

Oracle 수업교안

1. 데이터베이스와 SQL

1.1 데이터베이스

처음 데이터베이스란 용어가 사용된 것은 1963년 “Development and Management of Computer Center Data Bases”에서 사용되었다. 그리고 같은 해 제너럴 일렉트릭(GE)사에서 “Integrated Data Store”라는 최초의 상용 DBMS(Data Base Management System)가 발표되었다. “Integrated Data Store”는 당시 제너럴 일렉트릭사에 근무하던 찰스 바크만(Charles Bachman)에 의해 개발되어 1965년 이를 기반으로 하여 DBTG(Data Base Task Group)에 의해 네트워크 모델이 시도 되었으며 같은 해 IBM에 의해 IMS(Information Management System)라는 상용 DBMS가 발표되었다.

1970년대 접어들어 Edgar Codd에 의해 관계 데이터 모델(“A Relational Model of Data for Large Shared Data Banks”)이 제안되어 오늘날 대부분의 RDBMS(Relational Data Base Management System)가 이 관계 데이터 모델을 기반으로 하고 있다. 그리고 1979년 세계 최초의 상용 RDBMS인 오라클이 발표되었다.

1.2 SQL(Structured Query Language)

우리가 관계형 데이터베이스 시스템(RDBMS, Relational Database Management System - 대표적인 RDBMS는 Oracle, MySQL, MSSQL 등이 있음)에 데이터를 저장하게 되면 RDBMS는 그 데이터를 물리적인 파일에 저장하게 되는데 Oracle를 예로 들자면 Oracle DBMS는 데이터를 물리적인 파일(.DBF, .ORA 등)로 컴퓨터 하드디스크에 저장해 관리 한다.

SQL은 RDBMS를 통해 데이터를 추가하고, 조회하고, 수정하고, 삭제하는 일련의 작업, 즉 CRUD(Create, Retrieve, Update, Delete) 작업을 위해 고안된 언어이다. CRUD는 용어에서도 알 수 있듯이 데이터베이스 객체인 테이블을 생성, 수정, 삭제하는 동작과 그 테이블에 데이터를 추가, 수정, 삭제하고 테이블에 입력된 데이터를 조회하는 동작을 의미한다. 일반적으로 프로그래밍 언어는 절차적 특징(처리과정을 일일이 기술하는 것)을 가지고 있지만 SQL은 이와 다르게 일정한 틀 안에서 패턴을 가지고 있는 구조화된(Structured) 언어이다.

1.3 SQL 표준

SQL 표준은 1986년 ANSI(American National Standard Institute - 미국표준협회)와 1987년 ISO(International Organization for Standardization - 국제표준화기구)에 의해 표준이 제정되었다. SQL이 국제표준으로 제정되었다는 것은 많은 DBMS에서 이를 지원한다는 것을 의미한다. 다시 말해 오라클에서 표준 SQL로 작성된 쿼리는 MySQL이나 MSSQL에서도 그대로 사용할 수 있다는 것을 의미한다. 실제로 현존하는 대부분의 DBMS는 표준 SQL을 지원하고 있다.

연 도	이 름	별 칭	내 용
1986	SQL-86	SQL1 SQL-86	1983년 ANSI와 ISO가 SQL 표준 개발 착수 1986년 ANSI가 SQL-86 표준 출간 1987년 ISO에서 SQL-86 표준으로 승인
1989	SQL-89		기본 키와 외래 키를 포함하도록 무결성 제약조건을 확장
1992	SQL-92	SQL2	기존 명령을 확정, 새로운 명령 추가
1999	SQL:1999	SQL3	정규표현식(Regular Expression), 제귀쿼리, 트리거, 객체지향, 멀티미디어 기능 등을 지원
2003	SQL:2003		XML 관련, 윈도우 함수, 표준화된 시퀀스, 자동 생성 값을 갖는 컬럼 등을 지원
2006	SQL:2006		XML 관련 기능 강화, XQuery

표 1-1 SQL 표준

1.4 SQL 명령의 종류

SQL 명령을 역할에 따라 구분하여 부르는 용어는 책에 따라 약간씩 다르지만 일반적으로 다음과 같이 5가지 정도로 나눌 수 있을 것이다.

데이터베이스(RDBMS)에 저장된 데이터를 조회하는 SQL 명령을 DQL(Data Query Language)이라고 하며 이 DQL에는 테이블 하나에서 조회하는 방식과 여러 테이블을 연결해 조회하는 조인(join)이라는 방식이 있으며 기준이 되는 SQL 쿼리 안에 쿼리를 적용해 조회하는 서브쿼리 방식과 특정 조건에 해당 하거나 범위 안에 드는 데이터를 조회할 수 있도록 다양한 데이터 조회 방식을 제공하고 있다.

테이블에 데이터를 추가하고, 수정하고, 삭제하는 데이터 조작에 필요한 SQL 명령을 데이터 조작용(DML, Data Manipulation Language)라고 한다. DML에서도 특정 데이터만 조작할 수 있도록 조건을 사용할 수 있다.

데이터가 저장되는 테이블, 순차적인 번호를 생성해 주는 시퀀스 등을 데이터베이스 객체라고 하는데, 이렇게 데이터베이스 안에서 사용되는 객체를 생성하고 수정, 삭제 하는데 사용하는 SQL 명령을 데이터 정의어(DDL, Data Definition Language)이라고 한다.

데이터에 대한 접근 권한을 지정하거나 회수 할 때 사용하는 SQL 명령어를 데이터 제어어(DCL, Data Control Language)라고 하며 트랜잭션 처리를 위해 조작된 데이터를 데이터베이스 파일에 적용하거나 취소하는 SQL 명령어를 TCL(Transaction Control Language)이라고 한다.

2. 오라클 데이터베이스

오라클은 오늘날 세계에서 가장 많이 사용되는 RDBMS로 현재는 클라우드 시장을 겨냥한 Oracle 12C를 비롯해 18C와 19C 버전의 제품이 출시된 상태이다. 이외에도 오픈소스로 출발하여 현재는 오라클사에 의해 관리되고 있는 MySQL, 마이크로소프트사의 MSSQL, IBM 사의 DB2 등이 많이 사용되고 있는 RDBMS이며 MariaDB와 Sysbase도 널리 사용되고 있는 RDBMS 이다.

오라클 제품은 4개미만의 프로세서를 지원하는 Oracle Database Standard Edition과 2개미만의 프로세서를 지원하는 Oracle Database Standard Edition One 그리고 엔터프라이즈급 성능과 보안을 지원하는 최상위 버전인 Oracle Database Enterprise Edition이 있다. 이외에도 설치와 관리가 비교적 쉽고 무료로 다운받아 사용할 수 있는 Oracle Database Express Edition이 있다. Express Edition은 단일 프로세서와 최대 1GB 메모리를 지원하기 때문에 학습용이나 소규모 기업에서 사용할 수 있는 성능을 가지고 있다.

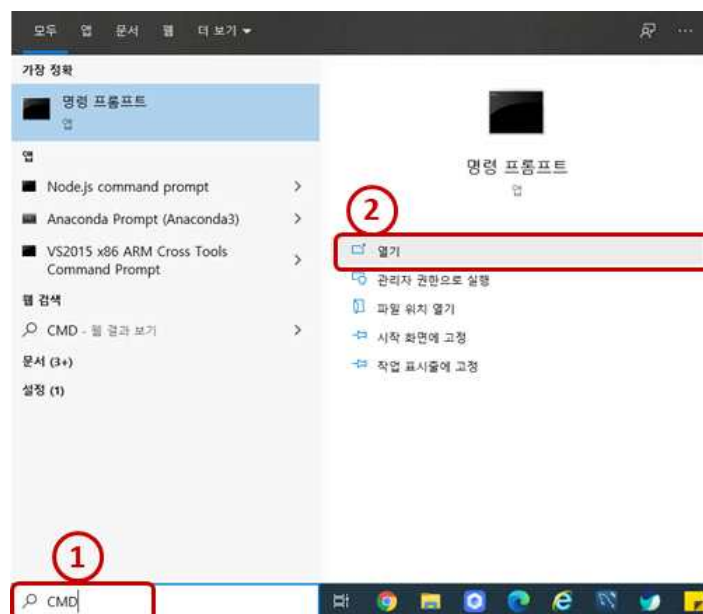
2.1 오라클 XE 및 도구 설치하기

별도로 제공하는 Oracle 11g XE 다운로드 및 설치.pdf 등을 참고

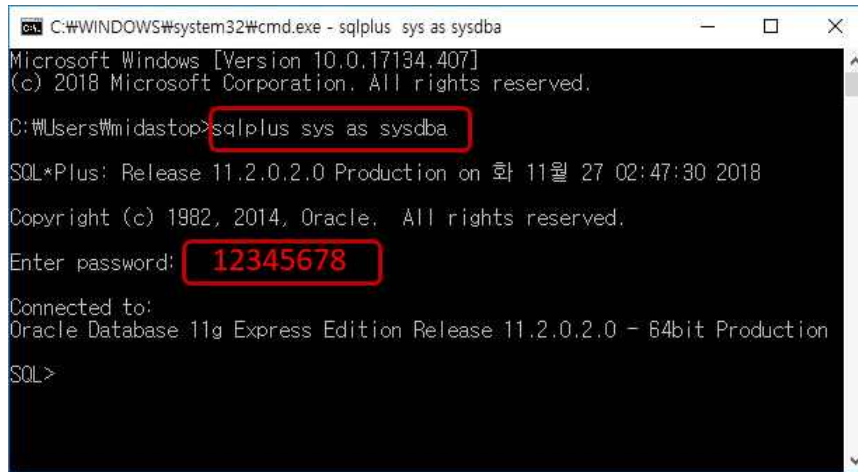
2.2 계정 활성화(오라클 설치 후 테스트)

▶ SYS 계정으로 접속하기

오라클 설치가 완료되면 아래 그림과 같이 작업표시줄의 검색어 입력란에 “CMD”를 입력하고 2번 “열기”를 클릭하여 명령 프롬프트를 실행해 오라클 시스템에 접속해 보자.



명령 프롬프트 창에서 다음 그림과 같이 sqlplus sys as sysdba 명령으로 sys 계정으로 오라클 시스템에 접속 할 수 있다. 참고로 오라클 DBMS에서 SYS와 SYSTEM 계정은 데이터베이스 관리자 (DBA, Database Administrator) 권한을 가지고 있다.



```
C:\WINDOWS\system32\cmd.exe - sqlplus sys as sysdba
Microsoft Windows [Version 10.0.17134.407]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Wmidastop>sqlplus sys as sysdba

SQL*Plus: Release 11.2.0.2.0 Production on 화 11월 27 02:47:30 2018

Copyright (c) 1982, 2014, Oracle. All rights reserved.

Enter password: 12345678

Connected to:
Oracle Database 11g Express Edition Release 11.2.0.2.0 - 64bit Production
SQL>
```

sqlplus sys as sysdba -> 비밀번호 입력 후 접속

sqlplus / as sysdba -> 비밀번호 입력 없이 바로 접속(오라클이 설치된 시스템에서만 가능)

▶ 잠긴 계정 풀기

우리가 실습에 사용할 HR 계정은 Oracle 11g Express Edition을 설치하게 되면 기본적으로 생성되어 있는 계정이지만 SYS와 SYSTEM과 같은 DBA 권한을 가진 계정처럼 바로 사용할 수 없도록 계정이 잠겨 있다. 그래서 DBA 권한을 가진 SYS 계정으로 접속해 HR 계정을 사용할 수 있도록 잠긴 계정을 풀어줘야 한다. 참고로 아래 SQL 문장은 가독성을 위해서 대소문자를 사용하였지만 SQL 쿼리는 실제 데이터를 비교하는 부분을 제외하고는 대소문자를 구문하지 않는다.

■ HR 계정 풀기

```
ALTER USER hr ACCOUNT UNLOCK;
```

■ HR 계정의 비밀번호를 hr로 설정

```
ALTER USER hr IDENTIFIED BY hr;
```

■ HR 계정과 비밀번호를 한 번에 설정

```
ALTER USER hr IDENTIFIED BY hr ACCOUNT UNLOCK;
```

■ 오라클 사용자 계정상태 조회

아래 SQL 문으로 검색해서 account_status가 LOCKED로 검색된 사용자는 계정이 잠겨있는 사용

자이며 EXPIRED로 검색된 사용자는 비밀번호 유효기간이 지난 사용자 이다.

참고로 dba_ 접두어로 시작하는 뷰는 데이터베이스 관리자를 위한 정보를 제공하는 뷰로 일반 사용자는 조회할 수 없다. 아래와 같이 조회하면 DBMS의 모든 사용자 계정과 계정 상태에 대해 조회할 수 있다.

```
SELECT username, account_status FROM dba_users;
```

특정 사용자의 계정 상태를 조회하려면 아래와 같은 쿼리를 사용하면 된다.

아래는 우리가 사용할 HR 계정의 상태를 조회한 예이다.

```
SELECT username, account_status FROM dba_users WHERE username='HR';
```

위에서 잠긴 계정을 풀지 않은 상태에서 조회하게 되면 아래와 같이 조회된다.

아래 조회된 내용을 살펴보면 ACCOUNT_STATUS가 EXPIRED & LOCKED로 나타나는데 이는 HR 계정이 현재 잠겨있는 상태이며 계정의 유효기간이 만료되었음을 의미한다.

USERNAME	ACCOUNT_STATUS
HR	EXPIRED & LOCKED

위에서 잠긴 계정만 풀고 계정 상태를 조회하면 여전히 계정의 유효기간 만료인 EXPIRED로 조회되는데 이때는 계정의 비밀번호를 설정(변경)해 주면 계정 사용이 가능한 OPEN 상태로 조회될 것이다.

▶ 현재 접속한 사용자 보기

```
SHOW USER
```

2.3 실습용 스키마 설치

수업시간에 제공한 EMP_EXAMPLE.sql, star.sql, UNION_SET.sql 파일을 컴퓨터의 D드라이브가 존재하면 D드라이브 루트에 그렇지 않으면 C 드라이브 루트에 복사한 후 Oracle SQL Developer에서 HR 계정으로 접속해 아래 명령을 사용해 테이블을 생성하고 테이블에 데이터를 추가하자.

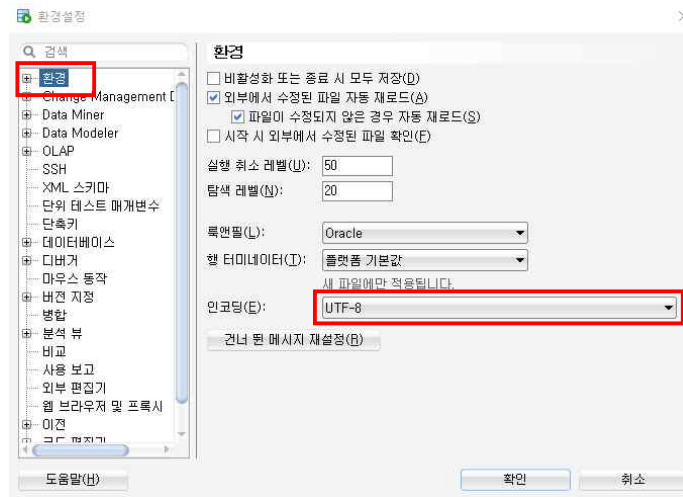
```
@D:\SQL_DATA\EMP_EXAMPLE.sql
```

```
@D:\SQL_DATA\star.sql
```

```
@D:\SQL_DATA\UNION_SET.sql
```

@D:\SQL_DATA\EXAMPLE_JOIN.sql

@D:\SQL_DATA\SUBQUERY.sql



실습용 스키마 설치시 sql 파일이 UTF-8로 인코딩되었기 때문에 한글이 깨져서 에러가 발생할 수 있다. 이럴 경우 Oracle SQL Developer의 도구 메뉴 -> 환경설정 메뉴를 선택하면 아래 그림과 같이 환경설정 대화상자가 나타난다. 이 대화상자에서 좌측의 환경을 선택하고 우측의 인코딩을 "UTF-8"로 변경한 후 Oracle SQL Developer를 다시 시작하여 작업하면 정상적으로 실행될 것이다.

3. 데이터 조회하기

아마도 데이터베이스에서 가장 많이 하는 작업이 테이블에서 데이터를 조회하는 일일 것이다.

앞에서 SQL 기본 형식과 종류에 대해 학습하면서 데이터를 조회하는 SELECT 문에 대해 간단히 알아보았다. 앞에서 알아본 SELECT 문은 테이블에서 모든 데이터를 조회하는 정도였는데 실무에서는 보다 복잡한 조건 등을 지정해 데이터를 조회하기 때문에 이장에서는 데이터를 조회하는 세부적인 방법에 대해 알아볼 것이다.

표 3-1은 SELECT 문의 기본 형식으로 [] 안에 기술된 명령은 생략이 가능하고 { } 안에 들어 있는 내용 중에서 하나를 선택해 사용할 수 있다.

SELECT [DISTINCT] {*, column[Alias], ...} FROM 테이블명

표 3-1 SELECT 문의 기본 형식

SELECT 문에 검색 조건이나 조인 조건을 지정하기 위해 WHERE 절과 같이 사용할 수 있으며 GROUP BY 절, HAVING 절, ORDER BY와 같이 사용할 수 있다. 또한 집합 연산자를 사용해 여러 개의 SELECT 문을 연결해 데이터를 조회할 수도 있다.

3.1 기본적인 데이터 조회

SELECT 문의 가장 기본적인 데이터 조회 방식인 테이블의 전체 데이터를 조회 한다거나 테이블에서 필요한 컬럼만 지정해 데이터를 조회하는 방법에 대해 알아보자.

▶ 전체 데이터 조회하기

emp 테이블에서 모든 데이터를 조회하는 SELECT 문이다.

```
SELECT * FROM emp;
```

▶ 특정 컬럼만 지정해 조회하기

emp 테이블에서 필요한 컬럼만 지정해 데이터를 조회하는 SELECT 문이다.

아래 SELECT 문과 같이 꼭 컬럼의 순서를 지켜서 데이터를 조회할 수 있는 것은 아니다.

```
SELECT empno, ename, sal FROM emp;  
SELECT ename, sal, job, empno FROM emp;
```

▶ 컬럼에 별칭을 지정해 데이터 조회하기

테이블의 거의 모든 컬럼 이름은 영어로 지정되어 있다. 이렇게 영어로 되어 있는 컬럼에 아래와 같이 한글 별칭을 지정해 데이터를 조회 할 수 있다.

아래 SELECT 문에서와 같이 컬럼 이름에 AS를 사용해 컬럼에 별칭을 지정할 수 있다. 또한 AS를 사용하지 않고 컬럼 이름 다음에 공백을 사용해 별칭을 지정할 수도 있다. 그리고 별칭에 특수 문자를 포함시키거나 대소문자를 구분하고 띄어쓰기를 포함하고 싶다면 별칭을 쌍 따옴표(“”)로 감싸주면 된다.

```
SELECT empno AS 사번, ename AS 이름, sal AS 급여, deptno AS 부서
FROM emp;
```

```
SELECT empno 사번, ename 이름, sal 급여, hiredate 입사일 FROM emp;
```

3.2 산술 연산자를 사용한 데이터 조회

만약 emp 테이블에 저장된 월급 컬럼을 사용해 사원의 월급과 연봉을 출력해야 한다면 각 사원의 월급 데이터에 12를 곱하여 출력해야 한다. 이럴 때 기존 컬럼의 데이터와 산술 연산자를 사용해 새로운 조회 컬럼을 출력할 수 있다.

산술 연산자는 사칙연산을 수행하거나 수치 값의 부호를 바꾸어 출력하고자 할 때 사용한다.

```
SELECT empno 사번, ename 이름, sal 급여, sal * 12 연봉, hiredate 입사일
FROM emp;
```

3.3 조건을 부여해 데이터 조회하기

앞에서 알아본 SELECT 문은 해당 테이블이 가지고 있는 모든 행을 조회하는 방식이었다. 그러나 emp 테이블에서 급여를 500만원 이상 받는 직원만 조회하거나 영업부에 속한 직원만 조회해야 한다면 기존의 방식으로는 불가능하다. 이럴 경우 특정 조건을 기준으로 데이터를 조회해야 하는데 이때 조건을 지정하기 위해 사용되는 것이 바로 WHERE 절이다.

WHERE 절에는 비교, 논리, IN, ALL, ANY, IS NULL 등과 같은 다양한 연산자를 사용해 하나 이상의 조건을 지정할 수 있다.

3.3.1 비교 연산자를 사용한 데이터 조회

WHERE 절에 조건을 지정하기 위해 같다(=), 같지 않다(<>), 크다(>), 크거나 같다(>=), 작다(<), 작거나 같다(<=)와 같은 비교 연산자를 사용할 수 있다.

▶ 수치 데이터 조회

- 사원 테이블에서 사원 번호가 1004인 사원 조회

```
SELECT empno 사번, ename 이름, job 직급 FROM emp
WHERE empno=1004;
```

■ 사원 테이블에서 월급이 450만원 이하인 사원 조회

```
SELECT empno 사번, ename 이름, job 직급, sal 월급 FROM emp
WHERE sal <= 450;
```

■ 사원 테이블에서 영업부 소속이 아니 사원 조회

```
SELECT empno 사번, ename 이름, deptno 부서 FROM emp
WHERE deptno <> 30;
```

▶ 날짜 데이터 조회

TO_DATE() 함수는 날짜 형식으로 구성된 문자열을 DATE 타입으로 변환해 주는 오라클 기본 함수로 이 함수를 사용할 때 년, 월, 일을 구분하는 구분자는 “/”, “-”, 공백 모두를 사용할 수 있으나 공백 보다는 구분자를 명확히 지정하고 동일한 구분자를 사용하는 것이 좋다.

■ 사원 테이블에서 2007년 3월 이전에 입사한 사원 조회

```
SELECT empno 사번, ename 이름, hiredate 입사일 FROM emp
WHERE hiredate < '2007/03/01';
```

```
SELECT empno 사번, ename 이름, hiredate 입사일 FROM emp
WHERE hiredate < '2007-03-01';
```

```
SELECT empno 사번, ename 이름, hiredate 입사일 FROM emp
WHERE hiredate < TO_DATE('2007-03-01', 'YYYY/MM/DD');
```

▶ 문자 데이터 조회

■ 사원 테이블에서 이름이 "홍길동"인 사원 조회

```
SELECT * FROM emp WHERE ename='홍길동';
```

▶ LIKE 연산자를 사용한 문자 데이터 조회

LIKE 연산자는 비교 연산자는 아니지만 비교 연산자와 같이 알아 볼 것이다.

앞에서 “=” 연산자를 사용해 이름이 “홍길동”인 사원을 검색했는데 “=” 연산자는 정확히 일치하는 데이터만을 조회 할 수 있다. 이렇게 정확히 일치하지 않고 특정 문자를 포함하거나 특정 문자로 시작하는 데이터를 조회할 때 유용하게 사용할 수 있는 것이 바로 LIKE 연산자 이다. LIKE 연산자는 조회하는 데이터가 정확히 일치하지 않아도 데이터를 검색할 수 있도록 와일드카드를 사용할 수 있는데 LIKE 연산자와 함께 사용할 수 있는 와일드카드는 아래와 같이 두 가지가 있다.

% : 문자가 없거나 하나 이상의 문자와 대체된다.

_ : 오로지 하나의 문자와 대체된다.

- 사원 테이블에서 이름이 "김"으로 시작하는 (성이 김씨인) 사원 조회
SELECT * FROM emp WHERE ename LIKE '김%';
- 사원 테이블에서 이름에 "정"자가 포함된 사원 조회
SELECT * FROM emp WHERE ename LIKE '%정%';
- 사원 테이블에서 이름에 "사"자가 두 번째 포함된 사원 조회
SELECT * FROM emp WHERE ename LIKE '_사%';
- 사원 테이블에서 이름이 "기"자로 끝나는 사원 조회
SELECT * FROM emp WHERE ename LIKE '%기';
- 사원 테이블에서 이름이 "김"으로 시작하지 않는 (성이 김씨가 아닌) 사원 조회
SELECT * FROM emp WHERE ename NOT LIKE '김%';

3.3.2 논리 연산자를 사용한 데이터 조회

지금까지 하나의 조건에 만족해 데이터를 조회하는 상황에 대해 알아보았다. 하지만 실무에서는 하나 뿐 아니라 그 이상의 조건을 만족하는 데이터를 조회하는 경우가 훨씬 많다.

예를 들어 입사일이 2006년 이후에 입사한 사원 중에 월급을 350만원 이상 받는 사원을 조회해야 할 경우, 또는 급여가 300만원부터 500만원 이하인 사원을 조회할 때는 하나의 조건으로는 해결 할 수 없다. 이렇게 조건이 하나 이상인 경우 논리 연산자를 사용하게 되는데 이 논리 연산자의 종류는 아래와 같다.

AND : 모든 조건이 참이어야 조회하는 데이터를 얻을 수 있다.

OR : 조건 중에 하나라도 참이면 조회하는 데이터를 얻을 수 있다.

NOT : 지정한 조건이 거짓이어야 조회하는 데이터를 얻을 수 있다.

▶ AND 연산자를 사용한 데이터 조회

- 사원 테이블에서 2006년 이후에 입사한 사원 중 월급이 350만원 이상인 사원 조회
SELECT empno 사번, ename 이름, sal 월급, hiredate 입사일 FROM emp
WHERE hiredate > TO_DATE('2006/12/31', 'YYYY/MM/DD') AND sal >= 350;
- 사원 테이블에서 월급이 350만원부터 500만원 사이를 받는 사원 조회
SELECT empno 사번, ename 이름, job 직급, sal 월급 FROM emp
WHERE sal >= 300 AND sal <= 500;
- BETWEEN 연산자를 이용한 월급이 350만원부터 500만원 사이를 받는 사원 조회
SELECT empno 사번, ename 이름, job 직급, sal 월급 FROM emp
WHERE sal BETWEEN 300 AND 500;

▶ OR 연산자를 사용한 데이터 조회

- 사원 테이블에서 경리부에 소속된 사원과 전산부에 소속된 사원 조회
SELECT empno 사번, ename 이름, deptno 부서 FROM emp
WHERE deptno=10 OR deptno= 40;
- 사원 테이블에서 관리자가 장동건 부장이거나 박중훈 부장인 사원 조회
SELECT empno 사번, ename 이름, mgr 관리자 FROM emp
WHERE mgr=1006 OR mgr=1013;
- IN 연산자를 이용한 경리부에 소속된 사원과 전산부에 소속된 사원 조회
SELECT empno 사번, ename 이름, deptno 부서 FROM emp
WHERE deptno IN(10, 40);

▶ NOT 연산자를 사용한 데이터 조회

- 사원 테이블에서 영업부 소속 직원이 아닌 사원 조회
SELECT empno 사번, ename 이름, deptno 부서 FROM emp
WHERE NOT deptno=30;
- 사원 테이블에서 입사일이 2007년이 아닌 사원 조회
SELECT empno 사번, ename 이름, hiredate 입사일 FROM emp
WHERE NOT (hiredate >= '2007/01/01' AND hiredate <= '2007/12/31');
- NOT BETWEEN 연산자를 이용한 입사일이 2007년이 아닌 사원 조회
SELECT empno 사번, ename 이름, hiredate 입사일 FROM emp
WHERE hiredate NOT BETWEEN '2007/01/01' AND '2007/12/31';
- NOT IN 연산자를 이용한 커미션이 80, 100, 200 이 아닌 사원 조회
SELECT empno 사번, ename 이름, job 직급, comm 커미션 FROM emp
WHERE comm NOT IN(80, 100, 200) OR comm IS NULL;

3.4 데이터 정렬과 중복 행 제거

SELECT 문으로 조회한 데이터를 정렬하기 위해 ORDER BY 절을 사용한다.
정렬에는 정렬해야 하는 데이터를 기준으로 오름차순 정렬과 내림차순 정렬이 있다.
ORDER BY 절에 정렬 옵션을 지정하지 않으면 기본 값은 오름차순 정렬이다.

▶ 데이터 정렬하여 조회하기

- 사원 테이블에서 입사일이 2007년이 아닌 사원을 조회하고 입사일로 오름차순 정렬
 SELECT empno 사번, ename 이름, hiredate 입사일 FROM emp
 WHERE NOT (hiredate >= '2007/01/01' AND hiredate <= '2007/12/31')
 ORDER BY hiredate;
- 사원 테이블에서 입사일이 2007년이 아닌 사원을 조회하고 입사일로 내림차순 정렬
 SELECT empno 사번, ename 이름, hiredate 입사일 FROM emp
 WHERE hiredate NOT BETWEEN '2007/01/01' AND '2007/12/31'
 ORDER BY hiredate DESC;
- 커미션이 80, 100, 200 이 아닌 사원을 조회하고 월급으로 내림차순 정렬
 SELECT empno 사번, ename 이름, job 직급, comm 커미션, sal 월급 FROM emp
 WHERE comm NOT IN(80, 100, 200) OR comm IS NULL
 ORDER BY sal DESC;

▶ 두 개 이상의 컬럼을 정렬해 조회하기

다음의 SQL 문은 WHERE 절은 없고 관리자 아이디 별로 오름차순 정렬하고 급여로 내림차순 정렬하는 쿼리이다. 이렇게 두 개 이상의 컬럼을 정렬하려면 정렬하려는 대상 컬럼에 정렬 옵션을 지정하고 콤마(,)로 구분해 여러 컬럼을 지정하면 된다. 또한 아래에서 관리자 아이디로 오름차순 정렬하게 되면 NULL 값은 맨 뒤에 위치하게 되는데 NULL 값을 처음으로 정렬되게 하려면 NULLS FIRST를 지정하면 된다.

- 관리자 아이디로 오름차순 정렬(null 데이터를 앞으로)하고 급여로 내림차순 정렬
 SELECT empno 사번, ename 이름, job 직급, sal 월급, mgr 매니저 FROM emp
 ORDER BY mgr ASC NULLS FIRST, sal DESC;
- 부서 아이디로 오름차순 정렬하고 부서별로 월급을 기준으로 오름차순 정렬
 SELECT ename 이름, job 직급, sal 월급, deptno 부서 FROM emp
 ORDER BY deptno, sal;

▶ DISTINCT 키워드로 중복 행 제거하기

회사에 몇 개의 부서가 존재하는지 검색하기 위해 deptno를 검색하게 되면 테이블에 저장된 사원 수 만큼의 중복된 데이터가 검색된다. 중복된 데이터를 검색에서 제외시키려면 DISTINCT 키워드를 사용해 조회하면 된다.

```
SELECT DISTINCT deptno 부서 FROM emp
ORDER BY deptno;
```

```
SELECT COUNT(DISTINCT job) FROM emp;
```

3.5 집합 연산자

집합 연산자는 여러 개의 SELECT 문을 연결해 마치 하나의 SELECT 문으로 조회한 것과 같이 취급할 수 있도록 지원하는 연산자 이다.

하나의 SELECT 문으로 조회한 데이터는 조건에 따라서 특정 테이블을 조회한 부분 집합으로써 각각의 개별 SELECT 문으로 조회한 결과 데이터를 하나의 집합으로 보고 여러 개의 SELECT 문으로 조회한 결과 데이터를 어떤 집합 연산을 적용해 연결할 지를 지정하는 연산자가 바로 집합 연산자 이다. 다시 말해 집합 연산자는 하나의 SELECT 문의 조회 결과가 하나의 집합이 되므로 여러 개의 SELECT 문의 조회 결과를 합집합(UNION), 교집합(INTERSECT), 차집합(DIFFERENCE) 형식으로 연산하여 조회할 수 있도록 지원하는 연산자를 말한다.

집합 연산자는 아래 표 3-1과 같이 4가지 유형을 사용할 수 있다.

집합 연산자	설 명
UNION	두 집합의 합집합 결과를 출력, 중복된 데이터를 제외하고 정렬함
UNION ALL	두 집합의 합집합 결과를 출력, 중복 데이터를 포함하고 정렬 안함
INTERSECT	두 집합의 교집합 결과를 출력, 정렬함
MINUS	두 집합의 차집합 결과를 출력

표 3-1 집합 연산자의 종류

집합 연산자를 사용할 경우 두 집합의 컬럼 명은 달라도 상관없으나 아래와 같이 주의해야 할 사항이 있다.

- 두 집합의 SELECT 문에 오는 컬럼의 개수가 동일해야 한다.
- 두 집합의 SELECT 문에 오는 컬럼의 데이터 형이 동일해야 한다.

3.5.1 UNION, UNION ALL 연산자를 이용해 조회하기

UNION, UNION ALL 연산자를 사용하면 각각의 SELECT 문으로 조회한 두 데이터 집합을 합쳐서 조회할 수 있다. 다시 말해 두 집합의 합집합을 조회할 수 있다. 이 둘의 차이점 있다면 UNION 연산자는 중복된 데이터를 제외시켜 합집합을 조회하고 정렬 하지만 UNION ALL 연산자는 중복된 데이터를 포함해 조회하고 정렬하지 않는다는 것이다.

▶ 그룹 활동을 하는 가수와 싱글 활동을 하는 가수 모두 조회하기

```
SELECT * FROM group_star
UNION
SELECT * FROM single_star;
```

```
SELECT * FROM group_star
UNION ALL
SELECT * FROM single_star;
```

▶ 컴퓨터 공학과에 소속된 교수와 학생 모두 조회하기

```
SELECT studno, name, deptno1
FROM student
WHERE deptno1 = 101
UNION
SELECT profno, name, deptno
FROM professor
WHERE deptno = 101;
```

```
SELECT studno, name, deptno1
FROM student
WHERE deptno1 = 101
UNION ALL
SELECT profno, name, deptno
FROM professor
WHERE deptno = 101;
```

3.5.2 INTERSECT 연산자를 이용해 조회하기

INTERSECT 연산자는 각각의 SELECT 문으로 조회한 데이터 집합에서 각 집합에 공통으로 조회된 데이터를 조회할 수 있다. 다시 말해 두 집합 모두에 소속된 데이터 즉 교집합을 조회할 수 있다.

▶ 그룹 활동과 싱글 활동 모두를 하는 가수 조회하기

```
SELECT * FROM group_star
INTERSECT
SELECT * FROM single_star;
```

▶ 컴퓨터 공학과 전자 공학을 복수 전공하는 학생 조회하기

```
SELECT studno 학번, name 이름, deptno1 학과
FROM student
WHERE deptno1 = 101
INTERSECT
SELECT studno 학번, name 이름, deptno1 학과
FROM student
WHERE deptno2 = 201;
```

3.5.3 MINUS 연산자를 사용해 조회하기

MINUS 연산자는 첫 번째 SELECT 문으로 조회한 데이터 집합에서 두 번째 SELECT 문으로 조회한 데이터 집합을 제외한 나머지를 조회할 수 있다. 다시 말해 첫 번째 쿼리의 결과에서 두 번째 쿼리의 결과에 존재하는 데이터를 제외한 나머지 데이터를 조회할 수 있다. 즉 차집합을 조회할 수 있다.

다.

▶ 싱글 활동만 하는 가수 조회하기

```
SELECT * FROM single_star  
MINUS  
SELECT * FROM group_star;
```

▶ 전임강사를 제외한 교수 조회하기

```
SELECT name, position FROM professor  
MINUS  
SELECT name, position FROM professor  
WHERE position='전임강사';
```

▶ 전체 사원 중에서 입사일이 2006년 이후에 입사한 직원 조회하기

```
SELECT empno 사번, ename 이름, hiredate 입사일 FROM emp  
MINUS  
SELECT empno 사번, ename 이름, hiredate 입사일 FROM emp  
WHERE hiredate < TO_DATE('2006/12/31', 'YYYY/MM/DD');
```


4. 오라클 기본함수

오라클은 다양한 형식의 데이터를 효율적으로 처리하기 위한 여러 함수를 제공하고 있다.

오라클에서 사용할 수 있는 함수는 오라클을 설치하면 기본적으로 내장되어 있는 내장 함수와 오라클이 제공하는 PL/SQL을 이용해 사용자가 직접 정의해 사용할 수 있는 사용자 정의 함수로 나눌 수 있다. 오라클에 기본 내장되어 있는 내장 함수를 기본함수라고도 부르며 우리는 이장에서 기본함수에 대해서 알아 볼 것이다.

내장 함수는 크게 단일 행 함수와 다중 행 함수로 나눌 수 있는데 단일 행 함수는 단일 행의 컬럼 값을 입력 인수로 사용해 각각의 행에 대해 값을 반환하는 함수이고 다중 행 함수는 여러 행의 컬럼 값을 입력 인수로 사용해 단 하나의 행을 반환하는 함수이다.

단일 행 함수에는 각 행에 대해 결과 값을 반환하는 숫자 함수, 문자 함수, 날짜 함수, 변환 함수, 조건 함수 등이 있으며 다중 행 함수에는 여러 행을 묶어 연산하는 그룹 함수가 있다.

4.1 지정할 테이블이 없을 때 DUAL을 사용하자

먼저 단일 행 함수 이전에 함수나 수식을 사용해 결과 값을 얻을 때 유용하게 사용할 수 있는 DUAL 이라는 모조 테이블(Dummy Table)에 대해 알아보자.

SQL에서 함수나 수식을 사용해 결과 값을 얻으려면 SELECT 문을 사용해야 하는데 SELECT 문은 FROM 절을 사용해 반드시 데이터를 읽어올 테이블이나 뷰를 지정해야 한다. 그렇지 않으면 FROM 절이 없다는 오류를 보게 될 것이다. 예를 들어 아래와 같이 ROUND 함수를 사용해 지정한 수의 소수점 둘째 자리에서 반올림한 결과를 얻고자 했지만 결과는 커녕 오류만 발생하고 말았다. 하지만 아래와 같은 SELECT 문은 어느 테이블에서도 읽어 올 수 있는 데이터가 아니기 때문에 마땅히 지정할 테이블이 없다. 바로 이럴 때 FROM 절에 지정할 수 있는 테이블이 DUAL 테이블 인 것이다.

```
SELECT ROUND(30.257, 1); <- 오류 발생
```

DUAL 테이블은 SYS 스키마에 속한 SYS.DUAL 테이블로 오라클 사용자 누구나 접근할 수 있도록 공개 시노님(PUBLIC SYNONYM)으로 만들어져 있다. 그래서 어느 사용자로 접속하든 DUAL 테이블을 사용할 수 있는 것이다.

4.2 숫자 함수

숫자 함수는 입력 인수로 숫자 형 데이터(NUMBER 등등)를 입력 받아 함수 내부에서 처리하고 그 결과 또한 숫자로 반환하는 함수를 숫자 함수(Numeric Function)라 한다.

▶ 지정한 자리에서 반올림 하는 함수

- ROUND(n, i)

ROUND 함수는 인수로 넘어오는 n의 값을 소수점 이하 i + 1 번째 자리에서 반올림해 반환하는 함수로 i 는 생략가능하다. i 가 생략되면 0 으로 인식한다. 예를 들어 i 가 3이라면 소수점 네 번째

자리에서 반올림한 수가 반환되고 i 가 생략되면 소수점 첫 번째 자리에서 반올림한 수가 반환된다. 또한 i 가 음수면 소수점을 기준으로 좌측 i 번째 자리에서 반올림된 수가 반환된다.

```
SELECT ROUND(7.123456789), ROUND(7.123456789, 0) FROM dual;
```

```
SELECT ROUND(7.123456789, 2), ROUND(7.123456789, 5) FROM dual;
```

```
SELECT ROUND(77777.12345, -2) FROM dual;
```

▶ 지정한 자리에서 잘라내는 함수

- TRUNC(n, i)

TRUNC 함수는 Truncate의 약자로 인수로 넘어오는 n의 값을 소수점 i + 1 번째에서 잘라낸 수를 반환하는 함수로 i 는 생략가능하다. i 가 생략되면 0 으로 인식한다. 예를 들어 i 가 3이라면 소수점 네 번째 자리를 잘라낸 수가 반환되고 i 가 생략되면 소수점 첫 번째 자리를 잘라낸 수가 반환된다. 또한 i 가 음수면 소수점을 기준으로 좌측 i 번째 자리에서 잘라낸 수가 반환된다.

```
SELECT TRUNC(7.123456789), TRUNC(7.123456789, 0) FROM dual;
```

```
SELECT TRUNC(7.123456789, 2), TRUNC(7.123456789, 5) FROM dual;
```

```
SELECT TRUNC(77777.12345, -2) FROM dual;
```

▶ 입력된 값과 작거나 같은 가장 큰 정수를 구하는 함수

- FLOOR(n)

FLOOR 함수는 인수로 넘어오는 n보다 작거나 같은 가장 큰 정수를 반환하는 함수로 인수로 넘어오는 값이 정수일 경우 넘어온 수 그대로 반환되고 양의 실수일 경우 소수점 아래 부분을 잘라낸 값이 반환된다. 또한 음의 실수일 경우 소수점 아래 부분을 잘라내고 -1을 더한 값이 반환된다.

```
SELECT FLOOR(7.12345), FLOOR(-7.12345), FLOOR(7), FLOOR(0.12345) FROM dual;
```

▶ 나머지를 구하는 함수

- MOD(n1, n2)

MOD 함수는 인수로 넘어오는 n1을 n2로 나눈 나머지를 구하여 반환하는 함수로 n2를 생략할 수는 없고 0은 지정할 수 있다. n2가 0이면 n1 값 그대로 반환한다.

```
SELECT MOD(7, 0), MOD(7, 2), MOD(7, -2) FROM dual;
```

▶ 제곱 값을 구하는 함수

- POWER(n1, n2)

POWER 함수는 인수로 넘어오는 n1을 n2 만큼 제공하여 반환하는 함수로 n2의 값은 실수를 지정할 수 있으나 이 경우 n1의 값은 양수여야 한다. n1의 값이 음수이고 n2가 실수이면 오류가 발생한다.

```
SELECT POWER(7, 0), POWER(7, 3) FROM dual;
```

```
SELECT POWER(7, 2.1) FROM dual;
```

```
SELECT POWER(-7, 2.1) FROM dual; <- 오류 발생
```

▶ 절대 값을 구하는 함수

- ABS(n)

함수명 ABS는 Absolute의 약자로 인수로 넘어오는 n의 절대 값을 반환한다.

```
SELECT ABS(-7) 음수, ABS(7) 양수 FROM dual;
```

4.3 문자 함수

문자 함수는 처리 결과를 문자형 데이터(CHAR, VARCHAR, VARCHAR2 등등)와 숫자 형 데이터(NUMBER)로 반환 한다. 숫자 함수는 숫자 형 데이터만 인수로 받아 그 결과로 숫자 형 데이터만을 반환 했지만 문자 함수의 인수와 처리 결과는 문자형 데이터뿐만 아니라 숫자 형 데이터도 될 수 있다.

▶ 대소문자 변환 함수

- UPPER(string), LOWER(string)

UPPER 함수는 인수로 넘어오는 문자열 string을 모두 대문자로 변환해 반환하는 함수이며 LOWER 함수는 string을 모두 소문자로 변환해 반환하는 함수이다.

```
SELECT UPPER('Hello ORACLE'), LOWER('HELLO Oracle') FROM dual;
```

▶ 일부만 추출하는 함수

- SUBSTR(string, position, length)

SUBSTR 함수는 인수로 넘어오는 문자열 string에서 position에 지정한 위치부터 length에 지정한 문자의 개수만큼 문자열을 추출해 주는 함수이다. 이외에 SUBSTRB(string, position, length)라는 함수가 있는데 이 함수 또한 position 위치부터 length에 지정한 만큼 문자를 추출해 주지만 position과 length 단위가 문자수가 아니라 Byte 수를 의미한다.

참고로 Oracle 11g나 Oracle 12c 에서는 이미 아스키코드에도 정의되어 있는 영문, 숫자, 일부 특수 문자는 1Byte로 저장되지만 한글과 같은 유니코드 문자는 “NLS_DATABASE_PARAMETERS”라는 시스템 뷰의 NLS_CHARACTERSET의 값이 “KO16MSWIN949”로 설정되어 있으면 한글을

2Byte로 취급하고 “AL32UTF8”로 설정되어 있으면 한글을 3Byte로 취급한다.

```
SELECT SUBSTR('Oracle Database Express Edition', 8, 8) FROM dual;
```

```
SELECT SUBSTR('오라클 데이터베이스 익스프레스 에디션', 5, 6) FROM dual;
```

```
SELECT SUBSTRB('Oracle Database Express Edition', 8, 8) FROM dual;
```

```
SELECT SUBSTRB('오라클 데이터베이스 익스프레스 에디션', 5, 6) FROM dual;
```

▶ 일부만 다른 문자열로 대체하는 함수

- REPLACE(string, search, replace)

REPLACE 함수는 인수로 넘어오는 문자열 string에서 search에 지정한 문자열을 찾아 replace로 지정한 문자열로 대체하여 반환하는 함수이다.

```
SELECT REPLACE('Oralce is difficult', 'difficult', 'easy') FROM dual;
```

```
SELECT position, REPLACE(position, '전임', '시간') FROM professor;
```

▶ 끝에 있는 문자열을 제거하는 함수

- LTRIM(string[, set]), RTRIM(string[, set])

LTRIM 함수는 인수로 넘겨받은 문자열 string에서 set으로 지정한 문자열을 왼쪽에서 제거한 결과를 반환하는 함수이며 RTRIM 함수는 string에 지정한 문자열에서 set으로 지정한 문자열을 오른쪽에서 제거한 결과를 반환하는 함수이다. set은 생략이 가능하며 생략할 경우 공백문자 한 자로 간주해 적용된다.

```
SELECT position, LTRIM(position, '전임') FROM professor;
```

```
SELECT dname, RTRIM(dname, '부') FROM dept;
```

▶ 양쪽 끝에 있는 문자열을 제거하는 함수

- TRIM([LEADING, TRAILING, BOTH] [trim_char] [FROM] trim_string)

TRIM 함수는 LTRIM 함수와 RTRIM 함수의 기능을 합쳐놓은 것과 같은 결과를 반환하는 함수로 인수로 넘겨받은 문자열 trim_string에서 trim_char에 지정한 문자를 양쪽에서 제거한 결과를 반환한다. trim_char는 생략할 수 있고 생략되면 LTRIM, RTRIM 함수와 같이 공백이 적용된 결과를 반환한다. 한 가지 주의할 사항은 trim_char에 지정하는 인수는 문자열이 아니라 한 문자만 지정할 수 있다. 만약 trim_char에 문자열을 지정하면 오류가 발생된다. trim_char에 한 문자만 지정할 수 있지만 양쪽 끝에 해당하는 문자가 여러 개 존재하면 이들 모두가 제거된다. 또한 LEADING 옵션을 지정하면 왼쪽, TRAILING은 오른쪽 그리고 BOTH는 양쪽에서 trim_char에 지정한 문자를 제거한 결과를 반환한다. 이들 옵션을 지정하지 않으면 BOTH 옵션이 기본 적용된다.

```
SELECT '    Hello Oracle    ', TRIM(BOTH FROM '    Hello Oracle    ')
FROM dual;
```

```
SELECT 'AABBCCDDAA', TRIM('A' FROM 'AABBCCDDAA') FROM dual;
```

▶ 문자열을 연결하여 반환하는 함수

- CONCAT(string1, string2)

CONCAT 함수는 인수로 넘어오는 문자열 string1에 string2를 연결해 반환하는 함수이다.

```
SELECT CONCAT('Oracle', ' Database') FROM dual;
```

▶ 문자열 길이를 반환하는 함수

- LENGTH(string)

LENGTH 함수는 인수로 넘어오는 문자열 string의 길이를 반환하는 함수로 여기서 길이는 글자 수를 의미한다. 이외에도 LENGTHB(string)라는 함수가 있는데 이 함수 또한 인수로 넘어온 문자열 string의 길이를 반환하지만 글자 수가 아니라 Byte 수를 반환한다. LENGTH 함수는 아스키코드에 정의된 1Byte 크기인 영문자, 공백, 숫자, 일부 특수문자와 3Byte 크기인 유니코드 문자 모두 하나의 문자를 1로 취급하지만 LENGTHB 함수는 아스키코드에 정의된 문자 하나를 1Byte로 유니코드 문자 하나를 3Byte로 취급한다.

```
SELECT id, LENGTH(id) "char", LENGTHB(id) "byte" FROM professor;
```

```
SELECT job, LENGTH(job) "char", LENGTHB(job) "byte" FROM emp;
```

```
SELECT LENGTH('홍길동 a#1') "char", LENGTHB('홍길동 a#1') "byte" FROM dual;
```

4.4 날짜 함수

날짜 함수는 날짜와 시간 데이터를 다루기 위해 오라클에서 제공하는 날짜 형 데이터(DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE 등등)를 대상으로 연산한 후 그 결과를 반환한다. 이때 날짜 형이라 함은 날짜와 시간 데이터를 포함하는 데이터를 말한다. 또한 날짜 함수가 반환하는 데이터는 대부분 DATE 형이지만 일부 날짜 함수는 숫자를 반환하기도 한다.

▶ 현재 시간을 반환하는 함수

- SYSDATE, CURRENT_DATE, SYSTIMESTAMP, CURRENT_TIMESTAMP

SYSDATE 함수와 SYSTIMESTAMP 함수는 현재 오라클이 구동되는 시스템 시간을 반환하는 함수이고 CURRENT_DATE 함수와 CURRENT_TIMESTAMP 함수는 현재 세션(Session)의 시간대(TIME ZONE)를 기준으로 현재 날짜를 반환하는 함수이다. 여기서 세션이란 오라클에 접속되어 있는 상태

를 말한다. 오라클 데이터베이스와 사용자가 같은 시간대(TIME ZONE)에 위치한 다면 아래와 같은 SQL 쿼리를 실행하면 SYSDATE 함수와 CURRENT_DATE 함수의 반환 값은 아래 그림과 같이 동일한 날짜를 반환한다.

```
SELECT SYSDATE, CURRENT_DATE FROM dual;
```

SYSDATE	CURRENT_DATE
2015-09-07 22:09:14	2015-09-07 22:09:14

오라클 서버와 접속 세션의 시간대가 같은 경우

하지만 오라클 데이터베이스 서버는 대한민국 서울의 본사에 있고 어떤 직원이 영국 런던으로 출장 가서 보고서를 작성하기 위해 본사에 있는 오라클 서버에 접속한 상태라면 오라클 서버와 사용자가 접속한 세션의 시간대가 다르므로 위의 SQL 쿼리는 아래 그림과 같이 서로 다른 날짜를 반환하게 된다.

SYSDATE	CURRENT_DATE
2015-09-07 22:11:54	2015-09-07 13:11:54

오라클 서버와 접속 세션의 시간대가 다른 경우

위에서 오라클 서버와 접속 세션의 시간대가 다른 경우에 대한 테스트는 현재 접속한 HR 계정의 세션 시간대를 영국 런던의 시간대로 변경하고 위의 쿼리를 실행한 결과이다.

영국의 런던은 이 도시 외곽에 있는 그리니치 천문대와 같은 시간대를 사용하고 있으며 그리니치 천문대는 경도 0도에 위치해 있다. 서울의 시간대는 그리니치 표준시 보다 9시간이 빠르다.

위의 4가지 함수는 모두 인수를 지정하지 않고 사용할 수 있다. 다만 SYSTIMESTAMP 함수와 CURRENT_TIMESTAMP 함수는 정밀도에 대한 인수를 0 ~ 9 까지 지정할 수는 있지만 생략 가능하다. 나머지 2가지 함수는 파라미터 자체가 없으므로 인수를 지정할 수 없다.

SYSDATE 함수와 CURRENT_DATE 함수의 반환 타입은 DATE 형이고 SYSTIMESTAMP 함수와 CURRENT_TIMESTAMP 함수의 반환 타입은 TIMESTAMP WITH TIME ZONE 형이다.

```
SELECT SYSTIMESTAMP(0), CURRENT_TIMESTAMP(9) FROM dual;
```

SYSTIMESTAMP(0)	CURRENT_TIMESTAMP(9)
15/09/07 22:42:22.000000000 +09:00	15/09/07 13:42:22.483000000 +00:00

▶ 포맷 조건을 기준으로 날짜 데이터를 반올림하는 함수

- ROUND(date, fmt)

ROUND 함수는 숫자뿐만 아니라 날짜 데이터의 반올림에도 사용할 수 있는 함수이며 인수로 넘겨 받은 날짜 데이터 date를 포맷 모델인 fmt에 지정한 단위로 반올림한 결과를 반환한다.

포맷 모델	반올림 단위
CC, SCC	4자리 연도의 끝 두 글자를 기준으로 반올림
SYYYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	년을 기준으로 반올림(7월 1일부터 반올림됨)
IYYY, IY, I	ISO 표준 년을 기준으로 반올림
Q	분기를 기준으로 반올림 (한 분기의 두 번째 달 16일부터 반올림 됨)
MONTH, MON, MM, RM	월을 기준으로 반올림(16일부터 반올림 됨)
DDD, DD, J	일을 기준으로 반올림

DAY, DY, D	한 주가 시작되는 날짜를 기준으로 반올림(일요일 시작)
HH, HH12, HH24	시를 기준으로 반올림
MI	분을 기준으로 반올림

표 4-1 ROUND 함수와 TRUNC 함수의 포맷 모델

```

SELECT ROUND(SYSDATE, 'CC') CENTURY,
       ROUND(SYSDATE, 'YYYY') YEAR,
       ROUND(SYSDATE, 'Q') QUARTER,
       ROUND(SYSDATE, 'MONTH') MONTH,
       ROUND(SYSDATE, 'DAY') WEEK,
       ROUND(SYSDATE, 'DD') DAY,
       ROUND(SYSDATE, 'HH24') H,
       ROUND(SYSDATE, 'MI') M
FROM dual;

```

▶ 포맷 조건을 기준으로 날짜 데이터를 잘라내는 함수
- TRUNC(date, fmt)

TRUNC 함수 또한 ROUND 함수와 같이 숫자뿐만이 아니라 날짜 데이터를 잘라내는데 사용할 수 있는 함수이며 인수로 넘겨받은 날짜 데이터 date를 포맷 모델인 fmt에 지정한 단위로 잘라낸 결과를 반환한다.

```

SELECT TRUNC(SYSDATE, 'CC') CENTURY,
       TRUNC(SYSDATE, 'YYYY') YEAR,
       TRUNC(SYSDATE, 'Q') QUARTER,
       TRUNC(SYSDATE, 'MONTH') MONTH,
       TRUNC(SYSDATE, 'DAY') WEEK,
       TRUNC(SYSDATE, 'DD') DAY,
       TRUNC(SYSDATE, 'HH24') H,
       TRUNC(SYSDATE, 'MI') M

```

```
FROM dual;
```

▶ 두 날짜 사이의 개월 수를 구하는 함수

- MONTHS_BETWEEN(date1, date2)

MONTHS_BETWEEN 함수는 인수로 넘겨받은 날짜 데이터 date2에서 date1 까지 개월 수를 구하는 함수로 $date1 - date2$ 의 결과를 개월 수로 반환한다.

```
SELECT SYSDATE today, hiredate 입사일,  
       ROUND(MONTHS_BETWEEN(SYSDATE, hiredate)) months  
FROM emp;
```

▶ 날짜에 개월 수를 더하는 함수

- ADD_MONTHS(date, integer)

ADD_MONTHS 함수는 인수로 넘겨받은 날짜 데이터 date에 integer로 지정한 개월 수를 더하여 날짜 데이터(DATE)로 반환하는 함수로 integer가 양수이면 date에 integer에 지정한 개월 수를 더한 날짜 데이터를 반환하고 음수이면 date에 integer에 지정한 개월 수를 빼서 반환한다.

```
SELECT SYSDATE today, ADD_MONTHS(SYSDATE, 12) FROM dual;
```

```
SELECT SYSDATE today, ADD_MONTHS(SYSDATE, -3) FROM dual;
```

▶ 지정한 요일에 가장 가까운 날짜를 구하는 함수

- NEXT_DAY(date, string)

NEXT_DAY 함수는 인수로 넘겨받은 date의 날짜 이후에 string에 지정한 요일이 최초로 도래하는 날짜를 반환하는 함수이다. string에 지정할 수 있는 문자열은 요일의 이름을 지정하거나 그 요일에 해당하는 정수를 지정하면 된다. 정수를 지정할 때는 일요일부터 토요일까지 1부터 7사이의 값을 지정할 수 있다. 이외의 문자나 다른 숫자를 지정하면 오류가 발생한다.

요일에 이름은 우리가 일상적으로 사용하는 “월”, “화”... “토”와 같은 축약형도 가능하다.

```
SELECT SYSDATE today, NEXT_DAY(SYSDATE, 2) "다음 월요일" FROM dual;
```

```
SELECT SYSDATE today, NEXT_DAY(SYSDATE, '월') "다음 월요일" FROM dual;
```

▶ 지정한 달의 마지막 날짜를 구하는 함수

- LAST_DAY(date)

LAST_DAY 함수는 인수로 넘겨받은 날짜 데이터 date와 같은 달의 마지막 날짜를 반환하는 함수이다. 예를 들면 date의 날짜가 9월에 속해 있으면 9월 30일을 반환한다.

```
SELECT SYSDATE today, LAST_DAY(SYSDATE) lastday FROM dual;
```


4.5 변환 함수

오라클에서 연산을 하게 되면 많은 부분 자동으로 데이터의 형을 변환을 해 준다. 이렇게 자동으로 데이터 형을 변환하는 것을 묵시적(암시적) 형 변환(Implicit) 이라고 한다.

오라클이 연산을 수행할 때 대부분 알아서 데이터 형을 변경해 주지만 때로는 사용자가 데이터 형을 변환해야 할 때가 있다. 이렇게 사용자가 강제로 데이터의 형을 변환하는 것을 명시적(Explicit) 형 변환이라 한다. 오라클은 사용자가 명시적으로 데이터 형을 변환할 수 있도록 형 변환 함수를 제공하고 있는데 이러한 함수에 대해 알아보도록 하자.

▶ 문자형 데이터로 변환해 주는 함수

- TO_CHAR(arg)

TO_CHAR 함수는 인수로 넘겨받은 데이터 arg를 문자열로 반환하는 함수이다.

arg에는 문자 데이터, 숫자 데이터, 날짜 데이터를 지정할 수 있다.

■ 날짜 데이터를 문자 데이터로 변환

```
SELECT SYSDATE, TO_CHAR(SYSDATE) FROM dual;
```

```
SELECT SYSDATE,  
       TO_CHAR(SYSDATE, 'YYYY/MM/DD HH24:MI:SS DY')  
FROM dual;
```

```
SELECT hiredate,  
       TO_CHAR(hiredate,  
               'YYYY"년" MM"월" DD"일" HH24"시" MI"분" SS"초("DAY")')  
FROM emp;
```

■ 숫자 데이터를 문자 데이터로 변환

```
SELECT 1234567, TO_CHAR(12345678, 'L99,999,999') FROM dual;
```

```
SELECT 12345.678, TO_CHAR(12345.678, '$999,999.0000') FROM dual;
```

▶ 날짜형 데이터로 변환해 주는 함수

- TO_DATE(arg, fmt)

TO_DATE 함수는 인수로 넘겨받은 데이터 arg를 DATE 타입으로 변환하여 반환하는 함수로 arg를 fmt에 지정한 날짜 포맷으로 변환한다. TO_DATE 함수와 더불어 많이 사용되는 함수에는 TO_TIMESTAMP(string, fmt) 함수가 있다. 이 TO_TIMESTAMP 함수는 인수로 넘겨받은 데이터

arg를 TIMESTAMP 타입으로 변환하여 반환한다.

```
SELECT TO_TIMESTAMP('2015-09-09 13:05:37',  
    'YYYY-MM-DD HH24:MI:SS') FROM dual;
```

```
SELECT * FROM emp WHERE hiredate > 20070330; <-- 오류 발생
```

```
SELECT * FROM emp WHERE hiredate > TO_DATE(20070330);
```

```
SELECT * FROM emp WHERE hiredate > TO_DATE('2007-03-30');
```

```
SELECT * FROM emp WHERE hiredate > '2007-03-30';
```

▶ 숫자형 데이터로 변환해 주는 함수

- TO_NUMBER(arg, fmt)

TO_NUMBER 함수는 인수로 넘겨받은 데이터 arg를 숫자형 데이터로 변환하여 반환하는 함수이다. 오라클은 숫자 형식으로 이루어진 문자열 데이터를 4칙 연산하게 되면 자동으로 숫자 형으로 변환한 후 4칙 연산을 한다. 하지만 문자열에 통화 기호나 천 단위 구분자 콤마(.)가 포함되어 있으면 문자열 데이터를 숫자 형으로 변환하지 못해 연산 오류가 발생하게 된다. 이럴 때 유용하게 사용할 수 있는 함수가 바로 TO_NUMBER 함수이다.

```
SELECT '5700' + '3000' FROM dual;
```

```
SELECT '$5,700' + '$3,000' FROM dual; <-- 오류 발생
```

```
SELECT TO_NUMBER('$5,700', '$999,999')  
    + TO_NUMBER('$3,000', '$999,999') FROM dual;
```

▶ NULL을 다른 값으로 변환해 주는 함수

- NVL(expr1, expr2)

NVL 함수는 NULL 값을 0 이나 다른 값으로 변환할 때 사용하는 함수로 인수로 넘겨받은 expr1이 NULL이면 expr2의 값으로 변환해 반환하고 expr1이 NULL이 아니면 expr1 값을 반환한다. 일반적으로 테이블에서 키를 제외한 컬럼은 NULL을 허용하는 경우가 있는데 이 NULL 데이터를 연산에 참여 시키거나 WHERE 절의 조건에 포함시킬 때 원하는 결과를 얻지 못하거나 오류의 원인이 된다.

아래 SQL 쿼리는 emp 테이블에서 직원들의 급여와 커미션을 포함한 급여 합계를 계산해 급여 합계가 300 이상인 직원을 조회하려고 한 것이다. 하지만 comm 컬럼의 데이터 중에 NULL 데이터가 저장된 행들은 sal + comm 결과가 NULL이 되므로 제대로 계산되지 못한다. 또한 WHERE 절에서도 제대로 조건을 체크할 수 없어 급여 합계가 300 이상인 직원이 조회되지 않고 누락되는 현상이 나타난다.

```
SELECT empno 사번, ename 이름, job 직급, sal 급여, sal + comm "급여 합계"
FROM emp
WHERE sal + comm > 300;
```

이럴 때 아래와 같이 NVL 함수를 이용하면 원하는 데이터를 조회할 수 있다.

```
SELECT empno 사번, ename 이름, job 직급, sal 급여,
       sal + NVL(comm, 0) "급여 합계" FROM emp
WHERE sal + NVL(comm, 0) > 300;
```

아래와 같이 NVL2 함수를 사용해 위와 동일한 결과를 얻을 수 있다.

```
- NVL2(expr1, expr2, expr3)
  expr1이 NULL 이면 expr3를 expr1이 NULL이 아니면 expr2를 반환 한다.
```

```
SELECT empno 사번, ename 이름, job 직급, sal 급여,
       NVL2(comm, sal + comm, sal) "급여 합계" FROM emp
WHERE NVL2(comm, sal + comm, sal) > 300;
```

4.6 조건 함수

조건 함수는 프로그래밍 언어에서 많이 사용되는 조건문 유형인 if~else문 또는 switch문과 비슷한 기능을 제공하는 함수를 말하는데 앞에서 알아본 함수들 보다 편리한 기능을 제공하기 때문에 활용도가 매우 높다. 이러한 조건 함수에는 DECODE 함수와 CASE 함수가 있다.

4.6.1 DECODE 함수

DECODE 함수는 switch 문 또는 if~else문과 유사한 기능을 제공하는 함수로 여러 가지 경우에 대해 조건과 일치하는 하나를 선택해 해당하는 데이터를 반환하는 함수이다.

DECODE 함수는 아래와 같은 형식으로 사용되며 아래에서 첫 번째 인수인 expression은 테이블의 특정 컬럼을 지정해 그 컬럼의 값을 비교할 수도 있고 연산식을 지정해 그 연산의 결과 값을 조건에 해당하는 condition1, condition2, ... 등과 비교해 해당하는 하나의 값을 결과로 반환한다. 다시 말해 DECODE 함수는 첫 번째 인수인 expression과 조건인 condition1, condition2, ... 등을 비교해 condition1에 해당하면 결과로 result1을 반환하고 condition2에 해당하면 결과로 result2를 반환한다. 만약 지정한 모든 조건이 해당되지 않으면 default를 반환한다. default는 생략할 수 있으며 만약 default를 생략한 경우에 일치하는 조건이 없다면 NULL이 반환된다.

DECODE 함수를 사용할 때 주의해야할 사항은 expression과 비교 대상인 조건(condition1, condition2, ... 등)은 서로 데이터 타입이 같아야 한다. 데이터 타입이 같지 않더라도 오라클에서 자동으로 형 변환이 가능한 타입이라면 문제 되지 않지만 자동으로 변환할 수 없는 경우에는 오류가 발생한다.

```
DECODE(expression, condition1, result1, condition2, result2, ... , default)
```

▶ 학과번호와 학과명 조회

```
SELECT name 이름, pay 급여,  
       DECODE(deptno,  
              101, '컴퓨터공학과',  
              102, '멀티미디어공학과',  
              103, '소프트웨어공학과',  
              201, '전자공학과',  
              202, '기계공학과',  
              203, '화학공학과',  
              301, '문헌정보학과') 학과  
FROM professor;
```

▶ 사원 테이블에서 15년 근속자와 20년 근속자 조회

```
SELECT empno 사번, ename 이름, hiredate 입사일,  
       DECODE(ROUND((SYSDATE - hiredate) / 365),  
              15, '15년 근속',  
              20, '20년 근속',  
              '-') 근속여부  
FROM emp;
```

4.6.2 CASE 함수

CASE 함수는 DECODE 함수와 마찬가지로 여러 가지 경우에 대해 하나를 선택해 해당하는 데이터를 반환하지만 DECODE 함수에 비해 보다 다양한 조건을 제시할 수 있다는 장점이 있다.

DECODE 함수는 조건과 비교할 때 동등 연산자(=)를 사용하지만 CASE 함수는 여러 가지 비교 연산자를 사용해 조건을 제시할 수 있다.

CASE 함수는 아래와 같이 기본형과 검색형 두 가지 유형이 존재한다.

기본형은 DECODE 함수와 동일한 동등연산을 수행해 값을 비교하고 해당하는 결과를 반환한다.

아래에서 기본형 CASE 함수의 사용법을 살펴보면 기준이 되는 비교대상과 WHEN 절의 비교 값을 비교해 일치하는 하나가 선택되어 THEN 절에 지정한 결과를 반환한다. 비교대상과 WHEN의 비교 값이 모두 일치하지 않을 경우 ELSE에 지정한 기본 결과 값이 반환된다. 검색형은 CASE WHEN 다음에 조건이 오는데 이 조건에 if문과 같이 여러 비교 연산자를 사용해 다양한 비교 조건을 제시할 수 있다.

검색형은 조건에 다양한 비교조건을 사용할 수 있기 때문에 동등연산만이 아닌 다양한 형식으로 비교해야 할 경우 유용하게 사용할 수 있다.

▶ 기본형 CASE 함수

```
CASE 비교대상 WHEN 비교 값1 THEN 결과1  
              WHEN 비교 값2 THEN 결과2  
              ....
```

```
        ELSE 기본 결과 값
    END
```

▶ **검색형 CASE 함수**

```
CASE WHEN 조건1 THEN 결과1
      WHEN 조건2 THEN 결과2
      .....
      ELSE 기본 결과값
    END
```

▶ **사원의 성별을 조회**

```
SELECT empno 사번, ename 이름,
       DECODE(gender, 'M', '남성', 'F', '여성') decode_gender,
       CASE gender WHEN 'M' THEN '남성'
                WHEN 'F' THEN '여성'
                ELSE ''
       END case_gender
FROM emp;
```

▶ **사원의 연령대 조회**

```
SELECT empno 사번, ename 이름, job 직급, birthday 생년월일,
       CASE WHEN ROUND((SYSDATE - birthday) / 365) >= 20
            AND ROUND((SYSDATE - birthday) / 365) <= 29
            THEN '20대'
            WHEN ROUND((SYSDATE - birthday) / 365) >= 30
            AND ROUND((SYSDATE - birthday) / 365) <= 39
            THEN '30대'
            WHEN ROUND((SYSDATE - birthday) / 365) >= 40
            AND ROUND((SYSDATE - birthday) / 365) <= 49
            THEN '40대'
            WHEN ROUND((SYSDATE - birthday) / 365) >= 50
            AND ROUND((SYSDATE - birthday) / 365) <= 59
            THEN '50대'
            ELSE '기타'
       END 연령대
FROM emp
ORDER BY 생년월일 DESC;
```

위의 SQL 쿼리를 아래와 같이 BETWEEN을 사용해도 같은 결과를 얻을 수 있다.

```

SELECT empno 사번, ename 이름, job 직급, birthday 생년월일,
CASE WHEN ROUND((SYSDATE - birthday) / 365) BETWEEN 20 AND 29
      THEN '20대'
      WHEN ROUND((SYSDATE - birthday) / 365) BETWEEN 30 AND 39
      THEN '30대'
      WHEN ROUND((SYSDATE - birthday) / 365) BETWEEN 40 AND 49
      THEN '40대'
      WHEN ROUND((SYSDATE - birthday) / 365) BETWEEN 50 AND 59
      THEN '50대'
      ELSE '기타'
END 연령대
FROM emp
ORDER BY 생년월일;

```

5. 그룹함수

그룹함수는 집계함수라고도 부르며 여러 개의 데이터를 그룹별로 구분해 통계 데이터를 얻기 위해 주로 사용된다. 이 그룹함수에 속한 함수는 그룹별로 합계(SUM), 평균(AVG), 개수(COUNT), 최댓값(MAX), 최솟값(MIN) 등을 조회할 수 있는 함수들이다. 그룹함수는 조회하는 데이터를 그룹으로 묶는 GROUP BY 절과 그룹의 결과를 제한하는 HAVING 절과 같이 많이 사용된다.

5.1 집계함수

집계란 단어를 국어사전에서 찾아보니 “이미 된 계산들을 한데 모아서 계산함. 또는 그런 계산”으로 정의되어 있다. 집계함수는 이와 같이 여러 행의 데이터를 그룹으로 묶어 합계, 평균, 개수, 최댓값, 최솟값을 구하는 연산을 수행하고 그 결과를 단일 행 값으로 반환하는 함수를 말한다.

▶ 전체 사원수와 급여 총액 조회하기

```
SELECT COUNT(gender) || '명', TO_CHAR(SUM(sal),  
    'L999,999' ) || '만원' "급여 총액" FROM emp;
```

▶ 회사의 직급 수 조회하기

집계함수는 DISTINCT를 지정할 수 있어 중복되는 직급을 제거하고 실제 직급의 수를 조회할 수 있다. 별도로 지정하지 않으면 기본으로 ALL이 적용된다.

```
SELECT COUNT(job) 직급수 FROM emp;
```

```
SELECT COUNT(DISTINCT job) 직급수 FROM emp;
```

▶ 최고 급여와 최저 급여 조회하기

```
SELECT MAX(sal) || '만원' 최고급여, MIN(sal) || '만원' 최저급여 FROM emp;
```

▶ 그룹함수와 NULL의 관계

집계함수는 다른 함수와 달리 컬럼 값이 NULL일 경우 NULL을 인식하지 못해서 연산 대상에서 제외시킨다. 즉 컬럼 값이 NULL이 아닌 행을 대상으로 연산하여 결과를 반환한다.

COUNT 함수는 NULL을 연산에서 제외하지만 “*”가 인수로 지정되면 NULL 이나 중복 값을 가진 행을 모두 포함한다. 즉 테이블의 모든 행의 개수가 반환된다.

```
SELECT SUM(comm) || '만원' "커미션 총액" FROM emp;
```

```
SELECT COUNT(comm) "커미션을 받는 직원 수",  
    ROUND(AVG(comm), 2) "평균 커미션" FROM emp;
```

```
SELECT COUNT(*) "사원 수" FROM emp;
```

▶ 급여 총액과 평균 급여 조회하기

```
SELECT SUM(sal) "급여 총액", AVG(sal) "평균 급여" FROM emp;
```

5.2 GROUP BY 절

앞에서 그룹함수를 사용해 급여 합계를 구하였고 최고 급여나 최저 급여 또는 직원의 평균 나이를 구하였다. 이는 테이블 전체를 대상으로 합계나 평균 등의 집계를 한 것이다. 하지만 경우에 따라서 테이블의 데이터를 특정 그룹으로 나누어 통계 데이터를 조회하는 경우가 많다. 예를 들자면 부서별 급여 합계, 직급별 급여 평균, 연령대별 급여 평균 등을 구해야 할 때도 있다. 이렇게 그룹을 짓는 것을 그룹핑(Grouping)이라고도 한다. 이렇게 특정 범위를 기준으로 그룹핑을 할 때 사용하는 것이 바로 GROUP BY 절이다. 다시 말해 GROUP BY는 어떤 컬럼을 기준으로 그룹함수를 적용할지를 지정하는 것이다.

▶ 부서별 급여 합계와 급여 평균 조회

SELECT 문장에서 GROUP BY 절을 사용할 경우 SELECT 절에 기술한 항목은 그룹함수와 상수를 제외하고 모든 항목이 GROUP BY절에 명시되어야 한다. 그렇지 않으면 오류가 발생한다.

아래는 DECODE 함수를 이용해 문자열을 반환하므로 해당하는 결과를 상수로 인식한다.

```
SELECT deptno 부서번호,  
       DECODE(deptno, 10, '경리부', 20, '인사부', 30, '영업부', 40, '전산부') 부서명,  
       SUM(sal) "급여 합계", ROUND(AVG(sal)) "급여 평균", COUNT(sal) "사원수"  
FROM emp  
GROUP BY deptno  
ORDER BY deptno;
```

▶ 영업부의 직급별 직원의 급여 합계와 급여 평균을 조회

아래 SQL 쿼리는 영업부에 속한 사원만을 조회하기 위해 WHERE 절을 사용해 deptno가 30인 사원만 조회하고 ORDER BY 절을 사용해 직급으로 정렬되도록 했다. 또한 영업부 직원만 조회하므로 부서명이 출력되도록 SELECT 절에 상수를 사용했다. 이렇게 그룹핑 할 때도 조건을 주기위해 WHERE 절을 사용할 수 있지만 한 가지 주의해야 할 것은 GROUP BY 절은 항상 WHERE 절 다음에 와야 하며 ORDER BY 절은 항상 GROUP BY 절 다음에 와야 한다는 것이다. 또한 ORDER BY 절에 기술하는 컬럼의 순서는 상관없으나 반드시 GROUP BY 절에 기술된 컬럼만 사용할 수 있다. WHERE 절에 조건으로 사용된 컬럼은 GROUP BY 절의 컬럼과는 상관없다.

```
SELECT '영업부' dept, job 직급,  
       TO_CHAR(SUM(sal), '999,999') "직급별 급여합계",  
       TO_CHAR(ROUND(AVG(sal)), '999,999') "직급별 급여평균"
```



```
FROM emp
WHERE deptno=30
GROUP BY deptno, job
ORDER BY job;
```

5.3 HAVING 절

SELECT 절과 GROUP BY 절을 함께 사용하는 SQL 문장에서도 일반적인 조건을 제시해 조회 범위를 제한하기 위해서 WHERE 절을 사용했다. 위에서 조회 범위를 영업부 사원으로 제한하기 위해서 WHERE 절에 deptno가 30인 사원에 대한 조건을 제시했다. 만약 그룹함수를 사용해 조건을 제시해야 한다면 WHERE 절에서 그룹함수를 사용할 수 있을까? 결론부터 말하면 WHERE 절에는 그룹함수를 사용해 조건을 제시할 수 없다. 이럴 경우에 사용할 수 있는 것이 바로 HAVING 절이다. 일반적인 조건은 WHERE 절에 제시하여 조회 범위를 제한하고 집계함수를 이용해 그룹에 대한 조건을 제시할 필요가 있는 경우 HAVING 절을 사용해 조건을 제시해야 한다. 다시 말해 HAVING 절은 집계함수의 조건절인 셈이다. 또한 HAVING 절은 GROUP BY 절과 함께 사용해야 한다. GROUP BY 절과 함께 사용하지 않아도 특별한 오류는 발생하지 않지만 아무런 의미 없는 쿼리가 된다. HAVING 절의 조건에는 그룹함수와 상수 그리고 GROUP BY 절에 사용된 컬럼을 지정할 수 있다.

▶ 남자 직원이 3명 미만인 부서 조회

```
SELECT deptno 부서번호, COUNT(gender) || '명' "남자 직원 수" FROM emp
WHERE gender = 'M'
GROUP BY deptno
HAVING COUNT(gender) < 3;
```

▶ 사장을 제외한 직급별 급여 합계가 1500이 넘는 직급 조회(직급 순으로 정렬)

```
SELECT job 직책, COUNT(*) 인원, SUM(sal) FROM emp
WHERE job <> '사장'
GROUP BY job
HAVING SUM(sal) >= 1500
ORDER BY CASE job WHEN '사원' THEN 1
              WHEN '대리' THEN 2
              WHEN '과장' THEN 3
              WHEN '차장' THEN 4
              WHEN '부장' THEN 5
              WHEN '사장' THEN 6
              ELSE 0
END;
```

6. 조인(Join)

관계형 데이터베이스에서 테이블을 설계할 때 중복된 데이터를 최소화 하고 데이터 무결성을 보장하기 위해서 정규화 과정을 거치게 된다. 이 정규화 과정을 거치면서 정규화 이론에 입각해 하나의 테이블에서 중복된 데이터를 분리해 중복을 최소화 하고 꼭 필요한 데이터만 저장하는 것이 일반적이다. 이렇게 중복데이터를 줄이고 데이터 무결성을 보장하기 위해 분리된 테이블 각각에 저장된 정보는 사용자의 다양한 요구사항을 만족시키는데 부족함이 많다. 다시 말해 각각의 개별 테이블을 놓고 보면 사용자가 필요로 하는 정보를 제공하는데 있어 그 만큼 정보로서의 가치가 많이 떨어진다고 볼 수 있다.

지금까지는 하나의 테이블에 대해 검색, 추가, 삭제 등에 대한 SQL 명령을 사용하였지만 실무에서는 하나 이상의 테이블을 연결해 데이터를 검색해야 하는 경우가 매우 많다. 이렇게 하나 이상의 테이블을 연결해 데이터를 조회하기 위해 사용하는 것이 바로 조인(Join)이다.

6.1 내부조인

조인은 크게 내부조인(Inner Join)과 외부조인(Outer Join)으로 나눌 수 있는데 내부조인은 내부적으로 어떤 특별한 처리를 해서 붙여진 이름이 아니라 외부조인과의 구분을 위해서 내부조인이라 부른다. 일반적으로 SQL에서 조인이라 함은 바로 이 내부조인을 의미하는 것으로 두 테이블을 조인하여 데이터를 조회할 때 양쪽 테이블 모두에 데이터가 있어야 조회되는 조인을 말한다.

아래 SQL 쿼리는 emp 테이블과 dept 테이블을 조인한 것으로 emp 테이블의 사원 정보 일부와 dept 테이블의 부서명 그리고 부서의 위치를 조회하는 쿼리이다.

WHERE 절은 테이블에서 데이터를 조회하는 조건을 제시하기도 하지만 아래와 같이 두 테이블을 연결하는 조인 조건을 제시할 때도 사용된다.

```
SELECT e.empno, e.ename, e.job, e.hiredate, e.deptno, e.sal, d.dname, d.loc
FROM emp e, dept d
WHERE e.deptno=d.deptno;
```

두 테이블은 공통 컬럼인 부서 번호(deptno)로 관계를 맺고 있으므로 WHERE 절에 두 테이블 모두 부서 번호(deptno)가 같은 값을 갖는 행을 연결하는 조인 조건을 제시했다.

내부조인은 위에서와 같이 WHERE 절의 조인 조건에 제시한 컬럼 값이 두 테이블 모두에 존재해야 결과로 조회되는 조인을 말한다.

▶ Cartesian Product

안시조인(Ansi Join)에서는 크로스조인(Cross Join, 상호조인)이라고도 하며 WHERE 절에 조인 조건을 제시하지 않기 때문에 조인하는 두 테이블에서 한쪽 테이블의 모든 행과 다른 쪽 테이블의 모든 행을 조인하게 되므로 조인하는 두 테이블의 행을 곱한 결과가 조회된다.

아래와 같이 FROM 절에 조인 대상이 되는 테이블을 나열하고 조인 조건을 제시하지 않았을 경우 두 테이블의 모든 행을 조합한 결과가 조회 된다. 즉 emp 테이블 모든 행(25) x dept 테이블의 모든 행(4) = 100개의 행이 결과로 조회된다. 이렇게 두 테이블의 행을 곱한 결과를 조회하는 쿼리를 카타시안 곱(Cartesian Product)이라고 한다.

실제 현장에서는 카타시안 곱의 결과만을 사용하는 경우는 거의 없고 카타시안 곱과 그룹 연산 등을 활용해 효율적인 쿼리를 작성하는데 사용된다.

```
SELECT e.empno, e.ename, e.job, e.deptno, e.sal, d.dname, d.loc
FROM emp e, dept d;
```

▶ 동등조인(Equi Join)

동등조인(Equi Join, 또는 등가조인)은 WHERE 절에 조인 조건을 제시할 때 동등연산자(=)를 사용하는 조인을 말하는데 가장 일반적으로 많이 사용되는 조인으로 대부분의 조인이 여기에 속한다. 다시 말해 동등조인은 조인에 참여하는 두 테이블에서 공통 컬럼의 값이 같은 행을 연결해 결과를 조회하는 조인을 말한다.

■ 두 개의 테이블을 조인하는 경우

다음의 SQL 쿼리는 사원 정보와 그 사원이 속한 부서명을 조회하기 위한 쿼리이다.

```
SELECT e.employee_id, e.first_name, e.salary, e.hire_date,
       d.department_id, d.department_name
FROM employees e, departments d
WHERE e.department_id = d.department_id;
```

■ 세 개의 테이블을 조인하는 경우

다음의 SQL 쿼리는 사원정보와 그 사원이 속한 부서명 그리고 직급을 조회하기 위한 쿼리이다.

아래와 같이 3개의 테이블을 조인하는 경우 employees 테이블이 departments 테이블과 jobs 테이블을 참조하고 있으므로 기준 테이블이 된다. 일반적으로 기준 테이블과 조인에 참여한 하나의 테이블을 조인한 결과를 가지고 나머지 하나의 테이블과 조인해 결과를 조회하게 된다.

아래의 경우 employees 테이블과 jobs 테이블이 공통 컬럼인 job_id 컬럼을 통해 조인한 결과를 가지고 다시 departments 테이블의 department_id 컬럼의 값과 일치하는 데이터를 찾아 최종 결과를 조회하게 된다.

```
SELECT e.employee_id, e.first_name, e.salary,
       d.department_name, j.job_title
FROM employees e,
     departments d,
     jobs j
WHERE e.department_id = d.department_id
     AND e.job_id = j.job_id;
```

■ 네 개의 테이블을 조인하는 경우

다음 SQL 쿼리는 “위의 세 개의 테이블을 조인하는 경우”에서 사원이 소속된 부서가 어디에 있는지 까지 조회하는 쿼리이다. 이를 위해서 부서의 위치 정보를 저장하고 있는 locations 테이블을 조

인에 참여시켜 네 개의 테이블을 조인하고 있다.

```
SELECT e.employee_id, e.first_name, e.salary,
       d.department_name, j.job_title, l.city
FROM employees e,
     departments d,
     jobs j,
     locations l
WHERE e.department_id = d.department_id
     AND e.job_id = j.job_id
     AND d.location_id = l.location_id;
```

■ 조인조건과 일반조건을 포함하는 경우

다음 SQL 쿼리는 직급이 과장인 사원의 사번, 이름, 직급, 급여, 소속 부서명을 출력하는 쿼리이다. 아래 쿼리와 같이 조인 조건과 조회 조건을 AND 연산자를 사용해 조건을 제시할 수 있다. 이럴 경우 연산 우선순위는 일반 조건으로 지정한 job이 과장인 사원을 먼저 검색한 후 그 결과를 가지고 조인 조건에 따라 조인을 시도하게 된다.

```
SELECT e.empno, e.ename, e.job, e.sal, d.dname FROM emp e, dept d
WHERE e.deptno = d.deptno
     AND e.job = '과장';
```

아래 SQL 쿼리는 employees, departments, jobs, locations 네 개의 테이블을 조인하고 사원의 FirstName, 부서명, 직급 그리고 워싱턴 주에 속한 도시에서 근무하는 사원의 정보를 출력한 것이다.

```
SELECT e.first_name, d.department_name, j.job_title, l.city
FROM employees e, departments d, jobs j, locations l
WHERE e.department_id = d.department_id
     AND e.job_id = j.job_id
     AND d.location_id = l.location_id
     AND l.state_province = 'Washington';
```

▶ 비동등조인(Non-Equi Join)

비동등조인(Non-Equi Join, 또는 비등가조인)은 WHERE 절에서 조인 조건을 제시할 때 동등연산자(=)가 아닌 그 이외의 연산자를 사용해 조인조건을 제시한 조인을 말한다. 동등조인(Equi Join)에서 공통 컬럼을 통해 조인이 이루어졌지만 비동등조인은 공통 컬럼이 아닌 다른 조인 조건을 제시해 결과를 조회한다. 예를 들면 아래 SQL 쿼리와 같이 특정 범위에 있는지를 조사하기 위해 대소 비교 연산자(>, >=, <, <=)와 AND 연산자를 연결한 조인 조건이나 BETWEEN 연산자를 사용한 조인 조건을 제시할 수도 있다.

아래는 고객의 마일리지 정보가 저장된 guest 테이블과 마일리지 포인트에 대한 사은품 정보가 저

장된 gift 테이블을 조인하고 고객의 마일리지 포인트 별로 받을 수 있는 사은품을 조회하는 쿼리이다.

```
SELECT gs.gname 고객명, gs.point 마일리지, gf.gname 사은품
FROM guest gs, gift gf
WHERE gs.point >= gf.g_start AND gs.point <= gf.g_end;
```

```
SELECT gs.gname 고객명, gs.point 마일리지, gf.gname 사은품
FROM guest gs, gift gf
WHERE gs.point BETWEEN gf.g_start AND gf.g_end;
```

▶ 자체조인(Self Join)

지금까지의 조인은 필요한 데이터가 두 개 이상의 테이블에 저장되어 있어 여러 테이블을 조인하여 데이터를 검색하는 것이었다. 하지만 필요한 데이터가 모두 하나의 테이블에 저장되어 있고 일반적인 SELECT 문으로 조회할 수 없을 때는 어떻게 해야 할까? 이런 경우에 사용할 수 있는 조인이 바로 자체조인(Self Join, 이하 셀프조인)으로 자기 자신과 조인을 맺는 것을 의미 한다.

셀프조인은 실제 조인에 참여하는 테이블은 하나지만 FROM 절에 같은 테이블을 두 번 지정하고 WHERE 절에서 조인 조건을 제시해 서로 다른 테이블과 조인하는 것처럼 행을 연결하는 것이다.

아래 SQL 쿼리는 emp 테이블에서 각 사원의 관리자가 누구인지를 알아보기 위해 emp 테이블 자신과 조인하여 각각의 사원에 대한 관리자를 조회하는 쿼리이다. emp 테이블은 사원 정보와 그 사원의 관리자에 대한 정보가 하나의 테이블에 저장되어 있기 때문에 일반적인 SELECT 문으로는 조회할 수 없어서 셀프조인을 사용하였다.

```
SELECT e.empno 사번, e.ename 이름, e.mgr "매니저 사번", m.ename "매니저 이름"
FROM emp e, emp m
WHERE e.mgr = m.empno;
```

6.2 외부조인

내부조인(Inner Join)에서는 조인 조건에 제시한 컬럼 값이 두 테이블 모두에 존재하는 행들만 결과로 조회 되었다. 하지만 외부조인(Outer Join)은 두 테이블을 조인할 때 어느 한쪽 테이블에만 데이터가 존재해도 조회될 수 있도록 하는 조인을 말한다.

아래 SQL 쿼리는 셀프조인에서 살펴봤던 emp 테이블에서 각 사원의 관리자가 누구인지를 조회하는 쿼리이다. 하지만 이 쿼리를 실행해 보면 emp 테이블에 저장된 사원의 수는 25명인데 반해 실제 조회되는 사원은 24명밖에 되지 않는다. 그 이유는 직급이 사장인 사원은 관리자가 없어서 mgr 컬럼의 데이터가 null이 되므로 조회되지 않는다.

```
SELECT e.empno 사번, e.ename 이름, e.mgr "매니저 사번", m.ename "매니저 이름"
FROM emp e, emp m
WHERE e.mgr = m.empno;
```

위의 쿼리를 아래와 같이 수정하고 실행하면 전체 사원 25명에 대한 관리자 정보를 확인할 수 있을 것이다. WHERE 절을 살펴보면 (+) 기호가 조건에 사용되었는데 이 기호가 바로 조인에 참여하는 테이블을 외부조인으로 연결하라는 명령이다.

외부조인은 WHERE 절에 제시한 조인 조건을 조인에 참여하는 두 테이블 중에서 어느 한 테이블에만 데이터가 존재하고 다른 테이블에는 해당 데이터가 존재하지 않더라도 결과에 포함시키는 조인이다. (+) 기호는 WHERE 절의 조인 조건에서 데이터가 존재하지 않는 테이블 쪽에 기술해야 한다. 또한 조인조건이 여러 개일 경우 모든 조인조건에 (+) 기호를 기술해야 한다. 만약 하나라도 (+) 기호를 기술하지 않으면 조인의 결과는 내부조인이 되며 OR 연산자와는 같이 사용할 수 없다. 그리고 동시에 여러 테이블과 외부조인을 맺을 수 없고 오직 하나의 테이블과만 외부조인을 할 수 있다. 아래 SQL 쿼리는 emp 테이블에서 사원과 관리자가 누구인지를 조회하는 쿼리로 관리자가 없는 사원이 누구인지도 출력하기 위해 외부조인으로 연결한 쿼리이다.

```
SELECT e.empno 사번, e.ename 이름, e.mgr "매니저 사번", m.ename "매니저 이름"
FROM emp e, emp m
WHERE e.mgr = m.empno(+);
```

아래의 SQL 쿼리는 employees 테이블에서 사원이 어느 부서에 소속되어 있는지를 조회하는 외부 조인 쿼리이다. 아래와 같이 외부조인을 하지 않으면 어느 부서에도 소속되지 않은 “Kimberely”라는 사원은 조회될 수 없다.

```
SELECT e.employee_id, e.first_name, e.department_id, d.department_name
FROM employees e, departments d
WHERE e.department_id = d.department_id(+);
```

외부조인은 아래 SQL 쿼리와 같이 모든 조인 조건에 (+) 기호를 기술해야 한다. 하나라도 (+) 기호를 기술하지 않으면 내부조인으로 조회된다.

```
SELECT e.employee_id, e.first_name, e.department_id,
       d.department_name, l.city
FROM employees e, departments d, locations l
WHERE e.department_id = d.department_id(+)
       AND d.location_id = l.location_id(+)
ORDER BY e.department_id;
```

6.3 ANSI 조인(Ansi Join)

앞에서도 언급했듯이 SQL은 국제 표준으로 재정되어 있으며 현재 대부분의 RDBMS에서 SQL 표준을 지원하고 있다. 오라클은 Oracle 9i 이전부터 ANSI(American National Standards Institute) SQL 표준을 지원하고 있으며 Oracle 9i부터 SQL/92 문법을 지원하고 있다. SQL/92에서 새로운 조인 문법이 등장했는데 이를 ANSI 조인(Ansi Join) 이라고 부른다.

▶ 크로스 조인(Cross Join)

앞에서 학습했던 카타시안 곱(Cartesian Product)을 안시조인에서는 크로스 조인이라고 부른다.

```
SELECT e.empno, e.ename, e.job, e.deptno, e.sal, d.dname, d.loc
FROM emp e CROSS JOIN dept d;
```

▶ 안시 내부조인(Ansi Inner Join)

아래 SQL 쿼리는 앞에서 내부조인을 알아볼 때 사용했던 쿼리이다.

```
SELECT e.employee_id, e.first_name, e.last_name, e.hire_date,
       d.department_id, d.department_name
FROM employees e, departments d
WHERE e.department_id = d.department_id;
```

다음은 위의 쿼리를 Ansi SQL 표준으로 변경한 쿼리이다.

이 쿼리를 살펴보면 FROM 절에 JOIN 이라는 단어를 사용한다. 이전에는 조인조건을 WHERE 절에 기술했지만 Ansi 표준에서는 더 이상 WHERE 절에 조인조건을 기술하지 않고 일반 조건만 지정할 수 있다. 대신에 ON과 USING을 사용할 수 있다.

```
SELECT e.employee_id, e.first_name, e.last_name, e.hire_date,
       d.department_id, d.department_name
FROM employees e INNER JOIN departments d
ON e.department_id = d.department_id;
```

공통 컬럼의 이름이 같은 경우 USING을 사용하면 아래와 같이 컬럼 명을 한 번만 기술해도 되고 SELECT 절에서도 테이블 별칭을 사용하지 않고 컬럼 명만 기술해야 한다.

```
SELECT e.employee_id, e.first_name, e.last_name, e.hire_date,
       department_id, d.department_name
FROM employees e INNER JOIN departments d
USING(department_id);
```

▶ 안시 외부조인(Ansi Outer Join)

기존 SQL에서 외부조인을 할 때는 아래와 같이 조인에 참여하는 테이블 중에서 데이터가 없는 테이블에 (+) 기호를 기술했다.

```
SELECT e.ename 이름, e.job 직급, em.ename 관리자 FROM emp e, emp em
WHERE e.mgr = em.empno(+);
```

하지만 ANSI 외부조인은 아래와 같이 LEFT, RIGHT, FULL을 사용해 보다 명확하게 외부조인을 지정할 수 있다. LEFT, RIGHT는 데이터가 존재하는 테이블이 어느 테이블인지를 지정 한다.

```
SELECT e.ename 이름, e.job 직급, em.ename 관리자
FROM emp e LEFT OUTER JOIN emp em
ON e.mgr = em.empno;
```

외부조인으로 연결해야할 테이블이 하나가 아니라 여러 개일 경우 FROM 절에 어떤 조인을 할 것인지 기술하고 ON을 사용해 각각 조인조건을 제시해야 한다.

```
SELECT e.employee_id, e.first_name, e.department_id,
       d.department_name, l.city
FROM employees e LEFT OUTER JOIN departments d
    ON e.department_id = d.department_id
    LEFT OUTER JOIN locations l
    ON d.location_id = l.location_id
ORDER BY e.department_id;
```

ANSI 외부조인은 기존 SQL로는 할 수 없었던 아래와 같은 새로운 형식의 조인(FULL OUTER JOIN)을 지원한다.

```
SELECT e.employee_id, e.first_name, d.department_name
FROM employees e, departments d
WHERE e.employee_id(+) = d.manager_id(+); <- (+)는 한 쪽 테이블에만 지정할 수 있다.
```

```
SELECT e.employee_id, e.first_name, d.department_name
FROM employees e FULL OUTER JOIN departments d
ON e.employee_id = d.manager_id;
```


7. 서브 쿼리(Sub Query)

서브 쿼리란 하나의 SQL 쿼리 내부에 또 다른 SELECT 문이 사용된 것을 말하는데 외부의 SQL 쿼리를 메인 쿼리(Main Query)라 하고 그 메인 쿼리 내부에 기술된 SELECT 문을 서브 쿼리라 한다. 다시 말해 SQL 쿼리 안에 들어있는 SELECT 쿼리를 서브 쿼리라고 하며 서브 쿼리는 SELECT, INSERT, UPDATE, DELETE 문에서 사용할 수 있다.

SELECT 문에서는 SELECT 절의 컬럼 항목, FROM 절, WHERE 절, HAVING 절에 서브 쿼리를 사용할 수 있다.

SELECT 절의 컬럼을 나열하는 항목으로 사용되는 서브 쿼리를 스칼라 서브 쿼리(Scalar Sub Query), FROM 절에 사용되는 서브 쿼리를 인라인 뷰(Inline View)라고 하며 WHERE 절에 사용된 서브 쿼리를 중첩 쿼리(Nested Query)라고도 한다.

서브 쿼리는 메인 쿼리가 실행되기 전에 테이블로부터 필요한 데이터를 읽어와 메인 쿼리에 전달하기 위해 사용된다.

자 그럼 먼저 서브 쿼리를 어떤 경우에 사용하게 되는지 부터 알아보도록 하자.

만약 emp 테이블에 있는 사원 중에서 사원의 평균 급여보다 급여를 많이 받는 사원의 사번, 이름, 급여 정보를 조회하려면 어떻게 해야 할까? 간단히 생각해 보면 먼저 emp 테이블에서 전체 사원의 평균 급여를 구하는 아래의 SQL 쿼리로 평균 급여가 얼마인지를 확인한 다음 다시 emp 테이블에서 평균 급여보다 많이 받는 사원의 리스트를 조회하면 될 것이다.

```
SELECT AVG(sal) FROM emp; <- emp 테이블에서 급여 평균을 조회한다.
```

```
SELECT empno, ename, sal FROM emp
```

```
WHERE sal > AVG(sal); <- WHERE 절에는 집계함수를 사용할 수 없어 오류 발생
```

```
SELECT empno, ename, sal FROM emp
```

```
WHERE sal > 426.8; <- emp 테이블에서 급여 평균을 조회한 값을 가지고
```

다시 평균 급여보다 급여가 많은 사원의 리스트를 조회 한다.

하지만 위와 같이 평균 급여보다 많은 급여를 받는 사원을 조회하기 위해 두 번의 SQL 쿼리를 데이터베이스에 요청해야 한다. 이 방법은 SQL 쿼리를 두 번 실행해야 하는 번거로움이 따르고 데이터베이스는 두 번의 요청을 처리하기 위한 동작을 해야 하므로 성능저하가 발생할 수 있다.

아래 SQL 쿼리의 WHERE 절과 같이 서브 쿼리를 사용하면 하나의 SQL 쿼리 문을 이용해 한 번의 요청으로 원하는 데이터를 간단히 조회할 수 있다. 서브 쿼리가 이렇게 유용하고 편리하다 보니 실무에서도 아주 많이 사용되는 SQL 기법이다.

```
SELECT empno, ename, sal FROM emp
```

```
WHERE sal > (SELECT AVG(sal) FROM emp); <- 서브 쿼리를 이용하면 간단히 해결
```

7.1 단일 행 서브 쿼리

서브 쿼리의 실행 결과가 오직 한 행인 서브 쿼리를 단일 행(Single Row) 서브 쿼리라 한다.

이 서브 쿼리는 단일행이면서 단일 컬럼인 결과를 반환하는 서브 쿼리를 말하는데 대부분 서브 쿼리에 집계함수가 포함된 쿼리가 많다.

아래 SQL 쿼리는 회사에서 제일 적은 급여를 받는 사원을 조회하는 쿼리이다.

```
SELECT ename 이름, job 직급, sal 급여
FROM emp
WHERE sal <= (SELECT MIN(sal) FROM emp);
```

아래 SQL 쿼리는 회사에서 제일 많은 급여를 받는 사원을 조회하는 쿼리이다.

emp 테이블과 dept 테이블을 조인해 부서명을 같이 출력 하였다.

```
SELECT e.ename, e.sal, d.dname
FROM emp e, dept d
WHERE e.deptno = d.deptno
      AND e.sal = (SELECT MAX(sal) FROM emp);
```

7.2 다중 행 서브 쿼리

서브 쿼리의 실행 결과가 다중 행이면서 단일 컬럼을 반환하는 서브 쿼리를 말한다.

단일 행 서브 쿼리에서 주로 집계함수를 사용했다면 다중 행 단일 컬럼을 반환하는 서브 쿼리는 일반적인 쿼리의 형태를 띠고 있다.

WHERE 절에 IN 연산자를 사용하여 서브 쿼리에서 반환되는 다중 행 단일 컬럼의 값들 중에서 하나와 일치하면 조회되도록 했다. 서브 쿼리의 결과로 다중 행이 반환되는 경우 IN 연산자가 많이 사용된다. IN 연산자는 메인 쿼리의 비교 조건이 서브 쿼리의 결과 중에서 하나라도 일치하면 참이 된다. 즉 여러 개의 비교 데이터를 OR 연산자를 사용해 조건을 제시한 것과 같다.

아래 SQL 쿼리는 급여를 530만원 이상 받는 사원과 같은 부서에 근무하는 사원의 정보를 조회하는 쿼리이다.

```
SELECT ename, hiredate, sal, deptno
FROM emp
WHERE deptno IN(SELECT deptno FROM emp WHERE sal >= 530);
```

아래 SQL 쿼리는 서울에 위치한 부서 정보를 출력하는 쿼리이다.

```
SELECT * FROM dept WHERE loc = '서울';
```

```
SELECT * FROM dept
WHERE loc IN(SELECT loc FROM dept WHERE loc='서울');
```

아래 SQL 쿼리와 같이 WHERE 절에서 ANY 연산자와 동등연산자(=)를 사용해 위와 동일한 데이터를 조회할 수 있다. ANY 연산자는 메인 쿼리의 비교조건이 서브 쿼리의 검색 결과와 하나 이상 일치하면 참이 된다. 즉 어느 것 하나라도 일치하면 참이 된다.

ANY 연산자가 IN 연산자와 다른 점이 있다면 ANY 연산자는 동등 연산자(=)를 포함해서 >, >=, <, <=, <>, != 등의 비교 연산자를 사용할 수 있다는 점이다.

```
SELECT * FROM dept
WHERE loc = ANY(SELECT loc FROM dept WHERE loc = '서울');
```

아래 SQL 쿼리는 서울이 아닌 도시에 위치한 부서 정보를 출력하는 쿼리이다.

```
SELECT * FROM dept
WHERE loc NOT IN (SELECT loc FROM dept WHERE loc = '서울');
```

```
SELECT * FROM dept
WHERE loc <> ANY (SELECT loc FROM dept WHERE loc = '서울');
```

아래 SQL 쿼리는 emp 테이블에서 부서번호가 20인 사원들 중 아무나(Any one) 한 명의 급여보다 많이 받는 사원을 조회하는 쿼리이다. 즉 부서번호가 20인 사원들 중에서 급여가 가장 적은 사원보다 급여를 많이 받는 사원들을 조회한 결과가 출력된다.

```
SELECT ename, sal FROM emp
WHERE sal > ANY(SELECT sal FROM emp WHERE deptno=20);
```

위의 쿼리는 바로 아래의 쿼리에서 서브 쿼리에 집계함수를 사용한 것과 동일하게 조회된다.

```
SELECT ename, sal FROM emp
WHERE sal > (SELECT MIN(sal) FROM emp WHERE deptno=20);
```

아래 SQL 쿼리는 부서번호가 40인 사원들의 급여보다 많은 사원을 모두 조회하는 쿼리이다.

아래 쿼리에서 사용한 ALL 연산자는 메인 쿼리의 비교 조건과 서브 쿼리의 검색 결과를 비교해서

모두 만족해야 참이 된다. 그러므로 아래 쿼리의 결과는 부서번호가 40인 직원들의 급여인 350 ~ 550 보다 많이 받는 직원이 조회된다. 이는 급여를 350보다 많고 550보다 많이 받는 직원을 조회하는 것과 같으며 결론적으로 부서번호가 40인 직원 중에서 최고 급여를 받는 직원보다 급여가 더 많이 받는 직원들이 조회된다.

```
SELECT ename, sal FROM emp
WHERE sal > ALL(SELECT sal FROM emp WHERE deptno=40);
```

위의 쿼리는 아래에서 서브 쿼리에 집계함수를 사용한 것과 동일하게 조회한다.

```
SELECT ename, sal FROM emp
WHERE sal > (SELECT MAX(sal) FROM emp WHERE deptno=40);
```

아래 SQL 쿼리는 부서번호가 10, 40인 직원의 정보를 출력하는 쿼리이다. 하나는 IN 연산자를 또 다른 하나는 동등 연산자(=)와 ANY 연산자를 사용해 동일한 결과가 조회되도록 했다.

```
SELECT e.ename, e.sal, d.dname FROM emp e, dept d
WHERE e.deptno = d.deptno
      AND e.deptno IN(10, 40);
```

```
SELECT e.ename, e.sal, d.dname FROM emp e, dept d
WHERE e.deptno = d.deptno
      AND e.deptno = ANY(10, 40);
```

아래 SQL 쿼리는 부서번호가 10, 30, 40이 아닌 직원을 조회하는 쿼리로 위에서 사용한 IN 연산자와 ANY 연산자에 부정 연산자를 조합하여 NOT IN과 <> ANY 연산자를 사용했다. 아래의 두 SQL 쿼리의 결과는 위의 두 SQL 쿼리의 결과처럼 동일하게 조회될 수 있을까? 결론부터 말하자면 두 쿼리의 결과는 완전히 다르게 조회된다. 왜냐하면 NOT IN 연산자를 사용한 아래 쿼리는 부서번호가 10, 30, 40이 아닌 직원 즉 부서번호가 20인 직원들만 조회되지만 그 아래 있는 ANY 연산자와 같지 않다는 의미의 비교 연산자(<>)를 사용한 쿼리의 결과는 서브 쿼리의 결과 중 어느 것 하나의 값과 같지 않다면 참이 되므로 그 행은 메인 쿼리의 결과에 포함된다. 다시 말해 부서번호가 각각 10 또는 30 또는 40이 아닌 데이터를 모두 조회하게 되는데 부서번호가 10인 것은 30과 40이 아니므로 결과에 포함되고 부서번호가 20인 것은 10, 30, 40이 아니므로 포함되고 부서번호가 30인 것은 10과 40이 아니므로 결과에 포함된다. 그리고 부서번호가 40인 것은 10과 30이 아니므로 결과에 포함된다. 그러므로 이 쿼리의 실행 결과는 테이블에서 deptno 값이 null이 아닌 모든 데이터가 결과로 조회된다.

```
SELECT e.ename, e.sal, d.dname, d.deptno FROM emp e, dept d
WHERE e.deptno = d.deptno
      AND e.deptno NOT IN(10, 30, 40);
```

```
SELECT e.ename, e.sal, d.dname, d.deptno FROM emp e, dept d
WHERE e.deptno = d.deptno
      AND e.deptno <> ANY(10, 30, 40);
```

7.3 다중 컬럼을 반환하는 서브 쿼리

서브 쿼리가 SELECT 문장이므로 하나 이상인 컬럼을 반환할 수도 있다. 서브 쿼리의 결과로 단일 행 다중 컬럼을 반환할 수도 있고 다중 행 다중 컬럼을 반환할 수도 있다.

다중 컬럼을 반환하는 서브 쿼리를 이용해 다른 테이블에 데이터를 추가 또는 수정할 수도 있고 여러 컬럼으로 구성된 복합키를 한 번에 비교할 수도 있다.

아래 SQL 쿼리는 professor 테이블에서 컴퓨터공학(101)과 교수 중에 급여가 제일 많은 교수의 정보를 출력하는 쿼리이다. 아래 쿼리에서 IN 연산자에 사용한 서브 쿼리는 2개의 컬럼으로 이루어진 1개의 행이 결과로 반환된다. 즉 서브 쿼리의 결과가 단일 행 다중 컬럼으로 조회되는 것이다.

```
SELECT p.name 교수명, p.pay 급여, p.hiredate 입사일, d.dname
FROM professor p, department d
WHERE p.deptno = d.deptno
      AND (p.deptno, p.pay) IN(SELECT deptno, MAX(pay) FROM professor
                              WHERE deptno = 101 GROUP BY deptno);
```

아래 SQL 쿼리는 emp 테이블에서 부서별로 최고 급여를 받는 사원의 정보를 조회하는 쿼리이다. 아래 쿼리에서 IN 연산자에 사용한 서브 쿼리는 2개의 컬럼으로 이루어진 4개 행이 결과로 반환된다. 다시 말해 아래에 사용된 서브 쿼리의 결과가 바로 다중 행 다중 컬럼으로 조회되는 것이다.

```
SELECT e.empno 사번, e.ename 이름, e.job 직급, sal 급여, d.dname 부서명
FROM emp e, dept d
WHERE e.deptno = d.deptno
      AND (e.deptno, e.sal) IN(SELECT deptno, MAX(sal) FROM emp GROUP BY deptno);
```

7.4 연관성 있는 서브 쿼리

지금까지 앞에서 알아본 서브 쿼리는 메인 쿼리와 관련 없이 서로 독립적으로 수행되는 서로 연관성이 없는 서브 쿼리였다. 이렇게 연관성이 없는 서브 쿼리는 메인 쿼리와는 별도로 서브 쿼리가 먼저 수행되어 그 결과를 메인 쿼리로 반환하는 경우가 대부분이다. 이번에는 메인 쿼리의 데이터를

서브 쿼리에서 사용하여 서브 쿼리를 수행한 결과가 다시 메인 쿼리로 반환되는 연관성 있는 서브 쿼리에 대해 알아보도록 하자.

연관성 있는 쿼리란 무엇일까? 어떤 데이터가 연관성이 있다는 것은 두 데이터가 관계를 맺고 있다는 것과 같은 의미로 메인 쿼리와 서브 쿼리가 조인되는 것을 말한다.

다시 말해 연관성 있는 쿼리는 메인 쿼리가 먼저 수행되어 그 결과 데이터를 서브 쿼리 안에서 참조하는 경우를 말하는데 이때 메인 쿼리의 데이터를 서브 쿼리에서 참조한다는 것은 메인 쿼리와 서브 쿼리 사이에 조인이 이루어져 메인 쿼리의 데이터를 서브 쿼리 안에서 사용하는 경우를 말하는 것이다.

아래 SQL 쿼리는 자신이 속한 부서의 평균 급여보다 많은 급여를 받는 사원의 정보를 조회하는 쿼리이다. 이 SQL 쿼리를 보면 메인 쿼리와 서브 쿼리 사이에 조인이 이루어진 것을 볼 수 있을 것이다. 이렇게 연관성 있는 서브 쿼리는 메인 쿼리로 부터 데이터(현재 행의 부서번호)를 넘겨받아 서브 쿼리가 실행되고 그 결과를 다시 메인 쿼리로 반환해 WHERE 절의 조건에 적용한 결과를 최종적으로 조회하게 된다.

```
SELECT e.ename 이름, e.sal 급여, d.dname
FROM emp e, dept d
WHERE e.deptno = d.deptno
      AND e.sal >= (SELECT AVG(es.sal) FROM emp es WHERE e.deptno = es.deptno)
ORDER BY e.deptno;
```

아래의 3가지 SQL 쿼리는 모두 동일한 결과를 조회하는 쿼리 이다. 아래에서 두 번째 세 번째 쿼리는 연관성 있는 서브 쿼리를 사용했지만 각각 IN 연산자와 EXISTS 연산자를 사용하였다. IN 연산자는 () 안에 여러 데이터 중에서 하나와 일치하면 참을 반환하고 EXISTS 연산자는 서브 쿼리의 결과가 존재하기만 하면 참을 반환한다. 참고로 EXISTS 연산자는 서브 쿼리에서만 사용할 수 있는 연산자 이다.

```
SELECT e.ename, e.sal, e.deptno FROM emp e, dept d
WHERE e.deptno = d.deptno
      AND e.deptno IN(10, 40);
```

```
SELECT e.ename, e.sal, e.deptno FROM emp e
WHERE e.deptno IN(SELECT d.deptno FROM dept d
                  WHERE d.deptno IN(10, 40)
                  AND d.deptno = e.deptno);
```

```
SELECT e.ename, e.sal, e.deptno FROM emp e
WHERE EXISTS (SELECT 0 FROM dept d
              WHERE d.deptno IN(10, 40)
              AND d.deptno = e.deptno);
```

7.5 스칼라 서브 쿼리(Scalar Sub Query)

SELECT 절의 리스트 항목으로 사용되는 서브 쿼리를 스칼라 서브 쿼리라 하며 한 번에 한 행의 결과를 반환한다.

아래는 SQL 쿼리는 emp 테이블에서 사원의 이름과 입사일 그리고 부서명을 출력하는 쿼리이다. 부서명을 조회하기 위해 SELECT 절의 리스트 항목으로 서브 쿼리를 사용하고 이 서브 쿼리 안에서 emp 테이블과 dept 테이블을 조인하고 있다. 이 경우에도 메인 쿼리에서 서브 쿼리에 필요한 부서 번호를 넘겨주고 서브 쿼리가 수행되어 그 결과를 다시 메인 쿼리로 반환한다.

스칼라 서브 쿼리는 조인이 필요하고 조회하는 데이터가 적을 경우 일반 조인 대신 사용하면 보다 좋은 성능을 발휘한다.

```
SELECT e.ename 이름, e.hiredate 입사일,  
       (SELECT d.dname FROM dept d WHERE e.deptno = d.deptno) 부서명  
FROM emp e;
```

7.6 인라인 뷰(Inline View)

SQL 쿼리의 FROM 절에서 테이블을 대신해 사용되는 서브 쿼리를 인라인뷰(Inline View)라고 한다. 아래 SQL 쿼리는 emp 테이블에서 평균 급여 이하를 받는 사원의 정보를 조회한 것으로 FROM 절에서 emp 테이블의 직원 중에 최저 급여와 평균 급여를 반환하는 서브 쿼리를 사용하였다. 아래와 같이 SELECT 절에서 집계 결과와 여러 컬럼의 정보를 같이 출력해야 하는 경우에 서브 쿼리를 활용하면 까다로운 문제를 보다 수월하게 풀 수 있다.

```
SELECT e.empno 사번, e.ename 이름, e.sal 급여,  
       d.dname 부서명, es.min 최저급여, es.avg 평균급여  
FROM emp e, dept d,  
       (SELECT MIN(sal) min, AVG(sal) avg FROM emp) es  
WHERE e.deptno = d.deptno  
      AND e.sal BETWEEN es.min AND es.avg  
ORDER BY e.sal DESC;
```

▶ 인라인 뷰와 ROWNUM 활용

만약 위의 쿼리에서 평균 급여 이하를 받는 사원 중에서 최저 급여를 받는 사원 5명만 조회해야 한다면 또는 평균 급여 이하를 받는 사원 중에서 급여 순으로 오름차순 하여 3번째부터 7번째까지의 최저 급여를 받는 사원을 조회해야 한다면 어떻게 하면 될까?

물론 ORDER BY 절에서 급여가 제일 작은 사원부터 출력되게 하면 최저 급여를 받는 사원 순으로 조회할 수는 있으나 최하위 급여 5명만 출력되게 하는 것은 위의 SQL 쿼리로 제대로 조회할 수 없다. 이럴 때 유용하게 사용할 수 있는 것이 의사 컬럼(Pseudo Column)인 ROWNUM 이다. 의사

컬럼이란 실제 테이블에는 존재하지 않지만 테이블에 있는 컬럼처럼 조회할 수 있는 컬럼을 말한다. ROWNUM은 쿼리 결과로 조회되는 각각의 행에 대한 순서 값을 가지고 있는 의사 컬럼으로 첫 번째 조회되는 행의 ROWNUM 값은 1 그 다음 조회되는 행의 ROWNUM 값은 2가 된다. ROWNUM은 실제 테이블에는 존재하지 않지만 쿼리 결과로 조회되는 행에 대한 순서 값을 가지고 있어서 조회한 결과 데이터에서 특정 위치에 있는 원하는 개수만큼 데이터를 조회하고 싶을 때 사용한다. 이번에는 emp 테이블이 아니라 게시판 테이블을 새로 만들어 데이터를 추가하고 인라인 뷰와 ROWNUM을 사용해 실무에서 활용할 수 있는 게시판의 페이징 처리기법에 대해서 살펴볼 것이다. 먼저 수업시간에 제공한 JSPBBS.sql 파일을 이용해 HR 스키마에 테이블을 생성하고 데이터를 추가하자. 성공적으로 데이터를 추가하였다면 다음 SQL 쿼리를 이용해 게시판 테이블의 데이터를 조회해 보자.

```
SELECT * FROM jspbbs ORDER BY no DESC;
```

위의 쿼리로 게시판 테이블을 조회하면 제일 마지막에 저장된 데이터가 제일 위에 조회되는데 이는 게시판 특성상 최신의 글을 맨 처음에 보여주기 위한 것으로 게시 글 번호인 no를 기준으로 내림차순 정렬하여 조회했기 때문이다.

게시판 테이블의 데이터를 조회한 결과를 보면 no 순으로 내림차순 정렬되어 게시판에 있는 모든 데이터인 200개가 조회 되었다. 만약 게시판 테이블에 데이터가 1,000개라면 위의 SQL 쿼리를 사용하면 1,000개가 모두 한 페이지에 출력될 것이다. 여기까지 뭐 그럭저럭 사용하는데 큰 문제는 없겠지만 만약 10,000의 데이터라면 한 페이지에 모든 게시 글 리스트를 출력하는 것은 아마도 문제가 되지 않을까하는 생각이 든다.

이를 보완하기 위해서 한 페이지에 10개의 게시 글 리스트가 출력될 수 있도록 수정하고 각 페이지로 이동할 수 있는 링크를 화면에 표시하는 방법을 생각해 볼 수 있다. 이렇게 게시 글을 조회할 수 있게 하려면 최신 게시 글 순으로 조회하여 1페이지는 1 ~ 10까지만 조회되고 2페이지는 11 ~ 20까지만 조회되고 또 3페이지는 21 ~ 30까지만 조회되어야 한다. 그러므로 게시판 테이블에서 게시 글이 조회될 때마다 조회되는 행의 순서 값을 가지고 1페이지, 2페이지 또는 3페이지와 같이 각 페이지에 해당하는 게시 글을 10개 씩 조회할 수 있도록 쿼리를 작성해야 하는데 이때 유용하게 사용할 수 있는 것이 바로 ROWNUM 이며 이 ROWNUM은 앞에서도 설명했듯이 조회되는 행의 순서 값을 가지고 있는 의사컬럼 이다.

자 그럼 ROWNUM을 사용해 한 페이지에 10씩 게시 글이 출력될 수 있도록 조회하는 쿼리를 작성해 보자. 먼저 아래 SQL 쿼리와 같이 작성해서 1페이지에 해당하는 게시 글 리스트와 3페이지에 해당하는 게시 글 리스트가 제대로 출력되는지 확인해 보자.

■ 1페이지에 해당하는 게시 글 리스트를 조회하는 쿼리

```
SELECT ROWNUM, bbs.* FROM jspbbs bbs
WHERE ROWNUM >= 1 AND ROWNUM <= 10
ORDER BY no DESC;
```

■ 3페이지에 해당하는 게시 글 리스트를 조회하는 쿼리

```
SELECT ROWNUM, bbs.* FROM jspbbs bbs
```



```
WHERE ROWNUM >= 21 AND ROWNUM <= 30
ORDER BY no DESC;
```

위의 SQL 쿼리를 실행해 보면 1페이지에 해당하는 게시 글 리스트를 조회하는 쿼리는 제대로 동작하는 것 같은데 3페이지에 해당하는 게시 글 리스트를 조회하는 쿼리는 데이터는커녕 아예 데이터를 자체를 가져오지 못하는 현상이 발생한다. 그 이유는 ROWNUM은 기준점을 1부터 시작해서 조회되는 행의 번호를 상대적으로 매기기 때문에 ROWNUM 1을 기준으로 처음부터 데이터를 조회할 수 있도록 WHERE 절에서 “ROWNUM >= 1 AND ROWNUM <= 10”와 같은 조건을 지정하면 1페이지에 해당하는 데이터는 제대로 조회될 수 있지만 ROWNUM이 1이 아닌 특정 구간부터 데이터를 조회할 수 있는 “ROWNUM >= 21 AND ROWNUM <= 30”과 같은 조건을 WHERE 절에 지정하면 데이터가 하나도 조회되지 않는다. 그러므로 WHERE 절에서 특정 구간의 ROWNUM을 직접 비교하는 방식으로 우리가 원하는 데이터를 제대로 조회할 수가 없다. 그럼 어떻게 해야 3페이지에 해당하는 데이터를 제대로 읽어올 수 있을까?

이 문제는 WHERE 절에서 ROWNUM을 직접 비교하지 않고 아래와 같이 ROWNUM이 포함된 서브 쿼리를 작성해서 ROWNUM이 번호를 매긴 행의 순서를 num으로 별칭을 지정해 메인 쿼리로 보내고 이 num을 메인 쿼리의 WHERE 절에서 사용하게 되면 이 문제를 간단히 해결할 수 있다.

■ 인라인 뷰를 사용해 3페이지에 해당하는 게시 글 리스트를 조회하는 쿼리

```
SELECT * FROM
```

```
(SELECT ROWNUM num, sub.* FROM jsbbbs sub ORDER BY no DESC)
```

```
WHERE num >= 21 AND num <= 30;
```

위의 SQL 쿼리를 실행하면 아래 그림과 같은 조회 결과를 볼 수가 있다. 조회된 결과를 자세히 살펴보면 num을 기준으로 10개의 게시 글 데이터를 잘 읽어오기는 하지만 게시 글 번호인 no를 살펴보면 뭔가 조금은 이상한 것을 볼 수 있다. 게시판 테이블인 jsbbbs 테이블에서 no는 게시 글이 쓰여진 순서를 의미하는데 no가 클수록 최신 게시 글이 된다. 최신 게시 글이 위쪽에 배치되도록 화면에 출력해야 하므로 조회할 때 no를 기준으로 내림차순 정렬하여 데이터를 조회했다. 우리가 jsbbbs 테이블에 게시 글 데이터를 200개 삽입하였으므로 한 페이지 당 10개씩의 게시 글 리스트를 출력하게 되면 3페이지에 해당하는 no의 구간은 게시 글이 삭제되지 않았다면 171 ~ 180이 되는데 조회된 결과인 아래 그림을 살펴보면 no의 구간이 이상하게 조회되었다. 참고로 no 컬럼의 조회된 값들이 오라클이 설치된 시스템에 따라서 오라클의 실행 계획이 다를 수 있어서 아래 그림과는 차이가 있을 수 있지만 그래도 3페이지에 해당하는 171 ~ 180이 조회되지 않는 것이다.

NUM	NO	TITLE	WRITER	CONTENT
30	123	관심을 가져...	관리자	안녕하세요관리자입니다.이번에 변경된 회원 약관에 대해
29	122	공식 사항 ...	회원1	아~ 관리자님~공식사항 잘 읽었습니다.그런데 궁금한 것이
28	121	공식 사항 ...	관리자	안녕하세요이번에 여쭙구 서쭙구 해서리...\r\n\r\n이렇게
27	120	감사합니다.	회원1	새 덕분에 궁금한게 해결 되었나니...나행입니다.슬겡하삼~
26	119	궁금한게 해...	회원8	안녕하세요님 덕분에 궁금한점이 해결 되었습니나.감사합니
25	118	서두 궁금했는데	회원7	안녕하세요서두 궁금한 점이 해결 되었습니나.감사합니나.
24	117	아 쏘 바꿈 ...	관리자	안녕하세요 관리자입니다.번서 멘터기를 지시않고 글 들
23	116	감사합니다.	관리자	안녕하세요남편을 감사합니나.\r\n\r\n\r\n\r\n\r\n
22	115	그러게요	회원3	아~ 쏘 바꿈 하시 알아노 사뵤으로 볼 쏘 알았쇼...^ ^관리
21	114	회원이면 낭...	회원3	ㅋㅋㅋ님들도 나 같은 생각을 가지고 게시곤요뵤 같은 회원

이렇게 조회되는 원인은 ROWNUM이 실행되는 SELECT 절과 ORDER BY 절의 순서 때문에 생기는 문제로 SELECT 절에 사용되는 ROWNUM이 조회되는 행의 번호를 매기고 나서 마지막으로 ORDER BY 절에 지정한 조건으로 정렬되기 때문에 생기는 문제이다.

이를 해결하기 위해서는 no가 큰 글이 최신 글이 되므로 먼저 ORDER BY에서 no를 기준으로 내림차순 정렬하여 조회한 데이터를 기준으로 ROWNUM이 행의 번호를 매길 수 있도록 인라인 뷰를 중첩해서 아래와 같이 SQL 쿼리를 작성하면 된다.

■ 중첩 인라인 뷰를 사용해 3페이지에 해당하는 게시 글 리스트를 조회하는 쿼리

```
SELECT * FROM
  (SELECT ROWNUM num, sub.* FROM
    (SELECT * FROM jsbbbs ORDER BY no DESC) sub)
WHERE num >= 21 AND num <= 30;
```

NUM	NO	TITLE	WRITER	CONTENT
21	180	감사합니다.	회원1	세 덕문에 궁금한게 해결 되었습니다...나행입니다.즐거하삼~
22	179	궁금한게 해...	회원8	안녕하세요님 덕문에 궁금한점이 해결 되었습니다.감사합니다
23	178	서두 궁금했네	회원7	안녕하세요서두 궁금한 점이 해결 되었습니다.감사합니다.
24	177	아 쏘 바꿈 ...	관리사	안녕하세요관리사입니다.번서 엔터키를 지시않고 글 늘
25	176	감사합니다.	관리사	안녕하세요남편글 감사합니다.\r\n\r\n\r\n\r\n그럼
26	175	그리게요	회원3	아~ 쏘 바꿈 하시잖아노 사똥으로 볼 쏘 알았쇼..^^관리
27	174	회원이면 낭...	회원3	ㅋㅋㅋ님들도 나 같은 생각을 가지고 게시군요똥 같은 회원!
28	173	별 말씀을 ...	회원1	안녕하세요관리사님~회원이면 낭연히...관심을 가져야셔.
29	172	낭연하쇼~	회원6	안녕하세요관리사님~회원이니까 관심을 갖는건 낭연하쇼..!
30	171	서두요~	회원5	서두 그게 궁금했네...ㅋㅋㅋ님께서 내신 해주시네요...

위의 SQL 쿼리를 실행한 결과를 살펴보면 위의 그림과 같은데 이 그림에서 no를 확인해 보면 3페이지에 해당하는 no의 구간이 잘 조회된 것을 확인할 수 있을 것이다.

8. 데이터 조작과 트랜잭션

새로운 데이터를 테이블에 추가하는 INSERT문, 기존 데이터를 수정하는 UPDATE문, 데이터를 삭제하는 DELETE문은 테이블에서 데이터를 조작하기 위한 명령으로 데이터 조작어 (DML, Data Manipulation Language)에 속하는 명령이다.

일반적으로 프로그래밍 언어에서 데이터베이스의 데이터를 조작하게 되면 바로 데이터베이스에 적용된다. 하지만 여러 번의 SQL 문장을 실행해야 하나의 완성된 작업이 완료된다면 여러 개의 SQL 문장을 하나의 논리적인 작업으로 묶어서 모두가 제대로 실행되어야 비로소 데이터베이스에 적용하고 만약 하나라도 문제가 발생하면 이전의 상태로 되돌려야 데이터 무결성을 유지 할 수 있다.

이렇게 여러 SQL 문장을 하나의 논리적인 작업 단위로 묶어서 처리하는 것을 트랜잭션이라고 한다. 하나의 트랜잭션 안에 있는 여러 개의 SQL 문장이 완료되면 최종적으로 데이터베이스에 적용하거나 하나라도 문제가 생겨서 현재 트랜잭션의 바로 이전으로 되돌릴 때 사용하는 명령을 트랜잭션 제어어(TCL, Transaction Control Language)라고 한다.

8.1 트랜잭션 제어

트랜잭션은 앞서도 언급했지만 하나의 작업을 완료하기 위해서 데이터베이스에 여러 번의 SQL 쿼리를 발행해야 하는 경우 즉 여러 번의 INSERT, UPDATE, DELETE를 발행해야 하나의 작업이 완성된다면 이 여러 번의 쿼리는 하나의 작업으로 묶어서 관리해야 하며 그렇지 않을 경우에 데이터 무결성을 보장할 수 없게 된다. 그러므로 하나로 묶인 SQL 문장이 모두 제대로 실행되면 최종적으로 데이터베이스에 반영하고 만약 하나라도 실패하게 되면 현재 작업 이전으로 되돌려야 하는데 이때 사용하는 명령에 대해 알아보자.

▶ 변경된 데이터 적용하기

INSERT, UPDATE, DELETE 문을 사용해 데이터를 변경하고 COMMIT 명령을 실행하면 변경된 데이터가 실제 데이터베이스 파일에 적용된다. INSERT, UPDATE, DELETE 문을 사용해 데이터를 변경하게 되면 곧 바로 데이터가 데이터베이스에 적용되는 것이 아니라 메모리에서만 변경되고 COMMIT 명령을 실행해야 비로소 데이터베이스 파일에 저장된다. COMMIT 명령은 이전의 ROLLBACK 명령이나 COMMIT 명령이 실행된 이후부터 현재 COMMIT 명령이 실행되기 바로 직전까지의 작업을 데이터베이스 파일에 반영한다.

```
COMMIT [WORK] [TO SAVEPOINT savepoint_name]
```

▶ 변경된 데이터 되돌리기

INSERT, UPDATE, DELETE 문을 사용해 데이터를 변경하고 ROLLBACK 명령을 실행하면 변경된 데이터를 이전의 상태로 되돌릴 수 있다. ROLLBACK 명령은 COMMIT 명령의 반대되는 개념으로 바로 이전의 ROLLBACK 명령이나 COMMIT 명령이 실행된 시점으로 변경된 사항을 되돌릴 수 있다. COMMIT 명령이나 ROLLBACK 명령은 데이터 조작어(DML)에 속하는 SQL 명령이 아니라 트랜잭션 처리를 위해 사용되는 SQL(TCL, Transaction Control Language) 명령이다.

```
ROLLBACK [WORK] [TO SAVEPOINT savepoint_name]
```

DML과 TCL 실습에 앞서 먼저 아래 SQL 쿼리를 이용해 실습용 테이블을 생성하고 데이터를 추가하자

-- member00 테이블 생성

```
CREATE TABLE member00(  
    no NUMBER,  
    id VARCHAR2(15),  
    password VARCHAR2(15),  
    name VARCHAR2(5 CHAR),  
    gender VARCHAR2(2 CHAR),  
    email VARCHAR2(20),  
    phone VARCHAR2(13),  
    reg_date DATE  
);
```

-- member00 테이블에 데이터 추가

```
INSERT INTO member00 VALUES(1, 'oracle', 'oracle1234',  
    '홍길동', '남성', 'oracle@naver.com', '010-1234-5678', SYSDATE);  
INSERT INTO member00 VALUES(2, 'javagirl', 'javamania',  
    '김하나', '여성', 'java@oracle.com', '010-4321-9876', SYSDATE);  
INSERT INTO member00 VALUES(3, 'database', 'mysql',  
    '최고집', '남성', 'database@gmail.com', '010-1111-7777', SYSDATE);
```

commit;

```
CREATE TABLE member01(  
    id VARCHAR2(15),  
    name VARCHAR2(5 CHAR),  
    pass VARCHAR2(15),  
    gender VARCHAR2(2 CHAR),  
    phone VARCHAR2(13 CHAR),  
    email VARCHAR2(20),  
    reg_date DATE  
);
```

8.2 데이터 추가하기

INSERT문은 테이블에 새로운 데이터를 추가하기 위해 사용하는 SQL 명령으로 다음과 같은 구문으로 구성되어 있다. 컬럼명은 생략할 수 있으며 컬럼명이 생략되는 경우 테이블에 생성된 컬럼의 개수와 컬럼의 순서를 지켜서 VALUES에 데이터를 지정해야 한다.

```
INSERT INTO 테이블명([컬럼명1, 컬럼명2..., 컬럼명n])
VALUES(데이터1, 데이터2..., 데이터n);
```

```
INSERT INTO member01 VALUES('midas', '마이더스', '12345678', '남성',
'041-1234-5678', 'midas@gmail.com', '2010-01-05');
```

```
INSERT INTO member01 VALUES('midas11', '왕호감', '45678901', '남성',
NULL, 'midas11@naver.com', SYSDATE);
```

-- 컬럼명을 나열할 경우 나열한 순서대로 데이터를 지정하면 된다.

```
INSERT INTO member01(id, name, gender, email, reg_date, pass)
VALUES('member01', '어머나', '여성', 'member01@daum.net', SYSDATE, '12345');
```

```
commit;
```

-- 다음과 같이 기존의 테이블이나 뷰의 조회 결과를 테이블의 데이터로 추가할 수 있다.

```
INSERT INTO member01
SELECT id, name, password, gender, phone, email, reg_date
FROM member00 WHERE no <= 2;
```

```
commit;
```

8.3 데이터 수정하기

UPDATE문은 테이블에 이미 저장된 데이터를 수정하기 위해 사용하는 명령으로 다음과 같은 구문으로 구성되어 있다. UPDATE 문에서 WHERE 절을 생략하면 테이블의 모든 행이 수정되므로 각별한 주의가 필요하다.

```
UPDATE 테이블명
SET 컬럼명1=데이터1, 컬럼명2=데이터2..., 컬럼명n=데이터n
WHERE 조건;
```

```
UPDATE member01
SET name='한국인', gender='여성', phone='010-5555-9999'
WHERE id='midas11';
```

```
commit;
```

-- 다음과 같이 서브쿼리를 이용해 테이블의 데이터를 수정할 수 있다.

```
UPDATE member01
    SET (phone, email) = (SELECT phone, email
                          FROM member00
                          WHERE id = 'database')
WHERE id = 'midas11';

commit;
```

8.4 데이터 삭제하기

DELETE문은 테이블에 저장된 데이터를 삭제하기 위해 사용하는 명령으로 다음과 같은 구문으로 구성되어 있다. DELETE도 WHERE 절을 생략하면 테이블의 모든 행이 삭제되므로 주의가 필요하다.

```
DELETE [FROM] 테이블명
WHERE 조건;
```

```
DELETE FROM member01
WHERE id='midas11';

commit;
```

9. 테이블 생성과 제약조건

데이터베이스에서는 데이터를 테이블에 보관하고 관리하는데 이 테이블은 마치 엑셀시트와 같이 열(컬럼, Column)과 행(로우, Row 또는 레코드, Record)의 2차원 구조로 되어 있다.

이 장에서는 데이터를 보관하고 관리하기 위해 사용되는 테이블을 생성하고 그 구조를 변경하고 테이블을 삭제하는 SQL 명령에 대해 알아볼 것이다.

테이블은 데이터베이스 객체이며 테이블과 같이 데이터베이스를 구성하는 객체를 생성하고 수정하고 삭제하는 SQL 명령을 데이터 정의어(DDL, Data Definition Language)라고 한다.

우리는 이장에서 DDL을 사용해 오라클 데이터베이스 객체인 테이블을 생성하고 변경하는 것에 대해 학습하면서 테이블을 정의할 때 필요한 컬럼의 데이터 타입에 대해서도 알아볼 것이다. 그리고 데이터 무결성을 유지하기 위해 테이블에 지정하는 제약조건에 대해서도 알아볼 것이다.

9.1 테이블 생성하기

테이블을 생성하기 위해서는 아래와 같은 형식으로 SQL 문을 작성해야 한다.

여기에는 테이블 이름을 정의하고 테이블을 구성하는 각 컬럼의 이름과 데이터 타입 그리고 무결성 제약조건을 정의해야 한다.

```
CREATE TABLE [schema.] table(  
    column datatype [DEFAULT expression] [column_constraint clause] [, ...]  
);
```

table은 테이블 이름이고 column은 컬럼 이름이다. 그리고 datatype에 column에 저장될 데이터의 타입을 지정한다. [DEFAULT expression]은 테이블에 데이터가 추가될 때 컬럼의 값이 입력되지 않으면 저장할 기본 값이다. [column_constraint]는 컬럼에 대해 정의하는 무결성 제약조건 이다. 무결성 제약조건은 데이터의 정합성(무모순성)을 유지하기 위해 컬럼에 지정하며 중복 없이 올바른 데이터만 테이블에 저장하고 관리하기 위한 제약조건 이다.

오라클에서 지원하는 데이터 타입은 크게 문자형, 숫자형, 날짜형과 대용량 데이터(LOB - Large Object)로 구분할 수 있으며 각각의 자세한 데이터 타입은 다음과 같다.

▶ 문자형 컬럼 정의

문자형 데이터 타입에는 CHAR, VARCHAR2, NCHAR, NVARCHAR2, LONG 타입이 있으나 고정 길이 문자형은 CHAR 타입과 가변길이 문자형은 VARCHAR2 타입이 많이 사용된다.

NCHAR 타입과 NVARCHAR2 타입은 국가별 문자 집합에 따라 크기가 적용되는데 NCHAR는 고정 길이이고 NVARCHAR2는 가변길이 문자형 데이터 타입이다. 또한 LONG 타입은 2GB까지 저장할 수 있는 가변길이 문자형 데이터 타입이나 오라클에서 LOB(Large Object) 타입이 나오고 나서 LOB 타입을 권장하고 있다.

CHAR 타입과 VARCHAR2 타입은 아래와 같이 SIZE와 그 SIZE의 단위로 BYTE 또는 CHAR를 지정할 수 있다. 단위를 생략하면 시스템 파라미터인 NLS_LENGTH_SEMANTICS에 지정한 단위가 적용된다.

- CHAR(SIZE [BYTE, CHAR])
- VARCHAR(SIZE [BYTE, CHAR])

```
CREATE TABLE char_table01 (
  name1 CHAR(6),
  name2 CHAR(6 CHAR),
  name3 VARCHAR2(6),
  name4 VARCHAR2(6 CHAR)
);
```

```
INSERT INTO char_table01 VALUES('ABC', '홍', 'ABC', '홍');
```

```
INSERT INTO char_table01 VALUES('ABC', '홍길', 'ABC', '홍길');
```

```
INSERT INTO char_table01 VALUES('ABC', '홍길동', 'ABC', '홍길동');
```

```
INSERT INTO char_table01 VALUES('ABC', '홍길동홍길동', 'ABC', '홍길동홍길동');
```

CHAR 타입은 고정길이 문자형이므로 컬럼에 저장된 데이터가 컬럼 SIZE 보다 작을 경우 공백(' ')으로 채워지지만 VARCHAR2 타입은 가변길이 문자형이므로 컬럼 SIZE 보다 저장된 데이터가 작아도 공백으로 채워지지 않는다.

앞에서 오라클 기본함수를 학습할 때도 언급했지만 이미 ANSI 코드로 정의되어 있는 영문 대소문자와 숫자 그리고 일부 특수문자는 1Byte로 저장되지만 한글과 같은 유니코드 문자는 시스템 설정에 따라서 달라지는데 “NLS_DATABASE_PARAMETERS”라는 시스템 뷰의 NLS_CHARACTERSET의 값이 “KO16MSWIN949”로 설정되어 있으면 한글을 2Byte로 취급하고 “AL32UTF8”로 설정되어 있으면 한글을 3Byte로 취급한다.

다음은 NLS_CHARACTERSET의 값이 “AL32UTF8”로 설정되어 있을 경우에 질의한 것으로 한글 한 문자는 3Byte로 취급하기 때문에 아래 그림과 같은 Byte의 크기가 된다.

```
SELECT LENGTHB(name1) name1, LENGTHB(name2) name2,
  LENGTHB(name3) name3, LENGTHB(name4) name4 FROM char_table01;
```

NAME1	NAME2	NAME3	NAME4
6	8	3	3
6	10	3	6
6	12	3	9
6	18	3	18

위의 그림에서 “NAME2” 컬럼의 조회 결과를 보면 좀 의아한 생각이 들것이다.
테이블을 생성할 때 “NAME2” 컬럼에 지정한 속성을 보면 아래와 같다.

name2 CHAR(6 CHAR)

이는 고정길이 여섯 문자로 지정해서 테이블을 생성하고 첫 행은 한글 한 문자를 추가했는데 길이가 8Byte로 조회되었다. 이는 한글 한 문자에 해당하는 3Byte와 나머지 다섯 문자는 공백으로 채워지기 때문에 1Byte씩을 차지하므로 결과가 8Byte가 된다. 다시 말해 한글 한 문자(3Byte) + 공백 한 문자(1Byte) x 공백 다섯 문자(5Byte) = 8Byte가 되는 것이다.

```
SELECT REPLACE(name1, ' ', '*') name1, REPLACE(name2, ' ', '*') name2,
       REPLACE(name3, ' ', '*') name3, REPLACE(name4, ' ', '*') name4
FROM char_table01;
```

NAME1	NAME2	NAME3	NAME4
ABC**	홍*****	ABC	홍
ABC**	홍길****	ABC	홍길
ABC**	홍길동***	ABC	홍길동
ABC**	홍길동홍길동	ABC	홍길동홍길동

아래 그림은 NLS_CHARACTERSET의 값이 “KO16MSWIN949”로 설정되어 있을 경우에 질의한 것으로 이 설정은 한글 한 문자를 2Byte로 취급하기 때문에 앞에서 “AL32UTF8”로 설정한 결과보다 크기가 작게 조회된 것을 확인할 수 있다.

NAME1	NAME2	NAME3	NAME4
6	7	3	2
6	8	3	4
6	9	3	6
6	12	3	12

▶ 숫자형 컬럼 정의

숫자형 데이터 타입에는 NUMBER, BINARY_FLOAT, BINARY_DOUBLE 타입이 있으나 NUMBER 타입이 가장 많이 사용된다. BINARY_FLOAT, BINARY_DOUBLE 타입은 부동소수점 수를 표현하지만 이름에서 알 수 있듯이 이진법의 정밀도를 사용하기 때문에 10진법의 정밀도를 사용하는 NUMBER 타입 보다 수의 정밀도가 떨어진다.

다른 DBMS에서는 정수형 INTEGER 타입과 실수형 DECIMAL 타입 등과 같은 타입을 제공하고 있다. 오라클에서도 이런 타입을 지정해 테이블을 생성할 수 있지만 내부적으로는 모두 NUMBER 타입으로 변환된다.

NUMBER 타입은 아래와 같이 prec(Precision - 정밀도)와 scale을 지정할 수 있는데 prec는 수의 전체 자리수를 의미하며 scale은 소수점 아래 자릿수를 의미한다.

참고로 prec는 최대 38자리까지 입력할 수 있다.

- NUMBER(prec | prec, scale)

```
CREATE TABLE num_table01(  
  num1 NUMBER,  
  num2 NUMBER(7),  
  num3 NUMBER(7, 2),  
  num4 NUMBER(5),  
  num5 NUMBER(6, -1)  
);
```

```
INSERT INTO num_table01 VALUES(12345.67, 12345.67, 12345.67, 12345.67, 12345.67);  
SELECT * FROM num_table01;
```

NUM1	NUM2	NUM3	NUM4	NUM5
12345.67	12346	12345.67	12346	12350

SELECT 쿼리의 결과인 위의 그림을 보면 NUMBER 타입에 prec와 scale 인수를 지정하지 않은 NUM1은 입력된 수 그대로 저장되고 prec를 7로 지정한 NUM2는 scale을 지정하지 않아서 소수점 아래 첫 번째 자리에서 반올림 되었다. 또한 수의 전체 prec를 7로 scale를 2로 지정한 NUM3은 입력된 수가 전체 7자리이고 소수점이 2자리이기 때문에 입력된 수 그대로 저장되었다. 그리고 prec를 5로 지정한 NUM4는 소수점 아래 자리를 지정하지 않아서 소수점 첫 번째 자리에서 반올림 한 수가 저장되었다. 끝으로 prec를 6으로 scale을 -1로 지정한 NUM5는 소수점을 기준으로 왼쪽 첫 번째 자리에서 반올림한 수가 저장되었다.

위에서 컬럼을 생성할 때 실제 입력되는 값의 정수부 길이 보다 prec를 작게 지정하였다면 아래와 같은 에러가 발생할 것이다. 예를 들어 num4 컬럼을 “num4 NUMBER(4)”로 지정하여 테이블을 생성하였다면 실제 입력되는 값이 prec에 지정한 4보다 크기 때문에 에러가 발생하게 된다. 또한 num3 컬럼은 num3 NUMBER(7, 2)를 지정하여 테이블을 생성하였기 때문에 이 num3 컬럼에 입력되는 데이터가 123456.789 라면 위와 동일한 에러가 발생하게 된다. 이는 scale에 지정한 2와 prec에 지정한 7에 의해서 소수점 아래 두 자리를 포함해서 총 7자리의 숫자가 되어야 하지만 정수부 6자리와 소수점 아래 자리 두 자리가 입력되어 전체 정밀도인 7을 초과했기 때문에 아래와 같은 에러가 발생한다.

QL 오류: ORA-01438: value larger than specified precision allowed for this column
01438. 00000 - "value larger than specified precision allowed for this column"

▶ 날짜형 컬럼 정의

날짜형 데이터 타입에는 DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL YEAR, INTERVAL DAY 등이 있으나 이 중에서 DATE 타입과 TIMESTAMP 타입이 가장 많이 사용된다. DATE 타입은 날짜와 시간 정보를 세기, 연도, 월, 일, 시, 분, 초 정보로 저장하며 율리우스력(Julian Date)를 사용해 BC 4312년 1월 1일부터 시작한다. DATE 타입은 시스템 파라미터인 NLS_DATE_FORMAT에 설정된 값으로 날짜 정보를 표시한다. TIMESTAM 타입은 DATE 타입을 확장한 날짜형 데이터 타입으로 연도, 월, 일, 시, 분, 초 정보로 날짜와 시간 정보를 저장하며 DATE 타입 보다 더 정밀한 시간 데이터를 표현할 수 있다. TIMESTAMP 타입은 시스템 파라미터인 NLS_TIMESTAMP_FORMAT에 설정된 값으로 날짜 정보를 표시한다. TIMESTAMP WITH TIME ZONE 타입은 지역 시간대(Local Time)와 그리니치 표준시와의 차이를 포함한 날짜와 시간정보를 저장할 수 있는 타입이다.

아래에서 fractional_second_precision는 밀리 초 정보의 정밀도를 지정할 수 있는 옵션으로 0 ~ 9까지 지정할 수 있다. 이를 생략하면 DEFAULT 값인 6으로 설정된다.

- DATE
- TIMESTAMP [fractional_second_precision]
- TIMESTAMP [fractional_second_precision] WITH TIME ZONE

먼저 현재 접속한 세션에서 위의 3가지 타입의 포맷을 아래의 SQL 쿼리를 사용해 변경하고 3가지 날짜타입 컬럼을 가진 테이블을 생성한 후 테스트 해 보자.

```
ALTER SESSION SET NLS_DATE_FORMAT='YYYY-MM-DD HH24:MI:SS';
```

```
ALTER SESSION SET NLS_TIMESTAMP_FORMAT='YYYY-MM-DD HH24:MI:SS.FF';
```

```
ALTER SESSION SET NLS_TIMESTAMP_TZ_FORMAT='YYYY-MM-DD HH24:MI:SS.FF TZR';
```

```
CREATE TABLE date_table01(  
    date1 DATE,  
    date2 TIMESTAMP,  
    date3 TIMESTAMP WITH TIME ZONE  
);
```

```
INSERT INTO date_table01 VALUES(SYSDATE, SYSDATE, SYSDATE);
```

```
INSERT INTO date_table01 VALUES(SYSDATE, SYSTIMESTAMP, SYSTIMESTAMP);
```

```
SELECT * FROM date_table01;
```

DATE1	DATE2	DATE3
2017-01-22 23:32:40	2017-01-22 23:32:40.000000000	2017-01-22 23:32:40.000000000 ASIA/SEOUL
2017-01-22 23:32:40	2017-01-22 23:32:40.941000000	2017-01-22 23:32:40.941000000 +09:00

9.2 무결성 제약조건

무결성 제약조건(Integrity Constraints)은 데이터베이스에서 데이터의 정합성(Consistency, 무모순성)을 유지하기 위해 컬럼에 지정하는 규칙으로 바람직하지 못한 데이터가 저장되는 것을 미연에 방지하기 위해 사용한다. 제약조건은 데이터를 조작하는 삽입, 삭제, 수정 등의 작업에서 이상현상(Anomaly)이 발생하지 않도록 데이터가 저장되는 테이블에 적용하는 제약이라고 생각하면 되겠다. 제약조건을 지정하는 방식에는 컬럼을 정의할 때 컬럼 레벨에 직접 지정하는 인라인 방식과 컬럼 정의 후에 CONSTRAINT 키워드를 사용해 테이블 레벨에 지정하는 아웃라인 방식이 있다.

▶ 제약조건 확인하기

테이블의 구조를 확인할 수 있는 DESC(Describe) 명령으로는 각 컬럼의 데이터 타입과 크기 그리고 NOT NULL 제약조건만 확인할 수 있다.

테이블에 지정된 제약조건은 데이터 디렉터리에 저장된 시스템 뷰인 USER_CONSTRAINTS를 조회하면 테이블에 지정된 모든 제약조건을 확인할 수 있다.

오라클은 데이터베이스 자원을 효율적으로 관리하기 위해 데이터베이스 이름, 사용자 계정에 대한 정보, 오라클 객체(테이블, 뷰, 시퀀스 등)에 대한 정보 등의 다양한 정보를 저장하는 시스템 테이블을 가지고 있는데 이를 데이터 디렉터리(Data Dictionary)라고 부른다. 데이터 디렉터리는 사용자를 추가 또는 변경하거나 테이블 생성 등의 작업을 할 때 오라클에 의해 자동으로 갱신되는 시스템 테이블로써 사용자는 데이터 디렉터리의 내용을 직접 수정하거나 삭제할 수 없다.

아래의 쿼리로 USER_CONSTRAINTS 시스템 뷰를 조회하면 EMP 테이블에 지정된 모든 제약조건을 조회할 수 있다. 주의할 사항은 WHERE 절에 지정하는 테이블 이름은 반드시 대문자로 지정해야 제약조건을 제대로 확인할 수 있다.

```
SELECT constraint_name, constraint_type FROM user_constraints
WHERE table_name='EMP';
```

▶ NOT NULL 속성 지정하기

테이블 생성시 컬럼에 NULL을 허용할지 허용하지 않을지 지정할 수 있는데 컬럼에 NOT NULL 제약조건을 지정하면 테이블에 데이터가 저장될 때 반드시 데이터가 입력되어야 한다.

NOT NULL 제약조건을 지정하지 않은 컬럼은 기본적으로 NULL 값을 허용하게 된다.

NOT NULL 제약조건은 인라인(컬럼 레벨) 방식으로만 지정할 수 있다.

```
CREATE TABLE null_table01(
  name VARCHAR2(5 CHAR) NOT NULL,
  age NUMBER(3) NOT NULL,
  text VARCHAR2(50)
);
```

```
INSERT INTO null_table01 VALUES('홍길동', 25, '좋은친구');
```

```
INSERT INTO null_table01(name, age) VALUES('홍길동', 30);
```

```
INSERT INTO null_table01(name) VALUES('임격정');
```

다음 SQL 쿼리 3개를 실행하면 첫 번째와 두 번째는 INSERT가 되지만 세 번째는 NOT NULL 제약조건에 위배되어 오류가 발생하게 된다.

오류 보고 -

ORA-01400: NULL을 ("HR"."NULL_TABLE01"."AGE") 안에 삽입할 수 없습니다

▶ 유일키 제약조건(Unique Constraint)

유일키 제약조건은 테이블에서 데이터를 유일하게 식별하기 위해 지정하는 제약조건이다. 그래서 이 제약조건이 설정된 컬럼에는 중복된 값이 저장될 수 없다.

유일키는 단일 컬럼을 유일키로 지정할 수 있고 한 개 이상의 컬럼을 묶어서 유일키로 지정할 수도 있다. 이렇게 한 개 이상의 컬럼을 묶어서 키로 지정하는 것을 복합키(Composite Key)라고 한다. 유일키는 테이블에서 유일한 값을 의미하므로 값이 없음을 의미하는 NULL은 유일키 비교대상에서 제외된다. 그래서 UNIQUE 제약조건만 지정한 컬럼은 NULL 값을 저장할 수 있다.

```
CREATE TABLE unique_table(
  id VARCHAR2(10) UNIQUE NOT NULL,
  name VARCHAR2(5 CHAR) NOT NULL,
  phone VARCHAR2(13) NOT NULL,
  email VARCHAR2(20) NOT NULL,
  text VARCHAR2(100),
  CONSTRAINT unique_table_uk_test UNIQUE(phone, email)
);
```

```
INSERT INTO unique_table(id, name, phone, email, text) VALUES('member', '강감찬',
  '010-2222-3333', 'test@naver.com', '유일키는 NULL을 비교 대상에서 제외시킨다.');
```

```
INSERT INTO unique_table(id, name, phone, email) VALUES('member1', '김유신',  
'010-2222-3333', 'test@naver.com');
```

위의 INSERT 문을 실행하게 되면 첫 번째 INSERT 문은 제대로 실행되지만 두 번째 INSERT 문이 실행될 때 아래와 같이 unique 제약조건 위반이라는 에러가 발생하게 된다.

오류 보고 -

ORA-00001: 무결성 제약 조건(HR.UNIQUE_TABLE_UK)에 위배됩니다

시스템에 따라서 한글 에러 메시지가 아닌 아래와 같은 영문 에러 메시지가 출력될 수 있다.

SQL 오류: ORA-00001: unique constraint (HR.SYS_C007268) violated
00001. 00000 - "unique constraint (%s.%s) violated"

▶ CHECK 제약조건

CHECK 제약조건은 컬럼에 입력되는 데이터를 조사하여 CHECK 제약조건에 설정된 조건에 해당하는 데이터만 입력될 수 있도록 하는 제약조건이다. CHECK 제약조건 또한 NULL을 허용하기 때문에 NOT NULL 속성을 별도로 지정하는 것이 좋다. 아래는 인라인 방식으로 CHECK 제약조건을 지정한 예이다.

```
CREATE TABLE mem(  
  id VARCHAR2(15)  
  name VARCHAR2(5 CHAR) NOT NULL,  
  gender VARCHAR2(2 CHAR) CHECK(gender IN('남성', '여성')),  
  sal NUMBER NOT NULL CHECK(sal BETWEEN 200 AND 500)  
);
```

- 정상적으로 데이터가 추가됨

```
INSERT INTO mem VALUES('hostman', '임꺽정', '남성', 300);
```

-- NULL이 문제없이 추가됨

```
INSERT INTO mem VALUES('jsphost', '홍길동', NULL, 400);
```

-- 남성 또는 여성 두 값 중 하나여야 하므로 에러 발생

```
INSERT INTO mem VALUES('hostman1', '임꺽정', '중성', 300);
```

위에서 세 번째 쿼리를 실행하면 체크 제약조건에 지정한 남성 또는 여성이 아니므로 다음과 같은 오류가 발생한다.

오류 보고 -

ORA-02290: 체크 제약조건(HR.SYS_C0011202)이 위배되었습니다

다음은 아웃라인 방식으로 CHECK 제약조건을 지정한 예이다.

```
CREATE TABLE mem01(  
  id VARCHAR2(15)  
  name VARCHAR2(5 CHAR) NOT NULL,  
  gender VARCHAR2(2 CHAR) NOT NULL,  
  sal NUMBER NOT NULL,  
  CONSTRAINT mem_gender_ck CHECK(gender IN('남성', '여성')),  
  CONSTRAINT mem_sal_ck CHECK(sal BETWEEN 200 AND 500)  
);
```

-- 정상적으로 데이터가 추가됨

```
INSERT INTO mem01 VALUES('hostman01', '이순신', '남성', 300);
```

-- NOT NULL 오류 발생

```
INSERT INTO mem01 VALUES('jsphost01', '강감찬', NULL, 400);
```

-- 범위를 넘어선 값이 입력되어 오류 발생

```
INSERT INTO mem01 VALUES('hostman01', '김유신', '남성', 600);
```

▶ DEFAULT

DEFAULT는 데이터가 입력되지 않으면 지정한 데이터를 기본 값으로 입력하기 위해 사용하는 속성으로 엄밀히 따져 제약조건은 아니다. DEFAULT 속성을 지정할 때 주의해야 할 사항은 반드시 데이터 타입 다음에 그리고 NOT NULL 제약조건 앞에 지정해야 한다.

```
CREATE TABLE default_dept(  
  deptno NUMBER  
  deptname VARCHAR2(10 CHAR) NOT NULL,  
  location VARCHAR2(10 CHAR) DEFAULT '서울' NOT NULL  
);
```

-- 정상적으로 데이터가 추가됨

```
INSERT INTO default_dept VALUES(1005, '인사부', '대전');
```

-- 기본 값이 입력됨

```
INSERT INTO default_dept(deptno, deptname) VALUES(1006, '관리부');
```

```
-- NOT NULL 오류
INSERT INTO default_dept VALUES(1007, '총무부', NULL);
```

DEFAULT 속성은 아래의 SQL 명령으로 변경 할 수 있다.

```
ALTER TABLE default_dept
MODIFY location DEFAULT '부산';
```

```
-- location 컬럼에 기본 값인 부산이 저장된다.
INSERT INTO default_dept(deptno, deptname) VALUES(1008, '전산부');
```

▶ 기본키 제약조건(Primary Key Constraint)

기본키는 유일키와 같이 테이블에서 데이터를 유일하게 구분할 수 있는 제약조건으로 기본키로 정의된 컬럼에는 유니크 인덱스(Unique Index)가 지정되기 때문에 중복된 값이 저장될 수 없다. 또한 기본키로 지정되면 자동으로 NOT NULL 속성이 지정되기 때문에 NULL 값을 가질 수 없다.

기본키는 테이블에서 한 행을 유일하게 구분할 수 있어야 하므로 기본키로 지정된 컬럼은 중복된 값이나 NULL 값을 가질 수 없는데 이를 기본키 제약조건(또는 개체 무결성 제약조건, Entity Integrity Constraint) 이라고 한다.

유일키는 NULL 값을 허용하지만 기본키로 지정된 컬럼은 NULL 값을 허용하지 않는 NOT NULL 속성을 가진다. 테이블을 정의할 때 어떤 컬럼을 기본키로 지정하게 되면 UNIQUE 인덱스가 생성되고 그 컬럼은 UNIQUE한 속성과 NOT NULL 속성을 가지게 된다.

기본키도 유일키와 마찬가지로 단일 컬럼을 기본키로 지정할 수도 있고 한 개 이상의 컬럼을 묶은 복합키(Composite Key)를 기본키로 지정할 수도 있지만 최소한의 컬럼을 이용해 기본키를 지정해야 한다.

테이블에 반드시 기본키를 지정해야 하는 것은 아니지만 데이터 무결성을 지키기 위해 기본키를 지정하는 것이 바람직한 방법이다.

■ 인라인 방식 기본키 제약조건 지정

```
CREATE TABLE pri_table01(
  no NUMBER PRIMARY KEY,
  name VARCHAR2(50 CHAR) CONSTRAINT pri_table01_notnull NOT NULL,
  address VARCHAR2(80 CHAR)
);
```

```
INSERT INTO pri_table01 VALUES(1, '홍길동', '서울 구로구 구로동');
```

```
INSERT INTO pri_table01(no, name) VALUES(1, '강감찬');
```



```
INSERT INTO pri_table01(name) VALUES('이순신');
```

위의 INSERT 문을 실행하게 되면 첫 번째 INSERT 문은 제대로 실행되지만 두 번째 INSERT 문이 실행될 때 아래와 같이 unique 제약조건 위반이라는 에러가 발생하게 된다.

오류 보고 -

ORA-00001: unique constraint (HR.SYS_C0011154)에 위배됩니다

시스템에 따라서 위의 한글 에러 메시지가 아니라 다음과 같은 영문 에러 메시지가 출력될 수 있다.

SQL 오류: ORA-00001: unique constraint (HR.SYS_C0011154) violated

00001. 00000 - "unique constraint (%s.%s) violated"

또한 세 번째 INSERT 문이 실행될 때 아래와 같이 테이블에 NULL을 추가 할 수 없다는 에러가 발생하게 된다.

이는 위에서 테이블을 생성할 때 “no” 컬럼을 PRIMARY KEY로 지정했기 때문에 “no” 컬럼은 UNIQUE한 속성과 NOT NULL 속성을 가지고 있기 때문에 발생하는 것이다.

오류 보고 -

ORA-01400: NULL을 ("HR"."PRI_TABLE01"."NO") 안에 삽입할 수 없습니다

시스템에 따라서 위의 한글 에러 메시지가 아니라 다음과 같은 영문 에러 메시지가 출력될 수 있다.

SQL 오류: ORA-01400: cannot insert NULL into ("HR"."PRI_TABLE01"."NO")

01400. 00000 - "cannot insert NULL into (%s)"

■ 아웃라인 방식 기본키 제약조건 지정

```
CREATE TABLE pri_table02(  
    no NUMBER,  
    name VARCHAR2(5 CHAR) NOT NULL,  
    address VARCHAR2(80 CHAR),  
    CONSTRAINT pri_table02_pk PRIMARY KEY(no)  
);
```

```
INSERT INTO pri_table02 VALUES(1, '홍길동', '서울 구로구 구로동');
```

```
INSERT INTO pri_table02(no, name) VALUES(1, '강감찬');
```

```
INSERT INTO pri_table02(name) VALUES('이순신');
```

위의 INSERT 문을 실행하게 되면 첫 번째 INSERT 문은 제대로 실행되지만 두 번째 INSERT 문이 실행될 때 앞서와 마찬가지로 unique 제약조건 위반이라는 에러가 발생하게 되고 세 번째 INSERT 문이 실행될 때 테이블에 NULL을 추가 할 수 없다는 에러가 발생하게 된다.

■ 테이블 생성 후 기본키 제약조건 추가

```
CREATE TABLE pri_table03(  
    no NUMBER,  
    name VARCHAR2(50 CHAR) NOT NULL,  
    address VARCHAR2(80 CHAR)  
);
```

```
ALTER TABLE pri_table03  
ADD CONSTRAINT pri_table03_pk PRIMARY KEY(no);
```

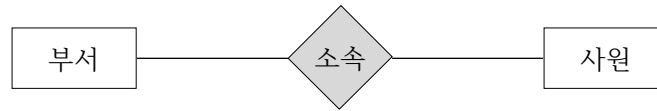
▶ 외래키 제약조건(Foreign Key Constraint)

관계형 데이터베이스에서 관계(Relationship)라는 말은 바로 테이블과 테이블 사이의 관계를 의미하며 이 관계는 각각의 테이블에서 공통적인 값을 갖는 컬럼을 통해 이루어진다.

우리가 실습용으로 사용하는 테이블 emp와 dept는 서로 관계를 맺고 있는 테이블로써 dept 테이블은 부서에 대한 정보를 저장하고 있으며 emp 테이블은 사원에 대한 정보를 저장하고 있다.

dept 테이블을 살펴보면 회사에 존재하는 모든 부서에 대한 정보를 저장하고 있고 이 부서들을 유일하게 구분할 수 있는 부서 번호(deptno) 컬럼을 PRIMARY KEY로 지정하고 있다. 또한 emp 테이블을 살펴보면 부서 테이블의 부서 번호(deptno) 컬럼과 동일한 이름을 가진 deptno 컬럼이 존재한다. emp 테이블의 부서 번호(deptno)는 그 사원이 어느 부서에 속해 있는지를 나타내고 있다. 즉 회사에서 모든 사원은 부서 테이블에 존재하는 하나의 부서에 소속되어 있어야 한다. 만일 어떤 사원의 정보가 부서 테이블(dept)에 존재하지 않는 부서 번호(deptno)를 가지고 있다면 이것은 올바른 데이터가 아닐 것이다. 이렇게 부서 테이블(dept)에 존재하는 부서 번호(deptno) 만을 사원 테이블(emp)에 저장하도록 하는 것을 참조 무결성(Referential Integrity) 이라고 한다. 부서 테이블과 사원 테이블의 관계는 아래 그림과 같이 소속이라는 관계가 성립된다.

소속이라는 관계는 부서 테이블(dept)과 사원 테이블(emp)에 공통으로 존재하는 부서 번호(deptno) 컬럼을 통해 이루어지며 사원 테이블의 부서 번호(deptno) 컬럼에는 부서 테이블(dept)에 존재하는 부서 번호(deptno) 컬럼의 값만 저장되게 해야 한다. 그러므로 사원 테이블(emp)에서 부서 테이블(dept)의 부서 번호(deptno) 컬럼에 존재하는 값을 참조하게 만들어야 한다. 이럴 경우 사원 테이블(emp)이 부서 테이블(dept)을 참조하는 종속관계가 성립되는데 이런 종속 관계에서 주체가 되는 부서 테이블(dept)을 부모 테이블이라 하고 이 부모 테이블을 참조하는 사원 테이블(emp)을 자식 테이블이라 한다.



부서에는 한 명 이상의 직원이 소속된다.
 직원은 하나의 부서에 소속되어 진다.

여기서 종속 관계의 주체라 함은 두 테이블 간의 관계가 위의 내용과 같이 능동형으로 표현됨을 의미한다. 주체 관계 구분이 모호한 경우에는 어느 테이블의 데이터가 먼저 정의되어 있어야 그 값을 다른 테이블에서 참조 할 수 있는지를 따져보고 이를 기준으로 먼저 정의되어야 하는 데이터가 속한 테이블을 부모 테이블로 이를 참조하는 테이블을 자식테이블로 구분하면 된다.

언뜻 생각해 보면 직원 테이블(emp)이 주체인 것처럼 보이지만 부서가 먼저 정의되어 있어야 직원 데이터를 저장할 때 그 직원이 소속된 부서를 지정할 수 있기 때문에 부서 테이블(dept)이 부모가 되고 직원 테이블이 자식이 된다.

자식인 직원 테이블(emp)의 부서 번호(deptno)를 통해 부모인 부서 테이블(dept)의 부서 번호(deptno)를 참조하고 있기 때문에 직원 테이블의 부서 번호(deptno) 컬럼을 외래키(FOREIGN KEY, 외부키라고도 함)라 하고 부서 테이블(dept)의 부서 번호(deptno) 컬럼을 부모키라고 한다. 부모키가 되기 위해서 해당 컬럼은 반드시 부모 테이블의 기본키(PRIMARY KEY)이거나 유일키(UNIQUE KEY)로 지정되어 있어야 한다. 이렇게 부모 테이블의 특정 컬럼을 참조하기 위해 자식 테이블의 특정 컬럼을 외래키(FOREIGN KEY)로 지정하는 것을 외래키 제약조건(Foreign Key Constraint) 또는 참조 무결성 제약조건(Referential Integrity Constraint)이라 한다.

외래키 제약조건도 기본키와 같이 한 개 이상의 컬럼을 묶어서 복합키(Composite Key)로 지정할 수 있으나 참조하는 부모 테이블의 기본키(PRIMARY KEY)나 유일키(UNIQUE KEY)도 반드시 복합키가 되어야 하며 키를 구성하는 순서와 개수도 모두 동일해야 한다. 또한 외래키 제약조건을 지정해도 NULL 값을 허용하므로 외래키 제약조건을 지정하는 컬럼에 NULL이 저장되지 않도록 NOT NULL 속성을 추가로 지정하는 것이 바람직한 방법이라 할 수 있다. 참고로 참조 무결성 제약조건은 자식 테이블에서 참조하고 있는 부모 테이블의 컬럼 값 중 하나가 일치하거나 NULL 만 입력 가능해야 한다는 조건이기 때문에 NULL 값이 허용된다.

외래키 제약조건 또한 다음과 같이 인라인 방식과 아웃라인 방식 그리고 테이블을 생성한 후에 ALTER 문을 사용해 제약조건을 추가할 수 있다.

먼저 다음과 같이 부모 테이블인 dept01 테이블을 생성하고 이를 참조하는 emp01 테이블에 3가지 방식을 사용해 외래키 제약조건을 지정해 보자.

```

CREATE TABLE dept01(
  deptno NUMBER,
  deptname VARCHAR2(10 CHAR) NOT NULL,
  location VARCHAR2(10 CHAR) NOT NULL,
  CONSTRAINT dept01_pk PRIMARY KEY(deptno)
);
  
```

```

INSERT INTO dept01 VALUES(1001, '관리부', '서울');
  
```

```
INSERT INTO dept01 VALUES(1002, '영업부', '대전');
```

```
INSERT INTO dept01 VALUES(1003, '전산부', '인천');
```

```
INSERT INTO dept01 VALUES(1004, '생산부', '충남');
```

■ 인라인 방식 외래키 제약조건 지정

```
CREATE TABLE emp01(  
  empno NUMBER PRIMARY KEY,  
  name VARCHAR2(5 CHAR) NOT NULL,  
  hire_date DATE NOT NULL,  
  deptno NUMBER NOT NULL  
    CONSTRAINT emp01_fk REFERENCES dept01(deptno)  
);
```

■ 아웃라인 방식 외래키 제약조건 지정

```
CREATE TABLE emp01(  
  empno NUMBER,  
  name VARCHAR2(5 CHAR) NOT NULL,  
  hire_date DATE NOT NULL,  
  deptno NUMBER NOT NULL,  
  CONSTRAINT emp01_pk PRIMARY KEY(empno),  
  CONSTRAINT emp01_fk FOREIGN KEY(deptno)  
    REFERENCES dept01(deptno)  
);
```

■ 테이블 생성 후 외래키 제약조건 추가

```
CREATE TABLE emp01(  
  empno NUMBER,  
  name VARCHAR2(5 CHAR) NOT NULL,  
  hire_date DATE NOT NULL,  
  deptno NUMBER NOT NULL  
);
```

```
ALTER TABLE emp01  
ADD CONSTRAINT emp01_pk PRIMARY KEY(empno);
```

```
ALTER TABLE emp01
```

```
ADD CONSTRAINT emp01_fk FOREIGN KEY(deptno)
REFERENCES dept01(deptno);
```

-- 정상적으로 데이터가 추가됨

```
INSERT INTO emp01 VALUES(100, '홍길동', SYSDATE, 1001);
```

-- 부모키가 없어 오류발생

```
INSERT INTO emp01 VALUES(101, '임꺽정', SYSDATE, 1000);
```

위의 INSERT 문을 실행하게 되면 첫 번째 INSERT 문은 제대로 실행되지만 두 번째 INSERT 문을 실행하면 참조하는 키가 없어서 아래와 같은 에러가 발생한다.

오류 보고 -

ORA-02291: 무결성 제약조건(HR.EMP02_FK)이 위반되었습니다- 부모 키가 없습니다

▶ 제약조건 변경하기

테이블에 이미 지정된 제약조건을 변경하기 위해서는 여러 가지 사항을 체크해야 한다.

테이블에 저장된 데이터가 변경하려는 제약조건을 벗어난 데이터가 이미 존재한다면 제약조건을 변경하는 순간 데이터 무결성 제약 조건을 위반하게 되므로 이런 경우에는 제약조건을 변경할 수 없다. 예를 들어 이미 NULL 데이터가 저장된 컬럼에 NOT NULL 속성을 지정하게 되면 제약조건이 지정되는 순간 제약조건 위반이 되므로 이 컬럼은 NOT NULL 제약조건으로 수정할 수 없게 된다.

테이블을 생성한 후 제약조건을 추가하기 위해서 ALTER TABLE 문에서 ADD CONSTRAINT 명령을 사용해 제약조건을 추가 할 수 있으나 NOT NULL 제약조건을 추가하기 위해서는 아래와 같이 ALTER TABLE 문에서 MODIFY 명령을 사용해 수정할 수 있다.

```
ALTER TABLE default_dept MODIFY deptno NOT NULL;
```

```
ALTER TABLE default_dept MODIFY deptno CONSTRAINT notnull_dept NOT NULL;
```

```
CREATE TABLE mem02(
  id VARCHAR2(15) PRIMARY KEY,
  name VARCHAR2(5 CHAR),
  gender VARCHAR2(2 CHAR) NOT NULL,
  sal NUMBER NOT NULL,
  CONSTRAINT mem_sal_ck CHECK(sal BETWEEN 200 AND 500)
);
```

```
INSERT INTO mem02(id, gender, sal) VALUES('hongmantoo', '남성', 200);
```

```
INSERT INTO mem02 VALUES('mantoo', '홍만이', '남성', 200);
```

```
INSERT INTO mem02 VALUES('kimhana', '김하나', '여성', 400);
```

-- 이미 NULL 데이터가 있으므로 오류 발생

```
ALTER TABLE mem02
```

```
MODIFY name CONSTRAINT mem02_notnull NOT NULL;
```

-- 이미 M과 F가 아닌 다른 데이터가 있으므로 오류 발생

```
ALTER TABLE mem02
```

```
MODIFY gender CONSTRAINT mem02_check CHECK(gender IN('M', 'F'));
```

-- 기존 데이터가 지정하려고 하는 범위에 있으므로 수정됨

```
ALTER TABLE mem02
```

```
MODIFY sal CONSTRAINT mem02_sal_check CHECK(sal BETWEEN 200 AND 600);
```

아래 SQL 쿼리로 사용자 제약조건을 조회해 보면 mem02_sal_check라는 이름의 제약조건이 추가된 것을 확인할 수 있다.

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, SEARCH_CONDITION  
FROM user_constraints  
WHERE table_name='MEM02';
```

▶ 제약조건 삭제하기

테이블에 이미 지정된 제약조건을 삭제하기 위해서는 제약조건을의 이름을 알아야 한다.

먼저 mem02 테이블에 어떤 제약조건이 지정되어 있는지를 검색하고 삭제할 제약조건을 아래와 같이 지정하면 된다.

```
ALTER TABLE mem02
```

```
DROP CONSTRAINT 제약조건 이름;
```

```
ALTER TABLE mem02
```

```
DROP CONSTRAINT sys_c008008;
```

-- PRIMARY KEY 제약 조건을 삭제했기 때문에 중복된 ID가 저장된다.

```
INSERT INTO mem02 VALUES('mantoo', '김공부', '남성', 600);
```

```
-- 아래 SQL 쿼리를 실행하면 PRIMARY KEY 제약조건이 삭제된 것을 확인할 수 있다.
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, SEARCH_CONDITION
FROM user_constraints
WHERE table_name='MEM02';
```

9.3 테이블 복사

기존의 테이블 내용을 그대로 복사하는 방법은 아래와 같이 아주 간단하다.

아래와 같이 테이블의 내용을 복사하게 되면 원본 테이블의 인덱스는 복사되지 않으며 제약조건도 NOT NULL 속성만 복사된다. 그리고 LONG 타입의 컬럼은 복사할 수 없다. 아래와 같은 방식은 원본 테이블을 그대로 복사하거나 WHERE 절의 조건을 지정해 테이블 구조를 복사할 수 있지만 PRIMARY KEY와 같은 제약조건은 ALTER 문을 통해 추가로 지정해 줘야 한다.

```
CREATE TABLE mem03 AS SELECT * FROM mem02;
```

```
-- 아래 쿼리를 실행해 제약조건이 제대로 복사되었는지 확인해 보자.
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, SEARCH_CONDITION
FROM user_constraints
WHERE table_name='MEM03';
```

9.4 테이블 수정하기

테이블을 생성한 후 테이블 이름을 변경하거나 컬럼을 추가하거나 삭제 또는 이미 존재하는 컬럼의 데이터 타입이나 크기를 변경해야 할 경우가 종종 발생하는데 이에 대해서 알아보자.

▶ 테이블 이름 변경하기

RENAME 문은 데이터베이스 객체의 이름을 수정할 때 사용하는 SQL 명령이다.

위에서 생성한 member01 테이블의 이름을 아래와 같이 members로 변경해 보자.

```
RENAME member01 to members;
```

RENAME SQL 문으로 테이블 이름이 제대로 변경되었는지 아래의 SQL 문을 사용해 확인해 보자. user_tables은 시스템 뷰(View)로 현재 접속한 사용자가 생성한 테이블의 정보를 조회할 수 있다. 참고로 user_ 접두어로 시작하는 뷰는 현재 접속한 사용자가 소유자가 되는 스키마 정보를 저장하고 있는 뷰이다.

```
SELECT table_name FROM user_tables WHERE table_name = 'MEMBERS';
```

▶ 컬럼 추가하기

새로운 컬럼을 추가하는 것은 테이블 구조를 변경하는 것이므로 오라클 데이터베이스 객체를 변경하는 ALTER 명령을 사용한다.

테이블에 새로운 컬럼을 추가하려면 아래와 같이 ALTER TABLE ... ADD 명령을 사용하면 된다.

```
ALTER TABLE 테이블명  
ADD ([컬럼명 데이터타입 DEFAULT 제약조건]  
[, 컬럼명 데이터타입 DEFAULT 제약조건]...);
```

```
CREATE TABLE member02(  
  id VARCHAR2(15) PRIMARY KEY,  
  name VARCHAR2(50 CHAR) NOT NULL,  
  sex VARCHAR2(2 CHAR),  
  phone VARCHAR2(13 CHAR),  
  CONSTRAINT member02_gender_ck CHECK(gender IN('남성', '여성'))  
);
```

```
ALTER TABLE member02  
ADD (age NUMBER(3) NOT NULL,  
  mobile VARCHAR(13 CHAR) NOT NULL,  
  address VARCHAR(100 CHAR) NOT NULL);
```

```
INSERT INTO member02 VALUES('mantoo', '홍만이', '남성', '032-5236-8541', 25,  
'010-1234-5678', '구로동');
```

```
commit;
```

```
-- 테이블 구조가 변경되었는지 확인  
DESC member02;
```

▶ 컬럼 이름 변경하기

기존 컬럼의 이름을 수정하는 경우 아래와 같이 ALTER TABLE ... RENAME COLUMN 명령을 사용하면 된다.


```
ALTER TABLE 테이블명  
RENAME COLUMN 변경하려는 컬럼명 TO 변경할 컬럼명;
```

```
-- 성별 컬럼을 sex에서 gender로 변경  
ALTER TABLE member02 RENAME COLUMN sex TO gender;  
  
-- 컬럼명이 변경되었는지 확인  
DESC member02;
```

▶ 컬럼의 데이터 타입 수정하기

기존의 컬럼에 데이터가 하나도 없는 경우에는 컬럼의 데이터 타입이나 크기를 변경하는 것이 자유롭지만 이미 데이터가 존재한다면 컬럼의 크기는 저장된 데이터의 크기보다 같거나 커야 한다. 숫자 타입도 저장할 수 있는 수의 범위를 늘리거나 전체 자리수를 늘릴 수 있다. 컬럼의 데이터 타입을 수정하려면 아래와 같이 ALTER TABLE ... MODIFY 명령을 사용하면 된다.

```
ALTER TABLE 테이블명  
MODIFY ([컬럼명 데이터타입 DEFAULT 제약조건]  
[, 컬럼명 데이터타입 DEFAULT 제약조건]...);
```

```
ALTER TABLE member02  
MODIFY (age NUMBER(3),  
        mobile CHAR(13 CHAR),  
        address VARCHAR2(150 CHAR));
```

```
-- 컬럼명의 데이터 타입과 크기가 변경되었는지 확인  
DESC member02;
```

▶ 컬럼 삭제하기

컬럼을 제거하려면 아래와 같이 ALTER TABLE ... DROP COLUMN 명령을 사용하면 된다. 이 명령으로 컬럼을 제거하게 되면 저장된 데이터도 함께 삭제된다.

```
ALTER TABLE 테이블명  
DROP COLUMN 컬럼명;
```

```
ALTER TABLE member02  
DROP COLUMN age;
```

```
ALTER TABLE member02  
DROP COLUMN mobile;
```

```
DESC member02;
```

9.5 테이블의 모든 데이터 삭제하기

TRUNCATE 문은 테이블에 있는 데이터를 삭제할 때 사용하는 SQL 명령으로 DML의 DELETE 문과 동일한 동작을 한다. 차이점이 있다면 TRUNCATE 문을 사용해 데이터를 삭제하게 되면 자동으로 COMMIT 명령이 적용되기 때문에 ROLLBACK 명령으로 데이터를 복구 할 수 없다. 그러므로 매우 신중하게 TRUNCATE 명령을 사용해야 한다.

아래 명령을 실행하고 ROLLBACK 명령을 실행한 후 테이블의 데이터를 조회해 보자.

```
TRUNCATE TABLE [스키마명.]테이블명
```

```
-- 테이블의 모든 데이터를 지운다.
```

```
TRUNCATE TABLE member02;
```

```
-- rollback 명령을 실행한 후 조회해 본다.
```

```
rollback;
```

```
SELECT * FROM member02;
```

9.6 테이블 삭제하기

테이블을 삭제하려면 아래와 같이 DROP TABLE 문을 사용해 데이터베이스에서 삭제할 수 있다.

DROP TABLE 문은 DDL 문에 속하므로 이 명령으로 삭제한 테이블은 ROLLBACK 명령으로 복원할 수 없기 때문에 DROP 문은 보다 신중하게 사용해야 한다.

테이블이 제거되면 참조 무결성 제약조건을 제외하고 제거하는 테이블에 연관된 제약조건, 인덱스, 트리거 등이 자동으로 삭제된다. 참조 무결성 제약조건(외래키 제약조건)이 지정되어 있는 부모 테이블을 삭제하려면 CASCADE CONSTRAINTS 옵션을 지정하면 되는데 이 경우 자식 테이블에 지정된 외래키 제약조건(기본키와 UNIQUE 키를 참조하는 참조 무결성 제약조건)이 자동으로 삭제된다. 삭제하려는 테이블이 다른 테이블에서 참조되고 있는 컬럼을 가지고 있는데 CASCADE CONSTRAINTS 옵션을 지정하지 않고 DROP TABLE 명령을 실행하면 오류가 발생한다. CASCADE CONSTRAINTS 옵션을 지정하지 않고 부모 테이블을 삭제하려면 부모 테이블을 참조하고 있는 모든 자식 테이블을 먼저 삭제한 후에 부모 테이블을 삭제하면 된다.

```
DROP TABLE [스키마명.]테이블명 [CASCADE CONSTRAINTS]
```

```
-- 먼저 아래 쿼리를 실행해 제약조건을 확인하고 다음에 DROP 명령을 실행하자.  
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, SEARCH_CONDITION  
FROM user_constraints  
WHERE table_name='EMP01';
```

```
DROP TABLE dept01 CASCADE CONSTRAINTS;
```

```
-- CASCADE CONSTRAINTS 옵션을 지정해 부모 테이블을 삭제했기 때문에  
-- emp01_fk라는 외래키가 삭제된 것을 확인할 수 있을 것이다.  
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, SEARCH_CONDITION  
FROM user_constraints  
WHERE table_name='EMP01';
```

10. 오라클 객체

앞에서 오라클 데이터베이스의 대표적인 객체라고 할 수 있는 테이블의 생성과 변경에 대해 알아보았다.

오라클 데이터베이스는 데이터 저장과 관리를 위한 다양한 기능을 제공하기 위해 테이블 외에도 여러 가지 객체를 제공하고 있다. 오라클을 보다 전문적으로 사용하기 위해서는 테이블 외에 실무에서 많이 사용하는 오라클 객체에 대해서 학습할 필요가 있다. 그래서 이번 챕터에서는 오라클에서 많이 사용되는 객체인 뷰(View), 인덱스(Index), 시퀀스(Sequence)에 대해 알아 볼 것이다.

10.1 뷰(View)

뷰(View)는 오라클 객체의 하나로 실제 테이블이나 또 다른 뷰를 기준으로 생성되는 가상 테이블이다. 뷰에는 실제 테이블과 같이 데이터가 저장되는 것이 아니라 SQL 쿼리만 저장되며 뷰라는 이름에서 알 수 있듯이 SQL 쿼리를 이용해서 실제 테이블에 저장된 데이터를 들여다 볼 수 있는 기능을 제공하는 오라클 객체이다. 뷰를 사용하는 이유는 보안과 편리성 때문인데 실제 테이블의 데이터 중에서 다른 사용자가 보면 안 되는 데이터가 있을 수 있고 이런 데이터는 다른 사용자에게 공개하지 않고 필요한 데이터만 조회할 수 있도록 뷰를 생성해 사용한다. 예를 들면 emp 테이블에서 급여 정보와 같은 민감한 데이터는 꼭 필요한 담당자만 볼 수 있어야 하고 다른 사용자에게는 공개해서는 안 되는 데이터이다. 이럴 때 필요한 컬럼만 SQL 쿼리를 이용해 뷰로 생성해 놓고 뷰를 통해서 필요한 데이터만 조회할 수 있도록 할 수 있다. 또한 여러 테이블을 조인한 복잡하고 긴 쿼리를 자주 사용해야 한다면 뷰를 생성해 놓고 간단한 SELECT 문을 사용해 데이터를 조회할 수 있다. 뷰가 없다면 매번 복잡한 쿼리를 실행해서 테이블에서 데이터를 조회해야 하는데 이럴 때 단 한 번 복잡한 쿼리를 사용해 뷰를 생성해 놓고 필요할 때 마다 SELECT 하는 것으로 동일한 데이터를 조회할 수 있다.

▶ 뷰 생성하기

뷰는 테이블이나 또 다른 뷰를 조회하는 SELECT 쿼리를 사용해 뷰에서 볼 수 있는 데이터를 정의하며 뷰 또한 오라클 객체이므로 아래와 같이 CREATE 문을 사용해 생성한다.

```
CREATE [OR REPLACE] VIEW 뷰이름 AS
SELECT 문장
[ WITH CHECK OPTION [CONSTRAINT 제약조건] ]
[ WITH READ ONLY ]
```

아래와 같이 AS 다음에 오는 서브 쿼리에 조인조건 없이 1개의 테이블을 기준으로 생성하는 뷰를 단순 뷰(Simple View)라 한다.

```
CREATE VIEW emp_view AS
SELECT empno, ename, job, hiredate, deptno FROM emp;
```

아래와 같이 OR REPLACE 옵션을 사용하면 기존에 있던 뷰를 삭제하고 그 이름을 그대로 사용해 새로운 뷰를 생성할 수 있다.

```
CREATE OR REPLACE VIEW emp_view AS
SELECT empno 사번, ename 이름, job 직급, birthday 생일, hiredate 입사일
FROM emp WHERE deptno IN(10, 30);

SELECT * FROM emp_view;
```

아래 SQL 쿼리는 emp 테이블에서 사장을 제외한 직급별로 최저 급여를 받는 사원의 이름, 직급, 입사일, 급여, 부서명에 해당하는 컬럼만 가지는 뷰를 생성하는 쿼리 이다.
아래와 같이 여러 테이블을 조인해 생성한 뷰를 복합 뷰(Composite View)라 한다.

```
CREATE OR REPLACE VIEW emp_min_sal AS
SELECT e.ename 이름, e.job 직급, e.hiredate 입사일, e.sal 급여, d.dname 부서명
FROM emp e, dept d
WHERE e.deptno = d.deptno
AND (e.job, e.sal) IN(SELECT job, MIN(sal) FROM emp
GROUP BY job
HAVING MAX(sal) NOT IN (SELECT MAX(sal) FROM emp))
ORDER BY e.sal DESC;
```

뷰에는 SQL 문장 외에 아무런 데이터도 저장되지 않는다. 뷰를 생성한 후 SELECT 쿼리를 이용해 뷰를 조회하게 되면 그 때 뷰 내부에 저장된 서브 쿼리가 실행된다. 위와 같이 뷰를 생성하게 되면 오라클은 뷰에 대한 정보를 데이터 디렉터리에 저장하고 사용자가 뷰에 접근하게 되면 서브 쿼리인 SELECT 쿼리가 실행된다. 이 때 실제 테이블인 emp 테이블로 접근해 SELECT 절의 리스트에 기술한 컬럼의 값을 가져오게 된다. 또한 뷰에는 실제 데이터가 저장되지 않기 때문에 제약조건이나 인덱스를 생성할 수 없다.

사용자가 생성한 뷰에 대한 정보는 데이터 디렉터리(시스템이 사용하는 테이블)에 저장되어 있는 user_views 라는 시스템 뷰를 통해 알 수 있다. 뷰 또한 가상 테이블 이므로 아래와 같이 DESCRIBE 명령을 사용해 뷰를 이루고 있는 컬럼을 확인할 수 있다.

```
DESC user_views;
```

user_views의 컬럼 구성을 확인했으면 아래의 SQL 쿼리를 통해 뷰의 내용을 확인할 수 있다.
아래 쿼리는 뷰의 이름과 뷰에 저장된 SQL 문장의 크기 그리고 실제로 뷰에 저장된 SQL 문장을 확인하는 쿼리이다.

```
SELECT view_name, text_length, text FROM user_views;
```

▶ 뷰 삭제하기

더 이상 필요가 없는 뷰는 아래와 같이 DROP VIEW 명령을 사용해 삭제할 수 있다.

뷰는 가상 테이블 형태로 존재하는 오라클 객체로써 뷰에는 SQL 쿼리만 저장되기 때문에 해당 뷰가 삭제되었다고 해서 기존 테이블의 구조나 데이터에는 영향을 주지 않는다.

DROP VIEW 뷰이름

```
DROP VIEW emp_view;
```

▶ 뷰를 통한 데이터 변경

앞에서 알아본 것처럼 뷰에는 SQL 문장만 저장되어 있다. 이렇게 데이터도 없는 가상 테이블을 통해서 실제 데이터를 추가하고 삭제할 수 있을까? 결론부터 말하자면 우리가 앞에서 생성한 뷰는 데이터의 변경이 가능한 뷰들이다.

뷰는 테이블의 데이터를 읽을 수만 있는 Read Only View와 데이터를 변경할 수 있는 Updatable View가 있다. 뷰를 생성할 때 WITH READ ONLY 옵션을 지정하지 않은 뷰는 기본적으로 Updatable 뷰가 되지만 뷰를 통해 기존 테이블의 데이터를 변경하기 위해서는 여러 가지 조건을 만족해야 한다.

기존 테이블에 제약조건이 여러 개 설정되어 있다면 테이블에서 직접 데이터를 변경하는 것과 마찬가지로 제약조건 모두를 만족해야 한다. 하지만 뷰는 테이블에서 일부 컬럼만 조회해서 생성했기 때문에 테이블에 설정된 제약조건을 만족하지 못하는 경우가 있을 수 있는데 이럴 경우 테이블의 데이터를 변경할 수 없다. 또한 Updatable 뷰를 생성했다 하더라도 뷰를 생성하는 서브 쿼리에서 아래와 같은 방법으로 뷰를 생성한 경우 기존 테이블의 데이터를 변경할 수 없다.

뷰에서 데이터를 변경하기 위해서는 테이블에 설정된 제약조건을 가진 컬럼이 뷰를 생성할 때 모두 서브 쿼리에 존재해야 하며 아래의 방법으로 뷰를 생성하지 말아야 한다.

- UNION, UNION ALL을 사용한 경우
- DISTINCT를 사용한 경우
- 그룹함수(SUM, MAX 등)를 사용한 경우
- GROUP BY, ORDER BY, MODEL, CONNECT BY, START WITH 절을 사용한 경우
- SELECT 절의 리스트 항목으로 서브 쿼리를 사용한 경우
- SELECT 절의 리스트 항목에 의사컬럼을 사용한 경우, 의사컬럼(ROWNUM 등)은 테이블에 존재하는 실제 컬럼이 아니기 때문에 데이터를 수정할 수 없다.

10.2 시퀀스(SEQUENCE)

시퀀스는 연속적으로 증가 또는 감소하는 일련번호 형식의 수를 생성해 주는 오라클 객체이다.

시퀀스는 말 그대로 일정한 규칙에 의해 숫자를 증가 시키거나 감소시켜 일련번호를 만들어 주는 자동번호 생성기라 할 수 있다.

테이블에 데이터를 저장할 때 데이터의 순서가 필요한 경우가 종종 있는데 이때 오라클에서는 시퀀스가 생성해 주는 일련번호를 사용해 순서가 필요한 데이터를 저장할 수 있다.

예를 들면 게시 글을 저장할 때 시퀀스를 통해 게시 글의 번호를 부여한다든지 학생 정보를 저장할 때 학번을 부여할 수도 있다. 또한 테이블에서 하나의 행을 유일하게 구분하기 위한 기본 키를 생성하는데 시퀀스를 사용하기도 한다.

기본 키는 데이터가 테이블에 저장될 때 그 행을 유일하게 식별할 수 있어야 한다.

예전에는 주민번호와 같이 중복되지 않는 값을 가질 수 있는 컬럼을 기본 키로 사용하였으나 요즘은 주민번호를 기본 키로 사용하는 경우는 거의 없다. 또한 테이블에서 유일한 값을 가질 수 있는 컬럼이 없다면 어떻게 해야 할까? 물론 두 개 또는 세 개의 컬럼을 연결해 복합키로 기본 키를 지정할 수도 있지만 이것 보다는 특정 규칙에 의해 연속적으로 증가하거나 감소하는 숫자를 기본 키로 사용하는 것이 편리하다. 이렇게 연속적으로 증가하거나 감소하는 숫자를 기본 키로 사용하기 위해 시퀀스를 통해 생성된 숫자를 사용할 수도 있다.

▶ 시퀀스 생성하기

시퀀스는 앞서서도 언급했듯이 연속적인 번호를 만들어주는 오라클 객체이므로 CREATE 문을 사용해 아래와 같이 생성할 수 있다.

CRATE SEQUENCE 시퀀스이름

[INCREMENT BY n]	-> 시퀀스가 증가되는 단위 지정, 0은 지정할 수 없음
[START WITH n]	-> 시퀀스가 생성해 주는 최초 숫자 지정
[MINVALUE n]	-> 시퀀스의 최소 숫자 지정
[MAXVALUE n]	-> 시퀀스의 최대 숫자 지정
[CYCLE NOCYCLE]	-> 시퀀스 값을 순환으로 사용할지 지정
[CACHE n NOCACHE]	-> CACHE 옵션을 사용할지 지정

시퀀스가 증가되는 단위를 지정하는 INCREMENT BY에 양수를 지정하면 시퀀스가 수를 한 번 생성할 때마다 지정한 수만큼 증가하고 음수를 지정하면 지정한 수만큼 감소한다. 다시 말해 양수를 지정하면 시퀀스는 지정한 수만큼 오름차순으로 수를 생성하고 음수를 지정하면 지정한 수만큼 내림차순으로 수를 생성하게 된다. INCREMENT BY는 생략할 수 있으며 생략되면 기본 값은 1이 된다.

START WITH에는 시퀀스에서 생성할 번호의 시작 값을 지정하는데 이 또한 생략할 수 있으며 생략하면 START WITH의 기본 값은 1이 된다.

MINVALUE는 시퀀스에서 생성할 번호의 최솟값으로 시작 값(START WITH) 이하, 최댓값(MAX VALUE) 미만인 값으로 지정할 수 있다. MINVALUE 또한 생략할 수 있으며 이를 생략하면 시퀀스가 오름차순인 경우 1이 되며 내림차순인 경우 10^{-26} 으로 설정된다.

MAXVALUE는 시퀀스에서 생성할 번호의 최댓값으로 시작 값(START WITH) 이상, 최솟값(MIN VALUE) 초과인 값으로 지정할 수 있다. MAXVALUE 또한 생략할 수 있으며 이를 생략하면 시퀀스가 오름차순인 경우 10^{27} 되며 내림차순인 경우 -1로 설정된다.

당연한 얘기겠지만 START WITH는 MINVALUE와 MAXVALUE 사이에 있어야 하고 START WITH를 생략하게 되면 오름차순은 MINVALUE가 되고 내림차순은 MAXVALUE가 된다.

CYCLE 옵션은 시퀀스가 생성하는 값이 MAXVALUE 또는 MINVALUE에 도달했을 때 MINVALUE 지정한 값부터 다시 시작할지를 지정하는 옵션으로 기본 값은 NOCYCLE 이다.

CACHE 옵션을 사용하면 시퀀스를 생성하기 위해 미리 값을 할당해 놓기 때문에 시퀀스에 접근하는 시간이 좀 더 빠르다. 그래서 동시 접속자가 많은 경우에 CACHE 옵션을 사용하면 속도향상에 도움이 된다. NOCACHE 일 경우는 값을 미리 할당하지 않는다.

CACHE 옵션에 지정한 n 개의 일련번호를 메모리에 항상 생성시켜 놓고 시퀀스를 호출하면 메모리에서 번호를 할당하게 된다.

시퀀스 객체를 생성하고 시퀀스가 어떻게 동작되는지 알아보기 위해 먼저 아래의 SQL 쿼리를 사용해 시퀀스 생성과 관리에 사용할 테이블을 생성하고 그 아래에 있는 시퀀스 생성 쿼리를 이용해 시퀀스를 생성하자.

```
CREATE TABLE seq_member(  
  no NUMBER PRIMARY KEY,  
  name VARCHAR2(5 CHAR) NOT NULL,  
  age NUMBER(3) NOT NULL  
);
```

```
CREATE SEQUENCE seq_member_no  
  START WITH 1  
  INCREMENT BY 1  
  MAXVALUE 10000;
```

시퀀스가 생성되었으면 현재 시퀀스 번호가 어떻게 되는지 아래와 같이 조회해 보자.

아래 쿼리는 현재 시퀀스의 값을 읽어오는 쿼리로 CURRVAL은 현재 시퀀스 값을 가지고 있는 의사 컬럼(Pseudo Column) 이다.

```
SELECT seq_member_no.CURRVAL FROM dual;
```

위의 쿼리로 seq_member_no의 현재 시퀀스 값을 읽으려 했지만 다음과 같은 오류가 발생한다. 이 오류를 살펴보면 현재 세션에 시퀀스가 정의되지 않았다는 오류 같은데 우리는 분명히 시퀀스를 생성하고 이 시퀀스의 현재 값을 읽으려 했건만 오류가 발생하고 말았다.

그런데 오류 메시지의 Cause 부분과 Action 부분을 잘 살펴보니 CURRVAL을 선택하기 전에 NEXTVAL을 먼저 선택하라는 내용이 보이는 것 같다.

```
ORA-08002: sequence SEQ_MEMBER_NO.CURRVAL is not yet defined in this session  
08002. 00000 - "sequence %s.CURRVAL is not yet defined in this session"
```

```
*Cause:      sequence CURRVAL has been selected before sequence NEXTVAL
```


***Action:** select NEXTVAL from the sequence before selecting CURRVAL

그래서 위에서 생성한 테이블에 데이터를 추가하기 위해 아래 SQL 쿼리를 사용해 한 행의 데이터를 추가 했다. 이 쿼리를 보면 PRIMARY KEY인 no 컬럼에 데이터를 추가하기 위해 NEXTVAL을 사용했다. NEXTVAL은 현재 시퀀스의 다음 값을 읽어오는 의사컬럼 이다.

그리고 CURRVAL을 조회해 보니 이번에는 오류 없이 현재 시퀀스 값이 조회 되었다.

앞에서와 같이 시퀀스를 생성하고 최초로 NEXTVAL을 호출하지 않으면 시퀀스를 제대로 사용할 수 없다는 것을 확인할 수 있었다. NEXTVAL은 한 번 호출될 때마다 INCREMENT BY에 지정한 만큼 무조건 값을 증가해 새로운 시퀀스 번호를 생성해 준다.

```
INSERT INTO seq_member VALUES(seq_member_no.NEXTVAL, '홍길동', 15);
```

```
INSERT INTO seq_member VALUES(seq_member_no.NEXTVAL, '임꺽정', 27);
```

```
INSERT INTO seq_member VALUES(seq_member_no.NEXTVAL, '장길산', 19);
```

시퀀스를 생성하고 CURRVAL과 NEXTVAL을 사용해 편리하게 일련번호를 얻을 수 있지만 모든 SQL 쿼리에서 사용할 수 있는 것은 아니다. 아래는 두 의사컬럼을 사용할 수 있는 경우와 사용할 수 없는 경우를 나타낸 것이다.

■ CURRVAL과 NEXTVAL을 사용할 수 있는 경우

- 서브 쿼리가 아닌 SELECT 절
- INSERT 문의 SELECT 절
- INSERT 문의 VALUES 절
- UPDATE 문의 SET 절

■ CURRVAL과 NEXTVAL을 사용할 수 없는 경우

- VIEW의 SELECT 절
- DISTINCT 키워드가 있는 SELECT 절
- GROUP BY, HAVING, ORDER BY 절이 있는 SELECT 절
- SELECT, DELETE, UPDATE 문의 서브 쿼리
- CREATE TABLE, ALTER TABLE 문의 DEFAULT 값

▶ 시퀀스 수정과 삭제하기

시퀀스도 오라클 데이터베이스 객체이므로 시퀀스를 수정하려면 아래와 같이 ALTER 명령을 사용해 수정할 수 있다. 참고로 시퀀스를 수정할 때는 START WITH 옵션은 수정할 수 없다.

```
ALTER SEQUENCE seq_member1_no  
MAXVALUE 200  
CACHE 5;
```

시퀀스를 삭제할 때는 다른 오라클 객체와 마찬가지로 DROP 명령을 사용하면 된다.
아래 SQL 쿼리는 시퀀스 seq_member1_no를 삭제하는 쿼리이다.

```
DROP SEQUENCE seq_member1_no;
```

10.3 인덱스(INDEX)

우리가 책을 읽을 때 특정 주제나 단어를 빠르게 찾기 위해 목차에서 찾아 해당 페이지로 이동하거나 아니면 일반적으로 도서의 뒤쪽에 작성되어 있는 찾아보기(색인, 인덱스) 목록에서 필요한 단어를 찾아 해당 페이지로 이동한다. 특정 단어를 기준으로 찾을 때는 아마도 목차 보다는 색인(Index)을 더 많이 이용할 것이다. 우리가 찾으려고 하는 내용을 보다 빠르게 찾기 위해 목차나 색인(Index)을 검색해 필요한 정보를 찾는 것처럼 오라클에서도 테이블에 저장된 데이터를 빨리 찾기 위해서 인덱스를 사용하게 된다. 대부분의 책 뒤쪽에 찾아보기 목록이 있는 것처럼 오라클에서도 테이블에 존재하는 인덱스 컬럼에 대한 인덱스 정보가 별도로 저장되어 관리된다. 오라클에서 데이터를 검색할 때 이 인덱스 정보를 이용해 테이블을 검색하게 된다. 이처럼 오라클에서 인덱스는 데이터를 검색할 때 SQL 명령의 처리 속도를 향상시키기 위해 컬럼에 대해 생성하는 오라클 데이터베이스 객체이다.

▶ 인덱스 생성과 삭제

인덱스는 테이블에 있는 컬럼을 지정해 만들 수 있지만 CREATE TABLE 문을 통해서 인덱스를 생성할 수는 없다. 인덱스는 앞에서 알아본 뷰, 시퀀스와 같은 오라클 데이터베이스 객체로 아래와 같이 CREATE 문을 사용해 생성할 수 있고 DROP 문을 사용해 인덱스를 삭제할 수 있다.

```
CREATE [ UNIQUE ] INDEX 인덱스 이름  
ON 테이블 이름 (컬럼1, 컬럼2, ... );
```

```
DROP INDEX 인덱스 이름;
```

사용자가 생성한 인덱스를 수동 인덱스라 하며 PRIMARY KEY나 UNIQUE 제약조건을 컬럼에 지정하면 오라클이 그 컬럼에 자동으로 생성해 주는 인덱스를 자동 인덱스라 한다.

컬럼에 PRIMARY KEY나 UNIQUE 제약조건이 지정되면 그 컬럼에 대한 인덱스를 오라클이 생성해 주는데 이렇게 오라클이 자동으로 생성해 주는 인덱스는 제약조건의 이름과 동일하다.

또한 인덱스를 구성하는 컬럼이 한 개인 경우 단일 인덱스(Single Index)라 하며 두 개 이상의 컬럼으로 구성되면 복합 인덱스(Composite Index)라 한다. 그리고 중복 값을 허용하지 않는 경우 UNIQUE 인덱스라고 하며 중복 값을 허용할 경우 NON UNIQUE 인덱스라 한다. 이외에도 인덱스의

구조나 성격에 따라서 여러 가지 종류의 인덱스로 구분된다.

인덱스의 종류는 꼭 한 가지 종류에 해당하는 것이 아니라 자동 인덱스이면서 복합 인덱스가 될 수도 있고 수동 인덱스이면서 단일 인덱스 이고 UNIQUE 인덱스가 될 수도 있다.

인덱스에 대해 알아보기 전에 아래의 SQL 쿼리를 사용해 테스트할 테이블을 만들어 emp 테이블로부터 데이터를 추가해 보자.

먼저 index_emp 테이블을 생성하고 그 테이블의 PRIMARY KEY로 사용할 번호를 시퀀스로부터 할당 받아 emp 테이블의 데이터와 함께 index_emp 테이블에 추가 할 것이다.

```
CREATE TABLE index_emp (  
    empno NUMBER,  
    ename VARCHAR2(50 CHAR),  
    hiredate DATE,  
    sal NUMBER,  
    deptno NUMBER  
);
```

```
CREATE SEQUENCE seq_indexemp_empno  
    MINVALUE 1  
    START WITH 1;
```

```
-- 아래는 emp 테이블의 25행의 데이터를 10만 번 반복해 index_emp 테이블에 추가하는  
-- PL/SQL 블록으로 250만 행의 데이터가 추가되므로 시스템에 따라서 20초 이상이 소요된다.  
BEGIN  
    FOR i in 1.. 50000 LOOP  
        INSERT INTO index_emp SELECT seq_indexemp_empno.NEXTVAL,  
            ename, hiredate, sal, deptno FROM emp;  
    END LOOP  
    COMMIT;  
END;  
/
```

▶ 인덱스와 검색 속도 비교

index_emp 테이블에서 사원 이름으로 조회할 때 시간이 얼마나 걸리는 지를 테스트하기 위해 아래 SQL 쿼리를 이용해 한 행의 데이터를 추가 하고 그 아래에 있는 SELECT 쿼리를 사용해 이름이 감사봉 사원을 조회하고 결과가 나타나기 까지 시간이 얼마나 걸렸는지 확인해 보자.

```
INSERT INTO index_emp VALUES(  
    seq_indexemp_empno.NEXTVAL, '감사봉', '2011-12-01', 350, 40);
```

```
SELECT * FROM index_emp WHERE ename = '감사봉';
```

아래는 index_emp 테이블의 ename 컬럼에 인덱스를 생성하는 쿼리이다. 아래 쿼리 문을 사용해 인덱스를 생성한 후 위의 SELECT 쿼리를 사용해 조회한 시간이 얼마나 걸리는지 확인해 보자.

```
-- 250만 행에 대해서 인덱스를 생성하기 때문에 시스템에 따라서 5초 이상 걸릴 수 있다.  
CREATE INDEX ind_indexemp_ename  
ON index_emp(ename);
```

10.4 데이터 사전(Data Dictionary)

오라클 데이터베이스에서 테이블은 크게 두 가지 유형으로 나눌 수 있는데 그 중 하나가 사용자가 직접 생성해 데이터를 저장하고 관리하는데 사용하는 사용자 테이블(User Table)이 있으며 또 다른 하나는 오라클 시스템이 시스템 관리를 위해 필요한 정보를 저장하고 관리하는데 사용하는 시스템 테이블이 있다.

시스템 테이블은 오라클 데이터베이스를 구성하여 자원을 효율적으로 관리하고 운영하는데 필요한 모든 정보를 저장하고 있는 특수한 테이블로써 데이터 사전(Data Dictionary)이라고 부르며 이 데이터 사전은 데이터베이스가 생성되는 시점에 오라클 내부에서 자동으로 만들어 진다.

데이터 사전에는 오라클 데이터베이스 운영에 필요한 사용자, 시스템 권한, 객체, 메모리 설정 등에 대한 중요한 데이터가 저장되어 관리되기 때문에 혹시라도 데이터 사전에 문제가 생기면 오라클 성능에 치명적 문제가 생기거나 아예 오라클 데이터베이스를 사용할 수 없게 될지도 모른다. 그러므로 일반 사용자는 데이터 사전에 직접 접속할 수 없고 데이터 사전 뷰(Data Dictionary View, 시스템 뷰라고도 함)를 통해서 데이터를 조회할 수 있다.

데이터 사전의 종류는 다음과 같은 형식의 이름으로 되어 있으며 일반 사용자는 조회할 수 없는 것도 있다. 아래 내용에서 XXX는 객체의 종류를 의미한다.

1. USER_XXX : 현재 데이터베이스에 접속한 사용자가 소유한 객체 정보를 조회할 수 있음
2. ALL_XXX : 현재 데이터베이스에 접속한 사용자가 소유한 객체 또는 다른 사용자가 소유한 객체 중 사용 허가를 받은 객체 , 즉 현재 접속한 사용자가 사용할 수 있는 모든 객체 정보를 조회할 수 있음
3. DBA_XXX : 데이터베이스 관리를 위한 정보(DBA 권한을 가진 SYSTEM, SYS만 조회 가능)
4. V\$XXX : 데이터베이스 성능관련 정보(DBA 권한을 가진 SYSTEM, SYS만 조회 가능)

▶ 현재 접속한 사용자가 생성한 데이터베이스 객체 조회

-- 현재 접속한 사용자가 생성한 테이블, 뷰, 인덱스, 시퀀스, 제약조건 조회

```
SELECT * FROM user_tables;  
SELECT * FROM user_views;  
SELECT * FROM user_indexes;  
SELECT * FROM user_sequences;
```

```
SELECT * FROM user_constraints;
```

▶ 현재 접속한 사용자가 사용할 수 있는 모든 데이터베이스 객체 조회

-- 현재 접속한 사용자가 사용할 수 있는 테이블, 뷰, 인덱스, 시퀀스, 제약조건 조회

```
SELECT * FROM all_tables;
```

```
SELECT * FROM all_views;
```

```
SELECT * FROM all_indexes;
```

```
SELECT * FROM all_sequences;
```

```
SELECT * FROM all_constraints;
```

▶ 데이터베이스 관리를 위한 정보 조회

-- DB 사용자, DBA 테이블, 뷰, 인덱스 시퀀스, 제약조건 조회

```
SELECT * FROM dba_users;
```

```
SELECT * FROM dba_tables;
```

```
SELECT * FROM dba_views;
```

```
SELECT * FROM dba_indexes;
```

```
SELECT * FROM dba_sequences;
```

```
SELECT * FROM dba_constraints;
```

▶ 데이터베이스 성능관련 정보 조회

-- V\$로 시작하는 Dynamic Performance table의 수는 100개도 넘음

-- 현재 데이터베이스 내의 lock이 걸린 object와 그 object를 access 하려는 session id 조회

```
SELECT * FROM v$access;
```

-- 모든 online 데이터파일의 backup 상태 조회

```
SELECT * FROM v$backup;
```

-- 현재 데이터베이스 인스턴스 상태 조회

```
SELECT * FROM v$instance;
```

-- 현재 OPEN된 세션 정보 조회

```
SELECT * FROM v$session;
```

-- 활동 중인 세션이 대기하고 있는 자원 또는 이벤트 조회

```
SELECT * FROM v$session_wait;
```

11. 사용자 계정과 권한 관리

데이터베이스 관리 시스템(DBMS)은 기업에서 필요한 데이터를 통합하여 저장하고 관리하는 시스템으로써 한 명의 사용자가 관리하기에는 너무나 방대하고 복잡한 구조로 이루어져 있다. 또한 기업에서 의사결정에 필요한 매우 중요한 데이터도 데이터베이스에 저장되고 관리되기 때문에 데이터에 접근할 수 있는 사용자 권한에도 많은 신경을 써야 한다. 그러므로 업무의 분할과 효율성 그리고 보안 등에 대한 여러 사항을 고려해 사용자를 나누고 데이터에 접근할 수 있는 적절한 권한을 부여해야 한다.

권한은 사용자가 특정 테이블에 접근할 수 있도록 하거나 해당 테이블에서만 데이터를 조작할 수 있도록 제한하는 것으로 데이터베이스 보안을 위해 오라클에서 사용하는 권한은 크게 시스템 권한(System Privileges)과 객체 권한(Object Privileges)으로 나눌 수 있다.

시스템 권한은 사용자 생성과 삭제, 데이터베이스에 접속하기 위한 세션 생성 그리고 테이블, 뷰, 인덱스 등과 같은 데이터베이스 객체의 생성과 삭제에 대한 권한으로 주로 DBA에 의해 부여되는 권한이다. 객체 권한은 오라클 객체를 조작할 수 있는 권한으로 테이블, 뷰를 조회하거나 데이터를 수정, 삭제하는 등의 작업과 시퀀스, 프로시저 함수 등을 실행할 수 있는 권한이다.

오라클 데이터베이스는 테이블, 뷰, 인덱스 등의 여러 객체가 사용자별로 생성되기 때문에 업무에 맞게 사용자를 생성하고 권한을 부여해 업무별로 데이터를 저장하고 관리할 수 있다.

데이터베이스에서는 데이터 간의 관계 및 데이터 구조와 제약조건 등에 대한 데이터를 저장하고 관리하기 위해서 정의하는 데이터베이스 구조 범위를 스키마(Schema)라고 하며 이 스키마를 통해 그룹 단위로 관리한다.

오라클에서는 사용자 계정과 스키마를 별도로 구별하지 않고 사용하며 한 사용자가 생성한 테이블, 뷰, 제약조건 등의 모든 데이터베이스 구조를 스키마라고 부르기도 한다. 예를 들면 우리가 사용하는 HR 스키마는 HR 사용자 계정을 포함해서 HR 계정으로 생성한 모든 데이터베이스 객체에 대한 구조를 포괄적으로 HR 스키마라고 한다.

11.1 오라클 시스템 권한

시스템 권한을 가진 데이터베이스 관리자(DBA - SYS, SYSTEM)가 사용자에게 권한을 부여할 때 사용하는 GRANT 문은 SQL 명령 중에서 DCL(Data Control Language) 명령에 해당한다. 시스템 권한을 가진 DBA는 표 11-1과 같이 SESSION, TABLE, VIEW, SEQUENCE, PROCEDURE 등과 같은 데이터베이스 객체를 생성하고 관리할 수 있는 시스템 권한과 표 11-2의 내용과 같이 TABLE, VIEW, SEQUENCE 등에 저장된 데이터를 추가, 수정, 삭제할 수 있는 객체 권한을 사용자에게 부여할 수 있다. 참고로 Oracle 11g R2 버전의 시스템 권한은 약 200여개가 넘는다.

구 분	시스템 권한	설 명
SESSION	CREATE SESSION	데이터베이스 접속 세션 생성 권한
TABLE	CREATE TABLE	사용자 스키마에 테이블 생성 권한
	CREATE ANY TABLE	모든 스키마에 테이블 생성 권한
	DROP ANY TABLE	모든 스키마의 테이블 삭제 권한

구 분	시스템 권한	설 명
VIEW	CREATE VIEW	사용자 스키마에 뷰 생성 권한
	CREATE ANY VIEW	모든 스키마에 뷰 생성 권한
	DROP ANY VIEW	모든 스키마의 뷰 삭제 권한
SEQUENCE	CREATE SEQUENCE	사용자 스키마에 시퀀스 생성 권한
	CREATE ANY SEQUENCE	모든 스키마에 시퀀스 생성 권한
	DROP ANY SEQUENCE	모든 스키마의 시퀀스 삭제 권한
SYNONYM	CREATE SYNONYM	사용자 스키마에 동의어 생성 권한
	CREATE ANY SYNONYM	모든 스키마에 동의어 생성 권한
	DROP ANY SYNONYM	모든 스키마의 동의어 삭제 권한
PROCEDURE	CREATE PROCEDURE	사용자 스키마에 함수 생성 권한
	CREATE ANY PROCEDURE	모든 스키마에 함수 생성 권한
	DROP ANY PROCEDURE	모든 스키마의 함수 생성 권한

표 11-1 Oracle DBMS의 시스템 권한

객 체	부여할 수 있는 권한
TABLE	SELECT, INSERT, UPDATE, DELETE, ALTER, DEBUG, ALL 등
VIEW	SELECT, INSERT, UPDATE, DELETE, UNDER 등
SEQUENCE	SELECT, ALTER
INDEX	EXECUTE
Package, Procedure, Function	EXECUTE, DEBUG
Directory	READ, WRITE
Library	EXECUTE
Operator	EXECUTE

표 11-2 Oracle DBMS의 객체 권한

▶ 사용자 계정 생성하기

먼저 사용자 계정과 관리 실습에 사용할 test1 이라는 사용자를 추가해 보자. 사용자 계정 생성은 위에서 학습했던 DDL(Data Definition Language) 문의 CREATE 문을 이용해 사용자를 생성할 수 있다. HR 계정은 사용자 계정을 추가할 수 있는 권한이 없으므로 시스템 권한을 가진 SYS로 접속하고 아래 SQL 문을 사용해 계정을 생성해야 한다.

아래는 test1 이라는 계정을 생성하고 비밀번호를 12345678로 설정하는 SQL문 이다.

```
CREATE USER test1 IDENTIFIED BY 12345678;
```

▶ 사용자에게 시스템 권한 부여하기

위와 같이 계정을 생성하고 test1으로 접속을 하려고 하면 아래와 같은 오류가 발생한다.

오류 내용을 살펴보면 “test1 계정이 오라클에 접속하기 위한 세션을 생성하는 권한이 없어 로그인 이 거부 되었다”라는 내용의 메시지인 것 같다. 결론부터 말하자면 계정을 새로 생성하고 그 계정이 오라클에 접속할 수 있는 세션 생성 권한을 가지고 있어야 오라클에 접속할 수 있다.

위에서 CREATE 문으로 사용자 계정만 생성했을 뿐 그 계정에 아무런 권한을 주지 않았기 때문에 발생한 오류 이다.

ERROR:

ORA-01045: user TEST1 lacks CREATE SESSION privilege; logon denied

Warning: You are no longer connected to ORACLE.

■ 오라클 접속 세션 생성 권한 부여하기

SYS 계정으로 접속해 아래와 같이 GRANT 문을 실행해 세션 생성 권한과 테이블 생성 권한을 부여한 후 test1 계정으로 접속하여 테이블을 생성해 보자.

```
GRANT CREATE SESSION TO test1;
```

■ 테이블 생성 권한 부여하기

test1 계정에 테이블을 생성할 수 있는 권한을 아래 SQL 문을 통해 부여하고 test1 계정으로 접속해 테이블을 생성하고 데이터를 추가해 보자.

```
GRANT CREATE TABLE TO test1;
```

test1 계정에 오라클에 접속할 수 있는 세션 생성 권한과 테이블 생성 권한을 부여했으므로 test1 계정으로 접속해 아래 SQL 문으로 테이블을 생성하고 이 테이블에 3명의 회원 정보를 추가하고 테이블의 내용을 조회해 보자.

```
CREATE TABLE member(no NUMBER, name VARCHAR2(10));
```

```
INSERT INTO member VALUES(1001, '오라클');
```

```
INSERT INTO member VALUES(1002, '어머나');
```

```
INSERT INTO member VALUES(1003, '홍길동');
```

만약 위에서 테이블을 생성할 때 다음과 같은 오류가 발생할 수 있는데 이는 테이블 스페이스에 대한 권한이 없기 때문이다.

ORA-01950: no privileges on tablespace 'SYSTEM'

01950. 00000 - "no privileges on tablespace '%s'"

***Cause:** User does not have privileges to allocate an extent in the specified tablespace.

***Action:** Grant the user the appropriate system privileges or grant the user space resource on the tablespace.

이런 오류가 발생하면 dba 권한을 가진 sys로 접속해 아래의 SQL 명령을 실행해서 테이블 스페이스를 조회하고 TEST1 계정에 SYSTEM 테이블 스페이스에 대한 권한을 부여하면 TEST1 계정에서 새로운 테이블을 생성할 수 있게 된다.

-- 테이블 스페이스 조회

```
SELECT * FROM dba_tablespaces;
```

-- TEST1 계정에 SYSTEM 테이블스페이스에 대한 권한 부여

```
ALTER USER test1 QUOTA UNLIMITED ON SYSTEM;
```

▶ 여러 시스템 권한 부여하기

DBA인 SYS 계정으로 접속해 아래 GRANT 문을 실행해 보자.

이 SQL 명령은 test1 계정에 여러 가지 시스템 권한을 한 번에 부여하는 예이다.

```
GRANT CREATE SESSION, CREATE TABLE, CREATE VIEW TO test1;
```

또한 아래와 같이 “WITH ADMIN OPTION”을 지정해 시스템 권한을 부여하게 되면 test1 계정은 자신이 DBA가 아니라 하더라도 자신이 부여받은 시스템 권한을 다른 사용자에게 부여 할 수 있게 된다.

```
GRANT CREATE USER, CREATE SESSION, CREATE TABLE, CREATE VIEW  
TO test1 WITH ADMIN OPTION;
```

▶ 특정 사용자에게 객체 권한 부여하기

바로 위에서 시스템 권한을 부여받은 test1 계정으로 접속해 아래와 같이 새로운 사용자를 생성하고 이 계정에 오라클에 접속할 수 있는 시스템 권한과 member 테이블에 접근할 수 있는 객체 권한을 부여하고 테스트 해 보자.

■ 새로운 사용자 계정 생성하기

```
CREATE USER test2 IDENTIFIED BY 12345678;
```

- test1계정에서 test2 계정에 오라클 접속 권한과 테이블 생성 권한 부여하기

GRANT CREATE SESSION, CREATE TABLE to test2;

- test2 계정에 test1의 member 테이블에 접근할 수 있는 권한 부여하기

test1 계정으로 접속해 아래 GRANT 문을 실행한 후 test2 계정으로 접속해 test1 스키마의 member 테이블의 데이터를 조회하고 추가, 수정, 삭제하는 SQL 명령을 실행해 보자.

GRANT SELECT, INSERT, UPDATE, DELETE ON member TO test2;

- test1 계정에 HR의 emp 테이블에 접근할 수 있는 권한 부여하기

HR 계정으로 접속해 emp 테이블에 접근할 수 있도록 아래의 GRANT 문을 실행하자.
아래와 같이 “WITH GRANT OPTION”을 지정해 객체 권한을 부여하게 되면 test1이 emp 테이블 객체의 소유자가 아니더라도 다른 사용자에게 자신이 부여받은 emp 테이블에 접근할 수 있는 권한을 부여할 수 있다.

GRANT SELECT, INSERT, UPDATE, DELETE ON emp TO test1
WITH GRANT OPTION;

- HR 계정으로부터 부여받은 권한을 test2 계정에 부여하기

이번에는 test1 계정으로 접속해 HR 계정으로부터 부여 받은 권한을 test2 계정에 부여하고 테스트 해 보자.

GRANT SELECT, INSERT, UPDATE, DELETE ON hr.emp TO test2;

위의 GRANT 문을 실행하고 test2 계정으로 접속해 HR 계정의 emp 테이블의 데이터를 조회해 보자.

▶ 사용자 권한 회수하기

부여한 권한을 회수할 때는 REVOKE 명령을 사용해 다른 사용자에게 부여한 시스템 권한이나 객체에 대한 권한을 회수할 수 있다.

아래는 test1에게 부여한 CREATE USER 시스템 권한을 회수하는 SQL 명령이다.

REVOKE CREATE USER FROM test1;

다음은 test1 사용자에게 부여된 여러 시스템 권한을 한 번에 회수하는 SQL 명령이다.

REVOKE CREATE USER, CREATE SESSION, CREATE TABLE, CREATE VIEW FROM test1;

이번에는 사용자에게 부여된 객체 권한을 회수하는 방법에 대해서 알아보자.

먼저 test1 계정으로 접속해 아래의 SQL 문을 사용해 자신에게 부여된 객체 권한을 조회해 보자.

그러면 HR 계정에서 test1 계정에게 부여한 emp 테이블의 SELECT, INSERT, UPDATE, DELETE 권한이 적용되어 있는 것을 확인할 수 있을 것이다. 아래 SQL 명령은 현재 접속한 사용자가 부여받은 권한을 조회할 수 있는 쿼리 이다.

SELECT * FROM user_tab_privs_recd;

이번에는 HR 계정으로 접속해 test1에게 부여했던 emp 테이블의 SELECT 권한을 아래의 SQL 문을 사용해 회수하자.

REVOKE SELECT ON emp FROM test1;

그리고 아래 SQL 문을 사용해 HR 계정에서 test1에게 부여했던 권한이 제대로 회수 되었는지 아래 SQL 명령을 사용해 조회해 보자. 아래의 SQL 명령은 현재 접속한 사용자가 다른 사용자에게 부여한 권한을 조회할 수 있는 쿼리 이다.

SELECT * FROM user_tab_privs_made;

11.2 롤(Role)을 이용한 권한 관리

앞에서 우리는 여러 개의 시스템 권한을 나열해 사용자 계정에 부여하는 방법에 대해 알아보았다. 이렇게 각각의 시스템 권한을 하나하나 체크해 사용자에게 부여하는 작업은 번거롭다는 생각이 든다. 그래서 오라클에서는 사용자 권한을 간편하게 부여하는 방법을 제공하고 있는데 이것이 바로 롤(Role) 이다. 롤은 사용자와 권한을 효과적으로 관리하기 위해 여러 가지 권한을 묶어 놓은 시스템 권한의 묶음 이다.

오라클을 설치하면 기본적으로 CONNECT ROLE, RESOURCE ROLE, DBA ROLE과 같은 기본적인 롤이 설치된다.

■ CONNECT ROLE

사용자가 데이터베이스에 접속하기 위한 세션, 그리고 테이블, 뷰, 시퀀스, 동의어 등을 생성할

수 있는 기본적인 시스템 권한 8가지를 묶어 놓은 롤이다.

■ RESOURCE ROLE

사용자가 데이터베이스 객체인 테이블, 뷰, 인덱스, 클러스터, 트리거, 프로시저를 생성할 수 있도록 시스템 권한을 묶어 놓은 롤이다.

■ DBA ROLE

여러 사용자가 소유한 데이터베이스 객체를 관리하고 사용자를 생성하고 변경, 제거할 수 있도록 모든 권한을 묶어 놓은 롤이다. 다시 말해 DBA ROLE은 데이터베이스 관리에 필요한 모든 시스템 권한을 하나로 묶어 강력한 권한을 부여할 수 있는 롤이다.

SYS 계정으로 접속해 아래 GRANT 문을 실행하여 test2 계정에 CONNECT, RESOURCE 롤을 부여하고 이 롤이 제대로 부여되었는지 테스트 해 보자.

■ test2 계정에 롤을 이용해 시스템 권한 부여하기

```
GRANT CONNECT, RESOURCE TO test2;
```

■ test2 계정에 부여된 롤 조회하기

test2 계정으로 접속해 아래 SQL 명령을 실행하여 test2에 부여된 롤을 조회해 보자.

```
SELECT * FROM USER_ROLE_PRIVS;
```

▶ 사용자 롤(Role) 정의

CREATE 문을 통해서 아래와 같이 사용자 롤을 생성하고 이 롤에 권한을 부여한 후 다른 사용자에게 롤을 부여할 수도 있고 DROP 문을 통해 사용자 롤을 삭제할 수 있다.

```
CREATE ROLE TEST2_ROLE;  
GRANT CREATE SESSION, CREATE TABLE TO TEST2_ROLE;  
GRANT TEST2_ROLE TO test2;
```

```
DROP ROLE TEST2_ROLE;
```

▶ 롤(Role) 회수하기

롤 회수는 아래와 같이 REVOKE 문을 사용해 회수할 수 있다. SYS 계정으로 접속해 아래 SQL 명령을 실행하고 test2 계정으로 접속해 롤이 제대로 회수되었는지 조회해 보자.

```
REVOKE TEST2_ROLE FROM test2;  
REVOKE RESOURCE FROM test2;  
REVOKE RESOURCE, CONNECT FROM test2;
```

12. 데이터베이스 모델링

데이터베이스 모델링이란 현실 세계의 업무적인 프로세스에 존재하는 수많은 데이터를 꼭 필요한 데이터만 선별하여 컴퓨터 세계의 물리적인 데이터로 저장하기 위한 일련의 과정을 말한다.

데이터베이스 모델링은 일반적으로 아래와 같이 5단계를 거쳐 완성하게 된다.

❶ 요구사항 분석

현실 세계에서 업무에 필요한 모든 요구사항을 분석하는 단계로 이 단계에서 요구사항 명세서를 작성한다.

❷ 개념적 데이터 모델링

요구분석 단계에서 작성된 요구사항 명세서를 토대로 실제 업무에 필요한 개체(Entity)를 도출하고 각 개체의 속성(Attribute)과 개체 간의 관계 정의를 통해 ERD(Entity Relationship Diagram)를 작성한다.

❸ 논리적 데이터 모델링

개념적 데이터 모델링에서 작성한 ERD를 바탕으로 맵핑 룰(Mapping Rule)을 적용해 관계형 데이터베이스 모델에 따라 스키마를 설계하고 정규화를 수행해 특정 DBMS에 맞는 스키마를 설계한다.

❹ 물리적 데이터 모델링

논리적 데이터 모델링에서 설계된 스키마의 세부적인 사항을 정의한다. 다시 말해 특정 DBMS에 맞는 각 컬럼의 데이터 타입과 크기 그리고 제약조건을 정의한다. 이 단계에서 역정규화를 수행하게 된다.

❺ 데이터베이스 구현

물리적 모델링을 통해 완성된 스키마를 DDL(Data Definition Language) 언어를 사용해 실제 데이터베이스 객체로 생성한다.

12.1 요구사항 분석

요구사항 분석(Requirements Analysis) 단계는 고객이 무엇을 원하는지 정확히 파악하기 위해서 관련 분야에 대한 기본적인 지식을 갖추고 있어야 하며 업무 자체에 초점을 두고 업무 프로세스를 파악해야 한다.

요구사항을 제대로 분석하기 위해서는 우선 고객이 업무에 사용하는 문서(서류, 거래명세표, 보고서 등)를 바탕으로 기존에 데이터로 관리되어지는 항목을 정확하게 파악할 수 있어야 하며 담당자와 인터뷰를 통해 세부적인 업무프로세스 또한 정확하게 파악해야 한다.

이 단계에서는 어떤 DBMS를 사용해야 할지, 스키마를 어떻게 구성할지는 고려하지 않는다. 단지 업무가 어떤 프로세스에 의해서 실행되는지 정확하게 파악할 수 있어야 한다.

고객의 요구사항을 분석하고 나면 다음과 같은 요구사항 명세서(정의서)를 결과물로 얻을 수 있어야 한다.

스넥을 유통하는 A사는 30명의 사원이 관리부, 영업부, 경리부에 소속되어 근무하고 있다.
 사원관리를 위해 사번, 이름, 직책, 입사일, 생일, 급여 정보가 필요하다.
 각 사원은 하나의 부서에만 소속되어야 하며 각 부서는 부서번호, 부서명을 가진다.
 상품의 입고와 출고 데이터를 관리할 수 있어야 하며 상품 판매에 대한 정보도 정확하게 관리할 수 있어야 한다.

12.2 개념적 데이터 모델링

데이터베이스 모델링에서 가장 먼저 해야 할 일이 바로 고객이 필요로 하는 데이터가 무엇인지 정확하게 파악해 현실 세계의 어떤 데이터를 데이터베이스화 할 것인지를 결정하는 일이다.

데이터베이스에 저장해야 할 데이터는 정확한 업무 분석과 고객의 요구사항을 분석해 얻어지는 데이터이며 이렇게 현실세계의 정보를 사람이 이해할 수 있도록 명확한 형태로 표현하는 단계를 개념적 데이터 모델링(Conceptual Data Modeling)이라고 한다.

▶ 개체 관계 모델(Entity Relationship Model)

개념적 데이터 모델링에서 가장 널리 사용되는 객체 관계 모델은 현실 세계에 존재하는 데이터와 그 데이터들 간의 관계를 사람이 쉽게 이해할 수 있도록 명확하게 표현하기 위해 사용되는 모델이다. 객체 관계 모델은 1976년 Peter Chen에 의해 제안된 것으로 개체 타입(Entity Type)과 관계 타입(Relationship Type)을 기본 개념으로 현실 세계의 어떤 데이터를 데이터베이스에 저장해야 하는지를 표현하기 위해 사용하는 모델이다.

개체 관계 모델은 어떤 DBMS를 사용할지를 고려하지 않고 단지 데이터를 속성(Attribute)으로 구성된 개체(Entity)와 개체들 간의 관계를 알기 쉽게 표현하기 위해 표 12-1과 같이 약속된 도형을 사용해 그림 12-1과 같은 ERD(Entity Relationship Diagram)를 작성한다.




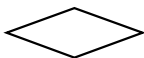


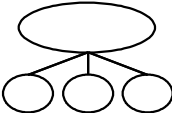

E-RD 표기법	의 미	E-RD 표기법	의 미
	개체 타입		개체의 속성
	키 속성		객체간의 관계
	다중 속성		유도 속성
	복합 속성		개체의 속성 연결 개체간의 관계 연결

표 12-1 E-R Diagram 표기법

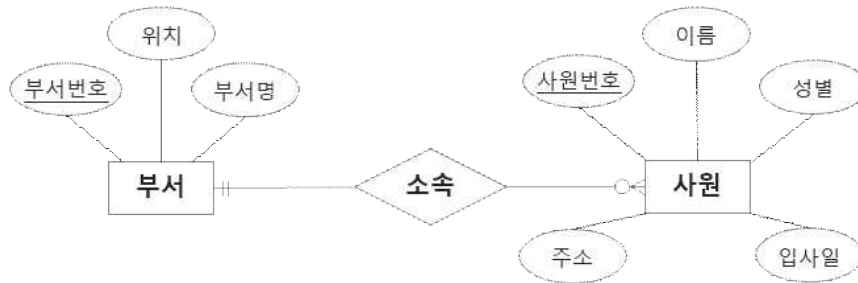


그림 12-1 직원과 부서 개체의 관계를 표현한 E-R Diagram

▶ 개체(Entity) 정의

고객의 요구사항에 의해서 데이터베이스에 저장되고 관리되어야 하는 데이터는 여러 가지 유형이 있을 수 있다. A라는 회사에서는 직원, 부서, 업무와 관련된 데이터가 될 수도 있고 B회사의 인터넷 쇼핑몰에서는 회원, 상품, 주문, 재고와 관련된 데이터가 될 수도 있다. 이처럼 데이터베이스에 저장되어 관리해야 할 데이터는 그 회사의 업무 성격과 범위 등에 따라서 달라질 수 있으며 이렇게 필요에 의해 데이터베이스에 저장되어 관리해야 할 정보의 대상(사람, 사물, 장소, 사건 등) 개체(Entity) 또는 개체 타입(Entity Type)이라고 한다. 개체는 회원, 직원 등과 같이 실제 물리적으로 존재하는 대상이 될 수도 있고 소속, 주문과 같이 개념적으로 존재하는 대상이 될 수도 있다.

각각의 개체(Entity)는 개체 인스턴스(Entity Instance)라 불리는 개별적인 객체(Object)의 집합으로 개체 인스턴스는 동일한 속성을 가지고 있는 개체를 구성하는 하나의 요소이다.

예를 들어 학생 테이블에 홍길동, 이순신, 강감찬 학생의 정보가 저장되어 있다고 하자. 여기서 학생 정보를 저장할 수 있는 학생 테이블이 개체(개체 타입)가 되고 이 학생 테이블을 구성하고 있는 각각의 학생들의 정보를 개체 인스턴스라고 한다.

개체를 정확하게 추출하기 위해서는 고객의 요구사항과 업무 프로세스를 요구사항 명세서로 작성하고 명세서의 내용 중에서 명사 위주로 개체를 추출한다.

■ 간단한 요구사항 명세서

인터넷 서점을 운영하는 A 서점은 회원들이 주문한 도서를 배송하는 작업이 주 업무입니다.

업무 처리에 있어 주된 자료는 회원 정보와 도서 정보이며 주문에 따라 발생하는 정보입니다.

서점관리 시스템 개발의 주된 목적은 기존 서점 관리 업무를 데이터베이스로 구축하여 전산처리가 가능하도록 하는 것입니다.

▶ 속성(Attribute) 정의

속성은 개체가 가지는 세부적인 정보로 개체의 성격, 분류, 수량, 상태, 특징 등을 나타낸다.

속성 또한 업무의 성격과 범위에 따라 달라질 수 있고 요구사항을 어떻게 분석하는가에 따라서 달라질 수 있다. 참고로 어떤 개체에서 열(Column)을 속성(Attribute)이라 하며 행(Row)을 튜플(Tuple)이라 한다.

■ 속성의 유형

☞ 기초속성 : 원래 가지고 있는 속성

예) 이름, 성별과 같이 더 이상 분해 할 수 없는 속성으로 이런 속성을 단순속성이라고도 한다.

- ☞ **추출속성** : 기초속성에서 데이터의 가공을 통해 얻어지는 속성
 예) $\text{단가} * \text{수량} = \text{금액}$
 추출속성은 다른 속성을 바탕으로 유도되어 결정되기 때문에 유도속성이라고 하며 유도속성을 결정하는데 사용된 속성을 저장 속성이라고 한다.
- ☞ **설계속성** : 설계자가 부여한 속성
 예) 주문 처리를 위해 필요한 주문 상태 속성
- ☞ **복합속성** : 몇 개의 단순속성으로 구성된 속성으로 분해할 수 없는 속성
 예) 입사일자, 주문일자 등은 년, 월, 일로 구성된 복합 속성이다.
- ☞ **단일 값 속성** : 한 개체(인스턴스)의 속성이 하나의 값만 갖는 속성
 예) 한 학생의 학번, 이름, 성별, 한 회원의 아이디, 이름, 성별은 하나이다.
- ☞ **다중 값 속성** : 한 개체(인스턴스)의 속성이 여러 개의 값을 갖는 속성
 예) 한 회원의 취미는 낚시, 여행, 독서 등 여러 취미를 가질 수 있다.
- ☞ **NULL속성** : NULL 값을 갖는 속성으로 개체에서 인스턴스의 속성이 특정 값을 갖고 있지 않을 때 이를 명시적으로 표현하기 위해 사용한다.

▶ 속성의 도메인 설정

도메인은 하나의 속성이 가질 수 있는 모든 가능한 원자 값들의 집합을 의미하는 것으로 도메인 설정은 속성이 가질 수 있는 값의 범위, 제약조건 및 특성을 세부적으로 정의하는 것이다.

다시 말해 도메인 설정은 속성의 이름, 속성에 저장되는 데이터의 형태와 크기, 포맷, NULL 값의 허용 여부, 데이터의 범위, 기본 값 등을 정의하는 것이다.

도메인 설정은 중복된 데이터와 잘못된 데이터가 저장되는 것을 방지하여 데이터베이스의 이상 현상(Anomaly)을 예방하기 위한 것으로 이렇게 어떤 속성의 값이 도메인에 속한 값만 저장할 수 있도록 정의하는 것을 도메인 무결성(Domain Integrity) 이라고 한다.

▶ 식별자(Identifier)

식별자란 어떤 개체에서 각각의 인스턴스(테이블에서 한 행의 데이터)를 유일하게 구분할 수 있는 속성을 가리킨다. 식별자는 단일 속성 또는 여러 속성을 묶어서 지정할 수 있으며 식별자가 각각의 인스턴스를 유일하게 구분할 수 있어야 한다는 것은 하나의 개체에서 식별자는 중복된 값을 가질 수 없음을 의미한다.

모든 개체는 인스턴스를 유일하게 구분할 수 있는 하나 이상의 식별자를 가지고 있어야 하고 이런 식별자를 다음과 같은 키로 선정해 사용한다.

■ 후보키(Candidate Key)

어떤 개체에서 각각의 인스턴스(테이블에서 한 행의 데이터)를 유일하게 구분하기 위해 사용되는 속성을 후보키로 선정하며 단일 속성 또는 여러 속성을 묶어 후보키로 구성할 수 있다.

어떤 개체에서 하나의 속성으로 각각의 인스턴스를 유일하게 구분할 수 없을 때 여러 속성을 묶어 후보키로 선정하는데 이렇게 여러 속성으로 구성된 키를 복합키(Composite Key)라 한다.

복합키는 여러 속성으로 구성되어 있기 때문에 키의 길이가 길고 데이터를 조작할 때 수행시간이 길어질 수 있다는 단점이 있어 개발자가 인위적으로 속성을 추가해 키로 사용하는 경우가 있는데 이렇게 인위적으로 만든 키를 대리키(Surrogate Key) 또는 인공키라 한다.

후보키는 개체에서 각각의 인스턴스를 유일하게 구분할 수 있기 때문에 일반적으로 후보키 중에서 기본키를 선정한다.

■ 기본키(Primary Key)

후보키 중에서 각각의 인스턴스를 유일하게 구분하는데 있어 가장 적합하여 특별히 선정된 키를 의미 한다. 후보키 중에서 길이가 짧고 해당 개체를 대표할 수 있는 속성을 골라 기본키로 선정하는 것 좋다. 참고로 기본키를 주 식별자라고도 부른다.

■ 대체키(Alternate Key)

후보키 중에서 기본키로 선정되지 못한 나머지 후보키를 대체키라 한다.

▶ 관계(Relationship) 정의

관계는 개체간의 업무적 연관성을 의미하며 객체간의 관계를 정의할 때는 요구사항 명세서에서 두 객체간의 동사로 표현된 문장을 중심으로 관계를 정의한다.

회원 - 대여 - 비디오

회원은 비디오를 대여한다.

비디오는 회원에게 대여되어 진다.

▶ 관계의 유형(관계 차수 정의)

관계의 유형은 A 개체에 소속된 인스턴스가 관계에 참여하는 B 개체에 소속된 인스턴스와 최대 몇 개의 관계를 맺는지에 따라서 1:1, 1:N, M:N으로 나눌 수 있다. 이렇게 하나의 개체에 소속된 인스턴스에 대해 관계를 맺는 다른 개체에 소속된 몇 개의 인스턴스와 대응되는 지를 카디널리티(Cardinality - 관계 대응 수)라고 한다. 하나의 개체에서 튜플(Tuple)의 수를 카디널리티라고도 하며 두 객체간의 관계에 대응하는 인스턴스의 수를 카디널리티라고도 부른다.

■ 1:1 관계

사원(부서장)과 부서의 관계는 아래와 같이 관리라는 관계가 존재하고 이 관계는 1:1의 관계가 성립된다.

사원(부서장) - 관리 - 부서

좌 -> 우 : 사원 중에는 부서를 관리하는 부서장이 존재하고

한 명의 부서장은 한 부서만 관리한다.

우 -> 좌 : 한 부서에는 한 명의 관리 사원이 존재한다.

■ 1:N 관계

사원과 부서의 관계는 아래와 같이 소속이라는 관계가 존재하고 이 관계는 1:N의 관계가 성립된다.

직원 - 소속 - 부서

좌 -> 우 : 하나의 부서는 한 명 이상의 직원이 소속된다.

우 -> 좌 : 한 명의 직원은 반드시 하나의 부서에 소속되어 있다.

■ M:N 관계

M:N은 이론상으로 존재하고 실제 데이터베이스에는 1:1 또는 1:N 관계만 존재해야 한다.

M:N은 여러 가지 문제점(대표적으로 데이터의 중복이 많이 발생)을 가지고 있기 때문에 논리적 데이터 모델링을 통해서 1:N의 관계를 도출해야 한다.

고객과 상품의 관계는 아래와 같이 주문이라는 관계가 존재하고 M:N의 관계가 성립된다.

아래와 같이 M:N의 관계가 성립되는 경우 논리적 데이터 모델링 단계에서 개체간의 1:N의 관계를 도출하기 위해 두 개체간의 관계인 주문을 개체(Entity)로 도출하고 주문 테이블로 구성하게 된다.

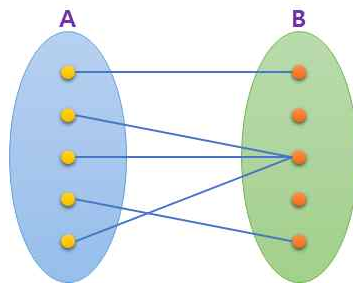
고객 - 주문 - 상품

좌 -> 우 : 한 명의 고객은 하나 이상의 상품을 주문할 수 있다.

우 -> 좌 : 하나의 상품은 한 명 이상의 고객에게 주문될 수 있다.

▶ 선택성

두 개체 간에 관계가 설정되었을 때 각 개체의 인스턴스(실제 테이블에 저장되는 데이터)가 관계에 반드시 참여해야 하는지 아니면 일부만 참여할 수 있는지를 나타내는 속성이다.



위의 그림과 같이 A와 B라는 개체가 존재하고 두 개체가 관계를 맺고 있다고 가정할 때 A의 모든 인스턴스는 두 개체 간에 관계에 참여하고 있으며 B의 인스턴스는 관계에 참여한 것도 있고 그렇지 않은 것도 있다. A 개체와 같이 모든 인스턴스가 관계에 참여하는 것을 필수 참여 속성으로 Mandatory라고 하며 B 개체와 같이 일부만 참여하는 선택 속성으로 Optional이라고 한다.

선택성은 다음과 같이 관계를 해석할 수 있을 것이다.

필수(Mandatory) : 반드시 ~ 해야만 한다.(한 명의 직원은 반드시 하나의 부서에 소속되어야 한다.)

선택(Optional) : ~ 일지도 모른다.(하나의 상품은 여러 회원에게 주문될지도 모른다.)

관 계	E-RD 표기법	설 명
1:1		양쪽 개체 모두 반드시 1개씩의 인스턴스가 존재
1:0, 1:1		왼쪽 개체는 반드시 1개, 오른쪽 개체는 없거나 1개의 인스턴스가 존재
1:1, 1:n		왼쪽 개체는 반드시 1개, 오른쪽 개체는 반드시 1개 또는 여러 개의 인스턴스가 존재
0:1, 1:1, 1:N		왼쪽 개체는 반드시 1개, 오른쪽 개체는 없거나 여러 개의 인스턴스가 존재

부서 - 소속 - 사원(1 : N)

각 부서는 하나 이상의 사원이 소속될지도 모른다. (선택)

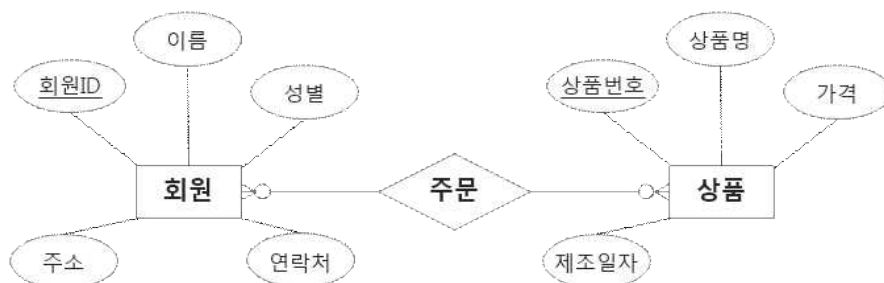
각 사원은 반드시 하나의 부서에 소속되어야 한다. (필수)



고객 - 주문 - 상품(N : M)

한 명의 고객은 하나 이상의 상품을 주문할지도 모른다. (선택)

하나의 상품은 한 명 이상의 고객에게 주문될지도 모른다. (선택)



12.3 논리적 데이터 모델링

개념적 데이터 모델링에서 작성된 ERD(Entity Relationship Diagram)를 바탕으로 특정 DBMS에 맞게 관계 스키마를 설계하는 단계를 논리적 데이터 모델링(Logical Data Modeling)이라 한다.

이 단계에서는 개념적 데이터 모델링을 논리적 데이터 모델링으로 전환하기 위해 필요한 맵핑 룰(Mapping Rule)을 적용해 관계형 데이터베이스 모델에 따라서 스키마를 설계하는 단계를 거쳐 데이터 정규화를 진행하게 된다.

▶ 관계형 모델

관계형 데이터베이스(RDB, Relational Database)는 1970년 당시 IBM 연구원으로 있던 E.F.Codd가 제안한 관계형 모델(relational model)을 바탕으로 개발 되었다. 관계형 모델이란 행과 열의 구조를 가진 테이블을 통해 데이터를 관리하는 것으로 데이터베이스는 최소한의 의미를 가진 테이블로 구성되어야 하고 각 테이블은 하나 이상의 컬럼으로 구성되며 각각의 테이블은 연관된 컬럼으로 관계를 맺고 있는 것을 의미한다. 다시 말해 테이블은 데이터를 효율적으로 저장하기 위해 하나의 테이블이 아닌 여러 개의 테이블로 나누어 최소한의 의미를 가지도록 데이터를 저장해야 한다. 이렇게 나누어진 테이블은 기본키(Primary Key)와 외래키(Foreign Key)를 통해 부모 테이블과 자식 테이블로 관계를 맺게 된다. 관계란 두 테이블 사이에 존재하는 업무적인 연관성이며 관계를 맺고 있는 두 테이블 중에 반드시 하나는 부모 테이블 되고 나머지 하나는 자식 테이블이 된다. 이렇게 부모 테이블의 기본키(Primary Key)가 자식 테이블의 외부키(Foreign Key)로 전이되어 두 테이블 간의 연관 관계를 정의한 것이 바로 관계형 데이터 모델이다.

두 테이블 간의 관계를 정의하고 능동형 표현으로 나타나는 쪽이 부모 테이블이 되고 수동형 표현으로 나타나는 쪽이 자식 테이블 된다.

예를 들면 사원 테이블과 부서 테이블의 관계는 아래와 같이 소속이라는 관계로 정의할 수 있다. 여기에서 능동형 표현으로 나타난 부서 테이블이 소속이라는 관계에서 주체가 되며 부모 테이블이 된다.

부서는 한 명 이상의 사원이 소속된다.

사원은 반드시 하나의 부서에 소속되어 진다.

주체관계가 모호한 경우가 있을 수 있는데 이럴 경우 어떤 테이블의 데이터가 먼저 정의되어야 하는지를 따져 보면 관계의 주체를 쉽게 파악할 수 있다.

사원 테이블과 부서 테이블의 관계에서 사원 테이블이 먼저 정의되고 사원의 정보를 저장하게 된다면 그 사원의 소속 부서를 정의할 수 없게 되므로 참조 무결성에 위배되는 경우가 발생한다. 이렇게 어떤 테이블의 데이터가 먼저 정의되어야 하는지를 기준으로 부모 테이블과 자식 테이블로 구분할 수 있다.

부모 테이블과 자식 테이블로 나누어진 두 테이블의 정보는 부모 테이블의 기본키와 자식 테이블의 외부키를 조인해 연관된 정보를 얻을 수 있게 된다.

부모 테이블의 기본키가 자식 테이블의 외부키로 전이되는 유형에 따라 식별관계와 비식별관계로 나뉘지는데 부모 테이블의 기본키가 자식 테이블의 기본키(기본키 + 외부키)로 전이되는 경우를 식별관계라 하며 부모 테이블의 기본키가 자식 테이블의 일반 컬럼(외부키)으로 전이되는 경우를 비식별관계라고 한다.

▶ 관계 스키마 정의

개념적 데이터 모델링에서 도출된 개체 타입과 관계 타입을 테이블로 정의하는 것을 관계 스키마 정의라고 한다. 또한 개념적 데이터 모델링에서 도출된 ERD(Entity Relationship Diagram)를 바탕으로 아래와 같이 단계별로 지켜야 하는 맵핑룰(Mapping rule)을 적용해 관계 스키마를 정의한다.

1단계 : 단순 개체(Entity)는 테이블로 정의한다.

2단계 : 객체에 소속된 속성(Attribute)은 컬럼으로 정의한다.

3단계 : 속성 중에서 식별자는 기본키(Primary Key)로 정의한다.

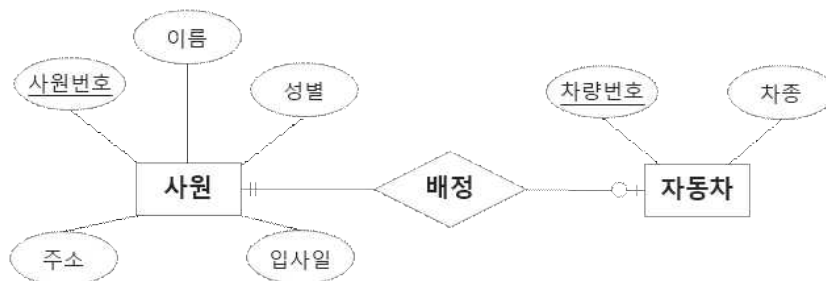
4단계 : 부모 테이블과 자식 테이블의 관계는 부모 테이블의 기본키를 자식 테이블의 외부키로 전이시켜 정의한다.



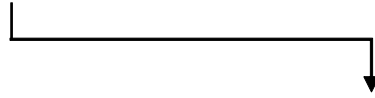
단순 개체(Entity)는 위의 그림과 같이 1단계에서 3단계까지의 맵핑룰에 의해 간단히 스키마를 정의할 수 있다. 하지만 두 개체간의 관계를 맺고 있는 개체는 관계 차수(1:1, 1:N, M:N)에 따라서 1단계부터 4단계까지의 맵핑룰을 적용해 스키마를 정의해야 한다.

▶ 1:1 관계

아래는 (주)00기획에서 부장급 이상의 관리 사원이 입사하면 자동차를 배정하는 회사 내규에 따라서 사원 개체와 자동차 개체의 1:1 관계 스키마를 정의하는 과정을 설명한 것이다.



컬럼	사원번호	이름	성별	주소	입사일
키	PK				
데이터	1001	홍길동	남	서울 구로구	20120106
	1002	임꺽정	남	경기 안양시	20141107
	1003	장보라	여	경기 부천시	20130725



컬럼	차량번호	차종	사원번호
키	PK		FK
데이터	3306	소나타	1001
	4205	그랜저	1003

매퍼를 4단계를 적용시켜 관계 스키마를 정의하기 위해 왼쪽에 위치한 사원 개체의 기본키인 사원 번호를 오른쪽에 위치한 자동차 개체의 외부키로 전이시켜 보면 위의 표에서와 같이 관계 스키마가 정의된다. 위에서와 같이 자동차 테이블에 자동차 정보가 추가될 때 그 자동차를 배정받은 사원이 누구인지 사원번호를 통해 명확히 알 수 있다. 즉 “하나의 자동차는 오로지 한 명의 사원에게 배정된다.”라는 관계를 해결할 수 있게 된다.

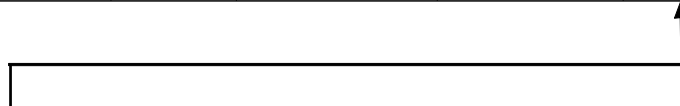
다음은 관계 스키마를 정의하기 위해 오른쪽에 위치한 자동차 개체의 기본키인 차량번호를 왼쪽에 위치한 사원 개체의 외부키로 전이시켜보자.

사원 중에 자동차를 배정 받은 사원은 차량번호를 통해 그 사원에게 배정된 자동차를 명확히 알 수 있으며 자동차를 배정 받지 못한 사원은 NULL 값이 저장 될 것이다. 즉 “사원은 자동차를 배정 받을 수도 있다”라는 관계를 해결할 수 있게 된다.

이렇게 1:1 관계인 경우 왼쪽 개체의 기본키를 오른쪽 개체의 외부키로 전이시키거나 오른쪽 개체의 기본키를 왼쪽 개체의 외부키로 전이시키더라도 두 개체 간의 관계를 해결할 수 있다.

즉 1:1 관계에서는 왼쪽 또는 오른쪽 모두 외부키를 전이시키더라도 사원과 자동차의 배정이라는 관계를 해결한 관계 스키마를 정의할 수 있게 된다.

컬럼	사원번호	이름	성별	주소	입사일	차량번호
키	PK					FK
데이터	1001	홍길동	남	서울 구로구	20120106	3306
	1002	임꺽정	남	경기 안양시	20141107	NULL
	1003	장보라	여	경기 부천시	20130725	4205



컬럼	차량번호	차종
키	PK	
데이터	3306	소나타
	4205	그랜저

위의 예에서 자동차 테이블을 부모 테이블로 하여 사원 테이블에 NULL 값을 저장하는 것 보다 사원 테이블을 부모 테이블로 해서 꼭 필요한 데이터만 자동차 테이블에 저장되도록 하는 것이 바람

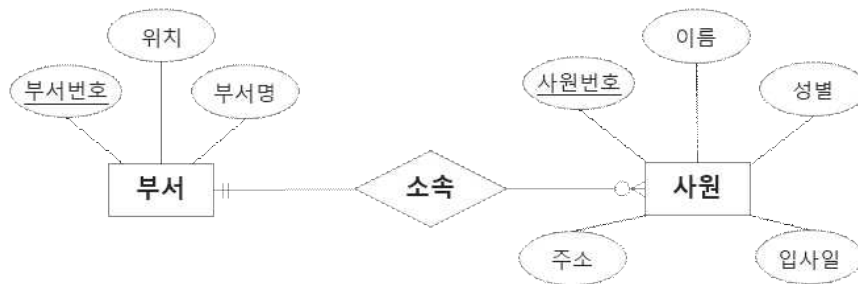
직한 관계 스키마 정의라 할 수 있을 것이다. 또한 아래와 같이 관계를 정의해 보면 어느 테이블이 부모 테이블이 되는지 더 명확해 진다. 그리고 선 후 관계를 따진다 하여도 부장급이상의 관리 사원이 있어야 자동차를 배정 할 수 있으니 이 또한 사원 테이블이 부모가 되는 것이 바람직한 관계 스키마 정의라 할 수 있을 것이다.

사원 중에는 자동차를 배정 받을지도 모른다.

자동차는 반드시 한 명의 사원에게 배정되어야 한다.

▶ 1:N 관계

이번에는 1:N의 관계인 사원 개체와 부서 개체 간의 소속이라는 관계를 해결하는 관계 스키마를 정의해 보자.



1:N의 관계에서 관계 스키마를 정의하기 위해 1:1에서와 같이 1측 개체의 기본키를 N측의 외부키로 전이시켜 보면 아래 표에서와 같이 관계 스키마가 정의된다.

아래와 같이 사원 테이블에 사원 정보가 추가될 때 마다 부서번호를 통해 그 사원이 소속된 부서를 알 수 있기 때문에 “하나의 부서에는 한 명 이상의 사원이 소속되어 진다”라는 관계를 해결할 수 있게 된다.

컬럼	부서번호	부서명	위치
키	PK		
데이터	100	관리부	서울
	200	영업부	대전
	300	전산부	서울

컬럼	사원번호	이름	성별	주소	입사일	부서번호
키	PK					FK
데이터	1001	홍길동	남	서울 구로구	20120106	100
	1002	임꺽정	남	경기 안양시	20141107	200
	1003	장보라	여	경기 부천시	20130725	100
	1004	이순신	남	서울 강남구	20141214	300
	1005	어머나	여	인천 남동구	20150305	200

이번에는 관계 스키마를 정의하기 위해 N측에 위치한 사원 개체의 기본키인 사원번호를 1측에 위치한 부서 개체의 외부 키로 전이 시켜보자.

이렇게 관계 스키마를 정의할 경우 아래와 같이 새로운 사원이 입사해서 부서에 소속될 때 마다 부서번호, 부서명, 위치 등의 데이터가 중복되어 부서 테이블에 저장되는 문제가 발생한다.

결론적으로 관계가 1:N인 경우는 1측의 기본키를 N측의 외부키로 전이시켜 관계 스키마를 정의하는 것이 중복데이터를 최소화 할 수 있는 바람직한 방법이라 할 수 있다.

컬럼	부서번호	부서명	위치	사원번호
키	PK			FK
데이터	100	관리부	서울	1001
	100	관리부	서울	1003
	200	영업부	대전	1002
	200	영업부	대전	1005
	300	전산부	서울	1004

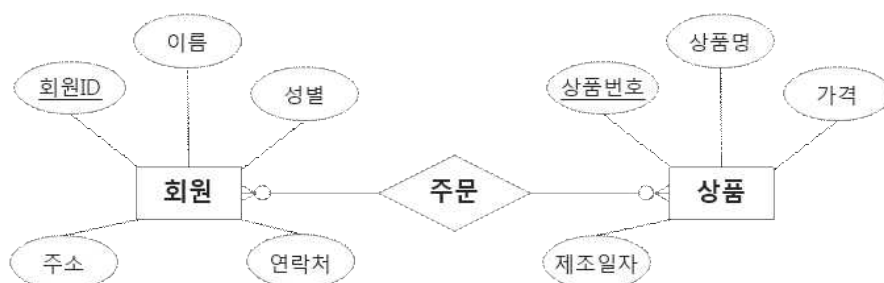
컬럼	사원번호	이름	성별	주소	입사일
키	PK				
데이터	1001	홍길동	남	서울 구로구	20120106
	1002	임꺽정	남	경기 안양시	20141107
	1003	장보라	여	경기 부천시	20130725
	1004	이순신	남	서울 강남구	20141214
	1005	어머나	여	인천 남동구	20150305

▶ M:N 관계

두 개체간의 M:N 관계인 경우에는 어떻게 관계 스키마를 정의해야 데이터를 효율적으로 저장할 수 있을까? 회원과 상품의 주문이라는 관계를 예로 들어 M:N의 관계에 대해서 알아보자. 우선 두 개체간의 관계를 따져 보면 아래와 같이 M:N의 관계가 성립될 것이다.

한 명의 회원은 하나 이상의 상품을 주문할 수도 있다.

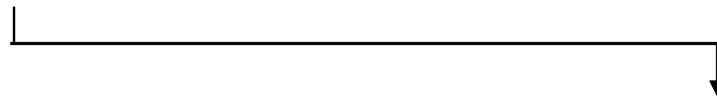
하나의 상품은 한 명 이상의 회원에게 주문될 수도 있다.



M:N의 관계에서 관계 스키마를 정의하기 위해 1:N에서와 같이 M측 개체의 기본키를 N측의 외부키

로 전이시켜 보면 아래 표에서와 같이 관계 스키마가 정의될 것이다. 이렇게 스키마를 정의하게 되면 아래와 같이 회원이 새우깡이라는 상품을 주문할 때마다 상품 테이블에는 붉은색 바탕의 행과 같이 새우깡에 대한 상품번호, 상품명, 가격, 제조일자가 계속해서 중복 저장되는 문제가 발생하게 된다.

컬럼	회원ID	이름	성별	주소	연락처
키	PK				
데이터	1001	홍길동	남	서울 구로구	010-1234-5678
	1002	임꺽정	남	경기 안양시	010-2345-6852
	1003	장보라	여	경기 부천시	010-9876-5432
	1004	이순신	남	서울 강남구	010-4256-1234
	1005	어머나	여	인천 남동구	010-9512-3574

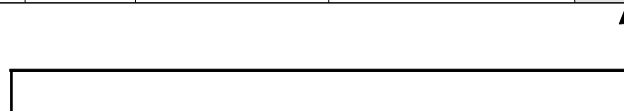


컬럼	상품번호	상품명	가격	제조일자	회원ID
키	PK				FK
데이터	P10001	새우깡	1500	20141225	1001
	P10002	양파링	1200	20150708	1003
	P10003	감자깡	1300	20150108	1002
	P10001	새우깡	1500	20141225	1005
	P10001	새우깡	1500	20141225	1004
	P10003	감자깡	1300	20150912	1001
	P10001	새우깡	1500	20141225	1002

그렇다면 N측 개체의 기본키를 M측 개체의 외부키로 전이시키면 어떻게 될까?

이 또한 아래와 같이 회원이 여러 가지 상품을 구매하게 되면 회원 테이블에는 하늘색 바탕의 행과 같이 한 회원의 회원ID, 이름, 성별, 주소, 연락처가 계속해서 중복 저장되는 문제가 발생하게 된다.

컬럼	회원ID	이름	성별	주소	연락처	상품번호
키	PK					FK
데이터	1001	홍길동	남	서울 구로구	010-1234-5678	P10001
	1002	임꺽정	남	경기 안양시	010-2345-6852	P10001
	1003	장보라	여	경기 부천시	010-9876-5432	P10003
	1004	이순신	남	서울 강남구	010-4256-1234	NULL
	1005	어머나	여	인천 남동구	010-9512-3574	NULL
	1001	홍길동	남	서울 구로구	010-1234-5678	P10003
	1003	장보라	여	경기 부천시	010-9876-5432	P10002
	1001	홍길동	남	서울 구로구	010-1234-5678	P10002
	1001	홍길동	남	서울 구로구	010-1234-5678	P10001
	1002	임꺽정	남	경기 안양시	010-2345-6852	P10003



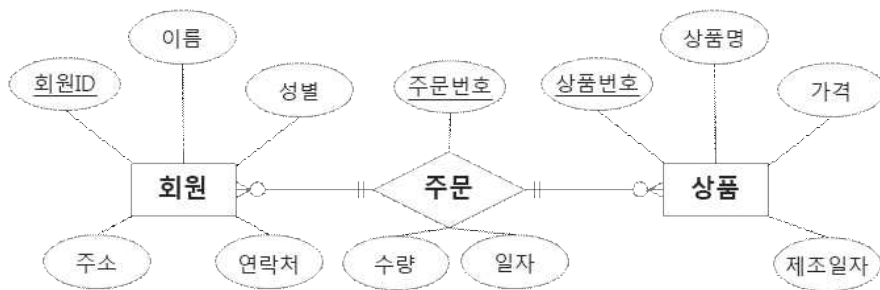
컬럼	상품번호	상품명	가격	제조일자
키	PK			
데이터	P10001	새우깡	1500	20141225
	P10002	양파링	1200	20150708
	P10003	감자깡	1300	20150108

이렇게 M:N 관계에서는 모두 데이터가 중복 저장되는 문제점을 가지고 있다.

개념적 데이터 모델링에서 두 개체가 M:N 관계인 경우 논리적 데이터 모델링을 통해서 1:1 또는 1:N의 관계로 해결되어야 한다. 그렇다면 어떻게 M:N의 관계를 1:1 또는 1:N의 관계로 만들 수 있을까? 그 해결책은 바로 두 개체 중간에 있는 관계를 이용하면 된다. 두 개체 중간에 존재하는 관계 또한 개체(Entity) 이므로 좌측과 우측을 교차하는 관계 개체를 새로운 관계 스키마로 정의하면 이 문제를 해결 할 수 있다.

아래는 회원과 상품 두 개체 중간에 있는 주문이라는 개체를 새로운 관계 스키마로 정의하기 위해 주문 개체가 가져야 하는 속성을 도출해서 새롭게 구성한 ERD이다.

회원이 어떤 상품을 주문했는지 주문이라는 관계를 새로운 관계 스키마로 정의해서 회원과 상품 사이에 주문 개체가 교차하게 만들었다. 이로써 회원과 주문이 N:1 관계, 상품과 주문이 N:1 관계가 되어 회원과 상품의 M:N 관계에서 데이터의 중복 저장에 대한 문제점을 해결하고 회원과 상품의 주문이라는 관계를 해결했다.



컬럼	회원ID	이름	성별	주소	연락처
키	PK				
데이터	1001	홍길동	남	서울 구로구	010-1234-5678
	1002	임꺽정	남	경기 안양시	010-2345-6852
	1003	장보라	여	경기 부천시	010-9876-5432
	1004	이순신	남	서울 강남구	010-4256-1234
	1005	어머나	여	인천 남동구	010-9512-3574

컬럼	주문번호	회원ID	상품번호	가격	일자
키	PK	FK	FK		
데이터	O10001	1001	P10002	1200	20150924
	O10003	1003	P10003	1300	20150927
	O10005	1001	P10001	1500	20150928
	O10007	1002	P10001	1500	20151003
	O10009	1001	P10002	1200	20151011

컬럼	상품번호	상품명	가격	제조일자
키	PK			
데이터	P10001	새우깡	1500	20141225
	P10002	양파링	1200	20150708
	P10003	감자깡	1300	20150108

12.4 정규화(Normalization)

정규화란 테이블 설계가 잘되었는지 평가하고 혹시 문제가 있다면 테이블을 고쳐나가는 과정이다. 다시 말해 테이블에 저장되는 데이터가 중복되지 않고 테이블에 꼭 필요한 속성들로만 구성되도록 속성을 분해해서 테이블(릴레이션, Relation)에 데이터가 추가, 수정, 삭제될 때 이상 현상이 발생하지 않는 테이블을 만들어가는 과정이라 할 수 있다. 이 과정에서 속성이 다른 테이블로 이동되거나 새로운 테이블을 생성해 속성을 이동 시킬 수도 있다.

정규화 이론은 1단계 ~ 6단계까지 있지만 일반적으로 정규화는 3단계 정도까지 시행하는 경우가 대부분이다. 참고로 테이블이 얼마나 정규화 되었는지의 정도를 정규형(Normal Form)이라고 부르며 숫자와 조합하여 1NF(제 1 정규형), 2NF(제 2 정규형), 3NF(제 3 정규형) 등으로 부른다.

▶ 제 1 정규형(First Normal Form, 1NF)

“릴레이션에서 모든 속성의 도메인이 원자 값을 가지면 제 1 정규형에 속한다.”

즉 테이블의 모든 속성은 여러 개의 값이 아닌 반드시 단일 값을 가져야 한다.

이는 테이블 각각의 컬럼에 저장되는 데이터가 더 이상 분해할 수 없는 원자 값으로 저장되어야 제 1 정규형을 만족한다고 할 수 있다. 그러므로 하나의 컬럼에 저장되는 데이터가 여러 개의 값으로 구성되는 속성이라면 원자 값을 갖도록 분해하여 아래와 같이 하나의 값이 저장되도록 해야 한다.

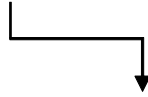
아래는 예를 들기 위해 간단하게 표현한 회원 테이블로 아래와 같이 회원의 이름과 취미 데이터만 저장되는 테이블은 아마도 없을 것이다. 보통 회원 테이블에는 회원 아이디, 이름, 생년월일, 주소 등등의 여러 가지 정보가 저장되기 때문에 취미 속성에 대한 원자 값 분해만 가지고는 같은 회원의 데이터가 중복되어 저장되는 것을 피할 수 없게 된다. 또한 이렇게 중복된 데이터가 쌓이게 되고 특정 회원의 정보가 변경되어 테이블에서 이를 수정해야 한다면 그 회원의 정보가 중복 저장된 모든 행의 데이터를 변경해야 하는 문제가 또 발생하게 된다. 이럴 경우 제 1 정규화 작업을 통해 새로운 취미 테이블을 만들고 회원 테이블의 기본키를 취미 테이블의 외부키로 전이시켜 1:N의 관계를 가지게 해야 한다. 이렇게 제 1정규화 작업이 완료되면 해당 릴레이션은 제 1 정규형을 만족하게 된다. 이때 부모 테이블인 회원 테이블의 기본키가 자식 테이블인 취미 테이블의 일반 속성으로 전이되어 외부키를 이루게 되는데 이렇게 부모 테이블의 기본키가 자식 테이블의 일반 속성으로 전이되는 것을 비식별 관계라고 한다.

이름	취미
홍길동	축구, 탁구
임꺽정	영화감상, 음악듣기
장보라	독서
이순신	등산, 음악듣기
어머나	게임



이름	취미
홍길동	축구
홍길동	탁구
임꺽정	영화감상
임꺽정	음악듣기
장보라	독서
이순신	등산
이순신	음악듣기
어머나	게임

컬럼	회원ID	이름	성별	주소	연락처
키	PK				
데이터	1001	홍길동	남	서울 구로구	010-1234-5678
	1002	임꺽정	남	경기 안양시	010-2345-6852
	1003	장보라	여	경기 부천시	010-9876-5432
	1004	이순신	남	서울 강남구	010-4256-1234
	1005	어머나	여	인천 남동구	010-9512-3574



컬럼	취미ID	회원ID	취미
키	PK	FK	
데이터	H1001	1001	축구
	H1002	1001	탁구
	H1003	1002	영화감상
	H1004	1002	음악듣기
	H1005	1003	독서
	H1006	1004	등산
	H1007	1005	게임

아래 테이블은 B 학원에서 진행 중인 과정에 대한 정보를 저장한 테이블이다.

이렇게 과정 데이터에 그 과정에서 사용되는 교재를 같이 저장할 경우 자바 웹 개발 과정을 제외한 나머지 과정의 교재를 저장하는 컬럼은 최소 1개 이상의 NULL 값을 가지게 된다.

테이블에 NULL 값이 많이 저장된다는 것은 쓸모없는 데이터가 저장되어 불필요한 메모리 공간의 낭비를 가져오게 된다. 아래의 테이블을 제 1 정규화를 통해 다시 설계해 보자.

과정 코드	과정명	교육 기간	수강료	교재1	교재2	교재3	교재4
R1001	자바 웹 개발	3개월	75만원	JSP	HTML5&CSS3	JavaScript & jQuery	오라클
R1002	자바 프로그래밍	1개월	30만원	Java	NULL	NULL	NULL
R1003	안드로이드 프로그래밍	3개월	80만원	Java	안드로이드	NULL	NULL

▶ 제 2 정규형(Second Normal Form, 2NF)

“릴레이션이 제 1 정규형을 만족하고 기본키가 아닌 모든 속성이 기본키에 완전 함수 종속이면 제 2 정규형에 속한다.”

즉 테이블의 모든 속성은 반드시 기본키에 직접 종속되어야 한다.

테이블을 구성하는 속성(컬럼)들 사이에는 의존성이 존재한다. 예를 들면 제 1 정규화가 완료된 회원 테이블에서 회원ID를 알면 그 회원의 이름을 알 수 있다. 반면 성별로는 이름이 홍길동인 회원을 찾을 수 없다. 이렇게 회원ID를 알면 이름이 결정되는 것을 이름이 회원ID에 종속적이라고 한다. 이를 다시 말하면 이름은 회원 ID에 의존적이지만 성별 속성에는 의존하지 않는다고 할 수 있다. 이렇게 어떤 테이블에서 어떤 속성 X 값을 알면 다른 속성 Y의 값이 유일하게 정해지는 의존 관계를 “X가 Y를 함수적으로 결정 한다.” 또는 “Y가 X에 함수적으로 종속되어 있다.” 라고 한다. 이 관계를 수학적으로 $X \rightarrow Y$ 와 같이 표기하고 X는 Y의 결정자, Y는 종속자라고 부른다.

보통 아래와 같이 두 개 이상의 속성으로 이루어진 기본키를 가진 테이블에서 기본키에 직접적으로 종속되지 못하고 기본키 조합에 종속되면서 또한 기본키의 일부분에도 종속되는 문제가 발생하는 경우가 있다. 이 경우 제 2 정규화를 통해 기본키에 직접 종속되도록 다음과 같이 테이블을 분리하는 작업을 수행한다.

<u>회원ID</u>	<u>이벤트번호</u>	당첨여부	회원이름
midas	E1001	Y	홍길동
midas	E1003	N	홍길동
jsp	E1005	Y	임꺽정
jsp	E1001	Y	임꺽정
servlet	E1005	N	이순신
servlet	E1003	Y	이순신
html5	E1003	N	강감찬
css3	E1005	Y	김보라

위의 테이블은 인터넷 쇼핑몰을 운영하는 A사의 이벤트에 참여한 회원의 당첨 여부를 저장한 테이블이다. 이 테이블은 회원ID와 이벤트번호를 조합해 기본키로 정의하고 있다.

이 테이블에서 회원ID가 회원이름을 유일하게 결정하는 결정자가 되고 회원이름은 종속자가 된다. 회원ID가 같으면 테이블의 모든 행에서 검색된 회원의 이름이 모두 동일하기 때문이다.

그리고 기본키인 [회원ID, 이벤트번호] 속성 집합은 당첨여부 속성을 유일하게 결정하는 결정자가 되는데 midas 회원이 참여한 E1001 이벤트의 당첨여부는 Y 값만 존재하기 때문이다.

여기서 회원이름도 기본키인 [회원ID, 이벤트번호] 속성 집합에 종속되어 있다.

이벤트 참여 테이블에서 함수적 종속관계를 기호로 나타내면 아래와 같다.

회원ID \rightarrow 회원이름

[회원ID, 이벤트번호] \rightarrow 당첨여부

[회원ID, 이벤트번호] \rightarrow 회원이름

이 테이블의 함수종속 관계에서 회원이름은 [회원ID, 이벤트번호] -> 회원이름과 같이 종속되어 있고 [회원ID, 이벤트번호]의 속성 집합의 일부분인 회원ID에도 종속되어 있다. 이런 경우 회원이름 속성이 [회원ID, 이벤트번호] 속성 집합에 부분 함수 종속되어 있다고 한다. 또한 당첨여부 속성은 [회원ID, 이벤트번호]의 속성 집합의 일부분이 아닌 전체에 종속되어 있는데 이럴 경우 당첨여부 속성이 [회원ID, 이벤트번호]의 속성 집합에 완전 함수 종속되었다고 한다.

이렇게 부분 함수 종속일 경우 아래와 같이 테이블을 분리해 부분 함수 종속을 제거해야 한다.

회원ID	회원이름
midas	홍길동
jsp	임꺽정
servlet	이순신
html5	강감찬
css3	김보라

회원ID	이벤트번호	당첨여부
midas	E1001	Y
midas	E1003	N
jsp	E1005	Y
jsp	E1001	Y
servlet	E1005	N
servlet	E1003	Y
html5	E1003	N
css3	E1005	Y

앞에서도 언급했지만 제 2 정규화는 기본키가 복합키(Composite Key)로 이루어진 경우 기본키 집합에 전체적으로 의존하지 않는 컬럼을 분리해 새로운 테이블로 이동하는 것이다.

다시 말해 기본키가 아닌 일반 컬럼은 모두 기본키에 완전 종속이어야 하며 그렇지 못한 컬럼은 분리해 새로운 테이블로 이동시켜야 제 2 정규형을 만족하다고 할 수 있다.

아래 테이블은 B 학원에서 진행 중인 과정을 수강하는 학생들의 과정별 성적 데이터를 저장하고 있는 학과 성적 테이블 이다. 이 테이블은 학번과 과정코드를 조합해야 유일키가 되기 때문에 두 컬럼을 조합해 복합키로 기본키를 정의하고 있다. 아래 테이블에서 기본키에 완전 함수 종속되지 않은 컬럼이 존재한다. 이를 제 2 정규화를 통해 다시 설계해 보자.

학번	과정코드	점수	과정명	기간
150623	R1001	91	자바 웹 개발	3개월
150721	R1002	88	자바 프로그래밍	1개월
150824	R1003	93	안드로이드 프로그래밍	3개월
150915	R1002	83	자바 프로그래밍	1개월
150924	R1001	87	자바 웹 개발	3개월

▶ 제 3 정규형(Third Normal Form, 3NF)

“릴레이션이 제 2 정규형을 만족하고 기본키가 아닌 모든 속성이 기본키에 이행적 함수 종속이 아니면 제 3 정규형에 속한다.”

즉 테이블 내의 모든 속성은 기본 키가 아닌 속성에 종속되지 말아야 한다.

제 3 정규형에서 새롭게 등장한 이행적 함수종속에 대해 간단히 살펴보자.

테이블을 구성하는 세 개의 속성 집합 X, Y, Z에 대해 $X \rightarrow Y$ 와 $Y \rightarrow Z$ 의 함수 종속관계가 존재하면 $X \rightarrow Z$ 가 성립하게 된다. 이럴 경우 Z가 X에 이행적으로 함수 종속되었다고 한다.

제 2 정규형을 만족하는 테이블이라 하더라도 함수 종속관계가 여러 개 존재하며 이행적 함수 종속 관계가 존재하게 되면 그 테이블은 이상 현상이 발생할 소지가 많다. 테이블에서 이행적 함수 종속 관계가 발생하는 이유는 그 테이블에 여러 개의 함수 종속 관계가 존재하기 때문이다.

테이블에서 모든 속성은 기본키에 직접적으로 종속되어야 하는데 기본키가 아닌 속성에 종속되는 경우가 발생한다. 이럴 경우 다음과 같이 제 3 정규화 작업을 통해서 테이블을 분리 시켜야 한다.

회원ID	등급	할인율
midas	VIP	15%
jsp	GOLD	10%
servlet	SILVER	5%
html5	GOLD	10%
css3	VIP	15%

위의 회원 테이블에서 회원ID가 등급을 결정하고 등급이 할인율을 결정하기 때문에 회원ID가 등급을 통해 할인율을 결정하는 이행적 함수 종속관계가 발생하게 된다.

실제로 할인율은 회원의 등급에 따라 달라지는 것인데 이행적 함수 종속관계로 인해 회원ID에 의해 할인율이 결정되는 이상 현상이 발생한다. 이런 경우 이행적 함수 종속관계가 발생하지 않도록 아래와 같이 테이블을 분리해야 한다.

회원ID	등급
midas	VIP
jsp	GOLD
servlet	SILVER
html5	GOLD
css3	VIP

등급	할인율
VIP	15%
GOLD	10%
SILVER	5%

위에서 이행적 함수 종속이 발생하는 경우를 자세히 살펴보면 할인율이 기본키인 회원ID에 종속적이어야 하는데 실제로는 등급에 종속적이고 등급은 다시 기본키인 회원ID에 종속적이기 때문에 발생하게 된다. 이 처럼 제 3 정규화 작업은 테이블에서 기본키에 직접적으로 의존하지 않고 일반 컬럼에 의존하는 컬럼을 분리해 새로운 테이블로 이동하는 것이다.

다음은 인터넷 쇼핑몰을 운영하는 H사의 주문 테이블이다.

이 테이블에는 어느 회원이 어떤 상품을 얼마나 주문했는지에 대한 정보가 저장되어 있다.

하지만 이 테이블은 함수 종속관계와 이행적 함수 종속관계가 존재하는 테이블로 이상 현상이 발생할 소지가 많다. 이 테이블을 제 3 정규화를 통해 다시 설계해 보자.

주문 번호	상품 번호	회원ID	이름	주소	연락처	등급	수량	단가
O1001	P1001	servlet	이순신	서울 구로구	010-1234-5678	GOLD	2	1,000
O1003	P1003	jsp	임꺽정	인천 부평구	010-2356-8947	VIP	5	5,000
O1004	P1010	servlet	이순신	서울 구로구	010-1234-5678	GOLD	2	3,000
O1005	P1007	servlet	이순신	서울 구로구	010-1234-5678	GOLD	3	1,500
O1007	P1001	html5	강감찬	경기도 부천시	010-9876-5432	SILVER	7	1,000

지금까지 학습한 관계스키마 정의와 정규화는 실무에서 많이 사용되는 이론이다.

일반적으로 정규화는 제 3 정규화까지 진행하며 제 3 정규화 과정 이후에도 테이블에서 주 식별자가 여러 개 존재할 경우가 있는데 이때는 BCNF(Boyce/Codd Normal Form, 보이스/코드 정규형)를 진행해 추가로 테이블을 분리하는 작업을 하게 된다. 이외에도 4NF, 5NF와 그 이상의 더 세분화된 정규화 이론이 있다.

실제 데이터베이스를 설계할 때 무조건적으로 전체 정규화를 통해 테이블을 분해하는 것은 아니다. 오히려 테이블을 세분화 시키는 것이 비효율적인 경우도 많다.

물리적 데이터 모델링 단계에서 테이블(릴레이션)의 분해가 비효율적인 경우가 발견되면 역정규화(De Normalization)를 통해 다시 테이블을 합치는 것이 바람직하다. 이런 경우 테이블을 무조건 합치기보다는 데이터의 무결성(도메인, 객체, 참조, 유일키 제약조건)과 정합성에 위배 되지 않는 범위에서 역정규화를 진행해야 한다.

12.5 물리적 데이터 모델링 실습

물리적 데이터 모델링(Physical Data Modeling)은 논리 데이터 모델링의 결과를 바탕으로 실제 데이터베이스를 구축할 때 사용할 이름, 데이터 타입, 제약조건, 인덱스 등을 설계하는 단계이다.

exERD를 이용한 논리/물리 ERD(Entity Relationship Diagram) 작성

테이블 정의서 작성

인덱스, 트리거 등을 정의

12.6 DB 모델링 및 구현 실습

개념적 모델링, 논리모델링, 물리모델링 작업을 통해 작성된 ERD를 바탕으로 DDL 문을 사용해 데이터베이스 객체를 생성하고 데이터베이스를 구축하는 것을 DB 구현이라 한다.

인터넷 서점에 대한 요구사항을 분석해 개념적 모델링, 논리모델링, 물리모델링 작업을 하고 테이블을 생성하고 실제 데이터를 테이블에 추가하는 실습을 해 보자.

이 때 테이블 외에 필요한 뷰, 인덱스, 시퀀스, 트리거 등과 같은 객체도 같이 생성하게 된다.