <i>new</i>	현재 라인의 <i>old</i> 텍스트를 <i>new</i> 텍스트로 변경합니다
C[HANGE] / <i>text</i> /	현재 라인의 <i>text</i> 를 삭제합니다.
CL[EAR] BUFF[ER]	SQL 버퍼의 모든 라인을 삭제합니다.
DEL	현재 라인을 삭제합니다.

지침서

- 명령어를 완료하기 전에 리턴 키를 눌렀을 경우, SQL*Plus 는 라인 번호를 나타냅니다.
- 종료 문자 (세미콜론이나 슬래시) 중 하나를 입력하거나 또는 리턴 키를 두 번 눌러서 SQL 버퍼를 종료합니다. 그러면 SQL 프롬프트가 나타납니다.

SQL*Plus 편집 명령어

- **I[INPUT]**
- **I[INPUT] text**
- **L[IST]**
- **L[IST] n**
- **L[IST] m n**
- **R[UN]**
- **n**
- **n text**
- **0 text**

SQL*Plus 편집 명령어 (계속)

명령어	설명
I[NPU T]	무한대의 라인을 삽입합니다.
I[INPUT] text	text 로 이루어진 한 라인을 삽입합니다.
L[IST]	SQL 버퍼의 모든 라인을 나열합니다.
L[IST] n	한 라인 (n 으로 명시된)을 나열합니다.
L[IST] m n	일정 범위(m 에서 n 까지)의 라인을 나열합니다.
R[UN]	버퍼의 SQL 문장을 디스플레이 하고 실행합니다.
n	현재 라인으로 만들기 위해서 라인을 명시합니다.
n text	라인 n 을 text 로 대체합니다.
0 text	첫째 라인 앞에 한 라인을 삽입합니다.

SQL 프롬프트마다 오직 하나의 SQL*Plus 명령어만을 입력할 수 있습니다. SQL*Plus 명령어는 버퍼에 저장되지 않습니다. SQL*Plus 명령어를 다음 라인에 계속 하려면 현재 라인의 끝에 하이픈(-)을 추가합니다.

치환 변수

- 값을 임시로 저장하기 위해서 **SQL*Plus** 치환 변수를 사용합니다.
 - **Single ampersand (&)**
 - **Double ampersand (&&)**
 - **DEFINE**과 **ACCEPT** 명령어
- **SQL** 문장간에 변수 값을 전달합니다.
- 머리말과 꼬리말을 동적으로 변경합니다.

치환 변수

SQL*Plus 에서 값을 임시로 저장하기 위해서 & 치환 변수를 사용할 수 있습니다.
ACCEPT 나 DEFINE 명령을 사용해서 SQL*Plus 에 변수를 미리 정의할 수 있습니다.
ACCEPT 는 사용자 입력 라인을 읽고 그것을 변수에 저장합니다.

제한된 범위의 데이터의 예

- 현재 분기나 명시된 날짜 범위에만 해당되는 리포트.
- 특정 부서 내의 사람만을 디스플레이합니다.

기타 상호작용의 영향

상호작용의 영향은 WHERE 절에 대한 사용자 상호작용에만 국한되지 않습니다. 똑같은 원칙 을 다른 목표를 실행하는데도 사용할 수 있습니다. 예를 들면:

- 머리말과 꼬리말을 동적으로 변경합니다.
- 사용자가 아닌 파일로부터 입력 값을 얻습니다.
- 하나의 SQL 문장으로부터 다른 SQL 문장으로 값을 전달합니다.

SQL*Plus 는 사용자 입력에 대한 타당성 검사(데이터형 제외)를 하지 않습니다. 사용자에게 대해서 만드는 프롬프트를 단순하고 모호하지 않게 하십시오.

& 치환 변수 사용

사용자에게 값을 프롬프트하기 위해서 &가 앞에 붙는 변수를 사용합니다.

```
SQL> SELECT empno, ename, sal, deptno  
2 FROM emp  
3 WHERE empno = &employee_num;
```

Enter value for employee_num: 7369

EMPNO	ENAME	SAL	DEPTNO
7369	SMITH	800	20

단일 앰퍼샌드(&)와 치환 변수

리포트를 실행할 때 사용자는 종종 리턴되는 데이터를 동적으로 제한하기를 원합니다. SQL*Plus 는 사용자 변수로써 이러한 융통성을 제공합니다. SQL 문장에서 각각의 변수를 인식 하기 위해서 &를 사용합니다. 각각의 변수에 값을 정의할 필요는 없습니다.

표기	설명
&user_variable	SQL 문장에 있는 변수를 지칭합니다. 변수가 존재하지 않으면 SQL*Plus 는 값(SQL*Plus 는 일단 사용된 새로운 변수는 버립니다.)에 대해서 사용자에게 프롬프트 합니다.

위의 예는 실행시에 종업원의 번호를 사용자에게 프롬프트하기 위한 SQL 문장을 생성합니다. 그리고 해당 종업원에 대해서 종업원 번호, 이름, 급여 그리고 부서 번호를 디스플레이합니다.

변수가 존재하지 않는다면, 단일 앰퍼샌드(&)를 사용하여 명령어가 실행될 때마다 항상 사용자에게 프롬프트 합니다.

실행시에 열 이름, 표현식 그리고 텍스트 명시

SQL>	SELECT	empno, ename, job, &column_name
2	FROM	emp
3	WHERE	&condition
4	ORDER BY	&order_column;

Enter value for column name:	sal
Enter value for condition:	sal>=3000
Enter value for order_column:	ename

EMPNO	ENAME	JOB	SAL
7902	FORD	ANALYST	3000
7839	KING	PRESIDENT	5000
7788	SCOTT	ANALYST	3000

실행 시에 열 이름, 표현식 그리고 텍스트를 정의(계속)

위의 예는 EMP 테이블로부터 종업원 번호, 이름, 업무 그리고 실행 시에 사용자에게 의해 명시되는 어떤 다른 열을 디스플레이합니다. 사용자는 또한 결과 데이터를 정렬하여 행과 열을 검색하기 위한 조건을 명시할 수 있습니다.

&& 치환 변수 사용

매번 사용자에게 프롬프트하지 않고 변수 값을 재사용하고자 한다면 &&를 사용합니다.

SQL>	SELECT	empno, ename, job, &&column_name
2	FROM	emp
3	ORDER BY	&column_name;

Enter value for column name:	deptno
------------------------------	--------

EMPNO	ENAME	JOB	DEPTNO
7839	KING	PRESIDENT	10
7782	CLARK	MANAGER	10
7934	MILLER	CLERK	10
...			

14 rows selected.

더블 앰퍼샌드(&&) 치환 변수

매번 사용자에게 프롬프트하지 않고 변수 값을 재사용하고자 한다면 더블 앰퍼샌드(&&) 치환 변수를 사용할 수 있습니다. 사용자는 오직 한번만 값에 대해 프롬프트될 것입니다. 슬라이드의 예에서 사용자는 변수 column_name 에 대해서 오직 한번만 값을 요구합니다. 사용자(deptno)에 의해서 제공된 값은 디스플레이와 데이터 정렬 양쪽에서 사용됩니다.

SQL*Plus 는 DEFINE 명령을 사용하여 공급된 값을 저장합니다. 그것은 변수 이름이 참조될 때 마다 다시 사용될 것입니다. 사용자 변수를 삭제하기 위해서 UNDEFINE 명령을 사용해야 한다.

사용자 변수 정의

- 다음의 두가지 **SQL*Plus** 명령어중 하나를 사용하여 변수를 미리 정의할 수 있습니다.
 - **DEFINE: CHAR** 데이터형의 사용자 변수를 생성합니다.
 - **ACCEPT:** 사용자 입력을 받고 그것을 변수에 저장합니다.
- **DEFINE** 명령어를 사용할 때 공백을 사용할 필요가 있다면, 공백을 단일 인용 표시로 둘러싸야 합니다.

사용자 변수 정의

SELECT 문장을 실행하기 전에 사용자 변수를 미리 정의할 수 있습니다. SQL*Plus 는 사용자 변수를 정의하고 설정하기 위해서 두개의 명령을 제공합니다: DEFINE 과 ACCEPT

명령	설명
DEFINE variable = value	CHAR 데이터형 사용자 변수를 생성하고 그것에 값을 할당합니다.
DEFINE variable	변수, 변수 값, 변수 데이터형을 디스플레이합니다.
DEFINE	값과 데이터형을 가진 모든 데이터형을 디스플레이

	합니다.
ACCEPT (see syntax on next slide)	사용자 입력 라인을 읽고 그것을 변수에 저장합니다.

ACCEPT 명령어 사용

```
ACCEPT dept PROMPT 'Provide the department name: '
SELECT *
FROM dept
WHERE dname = UPPER('&dept')
/
```

```
Provide the department name: Sales

DEPTNO DNAME          LOC
-----
30 SALES              CHICAGO
```

ACCEPT 명령어 사용

ACCEPT 명령은 DEPT 라는 변수에서 읽습니다. 프롬프트는 변수를 사용자에게 요구할 때 "Provide the department name:"라는 메시지를 디스플레이 합니다. 그런 다음에 SELECT 문장은 사용자가 입력한 부서 값을 받아서, 그것을 DEPT 테이블로부터 적절한 행을 검색하는데 사용합니다.

사용자가 부서 이름에 대해서 올바른 값을 입력한다면, SELECT 문장은 사용자가 입력한 값을 받아서 WHERE 절에서 그것을 DNAME 과 비교하는, 다른 어떤 SELECT 문장과 똑같은 방법으로 실행합니다.

& 문자는 ACCEPT 명령에서 DEPT 변수와 함께 나타나지 않음을 주목하십시오. &는 오직 SELECT 문장에만 나타납니다.

지침

- ACCEPT와 DEFINE 명령 모두는 변수가 존재하지 않으면 변수를 생성합니다. 존재한다면 자동적으로 재정의할 것입니다.
- DEFINE 명령을 사용할 때, 공백을 포함하는 스트링을 둘러싸기 위해서 단일 인용

부호(' ')를 사용합니다.

- 다음에 대해서 ACCEPT 명령을 사용합니다.

-사용자 입력을 받아들일 때 여러분이 원하는 프롬프트를 제공합니다. 그렇지 않으면, 디폴트로 “Enter value for 변수명”이 나타납니다.

-NUMBER 또는 DATE 데이터형 변수를 명시적으로 정의합니다.

-보안 문제때문에 사용자 입력을 숨깁니다.

DEFINE 명령어 사용

- 부서 이름을 유지하기 위한 변수를 생성합니다.

```
SQL> DEFINE deptname = sales
SQL> DEFINE deptname
```

```
DEFINE DEPTNAME          = "sales" (CHAR)
```

- 그 변수를 사용합니다.

```
SQL> SELECT *
2 FROM dept
3 WHERE dname = UPPER('&deptname');
```

DEFINE 명령 사용

변수를 생성하기 위해서 DEFINE 명령을 사용할 수 있으며 생성한 다음에 변수를 사용합니다. 위의 예는 부서 이름 SALES 를 포함하는 DEPTNAME 변수를 생성합니다. 그런 다음에 SQL 문장은 sales 부서의 번호와 지역을 디스플레이하기 위해서 이 변수를 사용합니다.

DEPTNO	DNAME	LOC

30	SALES	CHICAGO

변수를 삭제하기 위해서 UNDEFINE 명령을 사용합니다.

```
SQL> UNDEFINE deptname
```

```
SQL> DEFINE deptname  
symbol deptname is UNDEFINED
```

SQL*Plus 환경 정의

- 현재 세션을 제어하기 위해서 **SET** 명령을 사용합니다.

```
SET system_variable value
```

- **SHOW** 명령을 사용하여 설정 내용을 검사합니다.

```
SQL> SET ECHO ON
```

```
SQL> SHOW ECHO  
echo ON
```

SQL*Plus 환경 정의

SET 명령을 사용하여 현재 운용중인 SQL*Plus 환경을 제어합니다.

구문형식에서:

system_variable	세션의 환경을 제어하는 변수입니다.
value	시스템 변수를 위한 값입니다.

SHOW 명령을 사용하여 설정 내용을 검사할 수 있습니다. 슬라이드의 SHOW 명령은 ECHO 가 ON 인 OFF 인지를 검사합니다.

모든 SET 변수 값을 보기 위해서 SHOW ALL 명령을 사용합니다.

명령 참조에 대해 보다 많은 정보를 알고자 한다면, SQL*Plus User's Guide and Reference, Release 8 을 참조하십시오.

SET 명령어 변수

- **ARRAYSIZE** {20 | *n*}
- **COLSEP** { _ | *text*}
- **FEEDBACK** {6 | *n* | **OFF** | **ON**}
- **HEADING** {**OFF** | **ON**}
- **LINESIZE** {80 | *n*}
- **LONG** {80 | *n*}
- **PAGESIZE** {24 | *n*}
- **PAUSE** {**OFF** | **ON** | *text*}
- **TERMOUT** {**OFF** | **ON**}

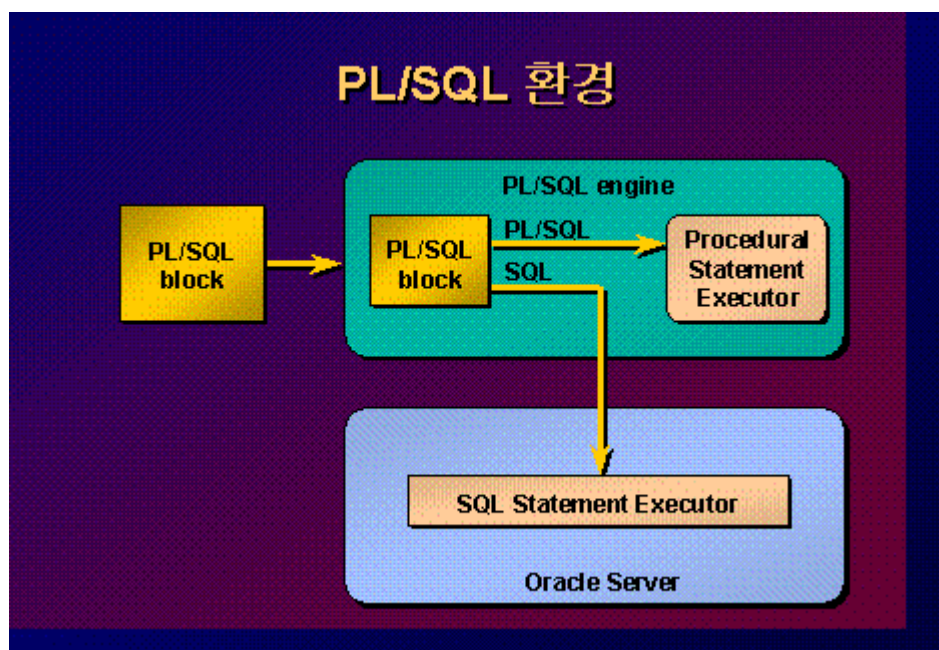
SET 명령 변수

SET 변수와 값	설명
ARRAY[SIZE] {20 n}	데이터베이스 데이터 패치 크기를 설정합니다.
COLSEP {_ text}	열 사이에 출력되는 텍스트를 설정합니다. 디폴트 스트링은 공백입니다.
FEED[BACK] {6 n OFF ON}	질의가 최소한 n 개의 레코드를 검색할 때 질의에 의해 리턴된 레코드의 개수를 디스플레이 합니다
HEA[DING] {OFF ON}	열 헤딩을 리포트에 디스플레이할 지의 여부를 결정합니다.
LIN[ESIZE] {80 n}	리포트에 대해서 라인당 n 개 까지의 문자 개수를 설정 합니다.
LONG {80 n}	LONG 값을 디스플레이하기 위해 최대 폭을 설정합니다.
PAGES[IZE] {24 n}	출력 페이지당 라인의 수를 명시합니다.
PAU[SE] {OFF ON text}	터미널을 제어하도록 해줍니다. (각각의 멈춤(pause))

	마다 리턴 키를 눌러야 합니다.)
TERM[OUT] {OFF ON}	결과를 화면에 디스플레이할 지의 여부를 결정합니다.

주: n 값은 숫자 값을 나타냅니다. 위의 밑줄 값은 디폴트 값을 가리킵니다. 변수에 값을 입력 하지 않는다면, SQL*Plus 는 디폴트 값을 가정합니다.

제 18 장 PL/SQL 환경



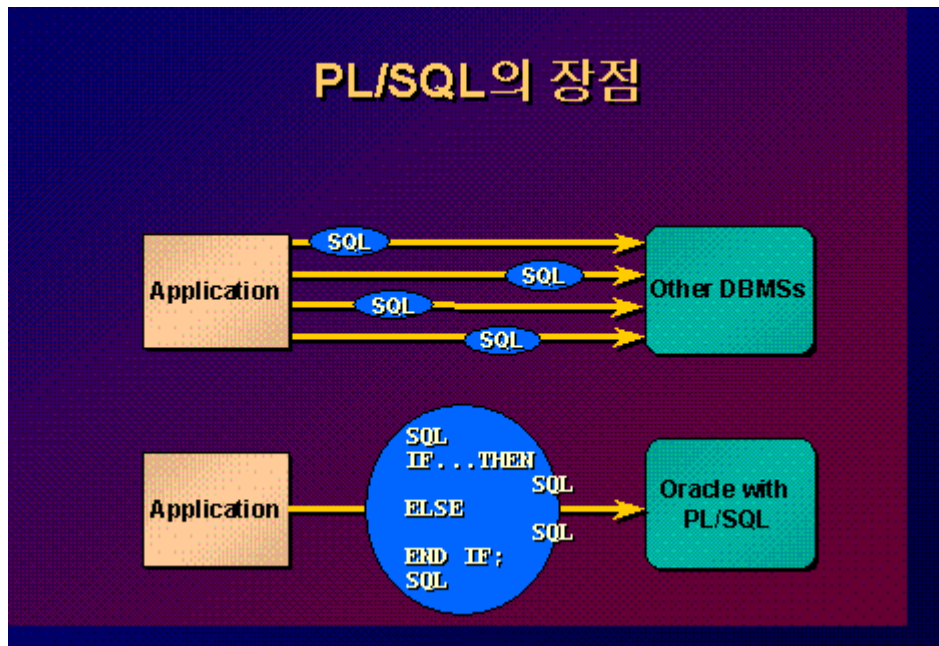
PL/SQL 엔진과 오라클 서버

PL/SQL 은 오라클 소유권의 제품은 아니고, 오라클 서버와 오라클 툴에 도입된 기술입니다.

PL/SQL 의 블록이 PL/SQL 엔진에 넘겨지고 처리됩니다. 사용될 엔진은 PL/SQL 블록이 어디에서 불리어지는가에 따라 다릅니다.

PL/SQL 블록을 Pro* program, 사용자-exit, SQL*Plus, 서버 관리자 등으로부터 보내게 되면 오라클 서버에 있는 PL/SQL 엔진이 처리를 합니다.그리고 나서 그 엔진은 블록내의 SQL 을 각각의 문장으로 분리시켜 SQL 문장 실행기로 넘깁니다. 즉 응용프로그램에서 오라클 서버로 블록을 송신하기 위해서 단일 전송만이 필요하다는 의미이며, 이렇게 함으로써 특히 **클라이언트-서버** 네트워크에서 성능을 향상 시킬 수 있습니다. **내장**된 서브프로그램도 데이터베이스에 연결된 응용프로그램들에 의해 참조가

되어집니다.



성능 향상

PL/SQL은 어플리케이션의 성능을 향상시킬 수 있습니다. 장점은 실행 환경에 따라 차이가 있습니다.

- PL/SQL은 단일 블록 내에서 sql 명령문을 함께 그룹화하고 단일 호출로써 서버로 전체 블록을 송신하기 위해 사용될 수 있습니다. 그러므로써 네트워크 교통량을 감소시킵니다. PL/SQL없이 SQL 명령문은 한번에 하나를 처리합니다. 각 SQL 명령문은 오라클 서버에의 또 다른 호출과 더 높은 성능 오버헤드를 초래할 수 있고 네트워크 환경에서 오버헤드는 중요해질 수 있습니다. 어플리케이션이 SQL문을 집중적으로 갖고 있다면 실행을 위해 오라클 서버로 그것을 송신하기 전에 SQL 명령문을 그룹화 하기 위해 PL/SQL 블록과 서브 프로그램을 사용할 수 있습니다.
- PL/SQL은 Developer Forms, Reports같은 오라클 서버 어플리케이션 개발 툴과 협력할 수 있습니다. 이 툴들에 절차적 처리 능력을 추가함으로써, PL/SQL은 성능을 향상시킵니다.

주: Developer 어플리케이션의 일부로써 선언된 프로시저와 함수는 데이터베이스에 저장된 프로시저 함수와 다르지만 그 일반적인 구조는 동일합니다. **내장**된 서브 프로그램은 데이터베이스 객체이며 데이터 사전에 저장됩니다. 그것들은 Developer 어플리케이션을 비롯한 많은 어플리케이션에 의해 액세스 될 수 있습니다

PL/SQL 블록 구조

- **DECLARE – Optional**
 - Variables, cursors, user-defined exceptions
- **BEGIN – Mandatory**
 - SQL statements
 - PL/SQL statements
- **EXCEPTION – Optional**
 - Actions to perform when errors occur
- **END; – Mandatory**

```
DECLARE
...
BEGIN
...
EXCEPTION
...
END;
```

PL/SQL 블록 구조

PL/SQL 은 프로그램을 논리적인 블록으로 나누게 하는 구조화된 블록 언어입니다. PL/SQL 블록은 3 섹션까지로 구성됩니다: 선언 (선택적), 실행 (필수적), 예외처리 (선택적). BEGIN 과 END 키워드만 필수적입니다. 변수들을 사용하는 블록에 대해 논리적으로 선언할 수 있습니다. 오류조건 (예외로 알려진) 은 블록 내에서 특별하게 처리될 수 있습니다. 변수들과 그 외의 식별자를 참조하고 선언함으로써 PL/SQL 블록 내에서 값을 저장하고 변경할 수 있습니다.

아래 테이블은 3 가지 블록 섹션을 기술합니다.

섹션	설 명	포 함
선언	실행과 선언 섹션에서 참조되는 모든 변수, 상수, 커서 , 사용자 정의 예외를 포함합니다.	선택적
실행	데이터베이스 내의 데이터를 조작하기 위한 SQL 명령문과, 블록에서 데이터를 조작하기 위한 PL/SQL 명령문을 포함합니다.	필수적
예외처리	실행 섹션에서 오류 또는 비정상적인 조건이 발생할 때 수행하기 위한 작업을 지정합니다.	선택적

PL/SQL 블록 구조

```
DECLARE
  v_variable VARCHAR2(5);
BEGIN
  SELECT      column_name
    INTO      v_variable
  FROM        table_name;
EXCEPTION
  WHEN exception_name THEN
    ...
END;
```

```
DECLARE
...
BEGIN
...
EXCEPTION
...
END;
```

- SQL 문장이나 PL/SQL 컨트롤 문장의 끝에 세미콜론(:)을 넣습니다.
- SQL 버퍼에서 익명의 PL/SQL 블록을 실행시키기 위해 슬래시를 넣습니다. 블록이 성공적으로 실행된다면, 처리되지 않은 오류 또는 **컴파일** 오류 없이, 메시지 출력은 아래와 같이 되어야 합니다.

PL/SQL procedure successfully completed

- SQL 버퍼를 닫기 위해 마침표(.)를 넣습니다. PL/SQL 블록은 버퍼 안에서 계속되는 하나의 문장으로 취급되고, 블록 내의 세미콜론은 버퍼를 닫거나 실행하지 않습니다.

주: PL/SQL에서, 오류는 exception 이라 불립니다.

섹션 키워드 DECLARE, BEGIN, 그리고 EXCEPTION은 세미콜론이 뒤따르지 않습니다. 그러나, END와 모든 다른 PL/SQL 문장은 문장을 종료시키기 위해서는 세미 콜론이 반드시 필요합니다. 동일 라인에 함께 문장을 나열할 수 있습니다. 그러나 이 방법은 명료함 또는 편집을 위해 권장되지 않습니다.



PL/SQL의 모든 단위는 하나 또는 그 이상의 블록을 포함합니다. 이 블록은 다른 것으로부터 하나로 완전히 분리되거나 중첩될 수 있습니다. 기본 단위(프로시저와 함수, 또는 서브프로그램, 그리고 익명의 블록)는 임의의 수의 중첩된 서브블록을 포함할 수 있는, 논리적 블록인 PL/SQL 프로그램을 구성합니다. 그러므로 한 블록은 다른 블록의 작은 부분이 되기도 하고 또는 코드 단위의 전체 중 일부가 될 수도 있습니다. PL/SQL의 2가지 형태인, 익명 블록과 서브프로그램 중 익명의 블록만 본 과정에서 설명합니다.

익명 블록

익명 블록은 이름이 없는 블록입니다. 그것은 실행되기 위한 어플리케이션 안에서 선언되고 실행시간 중에 실행을 위해 PL/SQL 엔진으로 전달됩니다. 선행[컴파일러](#) 프로그램과 SQL*Plus 또는 서버 관리자에서 익명의 블록을 [내장](#)할 수 있습니다.

서브프로그램

서브프로그램은 매개변수를 취할 수 있고 호출할 수 있는 이름있는 PL/SQL 블록입니다. 프로시저 또는 함수로 선언될 수 있습니다. 일반적으로 어떤 작업을 수행하기 위해 프로시저를 사용하고, 값을 계산하기 위해 함수를 사용합니다. 서버 또는 어플리케이션 수준에서 서브프로그램을 저장할 수 있습니다. Developer/2000 컴포넌트(Forms, Reports, Graphics)를 사용하여

어플리케이션(Forms 또는 Reports)의 일부분으로 함수와 프로시저를 선언할 수 있고 필요할 때마다 동일 어플리케이션의 트리거 (다음페이지를 보십시오.), 함수, 다른 프로시저에서 그것들을 호출할 수 있습니다.

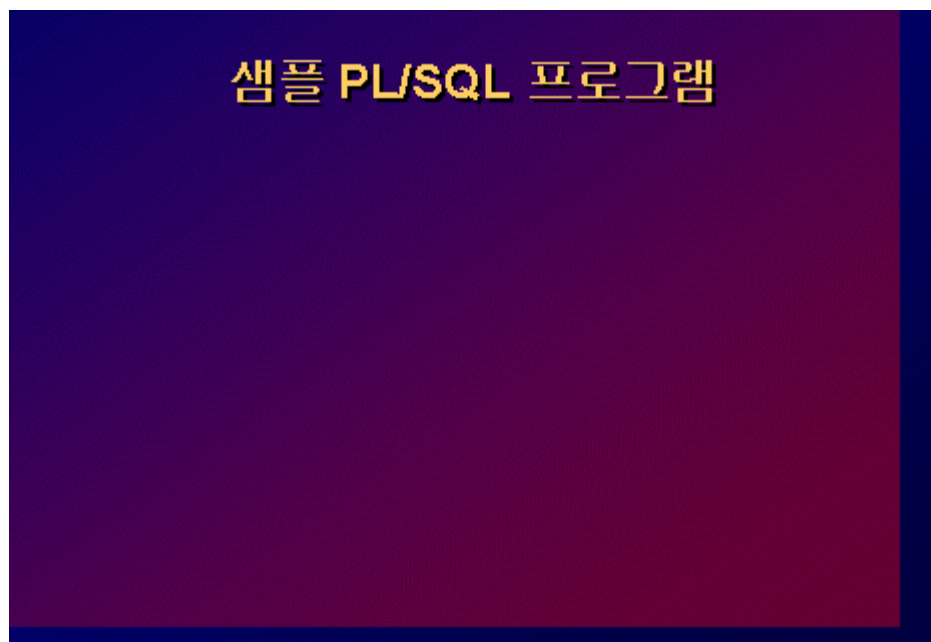
주: 함수는 값을 반환해야 함을 제외하고는 프로시저와 동일합니다. 프로시저와 함수는 다음 PL/SQL 과정에서 설명합니다.



아래 테이블은 기본 PL/SQL 블록을 사용하는 다양한 다른 PL/SQL 프로그램 구성을 약술하고 있습니다.

프로그램 구성	설 명	사용 환경
익명 블록	어플리케이션에 내장 되거나 대화식으로	모든 PL/SQL 환경
내장된 프로시저	매개변수를 받아들일 수 있고 이름을	오라클 서버

	내에서 저장됩니다.	
어플리케이션 프로시저 또는 함수	매개변수를 받아들일 수 있고 이름을 이용하여 반복적으로 호출할 수 있는 이름이 있는 PL/SQL 블록은 Developer/2000 어플리케이션에 저장되거나 공유 라이브러리에 저장됩니다	Developer/2000 컴포넌트- 예를 들면, Forms
패키지	관련된 프로시저, 함수들을 묶어 이름 붙인 PL/SQL 모듈입니다.	Oracle Server 와 Developer/2000 컴포넌트? 예를 들면, Forms
데이터베이스 트리거	데이터베이스 테이블과 관련되고 DML 명령문 에 의해 트리거될 때 자동적으로 실행됩니다.	Oracle Server
어플리케이션 트리거	PL/SQL 블록은 어플리케이션 이벤트와 관련되고 자동적으로 실행됩니다.	Developer/2000 컴포넌트- 예를 들면, Forms



***** 부서/직원 리포트 *****

부서 : 부서번호	부서명	위치
10	ACCOUNTING	NEW YORK

사원 : 사번	이름	급여	급여 등급
7871	Anderson	2000	***
7890	Andrew	700	*
7870	Scott	2400	****
7866	David	900	*
7891	Sylvia	3200	*****

2 명의 급여가 인상되었습니다.

위와 같은 리포트를 뽑아내고 \$1000 보다 적은 급여를 받는 직원의 급여를 10%인상하는 프로그램을 작성하여 봅시다.(PL/SQL 변수 설명 중 v_...를 v\$...로 설명을 하였으나, 실제로는 v_...가 맞습니다.)

```
ACCEPT p_deptno PROMPT 'Please enter the department number : '
```

```
DECLARE
```

```
    v_deptno    NUMBER(2);
    v_dname     CHAR(14);
    v_loc       CHAR(13);
```

```
    v_empno     emp.empno%TYPE;
    v_ename     emp.ename%TYPE;
    v_sal       emp.sal%TYPE;
    v_grade     salgrade.grade%TYPE;
    v_dno       emp.deptno%TYPE := &p_deptno;
```

```
CURSOR emp_cursor IS
    SELECT empno, ename, sal
    FROM emp
    WHERE deptno = &p_deptno;

e_no_emp      EXCEPTION;
```

```
BEGIN
```

```
    -- print out the report title
    DBMS_OUTPUT.PUT_LINE('*****부서/직원 리포트*****');
    SELECT deptno, dname, loc
    INTO v_deptno, v_dname, v_loc
    FROM dept
    WHERE deptno = v_dno;
```

```

DBMS_OUTPUT.PUT_LINE('부서 : 부서번호   부서명           위치');
DBMS_OUTPUT.PUT_LINE('      ' || v_deptno || '      ' || v_dname ||
                      '      ' || v_loc || '      ');
DBMS_OUTPUT.PUT_LINE('사원 : 사번   이름   급여   급여등급');
OPEN emp_cursor;
FETCH emp_cursor INTO v_empno, v_ename, v_sal;
/* according to the salary amount,
   print out the stars */
WHILE emp_cursor%FOUND LOOP
    SELECT grade
        INTO v_grade
        FROM salgrade
        WHERE v_sal > losal AND v_sal < hisal;
    DBMS_OUTPUT.PUT('      ' || v_empno || '      ' || v_ename ||
                  '      ' || v_sal || '      ');

    FOR I in 1..v_grade LOOP
        DBMS_OUTPUT.PUT('*');
    END LOOP;

    DBMS_OUTPUT.PUT_LINE(' ');
    FETCH emp_cursor INTO v_empno, v_ename, v_sal;
END LOOP;
CLOSE emp_cursor;
UPDATE emp
    SET      sal = sal * 1.1
    WHERE    sal < 1000;
IF SQL%ROWCOUNT = 0 THEN
    RAISE e_no_emp;
ELSE
    DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || '명의 급여가 ' ||
                        '인상되었습니다. ');
END IF;
COMMIT;
EXCEPTION
    WHEN no_data_found THEN
        DBMS_OUTPUT.PUT_LINE('There is no matching data. ');
    WHEN too_many_rows THEN

```

```

        DBMS_OUTPUT.PUT_LINE('More than one matching data. ');
    WHEN e_no_emp THEN
        DBMS_OUTPUT.PUT_LINE('No employee with a smaller ' ||
                               'than $1000. ');
    --in case of any other exception
    WHEN others THEN
        DBMS_OUTPUT.PUT_LINE('Some other error occurred. ');
END;

```

제 19 장 PL/SQL 변수 선언

PL/SQL 변수 선언

구문

```

identifier [CONSTANT] datatype [NOT NULL]
[:= | DEFAULT expr];

```

예

```

Declare
  v_hiredate      DATE;
  v_deptno        NUMBER(2) NOT NULL := 10;
  v_location      VARCHAR2(13) := 'Atlanta';
  c_comm          CONSTANT NUMBER := 1400;

```

PL/SQL 변수 선언

PL/SQL 블록에서 그것들을 참조하기 전에 선언섹션에서 모든 PL/SQL 식별자를 선언할 필요가 있습니다. 초기값을 할당하기 위해 옵션을 가집니다. 변수를 선언하기 위해 변수에 대한 값을 할당할 필요는 없습니다. 선언에서 다른 변수를 참조한다면, 이전 문장에서 개별적으로 그것들을 반드시 선언해 놓아야 합니다.

구문에서

identifier 변수의 이름입니다

CONSTANT 변수를 변경할 수 없도록 하기 위해 변수를 제약합니다

상수는 초기화 되어야 합니다

datatype 스칼라, 조합, 참조, LOB 데이터형 입니다. (이장에서는

스칼라와 조합 데이터형만 논합니다).

NOT NULL 값을 포함해야만 하도록 하기 위해 변수를 제약합니다.

NOT NULL 변수는 초기화되어야 합니다

expr 리터럴, 다른 변수, 또는 연산자나 함수를 포함하는 표현이 될 수 있는 PL/SQL 표현식입니다.

PL/SQL 변수 선언

구문

```
identifier [CONSTANT] datatype [NOT NULL]  
[:= | DEFAULT expr];
```

예

```
Declare  
  v_hiredate      DATE;  
  v_deptno        NUMBER(2) NOT NULL := 10;  
  v_location      VARCHAR2(13) := 'Atlanta';  
  c_comm          CONSTANT NUMBER := 1400;
```

실습

1. 아래 선언문 각각을 평가합니다. 이것들 중 어떤 것이 올바르지 않으며 왜 그런가를 설명하십시오.

a.	DECLARE
----	---------

	<code>v_id</code>	<code>NUMBER(4);</code>
--	-------------------	-------------------------

b.	DECLARE <code>v_x, v_y, v_z</code>	<code>VARCHAR2(10);</code>
----	---------------------------------------	----------------------------

c.	DECLARE <code>v_birthdate</code>	<code>DATE NOT NULL;</code>
----	-------------------------------------	-----------------------------

변수 값 지정

구문

```
identifier := expr;
```

예
미리 정의된 고용일자를 새로운 종업원에 대해 설정합니다

```
v_hiredate := '31-DEC-98';
```

종업원 이름을 "Maduro"로 설정합니다.

```
v_ename := 'Maduro';
```

변수 값 지정

변수 값을 지정하거나 재지정하기 위해, PL/SQL 지정 문장을 씁니다. 지정(assignment) 연산자(:=)의 좌측에 새 값을 받기 위한 변수를 적습니다.

구문에서,

`identifier` 스칼라 변수의 이름입니다.

`expr` 변수, 리터럴, 함수 호출이 될 수 있지만 데이터베이스 열은

안됩니다.

지정 변수 값 예는 아래와 같이 정의됩니다:

- 현재 급여 V_SAL의 값으로 최대 급여 V_MAX_SAL을 설정합니다.
- v_ename “Maduro” 이름을 저장합니다.

변수 값을 지정하기 위한 다른 방법은 데이터베이스 값에서 인출하거나 선택하는 것입니다.

아래 예에서, 종업원의 급여를 선택할 때 10% 보너스를 계산하도록 합니다.

```
SQL> SELECT      sal * 0.10
2 INTO          bonus
3 FROM          emp
4 WHERE         empno = 7369;
```

이때 다른 계산에서 변수 bonus 를 사용할 수 있거나 그 값을 데이터베이스 테이블에 삽입할 수 있습니다.

주: 데이터베이스에서의 값을 변수로 지정하기 위해, SELECT 나 FETCH 문장을 사용합니다.

기본 스칼라 데이터형

- VARCHAR2 (*maximum_length*)
- NUMBER [(*precision*, *scale*)]
- DATE
- CHAR [(*maximum_length*)]
- LONG
- LONG RAW
- BOOLEAN
- BINARY_INTEGER
- PLS_INTEGER

데이터형	설 명
------	-----

VARCHAR2 (maximum_length)	변수 길이 문자 데이터에 대한 기본형은 32767 바이트 까지 입니다. VARCHAR2 변수와 상수에 대한 디폴트 크기는 없습니다.
NUMBER [(precision, scale)]	고정(fixed)과 유동(floating) 포인트 숫자에 대한 기본형
DATE	날짜와 시간에 대한 기본형. DATE 값은 자정 이후의 초 단위로 날에 대한 시간을 포함합니다. 날짜의 범위는 BC 4712 와 AD 9999 사이 입 니다.
CHAR [(maximum_length)]	고정 길이 문자 데이터에 대한 기본형은 32767 바이트 까지입니다. maximum_length 를 지정하지 않는다면 디폴트 길이는 1 로 설정됩니다
LONG	고정 길이 문자 데이터에 대한 기본형은 32760 바이트 까지 입니다. LONG 데이터베이스 열의 최대 폭은 2147483647 바이트입니다.
LONG RAW	이진 데이터와 바이트 문자열에 대한 기본형은 32760 바이트까지 입니다. LONG RAW 데이터는 PL/SQL 에 의해 해석되지 않습니다.
BOOLEAN	논리적 계산에 사용되는 3 가지 가능한 값 가운데 기본형 TRUE, FALSE, NULL.
BINARY_INTEGER	-2147483647 과 2147483647 사이의 정수에 대한 기본형.
PLS_INTEGER	-2147483647 과 2147483647 사이의 signed 정수에 대한 기본형. PLS_INTEGER 값은 NUMBER 와 BINARY_INTEGER 값보다 적은 기억장치를 필요로 합니다.

주: LONG 데이터형은 LONG 값의 최대 길이가 32760 바이트인 것을 제외하고는
VARCHAR2 와 유사합니다. 그러므로 32760 바이트보다 더 긴 값은 LONG 데이터베이스
열에서 LONG PL/SQL 변수로 선택될 수 없습니다.

기본 스칼라 데이터형

- VARCHAR2 (*maximum_length*)
- NUMBER [(*precision*, *scale*)]
- DATE
- CHAR [(*maximum_length*)]
- LONG
- LONG RAW
- BOOLEAN
- BINARY_INTEGER
- PLS_INTEGER

실습(계속)

2. 각 아래 지정에서 결과 표현식의 데이터 형을 결정합니다.

a.	<code>v_days_to_go := v_due_date - SYSDATE;</code>
----	--

b.	<code>v_sender := USER ' : ' TO_CHAR(v_dept_no);</code>
----	---

c.	<code>v_sum := \$100,000 + \$250,000;</code>
----	--

d.	<code>v_flag := TRUE;</code>
----	------------------------------

e.	<code>v_n1 := v_n2 > (2 * v_n3);</code>
----	--

f.	<code>v_value := NULL;</code>
----	-------------------------------

%TYPE 속성과 변수 선언

예

```
...  
v_ename          emp.ename%TYPE;  
v_balance        NUMBER(7,2);  
v_min_balance    v_balance%TYPE := 10;  
...
```

%TYPE 속성과 변수선언

종업원의 이름을 저장하기 위해 변수를 선언합니다.

<pre>... v_ename emp.ename%TYPE; ...</pre>

10 인 최소 잔액과 은행 계정의 잔액을 저장하기 위해 변수를 선언합니다.

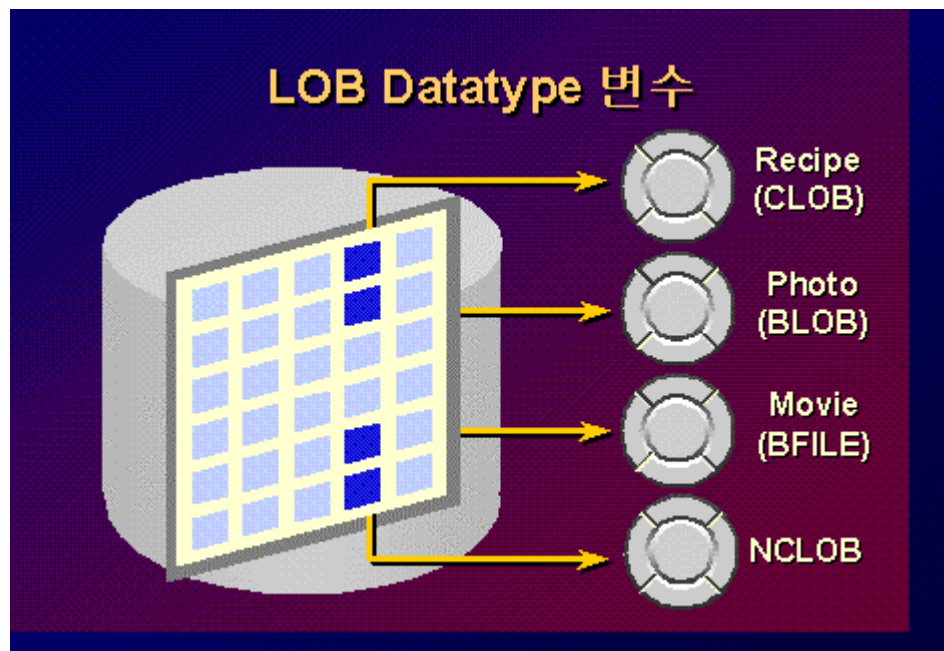
<pre>...</pre>

```

v_balance      NUMBER(7,2);
v_min_balance  v_balance%TYPE := 10;
...

```

NOT NULL 열 제약조건은 %TYPE 을 사용하여 선언된 변수에 대해서는 적용되지 않습니다.그러므로 NOT NULL 로 정의된 데이터베이스 열을 사용하여 %TYPE 속성을 사용하는 변수를 선언한다면 변수에 NULL 값을 지정할 수 있습니다



LOB 데이터형 변수

LOB (large object) Oracle8 데이터형으로 구조화되지 않은 데이터(텍스트, 그래픽이미지, 비디오 클립, 소리 웨이브 폼 같은)블록을 4기가 바이트 크기까지 저장할 수 있습니다. LOB 데이터형은 데이터에 대한 랜덤 액세스를 지원합니다.

- CLOB (character large object) 데이터형은 데이터베이스 내의 단일 바이트 문자 데이터의 대형 블록을 저장하기 위해 사용됩니다.
- BLOB (binary large object) 데이터형은 행의 안팎에 데이터 베이스 내의 대형 이진 객체를 저장하기 위해 사용됩니다.
- BFILE (binary file) 데이터형은 데이터 베이스 외부의 운영 시스템 파일의 대형 이진 객체를 저장하기 위해 사용됩니다.
- NCLOB (national language character large object) 데이터형은 데이터베이스 내의 단일 바이트,또는 고정 길이의 멀티바이트 NCHAR 데이터를 행의 안팎에 저장하기 위해 사용됩니다.

바인드 변수

일년 급여를 SQL*Plus 호스트변수로 저장합니다.

```
:g_monthly_sal := v_sal / 12;
```

- 호스트 변수로서 non-PL/SQL 변수를 참조합니다.
- 콜론(:)으로 참조 접두어를 만듭니다.

변수 지정값

호스트 변수를 참조하기 위해, 선언된 PL/SQL 변수와 호스트 변수를 구별하기 위해 콜론(:)으로 참조 접두어를 만들어야 합니다.

예

```
:host_var1 := v_sal;  
:global_var1 := 'YES'
```

예

```
SQL> VARIABLE return_code NUMBER  
... :return_code := 10 ;  
SQL> PRINT return_code
```

제 20 장 PL/SQL 에서의 SELECT 문장

***** 부서/직원 리포트 *****

부서 : 부서번호	부서명	위치
10	ACCOUNTING	NEW YORK

사원 : 사번	이름	급여	급여 등급
7871	Anderson	2000	***
7890	Andrew	700	*
7870	Scott	2400	****
7866	David	900	*
7891	Sylvia	3200	*****

2 명의 급여가 인상되었습니다.

위와 같은 리포트를 뽑아내고 \$1000 보다 적은 급여를 받는 직원의 급여를 10%인상하는 프로그램을 작성하여 봅시다.

```
ACCEPT p_deptno PROMPT 'Please enter the department number : '
```

```
DECLARE
```

```
    v_deptno    NUMBER(2);
```

```
    v_dname     CHAR(14);
```

```
    v_loc       CHAR(13);
```

```
    v_empno     emp.empno%TYPE;
```

```
    v_ename     emp.ename%TYPE;
```

```
    v_sal       emp.sal%TYPE;
```

```
    v_grade     salgrade.grade%TYPE;
```

```
    v_dno       emp.deptno%TYPE := &p_deptno;
```

```
    CURSOR emp_cursor IS
```

```
        SELECT empno, ename, sal
```

```
        FROM emp
```

```
        WHERE deptno = &p_deptno;
```

```
    e_no_emp    EXCEPTION;
```

```
BEGIN
```

```
    -- print out the report title
```

```

DBMS_OUTPUT.PUT_LINE('*****부서/직원 리포트*****');
SELECT deptno, dname, loc
      INTO v_deptno, v_dname, v_loc
      FROM dept
      WHERE deptno = v_dno;
DBMS_OUTPUT.PUT_LINE('부서 : 부서번호   부서명           위치');
DBMS_OUTPUT.PUT_LINE('      ' || v_deptno || '      ' || v_dname ||
                      '      ' || v_loc || '      ');
DBMS_OUTPUT.PUT_LINE('사원 : 사번   이름   급여   급여등급');
OPEN emp_cursor;
FETCH emp_cursor INTO v_empno, v_ename, v_sal;
/* according to the salary amount,
   print out the stars */
WHILE emp_cursor%FOUND LOOP
    SELECT grade
          INTO v_grade
          FROM salgrade
          WHERE v_sal > losal AND v_sal <= hisal;
    DBMS_OUTPUT.PUT('      ' || v_empno || '      ' || v_ename ||
                    '      ' || v_sal || '      ');
    FOR I in 1..v_grade LOOP
        DBMS_OUTPUT.PUT('*');
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('');
    FETCH emp_cursor INTO v_empno, v_ename, v_sal;
END LOOP;
CLOSE emp_cursor;
UPDATE emp
      SET      sal = sal * 1.1
      WHERE    sal < 1000;
IF SQL%ROWCOUNT = 0 THEN
    RAISE e_no_emp;
ELSE
    DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || '명의 급여가 ' ||
                          '인상되었습니다. ');
END IF;

```

```

COMMIT;
EXCEPTION
  WHEN no_data_found THEN
    DBMS_OUTPUT.PUT_LINE('There is no matching data. ');
  WHEN too_many_rows THEN
    DBMS_OUTPUT.PUT_LINE('More than one matching data. ');
  WHEN e_no_emp THEN
    DBMS_OUTPUT.PUT_LINE('No employee with a smaller ' ||
                          'than $1000. ');
  --in case of any other exception
  WHEN others THEN
    DBMS_OUTPUT.PUT_LINE('Some other error occurred. ');
END;

```

PL/SQL에서 SELECT 문장

데이터베이스에서 SELECT로 데이터를 읽어
들입니다.

구문

```

SELECT select_list
INTO    {variable_name[, variable_name]...
        | record_name}
FROM    table
WHERE   condition;

```

PL/SQL 을 사용하여 데이터 읽기

데이터베이스에서 데이터를 읽어 들이기 위해 SELECT 문장을 사용합니다.

구문에서,

select_list	최소한 한 개 이상의 열의 목록이며, SQL 표현식, 행 함수, 그룹 함수를 포함할 수 있습니다.
variable_name	읽어 들인 값을 저장하기 위한 스칼라 변수입니다.
record_name	읽어 들인 값을 저장하기 위한 PL/SQL RECORD 입니다.
table	데이터베이스 테이블 명을 지정합니다.
condition	PL/SQL 변수와 상수를 포함하여 열명, 표현식, 상수, 비교연산자로 구성됩니다.

SELECT 문장에 대한 모든 구문형식을 이용하십시오.

호스트 변수는 콜론으로 접두어를 붙여야 함을 명심하십시오.

PL/SQL에서 SELECT 문장

INTO 절이 요구됩니다.

예

```
DECLARE
  v_deptno  NUMBER(2);
  v_loc     VARCHAR2(15);
BEGIN
  SELECT    deptno, loc
    INTO    v_deptno, v_loc
  FROM      dept
  WHERE     dname = 'SALES';
  ...
END;
```

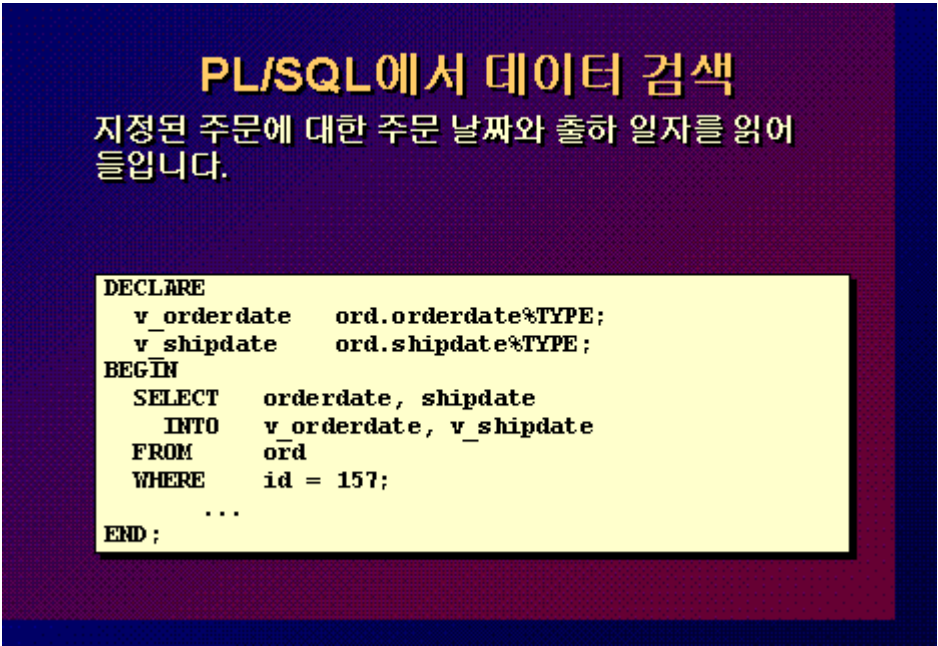
INTO 절

INTO 절은 필수이며 SELECT 와 FROM 사이에 위치합니다. 이것은 SELECT 절에서 SQL 이 리턴하는 값을 저장하기 위한 변수의 이름을 지정하기 위해 사용됩니다. 선택된 각 항목 당 하나의 변수를 제공해야 하고, 순서는 선택된 항목에 대해 같아야 합니다. PL/SQL 변수 또는 호스트 변수도 INTO 절에 사용합니다.

질의는 하나의 행만 리턴해야 합니다.

PL/SQL 블록 내의 SELECT 문장은 다음 규칙을 적용하는, Embbeded SQL 의 ANSI 범주에 속합니다. 질의는 하나의 행만 리턴해야 합니다. 하나의 행 이상 또는 행이 없는 것은 에러를 생성합니다.

PL/SQL 은 NO_DATA_FOUND 와 TOO_MANY_ROWS 예외(예외처리는 후속장에서 설명합니다.)로 블록의 예외 섹션에서 추적할 수 있는 표준 예외를 구성하여 이 에러를 처리합니다. 단일 행을 리턴하기 위해 SELECT 문장을 만들어야 합니다.



PL/SQL에서 데이터 검색
지정된 주문에 대한 주문 날짜와 출하 일자를 읽어 들입니다.

```
DECLARE
  v_orderdate  ord.orderdate%TYPE;
  v_shipdate   ord.shipdate%TYPE;
BEGIN
  SELECT  orderdate, shipdate
    INTO  v_orderdate, v_shipdate
  FROM    ord
 WHERE   id = 157;
  ...
END;
```

지침

PL/SQL 에서 데이터를 읽어 들이기 위해 이 지침들을 이행합니다.

- 세미콜론(;)으로 개별 SQL 문장을 종료합니다.
- SELECT 문장이 PL/SQL에 **내장**될 때 INTO 절을 필요로 합니다.

- WHERE 절은 선택적이며 입력 변수, 상수, 리터럴, 또는 PL/SQL 표현식을 지정하기 위해 사용될 수 있습니다.
- SELECT 절에서의 데이터베이스 열과 INTO 절에서 출력 변수의 수를 동일하게 지정합니다. 그것들은 위치상으로 대응되고 데이터형은 일치해야 함을 명심하십시오.



PL/SQL을 사용한 데이터 조작

DML(data manipulation) 명령어를 사용하여 데이터베이스에서 데이터를 조작합니다. PL/SQL에서 제한 없이 DML 명령 INSERT, UPDATE 그리고 DELETE를 생성할 수 있습니다. PL/SQL코드에 COMMIT 또는 ROLLBACK 문장을 포함함으로써 행 잠금(테이블 잠금)은 해제됩니다

- INSERT 문장은 테이블에 데이터의 새 행을 추가합니다.
- UPDATE 문장은 테이블에서 존재하는 행을 수정합니다.
- DELETE 문장은 테이블에서 원치 않는 행을 제거합니다

데이터 삽입

EMP 테이블에 새 종업원에 대한 정보를 추가합니다.
예

```
DECLARE
  v_empno      emp.empno%TYPE;
BEGIN
  SELECT
    empno_sequence.NEXTVAL
  INTO
    v_empno
  FROM
    dual;
  INSERT INTO emp (empno, ename, job, deptno)
  VALUES (v_empno, 'HARDING', 'CLERK', 10);
END;
```

데이터 삽입

- USER와 SYSDATE 같은 SQL 함수를 사용합니다.
- 데이터베이스 시퀀스를 사용하여 기본 키 값을 생성합니다.
- PL/SQL 블록에서 값을 얻습니다.
- 디폴트 값을 이용합니다.

주: INSERT 명령에서 식별자와 열 이름에 모호성이 있는 것은 불가능합니다.

INSERT 절에 있는 식별자는 데이터베이스 열 이름이어야 합니다.

데이터 갱신

EMP 테이블에서 분석가인 모든 종업원의 급여를 올립니다.

예

```
DECLARE
  v_sal_increase emp.sal%TYPE := 2000;
BEGIN
  UPDATE emp
  SET
    sal = sal + v_sal_increase
  WHERE
    job = 'ANALYST';
END;
```

데이터 갱신과 삭제

지정 연산자의 좌측에 있는 식별자는 항상 데이터 베이스 열이지만, 오른쪽에 있는 식별자도 데이터베이스 열 또는 PL/SQL 변수도 될 수 있기 때문에 UPDATE 문장의 SET 절에서 모호성이 존재할 수 있습니다. PL/SQL 에서의 SELECT 문장과 달리 수정된 행이 없으면 에러가 발생하지 않습니다.

주: PL/SQL 변수 지정은 항상 := 을, SQL 열 지정은 항상 = 을 사용합니다. 열명과 식별자명이 WHERE 절에서 같다면, 오라클 서버는 이름에 대해서 우선 데이터베이스를 조사함을 상기하십시오.

데이터 삭제

EMP 테이블에서 부서10에 속하는 행을 삭제합니다.

예

```
DECLARE
    v_deptno    emp.deptno%TYPE := 10;
BEGIN
    DELETE FROM emp
        WHERE deptno = v_deptno;
END;
```

데이터 삭제

지정된 주문을 삭제합니다.

```
DECLARE
    v_ordid      ord.ordid%TYPE := 605;
BEGIN
    DELETE FROM    item
        WHERE      ordid = v_ordid;
END;
```

이름 지정 규약

```
DECLARE
  order_date ord.orderdate%TYPE;
  ship_date  ord.shipdate%TYPE;
  v_date DATE := SYSDATE;
BEGIN
  SELECT orderdate, shipdate
  INTO   order_date, ship_date
  FROM   ord
  WHERE  shipdate = v_date;
END;
```

이름 지정 규약

PL/SQL 변수명과 데이터베이스 열명을 구분해 주는 이름 지정 규약을 따릅니다. WHERE 절에서 모호성을 회피합니다.

- 데이터베이스 열과 식별자는 다른 이름을 가져야 합니다.
- PL/SQL이 테이블의 열을 첫번째로 조사하기 때문에 구문 오류가 일어날 수 있습니다.

슬라이드에서 보여준 예는 다음과 같이 정의됩니다. 출하 일자가 오늘인 경우, ORD 테이블에서 주문일과 출하 일자를 읽어 들입니다.

PL/SQL은 식별자가 데이터베이스내의 열인지 아닌지를 먼저 확인하고, 아니면 PL/SQL의 식별자가 되는 것으로 가정합니다.

주: SELECT 절에서 임의의 식별자는 데이터베이스 열 이름이어야 하기 때문에 SELECT 절에서 모호성의 가능성은 없습니다. INTO 절에서 식별자는 PL/SQL 변수여야 하기 때문에 INTO 절에서 모호성의 가능성은 없습니다. WHERE 절에서만 혼란의 가능성이 있습니다. NO_DATA_FOUND와 다른 예외에 대한 자세한 내용은 다음 장에서 설명됩니다.

COMMIT과 ROLLBACK 문장

- COMMIT 또는 ROLLBACK을 수행하기 위해 처음에 DML 명령어로 트랜잭션을 시작합니다.
- 트랜잭션을 명시적으로 종료시키기 위해 COMMIT 과 ROLLBACK SQL 문장을 사용합니다.

트랜잭션 제어

COMMIT 과 ROLLBACK SQL 문장으로 트랜잭션 논리를 제어함으로써 데이터베이스를 영구적으로 변경하게 합니다. 오라클 서버에서와 마찬가지로, DML 트랜잭션은 COMMIT 또는 ROLLBACK 을 수행한 다음에 시작하고 성공적인 COMMIT 또는 ROLLBACK 다음에 종료합니다. 이 작업은 PL/SQL 블록에서 발생할 수 있거나 호스트 환경(예를 들면, SQL*Plus 섹션 종료는 자동적으로 미결 트랜잭션을 커밋) 에서 이벤트의 결과로서 발생할 수 있습니다.

COMMIT 문장

COMMIT 은 데이터베이스에 대한 모든 미결 변경을 영구화함으로써 현재 트랜잭션을 종료합니다.

구문

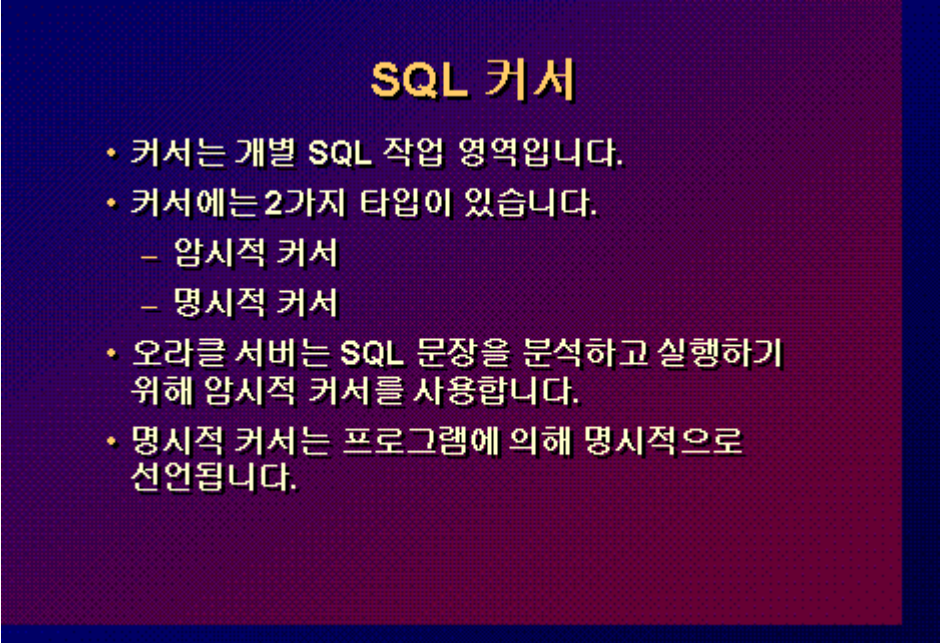
```
COMMIT [WORK];
```

```
ROLLBACK [WORK];
```

여기서: WORK ANSI 표준을 따르기 위해 사용합니다.

주: 호스트 환경마다 그 사용에 대해 약간의 제한이 있을 수 있지만 트랜잭션 제어 명령은 PL/SQL 에서 모두 유효합니다. 블록에서 또한 명시적으로 잠금 명령 (LOCK TABLE 과 SELECT ... FOR UPDATE 같은)을 이용할 수 있습니다. (후속 장에서 FOR-

UPDATE 에 대한 자세한 내용은 설명할 것입니다.) 그것들은 트랜잭션 종료까지 영향을 미치며 하나의 PL/SQL 블록은 하나의 트랜잭션을 반드시 의미하지는 않습니다.



SQL 커서

- 커서는 개별 SQL 작업 영역입니다.
- 커서에는 2가지 타입이 있습니다.
 - 암시적 커서
 - 명시적 커서
- 오라클 서버는 SQL 문장을 분석하고 실행하기 위해 암시적 커서를 사용합니다.
- 명시적 커서는 프로그램에 의해 명시적으로 선언됩니다.

SQL 커서

SQL 문장을 실행할 때마다 오라클 서버는 명령이 분석되고 실행되는 곳에서 메모리 영역을 개방합니다. 이 영역은 **cursor** 라 불립니다. 블록의 실행 부분이 SQL 문장을 실행할 때, PL/SQL 은 SQL 식별자를 가지는 암시적 커서를 생성합니다. PL/SQL 은 자동적으로 이 커서를 관리합니다. 명시적 커서는 명시적으로 선언되고 프로그래머에 의해 명명됩니다. PL/SQL 커서에 적용될 수 있는 이용 가능한 4 가지 속성이 있습니다.

주: 명시적 커서에 대한 자세한 내용은 후속 장에서 논합니다.

자세한 내용은 *PL/SQL User's Guide and Reference, Release 7.3* 또는 8.0, “Interaction with Oracle.”를 참조하십시오.

SQL 커서 속성

SQL 커서속성을 사용하여, SQL문장의 결과를 테스트 할 수 있습니다.

SQL%ROWCOUNT	가장 최근의 SQL문장에 의해 영향을 받은 행의 수(정수값)
SQL%FOUND	가장 최근의 SQL문장이 하나 또는 그 이상의 행에 영향을 미친다면 부울 속성은 TRUE로 평가됩니다.
SQL%NOTFOUND	가장 최근의 SQL문장이 어떤 행에도 영향을 미치지 않았다면 부울 속성은 TRUE로 평가됩니다
SQL%ISOPEN	PL/SQL이 실행된 후에 즉시 암시적 커서를 닫기 때문에 항상 FALSE로 평가됩니다.

SQL 커서 속성

SQL 커서 속성은 암시적 커서가 마지막으로 사용된 때 일어난 일을 평가할 수 있도록 해 줍니다. PL/SQL 문장에서 이 속성을 함수처럼 사용합니다. 이것들은 SQL 문장에서 사용할 수 없습니다. 데이터 조작 문장의 실행에 대한 정보를 얻기 위해 블록의 예외 섹션에서, SQL%ROWCOUNT, SQL%FOUND, SQL%NOTFOUND, 그리고 SQL%ISOPEN 속성을 사용할 수 있습니다. PL/SQL은 예외를 리턴하는 SELECT 문장과 달리, 아무런 행에 영향을 미치지 않는 DML 문장을 실패한 것으로 간주하지 않습니다.

SQL 커서속성

ITEM 테이블에서 지정된 명령 번호를 가지는 행을 삭제합니다. 삭제된 행의 수를 인쇄 합니다.

예

```
VARIABLE rows_deleted
DECLARE
  v_ordid NUMBER := 605;
BEGIN
  DELETE FROM item
  WHERE   ordid = v_ordid;
         :rows_deleted := SQL%ROWCOUNT
         || ' rows deleted. ');
END;
PRINT rows_deleted
```

제 21 장 IF 문

***** 부서/직원 리포트 *****

부서 : 부서번호	부서명	위치
10	ACCOUNTING	NEW YORK

사원 : 사번	이름	급여	급여 등급
7871	Anderson	2000	***
7890	Andrew	700	*
7870	Scott	2400	****
7866	David	900	*
7891	Sylvia	3200	*****

2 명의 급여가 인상되었습니다.

위와 같은 리포트를 뽑아내고 \$1000 보다 적은 급여를 받는 직원의 급여를 10%인상하는 프로그램을 작성하여 봅시다.

```
ACCEPT p_deptno PROMPT 'Please enter the department number : '
```

```
DECLARE
```

```
  v_deptno    NUMBER(2);
  v_dname     CHAR(14);
```



```

v_loc          CHAR(13);
v_empno        emp.empno%TYPE;
v_ename        emp.ename%TYPE;
v_sal          emp.sal%TYPE;
v_grade        salgrade.grade%TYPE;
v_dno          emp.deptno%TYPE := &p_deptno;
CURSOR emp_cursor IS
    SELECT empno, ename, sal
    FROM emp
    WHERE deptno = &p_deptno;
e_no_emp       EXCEPTION;

BEGIN
    -- print out the report title
    DBMS_OUTPUT.PUT_LINE('*****부서/직원 리포트*****');
    SELECT deptno, dname, loc
    INTO v_deptno, v_dname, v_loc
    FROM dept
    WHERE deptno = v_dno;
    DBMS_OUTPUT.PUT_LINE('부서 : 부서번호   부서명           위치');
    DBMS_OUTPUT.PUT_LINE('      ' || v_deptno || '      ' || v_dname ||
        '      ' || v_loc || '      ');
    DBMS_OUTPUT.PUT_LINE('사원 : 사번   이름   급여   급여등급');
    OPEN emp_cursor;
    FETCH emp_cursor INTO v_empno, v_ename, v_sal;
    /* according to the salary amount,
       print out the stars */
    WHILE emp_cursor%FOUND LOOP
        SELECT grade
        INTO v_grade
        FROM salgrade
        WHERE v_sal > losal AND v_sal < hisal;
        DBMS_OUTPUT.PUT('      ' || v_empno || '      ' || v_ename ||
            '      ' || v_sal || '      ');

        FOR I in 1..v_grade LOOP
            DBMS_OUTPUT.PUT('* ');

```

```

    END LOOP;

    DBMS_OUTPUT.PUT_LINE(' ');

    FETCH emp_cursor INTO v_empno, v_ename, v_sal;

    END LOOP;

    CLOSE emp_cursor;

    UPDATE emp
        SET      sal = sal * 1.1
        WHERE    sal < 1000;

    IF SQL%ROWCOUNT = 0 THEN
        RAISE e_no_emp;
    ELSE
        DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || '명의 급여가 ' ||
                               '인상되었습니다. ');

    END IF;

    COMMIT;

EXCEPTION

    WHEN no_data_found THEN
        DBMS_OUTPUT.PUT_LINE('There is no matching data. ');

    WHEN too_many_rows THEN
        DBMS_OUTPUT.PUT_LINE('More than one matching data. ');

    WHEN e_no_emp THEN
        DBMS_OUTPUT.PUT_LINE('No employee with a smaller ' ||
                               'than $1000. ');

    --in case of any other exception

    WHEN others THEN
        DBMS_OUTPUT.PUT_LINE('Some other error occurred. ');

END;

```

구문

단순 IF 문장:

```
IF v_ename = 'OSBORNE' THEN
    v_mgr := 22;
END IF;
```

PL/SQL IF 문장 구조는 다른 절차적인 언어의 IF 문장구조와 유사합니다.

구문에서

이것은 IF 표현식이 TRUE를 생성할 때만 일련의 문장들을 실행하도록 합니다.

관련시키는 절입니다

그것들은 몇 개의 중첩 IF, ELSE 와 ELSIF 를 포함하는 IF 문장을 포함할 수 있습니다.

또는 NULL 을 생성하고 이때 ELSIF 키워드는 추가적인 조건을 작성하도록 합니다.

ELSE 위의 조건들을 다 만족시키지 못하였을 때 다음의 문장들이 수행됩니다.

IF-THEN-ELSE 문장

주문일자와 출하일자 사이가 5일 미만인 경우에 주문 플래그를 설정합니다.

```
...  
IF v_shipdate - v_orderdate < 5 THEN  
    v_ship_flag := 'Acceptable';  
ELSE  
    v_ship_flag := 'Unacceptable';  
END IF;  
...
```

예

종업원 이름이 King 이라면 관리자로 작업을 설정합니다. 종업원 이름이 King 이 아니면 Clerk 으로 설정합니다.

```
IF v_ename = 'KING' THEN  
    v_job := 'MANAGER';  
ELSE  
    v_job := 'CLERK';  
END IF;
```

IF-THEN-ELSIF 문장

주어진 입력 값에 대해, 계산 값을 리턴합니다.

예

```
...  
IF v_start > 100 THEN  
    v_start := 2 * v_start;  
ELSIF v_start >= 50 THEN  
    v_start := .5 * v_start;  
ELSE  
    v_start := .1 * v_start;  
END IF;  
...
```

IF-THEN-ELSIF 문장

가능하면, 중첩 IF 문장 대신에 ELSIF 절을 사용합니다. 코드를 읽고 이해 하기가 더 쉬우며 로직은 정확하게 식별됩니다. ELSE 절 안의 작업이 순수하게 다른 IF 문장으로 구성된다면, 이것은 ELSIF 절을 사용하는 것이 더욱 편리합니다. 조건과 수행이 각각 종료시에 중첩 END IF 에 대해 일일이 요구하지 않음으로써 코드를 더 명확하게 만들어 줍니다.

예

```
IF condition1 THEN
    statement1;
ELSIF condition2 THEN
    statement2;
ELSIF condition3 THEN
    statement3;
END IF;
```

위의 IF-THEN-ELSIF 문장 예는 다음과 같이 더 자세히 정의됩니다:
입력된 주어진 값에 대해, 계산 값을 리턴합니다. 입력 값이 100 이상이면, 이때 계산 값은 입력 값의 두 배가 됩니다. 입력 값이 50 과 100 사이라면, 이때 계산 값은 시작 값의 10%가 됩니다.
주: null 값을 포함하는 임의의 수 표현식은 null 로 평가합니다.

논리 테이블

비교 연산자로 단순 부울 조건을 구축합니다.

AND	TRUE	FALSE	NULL	OR	TRUE	FALSE	NULL	NOT	
TRUE	TRUE	FALSE	NULL	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	NULL	FALSE	TRUE
NULL	NULL	FALSE	NULL	NULL	TRUE	NULL	NULL	NULL	NULL

비교 연산자와 부울 조건

논리 연산자 AND, OR, NOT 을 가지고 단순한 부울 조건을 조합함으로써 복잡한 부울 조건을 구축할 수 있습니다. 슬라이드에 보여진 논리 테이블에서, FALSE 는 AND 조건에 대해 우선순위를 갖고 TRUE 는 OR 조건에 대해 우선순위를 갖습니다. AND 는 양쪽 연산자가 TRUE 일 경우에만 TRUE 를 반환합니다. OR 은 양쪽 연산자가 FALSE 일 경우만 FALSE 를 반환합니다. NULL AND TRUE 는 첫 번째 연산자가 TRUE 로 평가되는지 아닌지 알려지지 않았기 때문에 항상 NULL 로 평가됩니다.

주: NULL 의 부정(NOT NULL)은 null 값들이 모호하기 때문에 null 값이 됩니다.

부울 조건		
각각의 경우에서 V_FLAG의 값은 무엇입니까?		
v_flag := v_reorder_flag AND v_available_flag;		
V_REORDER_FLAG	V_AVAILABLE_FLAG	V_FLAG
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
NULL	TRUE	NULL
NULL	FALSE	FALSE

논리 조건 구축

AND 로직 테이블은 위의 슬라이드에서 부울 조건에 대한 가능성을 평가하는데 도움을 줄 수 있습니다.

반복 제어: LOOP 문

- Loop 문은 문장 또는 일련의 문장들을 여러 번 반복합니다.
- 3가지 루프 유형이 있습니다:
 - Basic loop
 - FOR loop
 - WHILE loop



반복적인 제어 : LOOP 문장

PL/SQL 은 문장 또는 일련의 문장들을 반복하기 위해 루프를 구조화하기 위한 많은 편의를 제공합니다.

루프문은 제어 구조의 두 번째 유형입니다.

- 조건 없이 반복적인 작업을 제공하기 위한 Basic 루프
- 카운트를 기본으로 작업의 반복 제어를 제공하기 위한 FOR 루프
- 조건을 기본으로 작업의 반복 제어를 제공하기 위한 WHILE 루프
- 루프를 종료하기 위한 EXIT 문장

자세한 내용은

PL/SQL *User's Guide and Reference, Release 7.3* 또는 *8.0*, “Control Structures”를 참조하십시오.

주 : FOR LOOP 의 다른 타입, [커서 FOR LOOP](#) 는 후속 장에서 논합니다.

기본 루프

구문

```
LOOP                                -- delimiter
  statement1;                       -- statements
  . . .                             -- EXIT statement
  EXIT [WHEN condition];            -- delimiter
END LOOP;
```

```
where:  condition    is a Boolean variable or
                    expression (TRUE, FALSE,
                    or NULL);
```

기본 루프

LOOP 와 END LOOP 키워드 사이에 일련의 문장들을 포함하는 LOOP 문장의 가장 단순한 형태는 기본(또는 basic) 루프입니다. 실행 흐름의 END LOOP 문장에 도달할 때 마다, 제어는 해당 LOOP 문장으로 되돌려 줍니다. 기본 루프는 루프에 들어갈 때 조건이 이미 일치했다 할지라도 적어도 한번은 문장이 실행됩니다.

- 기본 루프를 통해 문장을 반복합니다.
- EXIT 문장이 없으면, 루프는 무한이 됩니다.

EXIT 문장

EXIT 문장을 사용하여 루프를 종료할 수 있습니다. 제어는 END LOOP 문장 후의 다음 문장으로 전달됩니다. IF 문장 내에서의 문장으로서 또는 ,루프 내에 있는 독립적인 문장으로서 EXIT 를 사용할 수 있습니다. EXIT 문장은 루프 내부에 위치해야 합니다. 후자인 경우에 , 루프의 조건적인 종료를 허용하기 위해 WHEN 절을 첨부할 수 있습니다. EXIT 문장에 직면할 때는, WHEN 절에서 조건이 평가됩니다. 조건이 TRUE 를 리턴하면, 루프는 끝나고 루프 후의 다음 문장으로 제어를 전달합니다. 기본 루프는 복수 EXIT 문장을 포함할 수 있습니다.

기본 루프

예

```
DECLARE
  v_ordid    item.ordid%TYPE := 101;
  v_counter  NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```

기본 루프

슬라이드에서 보여 준 기본 루프 예는 다음과 같이 정의됩니다.

주: 기본 루프는 조건이 루프 안에 들어갈 때 만족되었다 할지라도 적어도 한번은 그 문장을 수행합니다.

FOR 루프

구문

```
FOR counter in [REVERSE]
  lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- 반복되는 수를 정하여 사용하려면 FOR 루프를 사용합니다.
- 인덱스를 선언하지 마십시오. 이것은 암시적으로 선언됩니다.

FOR 루프는 기본 루프와 동일한 일반 구조를 가집니다. 그리고, PL/SQL 이 수행할 반복되는 수를 정하기 위해 LOOP 키워드의 앞에 제어 문장을 사용합니다.

구문에서,

counter 상단이나 하단 바운드에 도달할 때까지 루프를 계속 반복함으로써
1 씩 자동적으로 증가되거나 감소되는(감소는 REVERSE 키워드가 사용된다면)값을 가진
암시적으로 선언된 정수입니다.

REVERSE 상단 바운드에서 하단 바운드까지 반복함으로써 인덱스가 감소되도록
합니다. 하위 값(하단 바운드)는 여전히 처음에 참조됨을 주의하십시오.

lower_bound 인덱스 값의 범위에 대한 하단 바운드를 지정합니다.

upper_bound 인덱스 값의 범위에 대한 상단 바운드를 지정합니다.

카운터를 선언하지 마십시오. 이것은 정수로 암시적으로 선언됩니다.

주: 일련의 문장들은 두 바운드에 의해 카운터가 결정되고 증가될 때마다 실행됩니다. 루프
범위의 하단 바운드와 상단 바운드는 리터럴, 변수, 표현식이 될 수 있지만 정수로
평가해야 합니다. 루프 범위 하단 바운드가 상단 바운드보다 더 큰 정수를 평가한다면,
일련의 문장들은 실행되지 않습니다. 예를 들면, 다음에서 문장은 한번만 실행됩니다.

```
FOR i IN 3..3 LOOP statement1; END LOOP;
```

FOR 루프

주문 번호 101 에 대해 10개의 새로운 항목을
입력합니다.

예

```
DECLARE
  v_ordid   item.ordid%TYPE := 101;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, i);
  END LOOP;
END;
```

FOR 루프

주: LOOP 문장의 하단과 상단 바운드는 꼭 숫자 리터럴이 될 필요가 없습니다. 그것들은 숫자값으로 변환되는 표현식이 될 수도 있습니다.

예

```
DECLARE
  v_lower  NUMBER := 1;
  v_upper  NUMBER := 100;
BEGIN
  FOR i IN v_lower..v_upper LOOP
    . . .
  END LOOP;
END;
```

WHILE 루프

구문

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

← Condition is evaluated at the beginning of each iteration.

조건이 TRUE인 동안 문장을 반복하기 위해 WHILE 루프를 사용합니다.

WHILE 루프

제어 조건이 더 이상 TRUE 가 아닐 때까지 일련의 문장들을 반복하기

위해 WHILE 루프를 사용할 수 있습니다. 조건은 반복이 시작될 때에 평가됩니다.
루프는 조건이 FALSE 일 때 종료됩니다. 루프를 시작할 때 조건이 FALSE 라면 , 이때는 더 이상 반복을 수행하지 않습니다

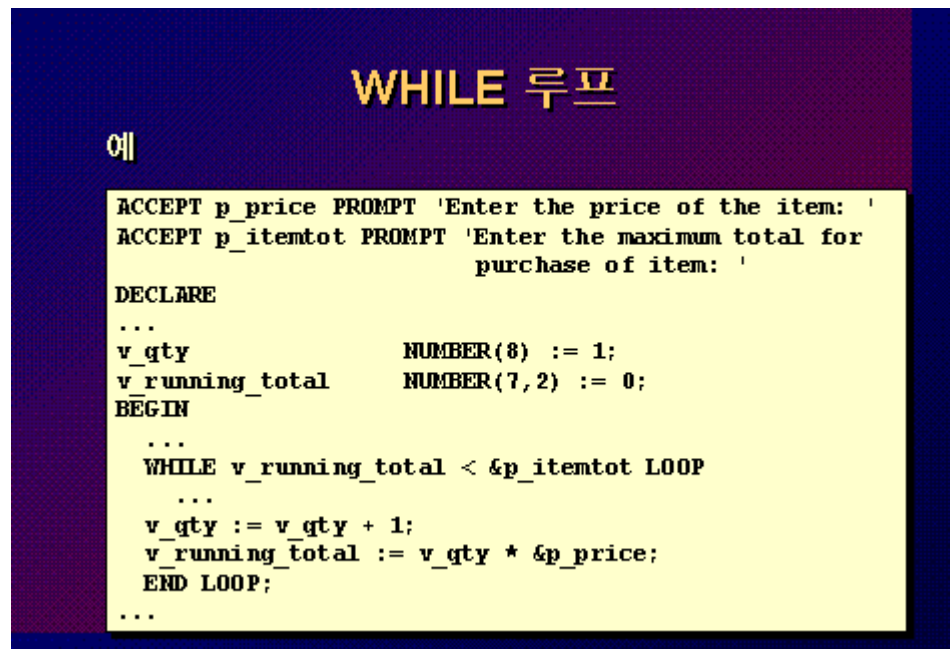
#구문에서,

condition 부울 변수 또는 표현식입니다 (TRUE, FALSE, 또는 NULL).

statement 하나 또는 그 이상의 PL/SQL 또는 SQL 문장이 될 수 있습니다.

조건 안에 포함된 변수가 루프의 안에서 변경될 수 없다면 이때 조건은 TRUE 를 계속 유지하고 루프는 종결되지 않습니다.

주: 조건이 NULL 을 리턴하면, 루프는 실행되지 않고 다음 문장으로 제어를 전달합니다



WHILE 루프

슬라이드 예에서 양(quantity)이 항목에서 소비로 허용된 최대 가격보다 더 이상 적지 않을 때까지 양(quantity)은 루프가 계속 반복되면서 증가합니다

중첩 루프와 레이블

```
...  
BEGIN  
  <<Outer_loop>>  
  LOOP  
    v counter := v counter+1;  
    EXIT WHEN v counter>10;  
    <<Inner_loop>>  
    LOOP  
      ...  
      EXIT Outer_loop WHEN total_done = 'YES';  
      -- Leave both loops  
      EXIT WHEN inner done = 'YES';  
      -- Leave inner loop only  
      ...  
    END LOOP Inner_loop;  
    ...  
  END LOOP Outer_loop;  
END;
```

중첩 루프와 레이블

여러 단계로 루프를 중첩할 수 있습니다. WHILE 루프 내에서 FOR 루프를, FOR 루프 내에서 WHILE 루프를 중첩할 수 있습니다. 대개 중첩 루프가 종결되면 예외가 발생하지 않는 한 둘러싸는 루프가 종결되지 않습니다. 레이블 명은 다른 식별자들과 동일한 규칙을 따릅니다. 레이블은 같은 라인 또는 다음 라인에서 문장 앞에 위치됩니다. 레이블 구분 문자(<<label>>) 안에 LOOP 라는 글자 앞에 레이블을 위치시킴으로서 루프를 레이블시킵니다. 루프가 레이블이 되면 END LOOP 문장 후에 루프 이름을 선택적으로 쓸 수 있습니다.

제 22 장 PL/SQL 블록 구문과 지침

PL/SQL 블록 구문과 지침

- 문장은 몇 라인 이상 계속될 수 있습니다.

식별자

- 30 문자까지 포함 할 수 있습니다.
- 더블 인용부호로 둘러싸지 않으면 예약어를 포함할 수 없습니다.
- 알파벳 문자로 시작해야 합니다.
- 데이터베이스 테이블 열 이름과 동일 이름을 가져서는 안됩니다.

PL/SQL 은 SQL 의 확장이기 때문에, SQL 에 적용하는 일반적인 구문은 PL/SQL 언어에 대해서도 적용 가능합니다.

- 문법적인 요소(예를 들면, 식별자 또는 리터럴)는 문법적인 요소의 일부로 혼돈될 수 없는 하나 또는 그 이상의 공백으로 분리될 수 있습니다.
- 문장은 줄에 나누어 분할될 수 있지만 키워드는 분할되어서는 안됩니다.

구분 문자

구분 문자는 PL/SQL 에 대해 특수한 의미를 가지는 단순 상징이거나 혼합 상징입니다.

단순 상징 기호	의 미	혼합 상징 기호	의 미
+	추가 연산자	<>	관계형 연산자
-	빼기/부정 연산자	!=	관계형 연산자
*	곱하기 연산자		접속 연산자
/	나누기 연산자	--	단일 라인 주석 지시자
=	관계형 연산자	/*	주석 구분문자 시작
@	원격 액세스 지시자	*/	주석 구분문자 종료

;	문장 종결자	:=	지정 연산자
---	--------	----	--------

자세한 내용은 *PL/SQL User's Guide and Reference, Release 7.3* 또는 *8.0*,
“Fundamentals.”를 참조하십시오.

식별자

식별자는 상수, 변수, 예외, [커서](#), [커서](#) 변수, 서브프로그램, 패키지를 포함하는 PL/SQL
프로그램 항목과 요소를 명명하기 위해 사용됩니다.

- 식별자는 30문자까지 포함할 수 있지만 알파벳 문자로 시작해야 합니다.
- 블록에서 사용된 테이블 열 이름과 동일한 이름으로 식별자 이름을 선택해서는 안 됩니다. PL/SQL 식별자가 동일 SQL 명령에 있고 열로 동일 이름을 가지고 있다면, 이때 오라클은 참조 중인 열로 간주합니다.
- 예약어는 더블 인용 부호(예를 들면 "SELECT")에 둘러싸여 있지 않으면 식별자로서 사용될 수 없습니다.
- 예약어는 읽기 쉽도록 대문자로 쓰여져야 합니다.

예약어 목록에 대한 자세한 내용은

PL/SQL User's Guide and Reference, Release 7.3 또는 *8.0*, “Appendix”를 참조하십시오

주석 코드

- 두개의 대쉬(-)를 앞에 붙여 단일 라인을 설명하는 주석을 만듭니다.
- 기호 /*와 */사이에 다중 라인 주석을 넣습니다.

예

```
...
v_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
    monthly salary input from the user */
  v_sal := v_sal * 12;
END; -- This is the end of the transaction
```

주석 코드

각 단계를 문서화하고 디버깅을 돕기 위해 코드에 주석을 합니다. 주석이 단일 라인에 있으면 두 대쉬(--)로, 주석 범위가 여러 줄이라면 기호/* 와 */사이에, PL/SQL 코드 주석을 다십시오. 주석은 철저하게 정보를 제공해야 하고 기능적인 논리 또는 데이터에 대한 어떤 조건 또는 기능을 강요해서는 안됩니다. 좋은 주석은 코드를 읽기 쉽게 하고 코드 유지를 위해 매우 중요합니다.

예

월 급여에서 일년급여를 계산합니다.

```
...  
    v_sal NUMBER(9,2);  
BEGIN  
    /* Compute the annual salary based on  
    the monthly salary input from the user */  
    v_sal := v_sal*12;  
END; -- This is the end of the transaction
```


PL/SQL에서 SQL 함수

- 사용 가능:

- 단일 행 숫자
- 단일 행 문자
- 데이터형 변환
- 날짜

- 사용 불가:

- GREATEST
- LEAST
- DECODE
- 그룹 함수

} SQL과 동일

PL/SQL 에서 SQL 함수

SQL 에서 이용 가능한 대부분의 함수는 PL/SQL 표현식에서도 유효합니다.

- 단일 행 숫자 함수
- 단일 행 문자 함수
- 데이터형 변환 함수
- 데이트 함수
- 그 밖의 함수

아래 함수는 절차적인 문장에서는 사용 불가능합니다.

- GREATEST, LEAST, 와 DECODE.
- 그룹 함수: AVG, MIN, MAX, COUNT, SUM, STDDEV 과 VARIANCE. 그룹 함수는 테이블에서 행 그룹에 적용되므로 PL/SQL 블록에 있는 SQL 문장에서만 이용 가능 합니다.

예

NUMBER_TABLE PL/SQL 테이블에서 저장된 모든 숫자의 합을 계산합니다.

이 예는 [컴파일](#) 오류를 생성합니다.

```
v_total := SUM(number_table);
```

중첩 블록과 변수 범위

- 문장은 실행 명령이 허용되는 곳 어디에서든지 중첩될 수 있습니다.
- 중첩 블록은 하나의 문장이 됩니다.
- 예외 섹션도 중첩 블록을 포함할 수 있습니다.
- 객체 범위는 객체를 참조할 수 있는 프로그램 영역입니다.

중첩 블록

SQL 에 비해 장점 가운데 하나는 PL/SQL 에서 문장이 중첩될 수 있다는 것입니다. 실행 문장이 허용되는 곳 어디에서든지 블록을 중첩할 수 있으므로, 중첩 블록 문장을 만드십시오. 그래서 실행 블록의 일부를 더 작은 블록으로 쪼갤 수 있습니다. 예외 섹션도 또한 중첩 블록을 포함할 수 있습니다.

변수 범위

객체의 범위는 객체를 조회할 수 있는 프로그램의 영역입니다.

선언된 변수를 실행 섹션에서 참조할 수 있습니다.

중첩 블록과 변수 범위

예

```
...  
  x BINARY_INTEGER;  
BEGIN  
  ...  
  DECLARE  
    y NUMBER;  
  BEGIN  
    ...  
  END;  
  ...  
END;
```

Scope of x

Scope of y

중첩 블록과 변수 범위

슬라이드의 중첩 블록에서 y 로 명명된 변수는 x 로 명명된 변수를 참조할 수 있습니다. 그러나 변수 x 는 변수 y 를 참조할 수 없습니다. 중첩 블록에서 변수 y 가 외부 블록에서의 변수 x 와 동일한 이름으로 주어지면 변수 y 의 값은 중첩 블록의 안에서만 유효합니다.

범위

식별자의 범위는 식별자를 참조할 수 있는 프로그램 단위의 영역(블록, 서브프로그램, 패키지)입니다.

가시성

식별자는 한정하지 않는 (unqualified) 이름을 사용하여 식별자를 참조할 수 있는 영역에서만 참조 가능합니다.

변수 범위 결정

연습

```
---
DECLARE
  V_SAL          NUMBER(7,2) := 60000;
3 V_COMM        NUMBER(7,2) := V_SAL / .20;
  V_MESSAGE      VARCHAR2(255) := ' eligible for commission';
BEGIN ...

  DECLARE
    V_SAL          NUMBER(7,2) := 50000;
    2 V_COMM        NUMBER(7,2) := 0;
    V_TOTAL_COMP   NUMBER(7,2) := V_SAL + V_COMM;
  BEGIN ...
    1 V_MESSAGE := 'CLERK not' || V_MESSAGE;
  END;

    4 V_MESSAGE := 'SALESMAN' || V_MESSAGE;
END;
```

연습

슬라이드의 PL/SQL 블록을 평가(evaluate)합니다. 범위 규칙에 따라 각각의 아래 값을 결정하십시오:

1. 서브 블록에서 V_MESSAGE 의 값.
2. 서브 블록에서 V_COMM 의 값.
3. 메인 블록에서 V_COMM 의 값.
4. 메인 블록에서 V_MESSAGE 의 값

PL/SQL에서 연산자

예

- 루프를 위한 인덱스 증가.

```
v_count := v_count + 1;
```

- 부울 플래그 값 설정.

```
v_equal := (v_n1 = v_n2);
```

- 종업원 번호가 값을 포함하는지 여부를 검증합니다

```
v_valid := (v_empno IS NOT NULL);
```

PL/SQL에서 연산자

Null로 작업 중일 때, 아래 규칙을 명심함으로써 몇 가지 일반적인 실수를 피할 수 있습니다.

- 관계 null 비교는 항상 NULL을 산출합니다.
- Null에 논리연산자 NOT을 적용하면 NULL을 산출합니다.
- 조건 제어 문장에서, 조건이 NULL을 산출한다면, 관련된 문장은 실행되지 않습니다

프로그래밍 지침

다음을 이용하여 코드 유지를 쉽게 합니다.

- 주석으로 코드 문서화
- 코드에 대한 대소문자 규약 개발
- 식별자와 다른 객체에 대한 이름 지정 규약 개발
- 들여쓰기를 이용하여 쉽게 읽기 향상

프로그래밍 지침

PL/SQL 블록을 개발할 때, 명확한 코드 생성과 유지를 경감하기 위하여 프로그래밍 지침을 수행합니다.

코드 규약

아래 테이블은 이름이 있는 객체에서 키워드를 구별하기 위해 대문자 또는 소문자 코드 쓰기에 유용한 지침을 제공합니다.

범주	대소문자 규약	예
SQL 명령문	대문자	SELECT, INSERT
PL/SQL 키워드	대문자	DECLARE, BEGIN, IF
데이터형	대문자	VARCHAR2, BOOLEAN
식별자와 매개변수	소문자	v_sal, emp_cursor, g_sal, p_empno
데이터베이스 테이블과 열	소문자	emp, orderdate, deptno

제 23 장 Cursor

***** 부서/직원 리포트 *****

부서 : 부서번호	부서명	위치
10	ACCOUNTING	NEW YORK

사원 : 사번	이름	급여	급여 등급
7871	Anderson	2000	***
7890	Andrew	700	*
7870	Scott	2400	****
7866	David	900	*
7891	Sylvia	3200	*****

2 명의 급여가 인상되었습니다.

위와 같은 리포트를 뽑아내고 \$1000 보다 적은 급여를 받는 직원의 급여를 10%인상하는 프로그램을 작성하여 봅시다.

```
ACCEPT p_deptno PROMPT 'Please enter the department number : '
```

```
DECLARE
```

```
    v_deptno    NUMBER(2);
    v_dname     CHAR(14);
    v_loc       CHAR(13);
    v_empno     emp.empno%TYPE;
    v_ename     emp.ename%TYPE;
    v_sal       emp.sal%TYPE;
    v_grade     salgrade.grade%TYPE;
    v_dno       emp.deptno%TYPE := &p_deptno;
```

```
CURSOR emp_cursor IS
```

```
    SELECLT empno, ename, sal
```

```
    FROM emp
```

```
    WHERE deptno = &p_deptno;
```

```
e_no_emp    EXCEPTION;
```

```
BEGIN
```

```
    -- print out the report title
```

```
    DBMS_OUTPUT.PUT_LINE('*****부서/직원 리포트*****');
```



```

EXCEPTION
    WHEN no_data_found THEN
        DBMS_OUTPUT.PUT_LINE('There is no matching data. ');
    WHEN too_many_rows THEN
        DBMS_OUTPUT.PUT_LINE('More than one matching data. ');
    WHEN e_no_emp THEN
        DBMS_OUTPUT.PUT_LINE('No employee with a smaller ' ||
                               'than $1000. ');
    --in case of any other exception
    WHEN others THEN
        DBMS_OUTPUT.PUT_LINE('Some other error occurred. ');
END;

```

커서

오라클 서버에 의해 실행되는 모든 SQL 문장은 개별 커서를 갖습니다.

- 암시적 커서 : 모든 DML과 PL/SQL SELECT 문장에 대해 선언됩니다.
- 명시적 커서 : 프로그래머에 의해 선언되고 이름 붙여집니다.

암시적, 명시적 커서

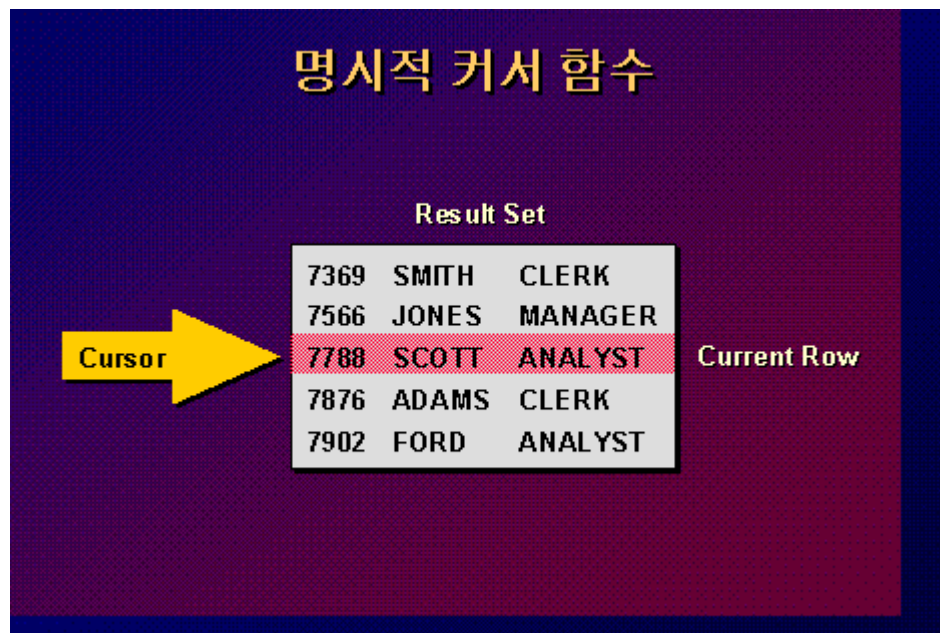
오라클 서버는 SQL 문장을 실행하고 처리한 정보를 저장하기 위해 Private SQL 영역이라 불리는 작업 영역을 사용합니다. private SQL 영역에 이름 붙이고 그것의 저장된 정보를 액세스하기 위해 PL/SQL 커서를 사용할 수 있습니다.

커서 유형	설명
-------	----

암시적	암시적 커서 는 하나의 행만 리턴하는 질의를 포함하여 모든 DML 과 PL/SQL SELECT 문장에 대해 PL/SQL 에 의해 암시적으로 선언됩니다.
명시적	하나 이상을 반환하는 모든 질의에 대해 명시적 커서 는 프로그래머에 의해 선언되고 이름 붙여지며 블록의 실행 섹션 안의 특정 명령문을 통해 조작됩니다.

암시적 커서

오라클 서버는 명시적으로 선언된 **커서**와 관련 없는 각 SQL 문장을 프로세스 하기 위해 **커서**를 암시적으로 개방합니다. PL/SQL 은 SQL **커서**로서 가장 최근의 암시적 **커서**를 참조할 수 있도록 해 줍니다. SQL **커서**를 제어하기 위해 OPEN, FETCH 그리고 CLOSE 문장을 사용할 수 없지만 가장 최근에 실행된 SQL 문장에 대한 정보를 얻기 위한 **커서** 속성을 사용할 수는 있습니다.



명시적 커서

다중 행 SELECT 문장에 의해 리턴되는 각 행을 개별적으로 처리하기 위해 명시적 **커서**를 사용합니다. 다중 행 질의에 의해 리턴된 행의 집합은 result set 이라 불립니다. 그것의 크기는 검색조건에 일치하는 행의 수입니다.

슬라이드에서 다이어그램은 result set 에서 current row 를 명시적 **커서**가 어떻게 가리키는가를 보여 줍니다. 이것은 프로그램이 한번에 한 행을 처리하는 것을 허용합니다.

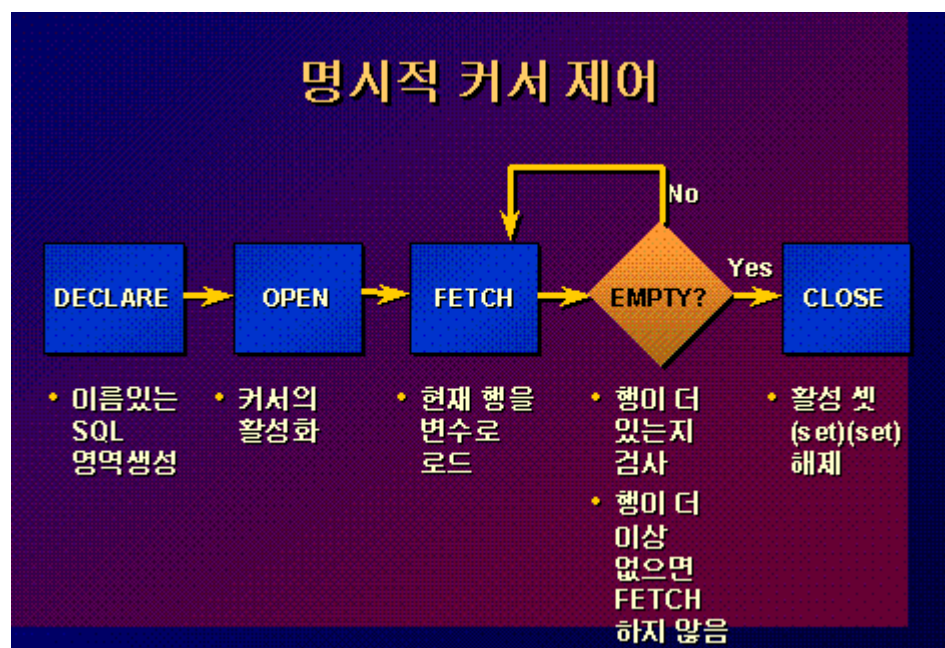
PL/SQL 프로그램은 질의에 의해 리턴된 행을 처리하고 그런 다음 **커서**를 닫습니다.

커서는 result set 에서 현재 위치를 표시합니다.

명시적 커서 함수

- 질의에 의해 리턴된 첫번째 행부터, 행 하나 하나씩 처리 할 수 있습니다.
- 현재 처리되는 행의 트랙을 유지합니다.
- 프로그래머가 PL/SQL 블록에서 수동으로 제어할 수 있습니다.

주: 암시적 커서에 대한 인출(fetch)은 배열 인출(fetch)이며, 두 번째 행의 존재는 여전히 TOO_MANY_ROWS 예외가 발생합니다. 그러므로, 다중 인출(fetch)을 수행하고 작업 영역에서 구문 분석된 질의를 재실행 하기 위해 명시적 커서를 사용할 수 있습니다.



명시적 커서 (계속)

커서에 대해 개념적인 이해는 하였으므로, 그것들을 사용하기 위한 단계를 살펴봅니다. 각 단계에 대한 구문은 다음에 있습니다.

4 가지 명령을 사용하는 명시적 커서 제어

1. 수행되기 위한 질의의 구조를 정의하고 이름을 지정함으로써 커서를 선언합니다.
2. 커서를 엽니다. OPEN 문장은 질의를 실행하고, 참조되는 임의의 변수를 바인드 합니다. 질의에 의해 식별된 행은 active set 이라 불리고 인출(fetch)가능합니다.
3. 커서에서 데이터를 인출(fetch)합니다. FETCH 문장은 커서에서 변수로 현재 행을

로드 합니다. 각 인출(fetch)은 활성 셋(set)에서 다음 행으로 그 포인터를 이동하도록 합니다. 그러므로 각 인출(fetch)은 질의에 의해 리턴되는 각기 다른 행을 액세스 합니다. 슬라이드에서 보여 준 흐름 다이어그램에서, 각 인출(fetch)시에 행이 존재하는 지를 검사합니다. 행이 발견되면 그것은 현재 행을 변수로 로드합니다. 그렇지 않으면 **커서**를 닫습니다.

4. **커서**를 닫습니다. CLOSE 문장은 행의 활성 셋(set)을 해제합니다. 이제 새로운 활성 셋(set)을 생성하기 위해 **커서**를 다시 열 수 있습니다

커서 선언

구문

```
CURSOR cursor_name IS  
select_statement;
```

- 커서 선언에서는 INTO 절을 쓰지 않습니다.
- 특정 시퀀스로 행을 처리해야 하면, 질의에서 **ORDER BY** 절을 사용합니다.

명시적 **커서** 선언

명시적으로 **커서**를 선언하기 위해 **CURSOR** 문장을 사용합니다.

질의 내에서 변수를 참조할 수 있지만, **CURSOR** 문장 전에 선언되어야 합니다

구문에서

cursor_name

PL/SQL 식별자입니다.

select_statement

INTO 절 없는 SELECT 문장입니다.

주: INTO 절은 FETCH 문장에 있기 때문에 **커서** 선언에서 INTO 절을 포함하지 마십시오.

커서 선언

예

```
DECLARE
  CURSOR c1 IS
    SELECT empno, ename
    FROM   emp;

  CURSOR c2 IS
    SELECT *
    FROM   dept
    WHERE  deptno = 10;
BEGIN
  ...
```

커서 선언 (계속)

하나씩 종업원을 읽어 들입니다.

```
DECLARE

  v_empno          emp.empno%TYPE;
  v_ename          emp.ename%TYPE;
  CURSOR c1 IS
    SELECT empno, ename
    FROM   emp;
BEGIN
  ...
```

주: 질의에서 변수를 참조할 수 있지만 **CURSOR** 문장 전에 선언합니다.

커서 열기

구분

```
OPEN cursor_name;
```

- 질의를 실행하고 활성 셋(set)을 생성하기 위해 커서를 엽니다.
- 질의가 아무 행도 리턴하지 않더라도 예외를 발생하지 않습니다.
- 인출(fetch) 후에 결과를 테스트하기 위해 커서 속성을 사용합니다.

OPEN 문장

질의를 수행하고 질의 검색조건을 충족하는 모든 행으로 구성된 결과 셋 (set) 을 생성하기 위해 커서를 엽니다. 커서는 이제 결과 셋(set)에서 첫번째 행을 가리킵니다.

구문에서

cursor_name 이전에 선언된 커서 이름입니다. OPEN 은 다음작업을 수행합니다.

1. 문맥 영역(context area)에 대해 동적으로 메모리를 할당하여 중요한 프로세싱 정보를 포함합니다.
2. SELECT 문장을 구문분석 합니다.
3. 입력변수를 바인드 합니다. - 즉, 그들의 메모리 주소를 획득하여 입력 변수에 대한 값을 설정합니다.
4. 결과 셋(set)을 식별합니다. - 즉, 검색 조건을 충족시키는 행의 집합입니다. OPEN 문장이 실행될 때 결과 셋(set)에 있는 행을 변수로 읽어 들이지 않습니다. 그대신 FETCH 문장이 행을 읽습니다.

5. 포인터는 활성 셋(set)에서 첫번째 행 앞에 위치합니다.

주: 커서가 개방될 때 질의가 아무 행도 리턴하지 않으면, PL/SQL 은 예외를 일으키지 않습니다. 그러나 인출(fetch)후에 커서의 상태를 테스트 할 수 있습니다. FOR UPDATE 절을 사용하는 커서를 선언할 경우에는, OPEN 문장은 그 행들을 잠급니다 (lock).

커서에서 데이터 인출(fetch)

구문

```
FETCH cursor_name INTO [variable1, variable2, ...]  
                        | record_name];
```

- 현재 행의 값을 출력변수로 읽어 들입니다.
- 동일 개수의 변수를 포함합니다.
- 열의 위치에 대응되도록 각 변수를 맞추십시오.
- 커서가 행을 갖고 있는지 테스트합니다

FETCH 문장

FETCH 문장은 결과 셋(set)에서 한번에 하나의 행을 읽어 들입니다. 각 인출(fetch) 후에 커서는 결과 셋(set)에서 다음 행으로 움직입니다.

#구문에서,

cursor_name 이전에 선언된 커서의 이름입니다.

variable 결과를 저장하기 위한 출력 변수입니다.

record_name 읽어 들인 데이터가 저장되는 레코드의 이름입니다. 레코드

변수는 %ROWTYPE 속성을 이용하여 선언 될 수 있습니다.

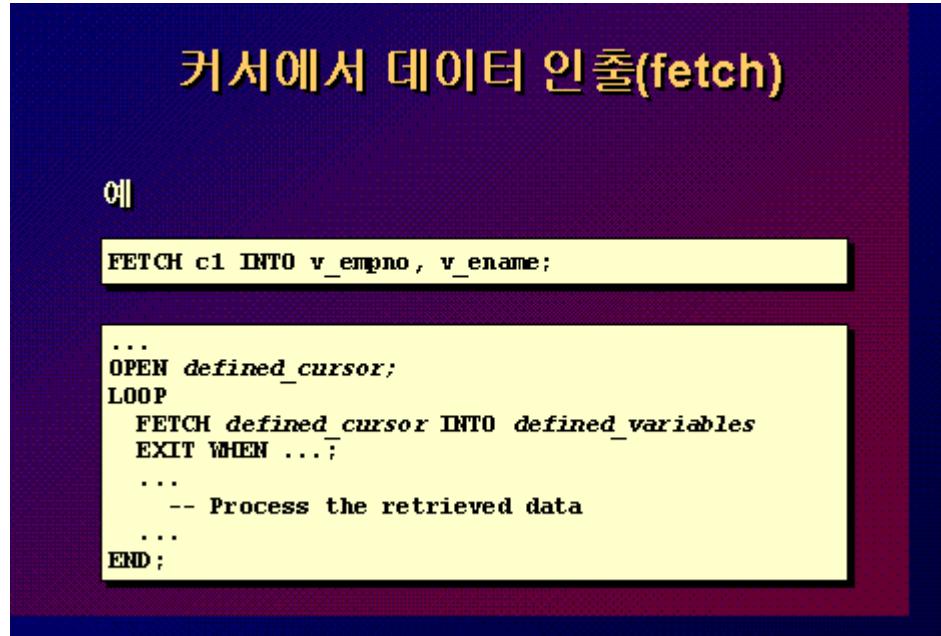
지침

- SELECT 문장의 열과 같은 개수의 변수를 FETCH 문장의 INTO 절에 포함시킵니다. 그리고 데이터 형은 같도록 하십시오.
- 열의 위치에 따라 대응되도록 각 변수를 일치시킵니다.
- 커서가 리턴할 행을 포함하는지 테스트합니다. FETCH 시 아무 값도 추출되지 않아도, 즉 활성 셋(set)에서 프로세스할 남겨진 행이 없는 경우에는 오류가 발생되지 않습니다.

주: FETCH 문장은 다음 작업을 이행합니다.

1. 활성 셋(set)에서 다음 행으로 포인터를 움직입니다.

2. 현재 행에 대한 정보를 출력 PL/SQL 변수로 읽어 들입니다.
3. 포인터가 활성 셋(set)의 끝에 위치하게 되면 **커서**는 FOR LOOP 를 빠져 나갑니다.



FETCH 문장 (계속)

현재 행의 값을 출력 변수로 읽어 들이기 위해 FETCH 문장을 사용할 수 있습니다.

인출(fetch) 후에 다른 문장들을 써서 변수를 조작할 수 있습니다. **커서**와 관련된 질에 의해 리턴되는 각 열 값에 대응하는 해당변수가 INTO 절에 있어야만 합니다. 또한 그들의 데이터 형은 같아야 합니다.

첫번째 10 명의 종업원을 하나씩 읽어 들입니다.

```

DECLARE
  v_empno          emp.empno%TYPE;
  v_ename          emp.ename%TYPE;
  CURSOR c1 IS
    SELECT empno, ename
    FROM   emp;
BEGIN
  OPEN c1;
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empno, v_ename;

```



```

...
END LOOP;
END ;

```

커서 닫기

구문

```
CLOSE   cursor_name;
```

- 행을 다 처리한 후에 커서를 닫습니다.
- 필요하다면 커서를 다시 OPEN 합니다.
- 일단 닫아진 커서에서 데이터를 인출(fetch)하려는 시도는 하지 마십시오.

CLOSE 문장

CLOSE 문장은 커서를 사용 할 수 없게 하고 결과 셋(set)은 정의가 해제됩니다.

SELECT 문장이 다 처리된 완성 후에 커서를 닫습니다. 필요하다면 커서를 다시 열 수 있습니다. 그러므로 활성 셋(set)을 여러 번 설정 할 수 있습니다.

cursor_name 이전에 선언된 커서의 이름입니다.

일단 닫아진 커서에서 데이터를 인출 (fetch) 하려는 시도는 하지 마십시오. 그렇지 않으면 INVALID_CURSOR 예외가 일어나게 됩니다.

주: CLOSE 문장은 context area 를 해제 합니다. 커서를 닫지 않고 PL/SQL 블록을 종료하는 것이 가능하다 할 지라도 리소스를 다시 사용 가능하게 하기 위해 명시적으로 선언된 임의의 커서를 닫는 습관을 들여야 합니다. 데이터베이스 매개변수 필드에서 OPEN_CURSORS 매개변수에 의해 결정되는 사용자마다 해당하는 커서의 수에는 최대 한계가 있습니다. 디폴트로 OPEN_CURSORS = 50 입니다.

```

...
FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empno, v_ename;

```



```

...
END LOOP;
CLOSE c1;
END;

```

명시적 커서 속성		
커서에 대한 상태 정보를 구합니다.		
속성	타입	기술
% ISOPEN	Boolean	커서가 열리면 TRUE를 리턴합니다.
% NOTFOUND	Boolean	가장 최근의 인출(fetch)이 행을 리턴하지 않으면 TRUE를 리턴합니다.
% FOUND	Boolean	가장 최근의 인출(fetch)의 행을 리턴하면 TRUE를 리턴합니다: % NOTFOUND 의 반대
% ROWCOUNT	Number	지금까지 리턴된 행의 총수를 리턴합니다.

명시적 커서 속성

명시적 커서로, 커서에 대해 상태 정보를 얻기 위한 4 가지 속성이 있습니다.

커서 또는 커서 변수에 붙을 때, 이 속성은 데이터 조작 문장의 실행에 대해 유용한 정보를 리턴합니다.

주: SQL 문장에서 직접적으로 커서 속성을 참조할 수 없습니다.

```

IF NOT c1%ISOPEN THEN
    OPEN c1;
END IF;
LOOP
    FETCH c1...

```

```

DECLARE
    v_empno emp.empno%TYPE;

```

```

v_ename                                emp.ename%TYPE;
CURSOR c1 IS
    SELECT empno, ename
    FROM    emp;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO v_empno, v_ename;
        EXIT WHEN c1%ROWCOUNT > 10 OR c1%NOTFOUND;
        ...
    END LOOP;
    CLOSE c1;
END;

```

매개변수와 커서

부서번호와 업무명을 WHERE 절로 전달합니다.
예

```

DECLARE
CURSOR c1
(v_deptno NUMBER, v_job VARCHAR2) IS
    SELECT empno, ename
    FROM    emp
    WHERE   deptno = v_deptno
    AND     job = v_job;
BEGIN
    OPEN c1(10, 'CLERK');
    ...

```

매개변수 데이터형은 스칼라 변수의 데이터형과 동일하지만, 크기는 제공하지 않습니다. 매개변수명은 커서 질의에서 참조하기 위한 것입니다. 다음 예에서, 두 변수와 커서가 선언됩니다. 커서는 두 매개변수로 정의됩니다.

```

DECLARE

job_emp            emp.job%TYPE := 'CLERK';

```

```
CURSOR c1 (v_deptno NUMBER, v_job VARCHAR2) is
SELECT ...
```

다음 문장은 둘 다 커서를 엽니다.

```
OPEN c1(10, job_emp);
OPEN c1(20, 'ANALYST');
```

FOR UPDATE 절

오늘 처리되었던 1000달러 금액에 대한 주문을
검색합니다.

예

```
DECLARE
CURSOR c1 IS
SELECT empno, ename
FROM emp
FOR UPDATE NOWAIT;
```

FOR UPDATE 절

주: 오라클 서버는 SELECT FOR UPDATE 에서 필요로 하는 행의 잠금을 얻을 수 없다면, 막연하게 기다립니다. SELECT FOR UPDATE 문장에서 NOWAIT 절을 사용할 수 있고 루프에서 잠금을 얻는데 실패하여 생기는 오류 코드를 테스트할 수 있습니다. 그러므로 PL/SQL 블록을 종료하기 전에 커서 OPEN 을 n 번 다시 시도할 수 있습니다. 대형 테이블이라 면, 테이블의 모든 행을 잠그기 위해 LOCK TABLE 문장을 사용함으로써 더 나은 성능을 얻을 수도 있습니다. 그러나 LOCK TABLE 을 사용할 때, WHERE CURRENT OF 절을 사용할 수 없고 WHERE column = identifier 를 사용해야 합니다. FOR UPDATE OF 절이 열을 참조 하는 것이 필수적이지는 않지만 이것은 더 쉽게 읽고 유지할 수 있게 하기 위해 추천되어집니다.

WHERE CURRENT OF 절

예

```
DECLARE
  CURSOR c1 IS
    SELECT ...
    FOR UPDATE NOWAIT;
BEGIN
  ...
  Open c1; fetch c1 into ... ;
  UPDATE ...
    WHERE CURRENT OF c1;
  ...
END LOOP;
COMMIT;
END;
```

WHERE CURRENT OF 절

명시적 커서에서 현재 행을 참조할 때 WHERE CURRENT OF 절을 사용합니다. 이를 통해서 명시적으로 ROWID를 참조하지 않고 현재 처리 중인 행을 갱신하고, 삭제할 수 있게 해 줍니다. 행을 OPEN 시에 잠기게 하기 위해서 커서 절의에 FOR UPDATE 절을 포함해야 합니다.

#구문에서,

cursor 선언된 커서의 이름입니다. 커서는 FOR UPDATE 절로 선언되어 있어야 합니다.

제 24 장 예외 처리

***** 부서/직원 리포트 *****

부서 : 부서번호	부서명	위치
10	ACCOUNTING	NEW YORK

사원 : 사번	이름	급여	급여 등급
7871	Anderson	2000	***
7890	Andrew	700	*
7870	Scott	2400	****
7866	David	900	*
7891	Sylvia	3200	*****

2 명의 급여가 인상되었습니다.

위와 같은 리포트를 뽑아내고 \$1000 보다 적은 급여를 받는 직원의 급여를 10%인상하는 프로그램을 작성하여 봅시다.

```
ACCEPT p_deptno PROMPT 'Please enter the department number : '
```

```
DECLARE
```

```
    v_deptno    NUMBER(2);
    v_dname      CHAR(14);
    v_loc        CHAR(13);
    v_empno      emp.empno%TYPE;
    v_ename      emp.ename%TYPE;
    v_sal        emp.sal%TYPE;
    v_grade      salgrade.grade%TYPE;
    v_dno        emp.deptno%TYPE := &p_deptno;
```

```
CURSOR emp_cursor IS
    SELECT empno, ename, sal
    FROM emp
    WHERE deptno = &p_deptno;
```

```
e_no_emp    EXCEPTION;
```

```
BEGIN
```

```
-- print out the report title
```

```
DBMS_OUTPUT.PUT_LINE('*****부서/직원 리포트*****');
```



```

EXCEPTION
    WHEN no_data_found THEN
        DBMS_OUTPUT.PUT_LINE('There is no matching data. ');
    WHEN too_many_rows THEN
        DBMS_OUTPUT.PUT_LINE('More than one matching data. ');
    WHEN e_no_emp THEN
        DBMS_OUTPUT.PUT_LINE('No employee with a smaller ' ||
                                'than $1000. ');
    --in case of any other exception
    WHEN others THEN
        DBMS_OUTPUT.PUT_LINE('Some other error occurred. ');
END;

```

PL/SQL로 예외 처리

- 예외란 무엇인가?
 - PL/SQL에서 실행 중에 발생하는 error처리를 의미합니다.
- 어떻게 발생되는가?
 - 오라클 에러가 발생합니다.
 - 여러분이 명시적으로 발생시킵니다.
- 어떻게 처리하는가?
 - 처리기로 트랩합니다.
 - 호출 환경으로 전달합니다.

개요

예외는 PL/SQL 블록의 실행 중에 발생하여 블록의 주요 작업 부분을 중단시킵니다. 항상 PL/SQL 예외가 발생할 때 블록은 항상 종료되지만, 마지막 조치 작업을 수행하도록 예외 처리기 부분을 작성할 수 있습니다.

예외를 일으키는 두 가지 방법

- 오라클 에러가 발생하면 관련예외는 자동적으로 일어납니다. 예를 들면 질의 문장의 데이터베이스에서 읽어 들인 행이 아무것도 없을 때 ORA-01403가 발생하면 이때 PL/SQL은 예외 NO_DATA_FOUND를 일으킵니다.

- 블록 내에서 RAISE 문장을 써서 명시적으로 예외를 일으킵니다. 예외는 사용자가 정의하거나 미리 정의 되어 있습니다

예외 유형

- 미리 정의된 오라클 서버 예외
- 미리 정의되지 않은 오라클 서버 예외
- 사용자 정의 예외

}

암시적으로 발생

명시적으로 발생

예외 유형

실행 중에 프로그램이 중단되지 않도록 예외에 대한 프로그램을 할 수 있습니다. 3 가지 종류의 예외가 있습니다.

예외	설명	처리용 지시
미리 정의된 오라클 서버오류	PL/SQL 코드에서 자주 발생하는 약 20 까지 오류	선언할 수 없고, 오라클 서버가 암시적으로 그것을 일으키도록 합니다.
미리 정의되지않은 오라클 서버오류	기타 표준 오라클 서버 오류	선언 섹션 내에서 선언하고, 오라클 서버가 암시적으로 그것을 일으키도록 합니다.
사용자 정의 오류	개발자가 정한 조건이 만족되지 않을 경우	선언섹션에서 선언하고, 명시적으로 일으킵니다.

주: Developer Forms 와 같이 **클라이언트**에 PL/SQL 문이 있는 어플리케이션은 독자적인 예외를 가집니다.

예외 트래핑(trapping)

구문

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

예외 트래핑(trapping)

PL/SQL 블록의 예외 처리 섹션 내에서 해당 루틴을 포함함으로써 모든 에러를 트랩할 수 있습니다. 각각의 에러 처리기는 WHEN 절로 구성되는데 그곳에 에러는 명시하고, WHEN 절 뒤에는 예외가 발생할 때 실행되기 위한 문장들이 옵니다.

#구문에서,

exception 선언섹션에서 선언된 미리 정의된 예외의 표준 이름 이거나 사용자 정의예외의 이름입니다.

statement 하나 이상의 PL/SQL 또는 SQL 문장입니다.

OTHERS 명시적으로 선언되지 않은 모든 예외를 트랩하는 예외 처리절입니다.

WHEN OTHERS 예외 처리기

예외처리 섹션은 지정된 예외만 트랩합니다. OTHERS 예외 처리기를 사용하지 않으면 다른 예외들은 트랩되지 않습니다. 이것은 아직 처리되지 않은 모든 예외를 트랩합니다. 그러므로, OTHERS 는 마지막에 정의되는 예외 처리기입니다.

OTHERS 처리기는 아직 트랩되지 않은 모든 예외들을 트랩합니다. 일부 오라클 툴들은 어플리케이션에서 이벤트를 발생시키기 위해 일으키는 개별적인 미리 정의된 예외들을 가지고 있습니다. OTHERS 는 또한 이 예외들도 트랩합니다.

미리 정의된 오라클 서버 예러

- 예외처리 루틴에서 미리 정의된 이름을 참조합니다.
- 미리 정의된(predefined) 예외의 예:
 - NO_DATA_FOUND
 - TOO_MANY_ROWS
 - INVALID_CURSOR
 - ZERO_DIVIDE
 - DUP_VAL_ON_INDEX

미리 정의된 오라클 서버 예러

해당 예외처리 루틴에서 표준 이름을 참조함으로써 미리 정의된 오라클 서버 예러를 트랩합니다.

미리 정의된 예외 리스트의 자세한 내용들은 *PL/SQL User's Guide and Reference, Release 7.3* 또는 *8.0*, "Error Handling."를 참조하십시오.

주: PL/SQL 은 STANDARD 패키지에 미리 정의된 예외를 선언합니다. 가장 일반적인 예외인 NO_DATA_FOUND 와 TOO_MANY_ROWS 예외는 사용하는 것이 좋습니다.

미리 정의된 예외

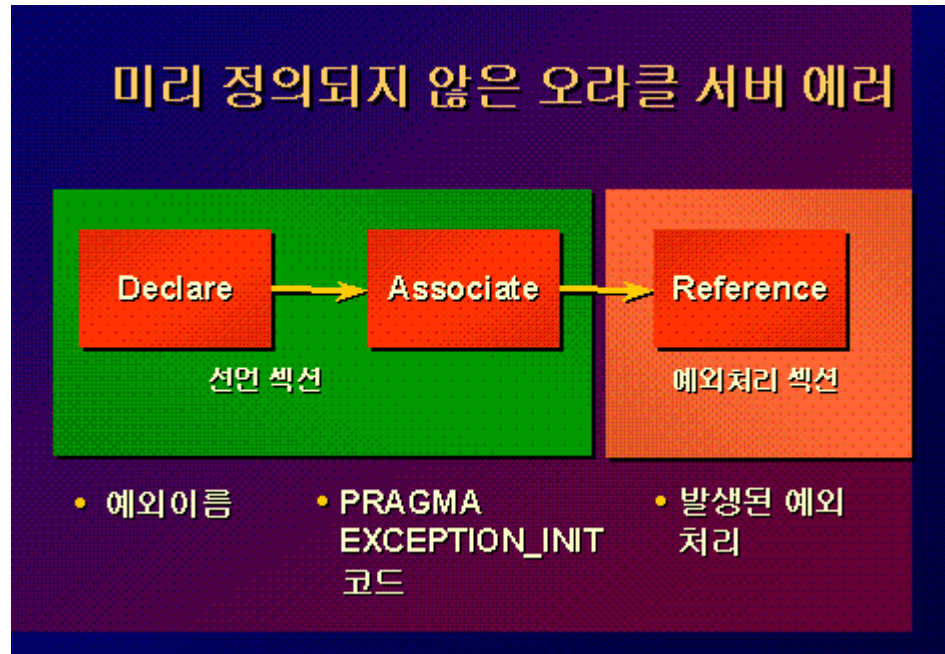
구문

```
BEGIN SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END ;
```

미리 정의된 오라클 서버 에러 트랩

각 예외에 대한 슬라이드의 예에서, 사용자를 위해 메시지가 인쇄 됩니다.

어떤 시점에서든지 하나의 예외만이 발생되고 처리됩니다.



미리 정의되지 않은 오라클 서버 에러

우선 에러를 선언하고 나서 OTHERS 처리기를 사용하여 미리 정의되지 않은 오라클 서버 에러를 트랩합니다. 선언된 예외는 암시적으로 발생합니다. PL/SQL 에서, pragma EXCEPTION_INIT 는 오라클 에러 번호와 예외 이름을 관련시키기 위해 컴파일러에게 알려 줍니다. 이름으로 임의의 내부 예외를 참조하고 지정 처리기를 사용합니다.

주: PRAGMA (pseudoinstruction 이라고도 함)는 PL/SQL 블록이 실행될 때 처리되지 않는 컴파일러 명령문임을 의미하는 키워드입니다. 블록 내에서 예외 이름이 발생하면 그것을 관련된 오라클 서버 에러 번호로 해독하기 위해 PL/SQL 컴파일러에게 지시합니다

미리 정의되지 않은 예러

무결성 제약조건 위반 오라클 서버 예러번호 - 2292 에 대한 트랩

```
DECLARE
  e_products_invalid EXCEPTION;
  PRAGMA EXCEPTION_INIT (
    e_products_invalid, -2292);
  v_message VARCHAR2(50);
BEGIN
  . . .
EXCEPTION
  WHEN e_products_invalid THEN
    :g_message := 'Product code
                  specified is not valid.';
  . . .
END;
```

1

2

3

미리 정의되지 않은 오라클 서버 예러 트랩

1. 선언섹션에서 예외 이름을 선언합니다.

#구문

```
exception          EXCEPTION;
```

구문에서: exception 예외의 이름입니다.

2. PRAGMA EXCEPTION_INIT 문장을 사용하여 표준 오라클 서버 예러 번호와 선언된 예외를 관련시킵니다.

#구문

```
PRAGMA EXCEPTION_INIT(exception, error_number);
```

구문에서: exception 이전에 선언된 예외입니다.

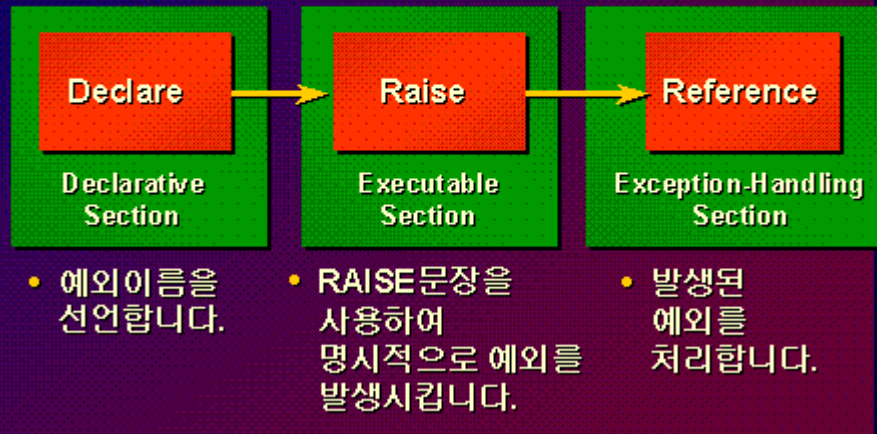
error_number 표준 오라클 서버 예러 번호입니다.

3. 해당 예외처리 루틴에서 선언된 예외를 참조합니다.

슬라이드에서, 재고품이 있다면 프로세싱을 중지하고 사용자에게 메시지를 보여줍니다.

자세한 내용은 *Oracle Server Messages, Release 7.3* 또는 *8.0* 을 참조하십시오.

사용자 정의 예외



사용자 정의 예외 트래핑

PL/SQL에서는 개별적으로 예외를 정의할 수 있습니다. 사용자 정의 PL/SQL 예외는 다음을 준수해야 합니다.

- PL/SQL 블록의 선언 섹션에서 선언합니다.
- RAISE 문장으로 명시적으로 발생시킵니다

사용자 정의 예외

예

```

[DECLARE]
  e_amount_remaining EXCEPTION;
. . .
BEGIN
  . . .
  RAISE e_amount_remaining;
  . . .
EXCEPTION
  WHEN e_amount_remaining THEN
    :g_message := 'There is still an amount
                  in stock.';
  . . .
END;
  
```

1

2

3

사용자 정의 예외 트래핑

명시적으로 선언하고 발생시킴으로써 사용자 정의 예외를 트랩할 수 있습니다.

1. 선언 섹션에서 사용자 정의된 예외에 대한 이름을 선언합니다.

#구문

```
exception EXCEPTION;
```

구문에서: exception 예외명입니다.

2. 실행 섹션에서 명시적으로 예외를 발생시키기 위해 RAISE 문장을 사용합니다.

#구문

```
RAISE exception;
```

구문에서: exception 앞에 선언된 예외명입니다.

3. 해당 예외 처리기 안에 선언된 예외를 참조합니다. 슬라이드 예에서, 이 고객은 만일 제품에 대한 어떤 재고가 남아 있다면, 데이터 베이스에서 제품을 제거할 수 없는 규칙을 가지고 있습니다. 이 규칙을 수행시키기 위해 어떠한 제약조건도 넣을 수 없기 때문에, 개발자는 어플리케이션에서 그것을 명시적으로 처리합니다. PRODUCT 테이블에서 DELETE 를 수행하기 전에, 제품의 재고가 있을 때는 INVENTORY 테이블을 질의합니다.

주: 동일한 예외를 다시 호출 환경으로 동일한 예외를 발생시키기 위해 예외 처리기에서 RAISE 문장을 사용합니다.

예외 트래핑 함수

- **SQLCODE**
에러 코드에 대한 숫자 값을 리턴합니다
- **SQLERRM**
에러 코드와 관련된 에러 메시지를 리턴합니다.

에러 트래핑 함수

예외가 발생할 때, 두 함수를 사용하여 관련된 에러코드 또는 에러 메시지를 확인할 수 있습니다.

코드 또는 메시지 값에 따라, 에러에 대해 취할 작업을 정할 수 있습니다. SQLCODE 로 내부 예외에 대한 오라클 에러 번호를 리턴합니다. 이때 에러 번호를 SQLERRM 으로 전달하면, 그 에러 번호와 관련된 메시지를 리턴합니다.

함수	설명
SQLCODE	에러코드에 대한 숫자 값을 리턴합니다.(NUMBER 변수에 저장할 수 있습니다.)
SQLERRM	에러번호와 관련된 메시지를 갖는 문자데이터를 리턴합니다.

SQLCODE 값 예

SQLCODE 값	설명
0	예외가 없습니다.
1	사용자 정의 예외.
+100	NO_DATA_FOUND 예외.
negative number	다른 오라클 서버 에러 번호.

예외 트래핑 함수

예

```
DECLARE
  v_error_code      NUMBER;
  v_error_message   VARCHAR2(255);
BEGIN
  ...
EXCEPTION
  ...
  WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQLCODE;
    v_error_message := SQLERRM;
    INSERT INTO errors VALUES(v_error_code,
                              v_error_message);
END;
```


예외 트래핑 함수

WHEN OTHERS 절에서 예외가 트랩될 때, 이 에러들을 식별하기 위한 함수를 사용할 수 있습니다. 슬라이드 예에서는 변수에 지정된 SQLCODE 값과 SQLERRM 값을 기술합니다. 이때 이 변수들을 SQL 문장에서 사용합니다.

SQLERRM 값을 변수에 저장하려 하기 전에 알려진 길이를 SQLERRM 값을 절단시킵니다.

예외 전달

서브블록은 예외를 처리하거나 둘러싸는 블록(상위블록)으로 예외를 전달할 수 있습니다.

```
DECLARE
    ...
    e_no_rows      exception;
    e_integrity    exception;
    PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
    FOR c_record IN emp_cursor LOOP
        BEGIN
            SELECT ...
            UPDATE ...
            IF SQLNOTFOUND THEN
                RAISE e_no_rows;
            END IF;
        EXCEPTION
            WHEN e_integrity THEN ...
            WHEN e_no_rows THEN ...
        END;
    END LOOP;
EXCEPTION
    WHEN NO_DATA_FOUND THEN ...
    WHEN TOO_MANY_ROWS THEN ...
END;
```

서브블록에서 예외 전달

예외를 처리한 후 서브블록은 정상적으로 종료되고 제어가 서브 블록의 END 문장으로 가서 바로 둘러싸는 블록(상위블록)으로 넘어갑니다. 그러나 PL/SQL 블록이 예외를 일으키고 그 블록이 예외에 대한 처리기를 가지고 있지 않다면, 예외는 처리기를 발견 할 때까지 계속해서 둘러싸는 블록에게 전달됩니다. 예외를 처리하는 블록이 하나도 없으면, 호스트 환경에 처리되지 않은 예외로 나타납니다. 예외가 둘러싸는 블록(상위 블록)으로 전달될 때, 그 블록 안에 남아 있는 나머지 실행 작업은 수행되지 않고 통과 됩니다. 이러한 처리 방법의 장점 가운데 하나는 둘러싸는 블록에 대해서는 보다 일반적 예외처리를 하게 하는 반면, 개별 블록에서는 독점적인 에러의 처리를 하는 문장을 쓸 수 있다는 것입니다.

RAISE_APPLICATION_ERROR

구문

```
raise_application_error (error_number,  
                        message[, {TRUE | FALSE}]);
```

- 내장된 서브프로그램에서 사용자 정의 에러 메시지를 생성하는 프로시저입니다.
- 실행 중인 내장된 서브프로그램에서만 호출됩니다.

표준화되지 않은 에러 코드와 에러 메시지를 리턴하는 RAISE_APPLICATION_ERROR 프로시저를 사용합니다. RAISE_APPLICATION_ERROR 로 어플리케이션에 대한 에러를 보고할 수 있고 처리되지 않은 예외가 리턴되지 않도록 합니다.

#구문에서,

error_number -20000 과 20999 사이의, 예외에 대해 지정된 번호입니다.

message 예외에 대한 사용자 지정 메시지입니다. 2048 바이트
 길이까지의 문자열입니다.

TRUE 선택적 부울 매개변수입니다. TRUE 이면, 에러는 이전의

| FALSE 에러 스택에 위치하게 됩니다. FALSE (디폴트)이면, 에러는 모든
 이전의 에러를 대체합니다.

예

```
...  
EXCEPTION  
  WHEN NO_DATA_FOUND THEN  
    RAISE_APPLICATION_ERROR (-20201,  
      'Manager is not a valid employee.');
```

END;

예

```
...  
DELETE FROM emp  
WHERE mgr = v_mgr ;  
IF SQL%NOTFOUND THEN  
    RAISE_APPLICATION_ERROR(-20202,'This is not a valid manager') ;  
END IF ;  
...
```

샘플 PL/SQL 프로그램 (정리)

***** 부서/직원 리포트 *****

부서 : 부서번호	부서명	위치
10	ACCOUNTING	NEW YORK

사원 : 사번	이름	급여	급여 등급
7871	Anderson	2000	***
7890	Andrew	700	*
7870	Scott	2400	****
7866	David	900	*
7891	Sylvia	3200	*****

2 명의 급여가 인상되었습니다.

위와 같은 리포트를 뽑아내고 \$1000 보다 적은 급여를 받는 직원의 급여를 10%인상하는 프로그램을 작성하여 봅니다.

```
ACCEPT p_deptno PROMPT 'Please enter the department number : '
```

```
DECLARE  
    v_deptno    NUMBER(2);  
    v_dname     CHAR(14);  
    v_loc       CHAR(13);  
    v_empno     emp.empno%TYPE;  
    v_ename     emp.ename%TYPE;
```

```

v_sal      emp.sal%TYPE;
v_grade    salgrade.grade%TYPE;
v_dno      emp.deptno%TYPE := &p_deptno;
CURSOR emp_cursor IS
    SELECT empno, ename, sal
    FROM emp
    WHERE deptno = &p_deptno;
e_no_emp    EXCEPTION;

BEGIN
    -- print out the report title
    DBMS_OUTPUT.PUT_LINE('*****부서/직원 리포트*****');
    SELECT deptno, dname, loc
    INTO v_deptno, v_dname, v_loc
    FROM dept
    WHERE deptno = v_dno;
    DBMS_OUTPUT.PUT_LINE('부서 : 부서번호   부서명           위치');
    DBMS_OUTPUT.PUT_LINE('      ' || v_deptno || '      ' || v_dname ||
        '      ' || v_loc || '      ');
    DBMS_OUTPUT.PUT_LINE('사원 : 사번   이름   급여   급여등급');
    OPEN emp_cursor;
    FETCH emp_cursor INTO v_empno, v_ename, v_sal;
    /* according to the salary amount,
       print out the stars */
    WHILE emp_cursor%FOUND LOOP
        SELECT grade
        INTO v_grade
        FROM salgrade
        WHERE v_sal > losal AND v_sal < hisal;
        DBMS_OUTPUT.PUT('      ' || v_empno || '      ' || v_ename ||
            '      ' || v_sal || '      ');
        FOR I in 1..v_grade LOOP
            DBMS_OUTPUT.PUT('* ');
        END LOOP;

        DBMS_OUTPUT.PUT_LINE(' ');
    END LOOP;

```

```

        FETCH emp_cursor INTO v_empno, v_ename, v_sal;
    END LOOP;
    CLOSE emp_cursor;
    UPDATE emp
        SET      sal = sal * 1.1
        WHERE    sal < 1000;
    IF SQL%ROWCOUNT = 0 THEN
        RAISE e_no_emp;
    ELSE
        DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || '명의 급여가 ' ||
                               '인상되었습니다.');
```

END IF;

```

    COMMIT;
EXCEPTION
    WHEN no_data_found THEN
        DBMS_OUTPUT.PUT_LINE('There is no matching data.');
```

WHEN too_many_rows THEN

```

        DBMS_OUTPUT.PUT_LINE('More than one matching data. ');
    WHEN e_no_emp THEN
        DBMS_OUTPUT.PUT_LINE('No employee with a smaller ' ||
                               'than $1000.');
```

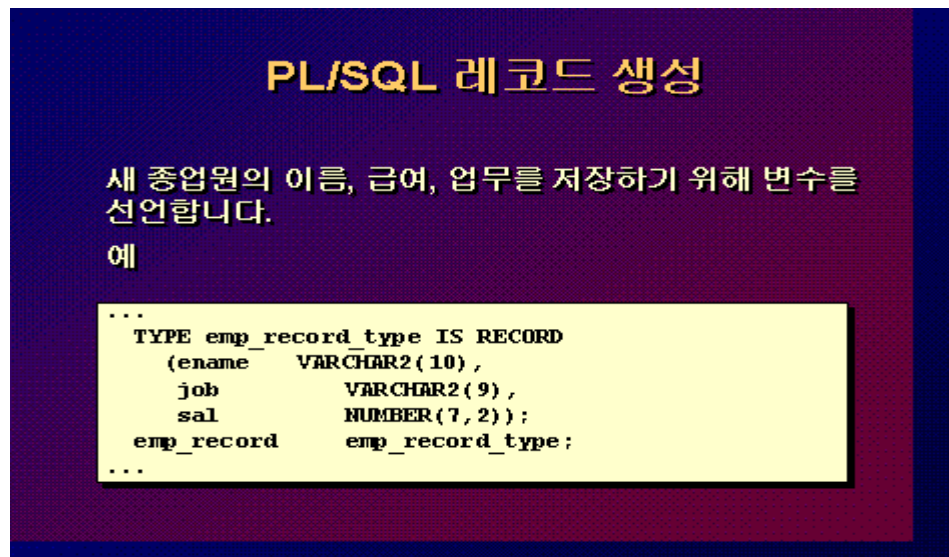
--in case of any other exception

```

    WHEN others THEN
        DBMS_OUTPUT.PUT_LINE('Some other error occurred.');
```

END;

제 25 장 PL/SQL 레코드



PL/SQL 레코드 생성

새 종업원의 이름, 급여, 업무를 저장하기 위해 변수를 선언합니다.

예

```
...  
TYPE emp_record_type IS RECORD  
  (ename   VARCHAR2(10),  
   job     VARCHAR2(9),  
   sal     NUMBER(7,2));  
emp_record emp_record_type;  
...
```

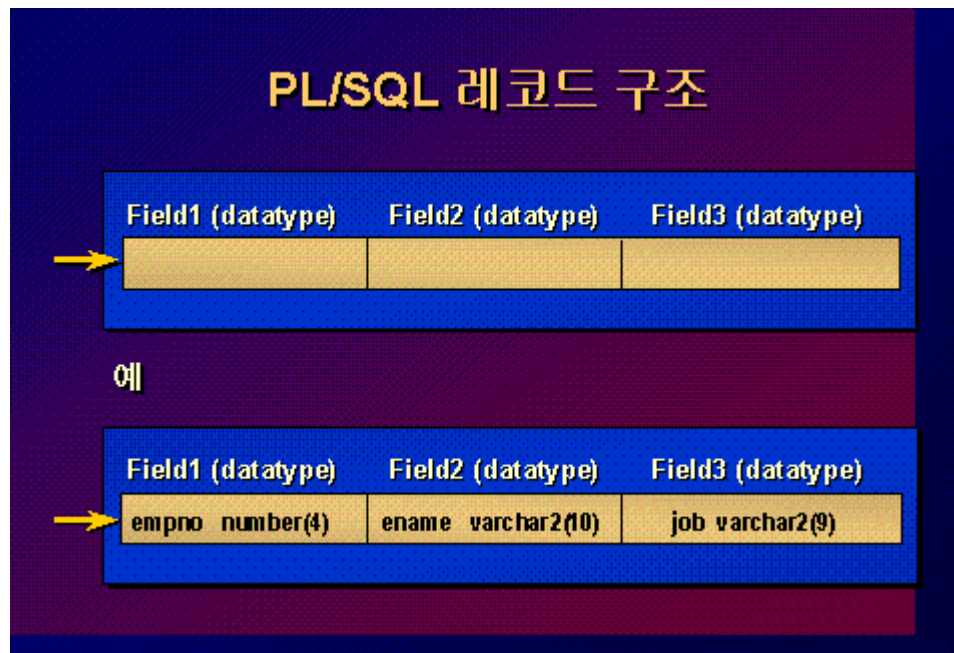
PL/SQL 레코드 생성

필드 선언은 변수 선언과 유사합니다. 각 필드는 유일한 이름과 특정 데이터형을 가지 고 있습니다.

스칼라 변수에 있는 것처럼 PL/SQL 레코드에 대해 미리 정의된 데이터형은 없습니다. 그러므로, 처음에 데이터형을 생성하고 그 데이터형을 사용하는 식별자를 선언합니다. 다음 예는 필드 데이터형을 지정하기 위해 %TYPE 속성을 사용할 수 있음을 보여 줍니다.

```
DECLARE  
  TYPE emp_record_type IS RECORD  
    (empno   NUMBER(4) NOT NULL := 100,  
     ename    emp.ename%TYPE,  
     job      emp.job%TYPE);  
  emp_record emp_record_type;  
...
```

주: 임의의 필드를 선언할 때에 NOT NULL 제약조건을 추가 할 수 있고 그래서 그 필드에는 null 지정을 방해합니다. NOT NULL 로 선언된 필드는 초기화되어야함을 명심하십시오.



레코드 참조와 초기화

레코드에서 필드는 이름으로 액세스 됩니다. 개별 필드를 참조하거나 초기화 하기 위해, 점 표기법을 다음과 같이 사용합니다.

```
record_name.field_name
```

예를 들면 다음과 같이 `emp_record` 레코드에서 `Job` 필드를 참조합니다.

```
emp_record.job ...
```

이때 다음과 같이 레코드에 대해 값을 지정합니다.

```
emp_record.job := 'CLERK';
```

블록이나 서브 프로그램에서 사용자 정의 레코드는 블록이나 서브 프로그램에 들어갈 때 생성되고 블록이나 서브 프로그램을 나갈 때 없어집니다.

레코드에 대한 값 할당

SELECT 또는 FETCH 문장을 사용함으로써 레코드에 공통 값들을 지정할 수 있습니다.

열 이름은 레코드의 필드와 동일한 순서로 나타나야 합니다. 두 레코드가 동일한

데이터형을 가지면 하나의 레코드를 다른 레코드에 지정할 수 있습니다

%ROWTYPE 속성

예

DEPT 테이블에 저장된 것과 동일하게 부서에 대한 정보를 저장하기 위해 변수를 선언합니다.

```
dept_record dept%ROWTYPE;
```

EMP 테이블에서 저장된 것과 동일하게 종업원 에 대한 정보를 저장하기 위해 변수를 선언합니다.

```
emp_record emp%ROWTYPE;
```

예

위의 첫 번째 선언은 DEPT 테이블의 행과 동일한 필드명과 필드 데이터형으로 레코드를 생성합니다. 필드는 DEPTNO, DNAME, 그리고 LOCATION 입니다.

다음 예에서 열 값을 item_record 로 이름 붙여진 레코드 안으로 지정합니다.

```
DECLARE
    item_record item%ROWTYPE;
    ...
BEGIN
    SELECT * INTO item_record
    FROM item
    WHERE ...
```

커서와 레코드

값을 PL/SQL RECORD 안으로 인출(fetch)하여
편리하게 활성 셋(set)의 행을 프로세스 합니다.

예

```
...  
CURSOR c1 IS  
  SELECT empno, ename  
  FROM   emp;  
  emp_record c1%ROWTYPE;  
BEGIN  
  OPEN c1;  
  . . .  
  FETCH c1 INTO emp_record;
```

커서와 레코드

여러분은 이미 테이블에서 열의 구조를 사용하기 위해 레코드를 정의할 수 있음을 보았습니다. 또한 명시적 커서에서의 열 목록을 기초로 하여 레코드를 정의할 수 있습니다. 이것은 단순히 인출(fetch)할 수 있기 때문에 활성 셋(set)의 행을 처리하기가 편리합니다.

그러므로 행 값은 레코드의 해당 필드 안으로 직접 로드됩니다.

커서 FOR 루프

구문

```
FOR record_name IN cursor_name LOOP  
  statement1;  
  statement2;  
  . . .  
END LOOP;
```

- 명시적 커서를 프로세스 하는 편리한 방법입니다.
- OPEN, FETCH, CLOSE가 자동 수행 됩니다.
- 레코드를 선언하지 마십시오. 자동적으로 선언됩니다.

커서 FOR 루프

커서 FOR 루프는 명시적 커서에서 행을 처리합니다. 루프에서 각 반복마다 커서를 열고 행을 인출(fetch)하고 모든 행이 처리되면 자동으로 커서가 닫히므로 사용하기가 편리합니다. 루프는 마지막 행이 인출(fetch)하고 자동적으로 종료됩니다.

#구문에서,

record_name	암시적으로 선언된 레코드의 이름입니다.
cursor_name	이전에 선언된 커서에 대한 PL/SQL 식별자입니다.

지침

- 루프를 제어하는 레코드를 선언하지 마십시오. 그것의 범위는 루프 내에서만입니다.
- 필요하다면 루프 내에서 커서 속성을 이용하십시오.
- 필요하다면 FOR 문 안에서 커서 이름 다음에 괄호로 커서에 대한 매개변수를 묶어 사용하십시오..
- 커서 작업이 수동으로 처리되어야 할 때는 FOR 루프를 사용하지 마십시오.

주: 루프가 시작 될 때에 질의를 정의할 수 있습니다. 질의 표현식은 SELECT 문장이라 불리고, 커서는 FOR 루프 내에서만 사용할 수 있습니다. 이름을 가지고 커서가 선언되지 않기 때문에 그 속성을 시험할 수는 없습니다.

커서 FOR 루프

더 이상 남겨진 것이 없을 때 까지 종업원을 하나씩 읽어 들입니다.

예

```
DECLARE
  CURSOR c1 IS
    SELECT empno, ename
    FROM   emp;
BEGIN
  FOR emp_record IN c1 LOOP
    -- implicit open and implicit fetch occur
    IF emp_record.empno = 7839 THEN
      ...
    END LOOP; -- implicit close occurs
  END;
```

#실습할것!

PL/SQL 테이블 구조

Primary Key	Column
...	...
1	Jones
2	Smith
3	Maduro
...	...
BINARY_INTEGER	Scalar

PL/SQL 테이블 구조

데이터베이스 테이블의 크기처럼 PL/SQL 테이블의 크기는 제한적이지 않습니다.

즉 PL/SQL 테이블에서 행의 숫자는 동적으로 증가될 수 있어 PL/SQL 테이블은 새 행이 추가 되듯이 증가합니다.

PL/SQL 테이블은 이름이 없는 한 열과 기본 키를 가질 수 있습니다.

열은 임의의 스칼라 또는 레코드 데이터형에 속할 수 있지만 기본

키는 BINARY_INTEGER 유형에 속해야 합니다.그 선언에서 PL/SQL 테이블은 초기화 할 수 없습니다.

PL/SQL 테이블 생성

```
DECLARE
  TYPE ename_table_type IS TABLE OF emp.ename%TYPE
    INDEX BY BINARY_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY BINARY_INTEGER;
  ename_table          ename_table_type;
  hiredate_table       hiredate_table_type;
BEGIN
  ename_table(1) := 'CAMERON';
  hiredate_table(8) := SYSDATE + 7;
  IF ename_table.EXISTS(1) THEN
    INSERT INTO ...
  ...
END;
```

PL/SQL 테이블 생성

스칼라 변수에 있는 것처럼 PL/SQL 테이블에 대해 미리 정의된 데이터형은 없습니다.

그러므로 처음에 데이터형을 생성하고 그 데이터형을 사용하여 식별자를 선언합니다.

PL/SQL 테이블 참조

구문

```
pl/sql_table_name(primary_key_value)
```

여기서: primary_key_value BINARY_INTEGER 형에 속합니다.

PL/SQL 테이블 ename_table 에서 3 번째 행을 참조합니다.

```
ename_table(3) ...
```

BINARY_INTEGER 의 크기 범위는 -2147483647 ... 2147483647 입니다. 그래서 기본 키 값은 음수가 될 수 있습니다. 인덱스는 꼭 1 로 시작할 필요가 없습니다.

주: table.EXISTS(i) 문장은 인덱스 i 인 행이 적어도 하나 반환되면 TRUE 를 반환합니다. 존재하지 않는 테이블 요소를 참조할 때 발생하는 오류를 막기 위해 EXISTS 문장을 사용합니다.

PL/SQL 테이블 메소드(method)사용

PL/SQL 테이블을 쉽게 사용하기 위해 :

- EXISTS
- COUNT
- FIRST 와 LAST
- PRIOR
- NEXT
- EXTEND
- TRIM
- DELETE

PL/SQL 테이블 메소드는 (method) 는 테이블에 대해 수행되는 **내장**된 프로시저 또는 함수이며 점 표기법을 사용하여 호출됩니다.

구문:

```
table_name.method_name[ (parameters) ]
```

방법	설명
EXISTS(n)	n 번째 요소가 PL/SQL 테이블에 존재한다면 TRUE 를 리턴합니다.
COUNT	PL/SQL 테이블이 현재 포함하는 요소의 숫자를 리턴합니다.
FIRST LAST	PL/SQL 테이블에서 처음과 마지막 (가장 작고 크고) 인덱스 숫자를 리턴합니다.
PRIOR(n)	PL/SQL 테이블에서 인덱스 n 이전의 인덱스 숫자를 리턴합니다.
NEXT(n)	PL/SQL 테이블에서 인덱스 n 다음의 인덱스 숫자를 리턴합니다
EXTEND(n,i)	PL/SQL 테이블의 크기 증가를 위해 EXTEND 는 PL/SQL 테이블에 대한 1 개의 NULL 요소 추가. EXTEND(n)은 PL/SQL 테이블에 n 개의 NULL 요소를 추가. EXTEND(n,i)는 PL/SQL 테이블의 i 번째 요소의 값을 n 개 추가.
TRIM	PL/SQL 테이블의 끝에서 한 요소를 제거. TRIM(n) 은 PL/SQL 테이블의 끝에서 n 개의 요소를 제거.
DELETE	DELETE 는 PL/SQL 테이블에서 모든 요소를 제거. DELETE(n)는 PL/SQL 테이블에서 n 번째 요소를 제거.

ELETE(m,n)는 PL/SQL 테이블에서 m....n 까지의 요소를 제거.

PL/SQL 레코드 테이블

- %ROWTYPE 속성으로 TABLE 변수를 정의합니다.
- 부서 정보를 저장하기 위해 PL/SQL 변수를 선언합니다.

예

```
DECLARE
TYPE dept_table_type IS TABLE OF dept%ROWTYPE
INDEX BY BINARY_INTEGER;
dept_table dept_table_type;
-- Each element of dept_table is a record
```

PL/SQL 레코드 테이블

데이터베이스 테이블의 모든 필드에 대한 정보를 저장하기 위해 필요한 테이블 정의는 하나이기 때문에 레코드 테이블은 PL/SQL 테이블의 기능성을 크게 증가 시킵니다.

레코드 테이블 참조

슬라이드에서 주어진 예에서, 이 테이블의 각 요소는 레코드이기 때문에 dept_table 레코드의 필드를 참조할 수 있습니다.

구문

<code>table(index).field</code>

예

<code>dept_table(15).location := 'Atlanta';</code>
--

location 은 DEPT 테이블에서 필드를 나타냅니다.

주: 데이터베이스 테이블에서 행을 나타내는 레코드를 선언하기 위해 %ROWTYPE 을 사용할 수 있습니다.

%ROWTYPE 속성과 조합 데이터형 RECORD 사이의 차이점은, RECORD 가 레코드에서

필드의 데이터형을 지정하거나 개별 필드를 선언할 수 있도록 해 준다는 것입니다.