Oracle ： SQL

박 시 우 (swparka@empal.com)

OCP(8i, 9i, 10g), MCSE, MCDBA, MCSD, MCSA, SCJP

저서 : Mastering Windows Server 2003 (삼각형 프레스)
Oracle(행정자치부 전자정부지원센터)
Oracle 백업 및 복구(행정자치부 전자정부지원센터)

차 례

1장. 관계형 데이터베이스 개념 ····· 1
2장. SELECT문 기초 ····· 17
3장. WHERE 및 ORDER BY ····· 31
4장. 단일행 함수 ····· 41
5장. JOIN ····· 58
6장. 그룹함수 ····· 71
7장. 서브 쿼리 I ····· 84
8장. SQL*Plus 변수와 환경설정 ····· 91
9장. 데이터 조작 ····· 103
10장. 테이블 생성 및 관리 ····· 116
11장. 제약조건 ····· 129
12장. 뷰(View)의 작성 ····· 139
13장. 데이터베이스 객체 ····· 146
14장. 사용자 접근 제어 ····· 154
15장. 집합 연산자 ····· 162
16장. Oracle 9i 날짜 함수 ····· 167
17장. ROLLUP과 CUBE ····· 174
18장. 서브쿼리 II ····· 183
19장. 계층형 쿼리 ····· 192
20장. Oracle 9i에서 향상된 DML과 DDL ····· 198
21장. 분석 함수 ····· 205
22장. Oracle 10g에서 향상된 SQL ····· 215
23장. Oracle 10g의 정규표현식 ····· 234

Chapter 1. 관계형 데이터베이스 개념

Oracle 데이터베이스를 배우기에 앞서 먼저 관계형 데이터베이스에 대한 개념을 간략하게 검토해 볼 필요가 있다. 그러나, 관계형 데이터베이스를 이해하기 위해서는 관계형 데이터 모델을 먼저 살펴봐야 하는데 이 책은 관계형 데이터 모델링의 방법론을 기술한 것이 아니므로 데이터 모델링에 대한 학문적 접근은 다루지 않지만, 오라클 데이터베이스를 사용하기 위해서는 필수적으로 관계형 데이터 모델의 구성요소들을 이해하고 개체관계도(ERD : Entity-Relational Diagram)를 분석 할 줄 알아야 한다. 관계형 데이터 모델링은 데이터베이스 시스템 구축에 있어서 기초가 되므로 잘못 설계된 데이터 모델은 추후 데이터베이스 전체를 재구축해야하는 엄청난 결과를 초래하게 될 수도 있다.

관계형 데이터 모델(Relational Data Model)

현재 상용 데이터베이스 제품의 대다수가 채용하고 있는 관계형 데이터 모델의 개념이 처음 언급된 것은 1970년대 E. F. Codd 박사의 "A Relational Model of Data for Large Shared Data Banks"라는 논문이었다. 그 당시, 사용되던 데이터 모델로는 계층형 데이터 모델, 네트워크형 데이터 모델이 있었지만 관계형 데이터 모델의 구조가 기존 데이터 모델에 비해 좀 더 유연하여 실세계를 좀 더 현실감 있게 반영할 수 있었기 때문에 많은 데이터베이스 시스템에 구현되었으며 이로 인하여 관계형 데이터 모델을 지원하는 관계형 데이터베이스 관리 시스템(RDBMS) 제품들이 데이터베이스 시장을 지배하게 되었다.

관계형 데이터 모델은 기본적으로 다음과 같은 핵심적인 3개의 구성요소로 이루어져 있으며, 실세계의 모든 업무체계를 아래의 3가지 구성요소로 모두 표현할 수 있다는 개념이다.

- 개체(Entity) : 시스템화하고자 하는 사건, 사물
- 관계(Relationship) : 개체간, 속성간의 연관성
- 속성(Attribute) : 개체, 관계성의 성질을 나타내는 더 이상 쪼갤 수 없는 정보의 단위

위에서 표현된 각 구성요소들이 결국에는 관계형 데이터베이스에 구현된다. 즉, 개체는 테이블이라는 2차원의 구조로 표현되며, 관계는 외래키(FK : Foreign Key), 속성은 테이블내의 컬럼으로 구축된다. 그러므로, 데이터베이스 시스템을 구축하기에 앞서 모델링하고자 하는 실세계를 개체, 관계, 속성으로 명확히 정의할 필요가 있다.

관계형 데이터베이스의 정의

관계형 데이터 모델을 전산화하여 논리적으로 구축해 놓은 것이 관계형 데이터베이스이다. 관계형 데이터베이스는 정보를 저장하기 위하여 아래와 같은 2차원의 테이블 구조를 사용한다. 예를 들어, 회사의 모든 사원 정보를 데이터베이스에 저장하고자 한다면 다음과 같은 사원 테이블이 필요하게 되는데, 이는 관계형 모델로 정의된 사원 개체가 관계형 데이터베이스 내에서 사원 테이블로 구현된 것이다.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	80/12/17	800		20
7499	ALLEN	SALESMAN	7698	81/02/20	1600	300	30
7521	WARD	SALESMAN	7698	81/02/22	1250	500	30
7566	JONES	MANAGER	7839	81/04/02	2975		20
7654	MARTIN	SALESMAN	7698	81/09/28	1250	1400	30
7698	BLAKE	MANAGER	7839	81/05/01	2850		30
7782	CLARK	MANAGER	7839	81/06/09	2450		10
7788	SCOTT	ANALYST	7566	87/04/19	3000		20
7839	KING	PRESIDENT		81/11/17	5000		10
7844	TURNER	SALESMAN	7698	81/09/08	1500	0	30
7876	ADAMS	CLERK	7788	87/05/23	1100		20
7900	JAMES	CLERK	7698	81/12/03	950		30
7902	FORD	ANALYST	7566	81/12/03	3000		20
7934	MILLER	CLERK	7782	82/01/23	1300		10

그림 1-1. EMP 테이블

관계형 데이터베이스의 용어

관계형 데이터베이스에서 모든 데이터는 2차원 테이블에 저장된다. 다음과 같이 사원 정보가 입력되어 있는 사원 테이블에서 각 번호가 의미하는 것은 다음과 같다.

	②					③		④
	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
	7369	SMITH	CLERK	7902	80/12/17	800		20
	7499	ALLEN	SALESMAN	7698	81/02/20	1600	300	⑤ 30
	7521	WARD	SALESMAN	7698	81/02/22	1250	500	30
	7566	JONES	MANAGER	7839	81/04/02	2975		20
	7654	MARTIN	SALESMAN	7698	81/09/28	1250	1400	30
	7698	BLAKE	MANAGER	7839	81/05/01	2850		30
	7782	CLARK	MANAGER	7839	81/06/09	2450	⑥	10
	7788	SCOTT	ANALYST	7566	87/04/19	3000		20
	7839	KING	PRESIDENT		81/11/17	5000		10
①	7844	TURNER	SALESMAN	7698	81/09/08	1500	0	30
	7876	ADAMS	CLERK	7788	87/05/23	1100		20
	7900	JAMES	CLERK	7698	81/12/03	950		30
	7902	FORD	ANALYST	7566	81/12/03	3000		20
	7934	MILLER	CLERK	7782	82/01/23	1300		10

그림 1-2. 사원 테이블의 구조

① 행(Row) : 특정한 한 명의 사원을 표현하기 위해 필요한 모든 데이터이다. 테이블 내의 각 행은 반드시 기본키에 의하여 식별 가능해야 하며, 기본키의 값은 절대 중복되면 안 된다. 테이블내의 행들은 임의 순서로 저장되어 있으나, 검색시 정렬이 가능하다.

② 기본키(PK : Primay Key) : 사번(EMPNO) 컬럼은 절대 중복 될 수 없으며, 값이 반드시 지정되어야 한다. 그러므로, 테이블내의 모든 행들은 기본키에 의해 식별될 수 있다.

③ 컬럼(Column) : 급여(SAL) 컬럼은 기본키가 아니므로 특별한 제약조건이 지정되지 않는

한, 값을 지정하지 않아도 되며 값이 중복되어도 상관 없다.

④ 외래키(FK : Foreign Key) : 외래키는 테이블간의 관계를 정의하는 컬럼이다. 부서코드(DEPTNO) 컬럼은 외래키로서 자기 자신 또는 다른 테이블의 기본키 또는 고유(Unique) 제약조건이 지정되어 있는 고유키 컬럼들을 참조한다. 여기서는 부서(DEPT) 테이블의 부서코드(DEPTNO) 컬럼을 참조하고 있다.

⑤ 필드(Field) : 행과 컬럼이 교차하는 지점을 필드(Field)라고 정의한다.

⑥ 널(Null) : 필드에 값이 지정되어 있지 않는 경우를 일반적으로 “NULL 값을 갖는다”라고 표현한다. NULL 값 자체도 다음의 두 가지 의미를 갖는데, “값이 아직 지정되지 않았다(Undefine)”는 의미와 “값을 아직 모른다(Unknown)”는 의미이다. 절대, 0(Zero)이나 공백(Space)과는 의미가 다르므로 혼동해서는 안 된다.

기본키(PK)와 외래키(FK)

ERD에서 표현된 개체는 추후 테이블로, 속성은 컬럼으로, 관계는 외래키로 구현된다고 기술한 바 있다. 여기서는 관계형 데이터베이스에서 아주 중요하게 다루어지는 기본키(PK : Primary Key)와 외래키(FK : Foreign Key)에 대하여 살펴본다. 기본키는 다음 그림과 같이 테이블내의 각 행을 유일하게 식별 할 수 있는 컬럼이며 외래키는 자기 자신 또는 다른 테이블의 기본키 또는 고유키들을 참조하는 컬럼이다.

PK 부서 테이블

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

PK 사원 테이블 FK

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	80/12/17	800		20
7499	ALLEN	SALESMAN	7698	81/02/20	1600	300	30
7521	WARD	SALESMAN	7698	81/02/22	1250	500	30
7566	JONES	MANAGER	7839	81/04/02	2975		20
7654	MARTIN	SALESMAN	7698	81/09/28	1250	1400	30
7698	BLAKE	MANAGER	7839	81/05/01	2850		30
7782	CLARK	MANAGER	7839	81/06/09	2450		10
7788	SCOTT	ANALYST	7566	87/04/19	3000		20
7839	KING	PRESIDENT		81/11/17	5000		10
7844	TURNER	SALESMAN	7698	81/09/08	1500	0	30
7876	ADAMS	CLERK	7788	87/05/23	1100		20
7900	JAMES	CLERK	7698	81/12/03	950		30
7902	FORD	ANALYST	7566	81/12/03	3000		20
7934	MILLER	CLERK	7782	82/01/23	1300		10

그림 1-3. 부서 테이블과 사원 테이블

예를 들어, 위 그림과 같이 관계형 데이터베이스에 부서 테이블과 사원 테이블이 구축되어

있다고 가정하자. 부서 테이블에는 부서 정보가 입력되어 있으며, 사원 테이블에는 사원들의 정보가 각각 입력되어 있다. 부서 테이블내에서 각각의 부서 정보를 유일하게 식별할 수 있는 컬럼을 기본키로 설정하게 되는데, 이 경우는 부서코드(DEPTNO)를 기본키로 설정했다. 그러나, 부서 테이블내에 입력된 실제 데이터를 보면 부서코드(DEPTNO) 컬럼 외에 부서명(DNAME), 위치(LOC) 컬럼 모두 기본키가 될 자격(모든 값이 고유하고 NULL 값이 존재하지 않음)은 충분히 있지만 부서명(DNAME)이나 위치(LOC) 컬럼은 추후 NULL 값이 입력되거나 데이터 값이 중복 입력될 소지가 충분히 있으므로 기본키의 후보로서 적절치 않다. 또한, 기본키는 해당 테이블의 대표 속성으로서의 의미를 가지기 때문에 부서코드(DEPTNO)가 기본키로서 적절함을 알 수 있다. 부서 테이블 또한 동일한 맥락에서 사번(EMPNO)이 기본키로 지정되었음을 확인할 수 있다.

외래키는 테이블간의 관계를 정의하는 컬럼으로 위 그림에서는 사원들의 소속부서 정보를 관리하고자 설정된 것이다. 만약, 사원의 소속부서 정보를 굳이 데이터베이스에 구축하여 관리하고자 할 필요가 없다면 이러한 관계 설정은 불필요한 것이 된다. 여기서는 부서 테이블의 기본키 컬럼을 사원 테이블의 외래키 컬럼으로 추가시켜 부서와 사원간의 관계를 상호 참조 할 수 있도록 지정하였다. 그러므로, 각각의 사원들이 어느 부서에 소속되어 있는지, 각각의 부서에 소속된 직원들은 누구인지 등의 정보를 검색할 수 있게 된다.

저자가 데이터 모델링을 강의하고 수강생들이 작성한 데이터 모델을 검토해보면 테이블간의 관계설정에 너무 부담을 갖고 불필요한 관계설정을 하게 되는 경우가 많은데, 반드시 기억해두어야 할 것은 이러한 관계 설정이 정말 필요한 것인지부터 먼저 규명을 해야 된다는 사실이다.

데이터 모델링의 4단계

이 책에서는 데이터 모델링에 대하여 자세히 다루지는 않지만, 개략적인 내용을 소개함으로써 ERD의 이해를 돕고자 한다. 데이터 모델링은 다음 그림과 같이 4단계에 걸쳐서 진행되며, 각 단계별 수행 작업과 산출물은 다음과 같다.

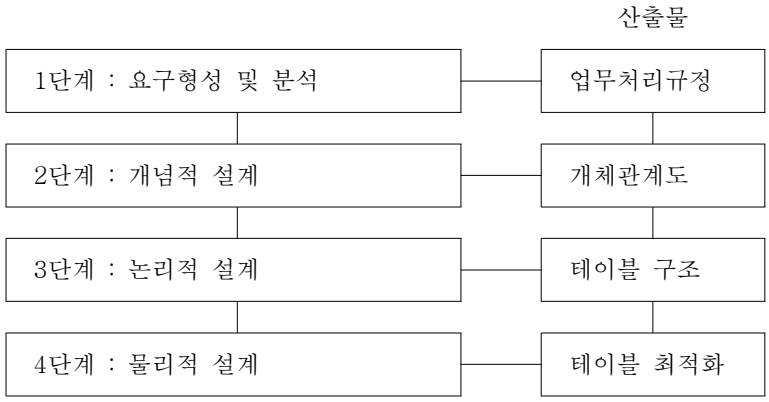


그림 1-4. 데이터 모델링의 4단계

1) 요구형성 및 분석

데이터베이스 시스템 구축을 위한 요구사항을 취합하여 분석한 후, 업무처리 규정을 도출하는 단계이다. 이 단계에서 필요한 자료는 업무분장표, 현행업무흐름도, 입출력 활용장표, 인터뷰 내용, 현 시스템분석도 등이 요구되며 이를 기반으로 최종 업무처리 규정이 산출 된다.

예) 업무처리규정

· 각 회사원은 한 부서에 소속된다.
· 각 부서에는 여러 명의 사원이 배치된다.
· 회사원에 관해서는 사번, 이름, 주소 등의 정보가 유지된다.
· 부서에 관해서는 부서코드, 부서명에 대한 정보가 유지된다.

2) 개념적 설계

앞 단계에서 산출된 업무처리규정을 이용하여 중심 데이터인 개체를 도출하고 개체 관계도(ERD)를 작성하는 과정이다. 데이터베이스를 처음 다루는 입문자에게 개체라는 개념이 상당히 어렵게 느껴질 수도 있으나 관계형 데이터 모델이 최종적으로 관계형 데이터베이스에 구축될 때, 개체가 테이블로 변환된다는 생각을 하면 쉽게 개념을 잡을 수 있을 것이다. 저자가 현업 관리자들을 대상으로 강의를 하다보면 간혹 ERD에 대해 전혀 모르거나, ERD를 제대로 관리하지 않아서 데이터베이스에 어떤 데이터가 있는지 또는 어떤 데이터가 이제 더 이상 필요치 않은지를 파악하지 못하는 경우도 있었다. 그로 인하여, 더 이상 필요하지 않은 수백 메가 분량의 데이터를 많은 시간을 소비해 가면서 여전히 백업을 수행하고 있는 경우도 있었다. 데이터베이스 관리자라면 반드시 ERD를 정기적으로 관리해야 한다.

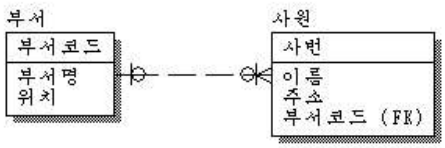


그림 1-5. 개체 관계도 예

3) 논리적 설계

전 단계에서 산출된 ERD로부터 개체를 테이블, 속성을 컬럼, 관계를 외래키로 적절히 변환하는 단계이다. 이 과정에서는 정규화라는 과정이 필수적으로 선행된다. 정규화란 자료의 손실이나 불필요한 정보의 도입 없이 데이터의 일관성, 최소한의 데이터 중복, 최대의 데이터 안정성 확보를 위한 안정적 자료구조로 변환하는 기법을 의미한다, 쉽게 설명하자면 동일한 테이블내에 같은 데이터들이 중복해서 입력이 되면 데이터가 입력, 수정, 삭제되는 조작 과정에서 여러 가지 이상현상이 발생하기 때문에 정규화는 이러한 일들이 발생되지 않도록 데이터 중복을 최소화하여 테이블을 여러 개로 분리시키는 일련의 과정을 말한다. 정규화에 관한 내용은 이 책의 범위를 벗어남으로 별도의 데이터 모델링 관련 서적을 참고하도록 한다.

예) 테이블로의 변환 예
부서 테이블(부서코드(PK), 부서명, 위치)
사원 테이블(사번(PK), 이름, 주소, 부서코드(FK))

4) 물리적 설계
앞 단계에서 변환된 관계형 테이블을 실제 데이터베이스 서버에 구현할 수 있도록 테이블 차트를 작성한다. 또한, 이 단계에서는 서버의 디스크 구조에 따라 데이터 파일의 배치, 인덱스 최적 설계, 트랜잭션 처리 알고리즘 등을 결정하여, 구현된 데이터베이스가 최대한의 성능을 발휘할 수 있도록 구현하는 단계이다.

예) 테이블 차트의 예
테이블명 : DEPT

컬럼명	키	Null/ Unique	체크제약	FK 참조 테이블	FK 참조 컬럼	데이터 타입	길이
DEPTNO	PK	NN, U				VARCHAR2	3
DNAME		NN				VARCHAR2	10
LOC						VARCHAR2	10

테이블명 : EMP

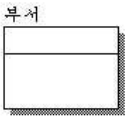
컬럼명	키	Null/ Unique	체크제약	FK 참조 테이블	FK 참조 컬럼	데이터 타입	길이
EMPNO	PK	NN, U				VARCHAR2	5
ENAME		NN				VARCHAR2	10
ADDRESS						VARCHAR2	10
DEPTNO	FK			DEPT	DEPTNO	VARCHAR2	3

ERD(Entity-Relationship Diagram)

ERD는 데이터베이스 모델링의 4단계 중에서 개념적 설계의 산출물로서 도출되어야 하는 결과이다. ERD는 수작업으로 작성할 수도 있으나, 최근의 데이터베이스들은 엄청난 수의 테이블을 포함하고 있기 때문에 ERD를 직접 손으로 작성한다는 것은 거의 불가능하다. 이를 위해서 상용 ERD 작성 툴을 많이 사용하는데, 그 대표적인 소프트웨어가 CA(Computer Associates)의 Erwin이라는 프로그램이다. 이 책에서는 Erwin을 사용하는 방법은 설명하지 않고, 기존의 Erwin으로 작성된 ERD를 분석하는 방법에 대하여 알아보도록 한다. Erwin의 사용방법을 학습하고자 하는 경우에는 Erwin 프로그램의 도움말 파일이나 CA의 홈페이지를 참고하도록 한다.

Erwin에서 관계형 데이터 모델의 기본이 되는 3개의 구성요소는 다음과 같이 표현된다.

■ 개체(Entity)
Erwin에서 개체는 사각형으로 표현되며, 사각형의 상단에 개체의 이름이 기술된다.

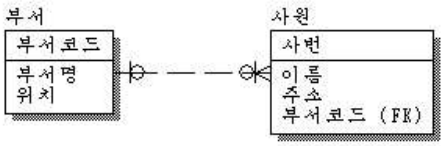


■ 속성(Attribute)

Erwin에서 개체의 속성은 사각형 내부에 기술하는데, 사각형의 위 칸에는 기본키 속성을, 아래 칸에는 일반 속성을 기술한다.

■ 관계(Relationship)

Erwin에서 관계는 개체 간에 실선 또는 점선으로 표시된다.



ERD에서 관계를 표시하는 방법은 관계의 카디널리티(Cardinality)에 따라 다르게 표현된다. 여기서, 카디널리티란 각 관계자에서 참여자의 수를 표현한 것으로, 쉽게 설명하면 한개 부서에 한명의 사원만이 배정되는 관계는 1:1이 되며, 한개 부서에 한명이상의 사원이 배정되면 1:M(Many)이 되는 것이다. M:M의 경우는 한개의 부서에 한명이상의 사원이 배정되며, 한명의 사원이 여러 부서에 동시에 소속될 수 있다는 의미가 된다. 예를 들면, 인사과에 홍길동, 경리과에 콩쥐 등 한개 부서에 한 사람만 배정되는 관계라면 1:1, 인사과에 홍길동, 콩쥐, 팥쥐 등 여러 명의 사원이 배정되는 관계라면 1:M, 인사과에 홍길동, 콩쥐, 팥쥐 등 여러 명의 사원이 배정되는 동시에 홍길동이 인사과, 총무과, 홍보과 등의 여러 부서에 소속되면 M:M 관계가 되는 것이다. 그러나, M:M 관계의 경우는 논리적인 테이블로 구현할 수 없기 때문에 M:M 관계를 테이블로 변환하여 두개의 1:M 관계로 반드시 변환하여야 한다는 점을 잊으면 안 된다.

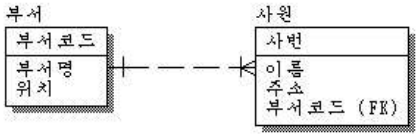
그러면 Erwin에서 이러한 카디널리티가 어떻게 표현되는지 살펴보자. Erwin에서 카디널리티가 1로 표현되는 개체 쪽은 실선으로 그리고, M으로 표현되는 개체 쪽은 까마귀 발(Crow's foot)과 같이 그린다. 위와 같은 경우는 한 개의 부서에 여러 명의 사원이 소속되는 관계를 표현한 것임을 알 수 있다.

또한, 관계로 표현된 점선의 양단에 ○ 기호를 확인 할 수 있는데, 이 기호는 선택도를 표시한 것이다. 즉, ○이 붙어 있는 경우를 선택(Optional), 그렇지 않은 경우를 필수(Mandatory)라고 표현하는데, 사원 개체 쪽에 ○이 붙어 있는 경우는 각 부서에 사원이 한명도 배정되지 않을 수도 있다는 의미이고 반대로, 부서 개체 쪽에 ○이 붙어 있는 경

우는 사원 중에서 어떠한 부서에도 소속되지 않은 사원이 있을 수 있음을 의미한다.

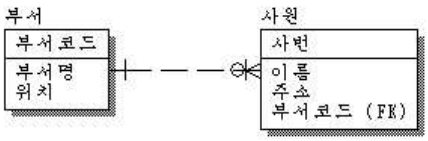
카디널리티와 선택도가 이해되었다면 다음과 같은 ERD를 각각 분석해보도록 하자. ERD를 읽을 때는 왼쪽 개체에서 오른쪽 개체로 한번 읽고, 그 반대로 한번 읽는다.

첫 번째 ERD는 부서와 사원 개체 간의 관계가 1:M이며 선택도를 표현하는 ○이 양쪽 모두 누락된 경우이다.



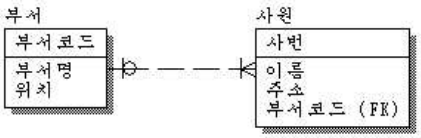
좌 → 우 : 각 부서는 한명 이상의 사원이 반드시 소속된다.(Mandatory)
우 → 좌 : 각 사원은 한개 부서에 반드시 배정된다.(Mandatory)

두 번째 ERD는 첫 번째 ERD에서 사원 개체에 ○이 추가된 경우이다.



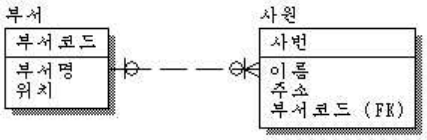
좌 → 우 : 각 부서는 한명 이상의 사원이 소속될 수도 있다.(Optional)
우 → 좌 : 각 사원은 한개 부서에 반드시 배정된다.(Mandatory)

세 번째 ERD는 첫 번째 ERD에서 부서 개체에 ○이 추가된 경우이다.



좌 → 우 : 각 부서는 한명 이상의 사원이 반드시 소속된다.(Mandatory)
우 → 좌 : 각 사원은 한개 부서에 배정될 수도 있다.(Optional)

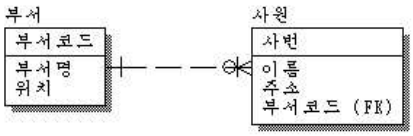
네 번째 ERD는 부서 개체와 사원 개체에 ○이 추가된 경우이다.



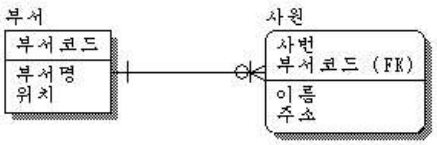
좌 → 우 : 각 부서는 한명 이상의 사원이 소속될 수도 있다.(Optional)
우 → 좌 : 각 사원은 한개 부서에 배정될 수도 있다.(Optional)

마지막으로 카디널리티와 선택도 이외에 한가지 더 알아두어야 할 것이 있는데 관계의

식별성이다. 지금까지 Erwin에서 표현되는 관계는 점선이었는데, 이 점선을 비식별(Non-Identifying) 관계라고 부른다. 비식별 관계로 표시되는 경우는 카디널리티가 1쪽인 개체의 기본키 속성이 M쪽인 사원 개체의 외래키로 구현되는 과정에서 일반 속성으로 추가된다는 의미이다. 즉, 아래 그림과 같이 부서 개체의 기본키인 부서코드 속성이 사원 개체의 외래키로 생성되며 일반 속성으로 추가된다는 의미이다.

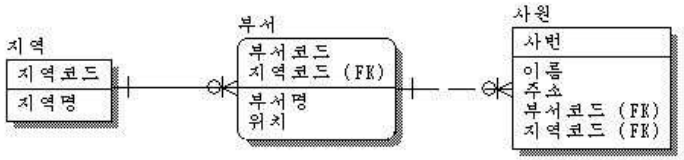


반면에 다음과 같이 실선으로 표시되는 관계를 식별(Identifying) 관계라고 부르는데, 이와 같은 경우, 카디널리티가 1쪽인 개체의 기본키 속성이 M쪽인 사원 개체의 외래키로 구현되는 과정에서 사원의 기본키 속성으로 추가된다는 의미이다. 즉, 아래 그림과 같이 부서 개체의 기본키인 부서코드 속성이 사원 개체의 외래키로 생성되며 기본키 속성으로 추가된다는 의미이다.



식별/비식별 관계는 두개의 개체 간에서는 큰 의미가 없기 때문에 다음과 같은 세 개의 개체로 구성된 식별/비식별 관계를 살펴보도록 하자. 먼저, 지역 개체, 부서 개체, 사원 개체로 구성되어 있으며 각각 1:M 관계로 설정되었다. 즉, 지역 개체의 기본키인 지역코드가 부서 개체의 외래키(일반 속성)로 추가되었고, 부서 개체의 기본키인 부서코드가 사원 개체의 외래키(일반 속성)로 추가되었음을 확인할 수 있다. 이러한 ERD에서 사원의 이름을 검색조건으로 지역명을 검색하고자 한다면 3개의 개체를 모두 조인(Join) 하여야 한다.

반면, 아래와 같이 지역 개체와 부서 개체간의 식별관계로 구성된 경우는 사원 개체에 지역 개체와 부서 개체의 기본키가 각각 포함되므로 지역 개체와 사원 개체가 직접 조인(Join)이 가능해지기 때문에 위의 경우보다 수행성능이 향상될 것임을 예상할 수 있다.



식별/비식별 관계는 구현상 장단점이 있기 때문에 상황에 맞추어 선택해야 한다. 조인에

대한 설명은 뒷 부분에서 다룰 것이므로 당장 이 부분을 이해하지 못하더라도 상관은 없다. 나중에 조인에 대해 학습 한 후, 다시 한번 가볍게 읽어 볼 것을 권장한다.

SQL 문장의 종류

SQL은 Structured Query Language의 줄임말로서 사용자는 SQL 문장을 이용하여 데이터베이스에 접근하고 데이터를 손쉽게 검색 및 조작할 수 있다. 또한, SQL 문장은 영어와 유사하기 때문에 쉽게 배울 수 있는 장점도 있다. Oracle 데이터베이스의 SQL은 산업 표준인 ANSI와 ISO를 준수하고 있기 때문에 다른 상용 데이터베이스 사용자라도 해당 데이터베이스가 관련 표준을 따르고 있다면 Oracle의 SQL 문장을 특별하게 수정할 필요 없이 원문 그대로 사용할 수 있다. 아래 표는 앞으로 설명하게 될 SQL 문장을 용도별로 분류한 것이다.

표 1-1. SQL 문장의 종류

문장	설명
SELECT	데이터베이스로부터 데이터를 검색
INSERT UPDATE DELETE MERGE	데이터베이스 내의 테이블에 새로운 행을 입력하거나, 기존의 행을 수정 또는 삭제하는 명령어로 데이터 조작어(DML : Data Manipulation Language)라고 함
CREATE ALTER DROP RENAME TRUNCATE	테이블을 생성, 변경, 삭제하는 명령어로 데이터 정의어(DDL : Data Definition Language)라고 함
COMMIT ROLLBACK SAVEPOINT	DML 문장에 의한 변경 사항을 관리하거나, 변경사항을 하나의 논리적 트랜잭션으로 포함시키는 명령어
GRANT REVOKE	데이터베이스와 데이터베이스를 구성하는 구조(테이블, 뷰 등)에 접근 권한을 부여하거나 회수하는 명령어로 데이터 제어어(DCL : Data Control Language)라고 함

Oracle 데이터베이스에 접속하는 방법

Oracle 데이터베이스에 접속하는 방법은 설치된 Oracle 데이터베이스의 버전에 따라 약간 다르다. 각 버전 별로 데이터베이스에 접속하는 방법을 살펴본다.

■ Oracle 9i 데이터베이스를 설치한 경우
기존 SQL*Plus를 사용하는 방법과 Apache 웹서버에 의해 제공되는 iSQL*Plus를 브라우저로 접속하는 방법이다.

· 도스 버전의 SQL*Plus에 의한 접속

1. **[시작]-[프로그램]-[보조 프로그램]-[명령 프롬프트]**를 차례로 클릭한다.

2. 명령 프롬프트에서 SQLPLUS SCOTT/TIGER를 입력한다.

3. SQL*Plus를 종료하려면 **EXIT**를 입력한다.

· 윈도우 버전의 SQL*Plus에 의한 접속

1. **[시작]-[프로그램]-[Oracle-OraHome92]-[Application Development]-[SQL Plus]**를 차례로 클릭한다.

2. 로그온 대화상자에서 사용자 이름에는 **SCOTT**, 암호에는 **TIGER**를 입력하고, **확인**을 클릭한다.

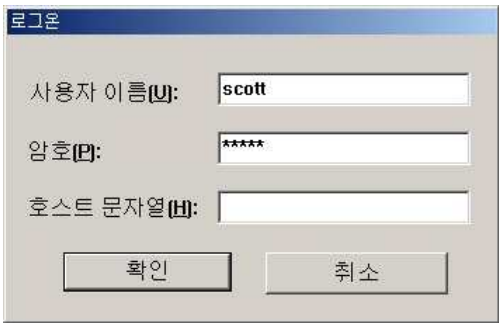


그림 1-6. SQL*Plus 로그온 대화상자

3. 사용자 이름과 암호가 올바르게 입력되었다면 오라클 데이터베이스에 정상적으로 로그온 된다.

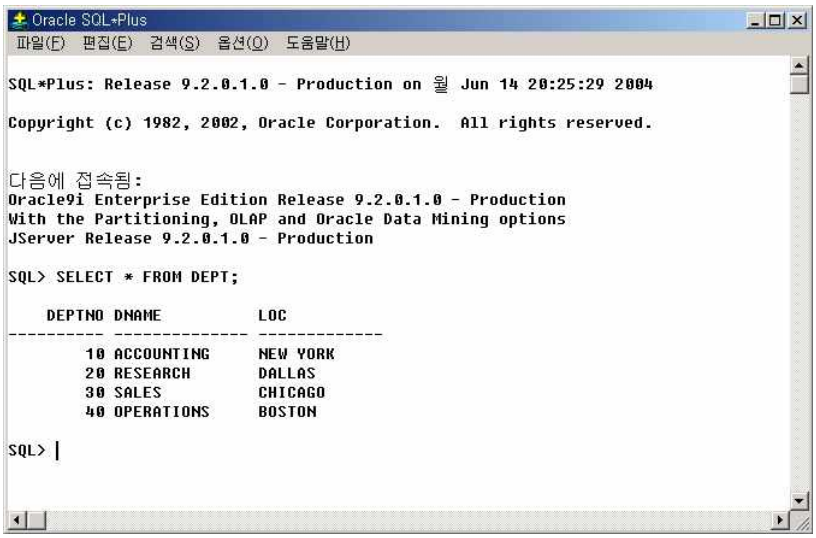


그림 1-7. SQL*Plus 작업환경

4. SQL*Plus를 종료하려면 **EXIT**를 입력한다.

· 브라우저에 의한 접속

1. 브라우저를 실행하고 주소에 **http://서버이름:포트번호/isqlplus**를 입력한다. 여기서, 서버이름은 Oracle 데이터베이스 서버가 설치된 컴퓨터의 호스트명 또는 IP 어드레스를

입력하며, 포트번호는 Oracle과 함께 설치되는 아파치 서버의 리스닝 포트를 기술한다. 아파치 서버의 포트는 아래 파일에서 확인 할 수 있다.

C:₩oracle₩ora92₩Apache₩Apache₩ports.ini

2. iSQL*Plus 로그온 페이지에서 사용자 이름에 SCOTT, 암호에 TIGER를 입력하고 로그인을 클릭한다.

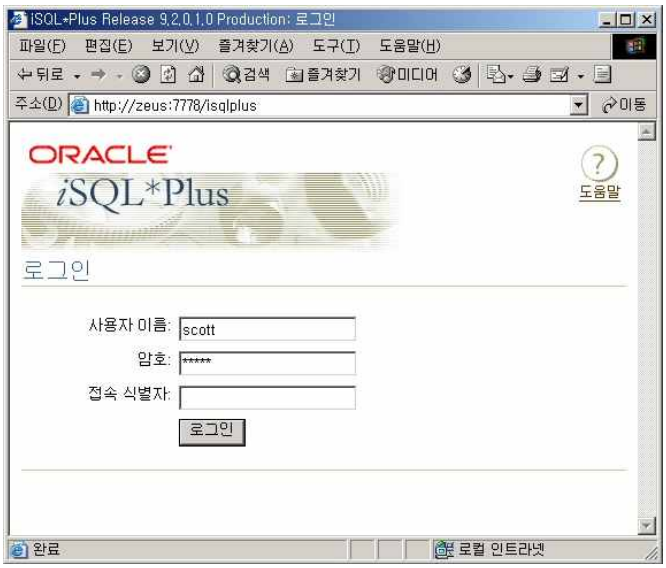


그림 1-8. iSQL*Plus 로그온 대화상자

3. 사용자 이름과 암호가 올바르게 입력되었다면 오라클 데이터베이스에 정상적으로 로그온 된다.

그림 1-9. iSQL*Plus 작업환경

■ Oracle 10g 데이터베이스를 설치한 경우
마찬가지로 첫 번째 방법은 기존 SQL*Plus를 사용하는 방법이고, 두 번째 방법은 브라우저를 이용하여 iSQL*Plus에 접속하는 방법이다. Oracle 10g부터는 Oracle을 설치하는 과정에서 Apache 웹서버가 디폴트로 설치되지 않는 대신에 iSQL*Plus HTTP 서비스가 설치된다.

· 도스 버전의 SQL*Plus에 의한 접속

1. **[시작]-[프로그램]-[보조 프로그램]-[명령 프롬프트]**를 차례로 클릭한다.

2. 명령 프롬프트에서 **SQLPLUS SCOTT/TIGER**를 입력한다.

3. SQL*Plus를 종료하려면 **EXIT**를 입력한다.

<참고>
Oracle 10g부터는 데이터베이스 설치 시, 보안 상의 이유로 인하여 관리자 계정(SYS, SYSTEM)을 제외한 모든 사용자 계정을 잠금 상태로 설정한다. 이런 경우에 명령 프롬프트에서 다음과 같이 관리자로 접속한 다음, 계정의 잠금 상태를 해제하도록 한다.

```
SQLPLUS / AS SYSDBA
SQL> ALTER USER SCOTT
  2  IDENTIFIED BY TIGER
  3  ACCOUNT UNLOCK;
SQL> EXIT
```

· SQL*Plus에 의한 접속

1. **[시작]-[프로그램]-[Oracle-OraDb10g-home1]-[응용 프로그램 개발]-[SQL Plus]**를 차례로 클릭한다.

2. 로그온 대화상자에서 사용자 이름에는 **SCOTT**, 암호에는 **TIGER**를 입력하고, **확인**을 클릭한다.

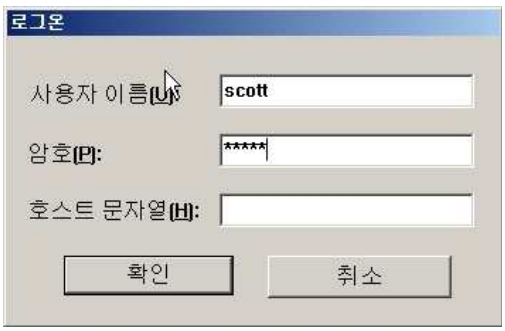


그림 1-10. SQL*Plus 로그온 대화상자

3. 사용자 이름과 암호가 올바르게 입력되었다면 오라클 데이터베이스에 정상적으로 로그온 된다.

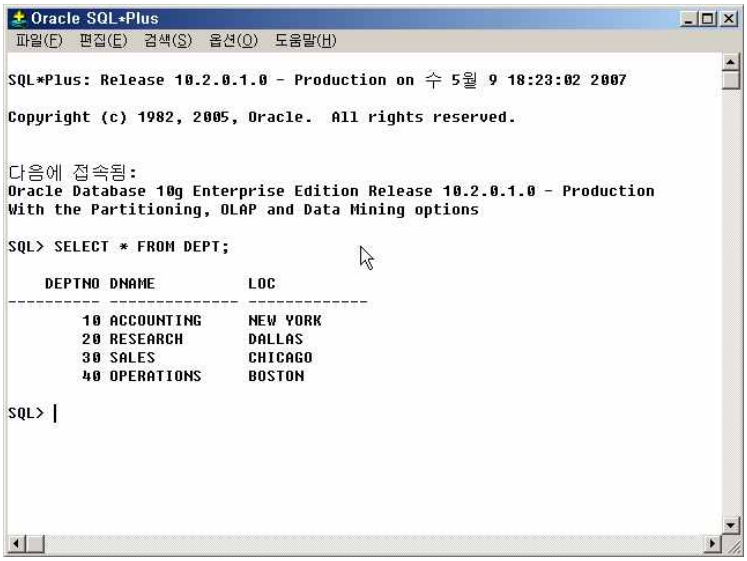


그림 1-11. SQL*Plus 작업환경

4. SQL*Plus를 종료하려면 **EXIT**를 입력한다.

· 브라우저에 의한 접속

1. 브라우저를 실행하고 주소에 **http://서버이름:포트번호/isqlplus**를 입력한다. 여기서, 서버이름은 Oracle 데이터베이스 서버가 설치된 컴퓨터의 호스트명 또는 IP 어드레스를 입력하며, 포트번호는 Oracle 데이터베이스와 함께 설치되는 iSQL*Plus HTTP 서비스의 리스닝 포트를 기술한다. 해당 서비스의 포트는 아래 파일에서 확인 할 수 있다.

   C:₩oracle₩product₩10.2.0₩db_1₩install₩portlist.ini

2. iSQL*Plus 로그온 페이지에서 사용자 이름에 **SCOTT**, 암호에 **TIGER**를 입력하고 로그인을 클릭한다.

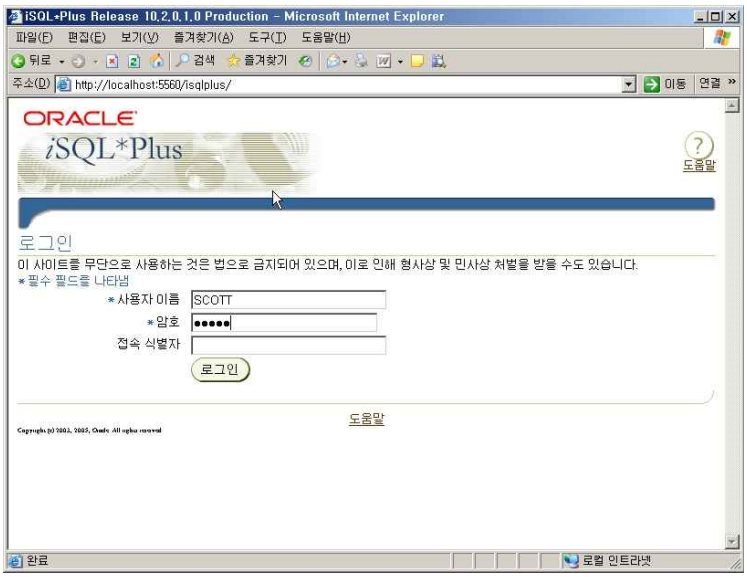


그림 1-12. iSQL*Plus 로그온 대화상자

3. 사용자 이름과 암호가 올바르게 입력되었다면 오라클 데이터베이스에 정상적으로 로그온 된다.

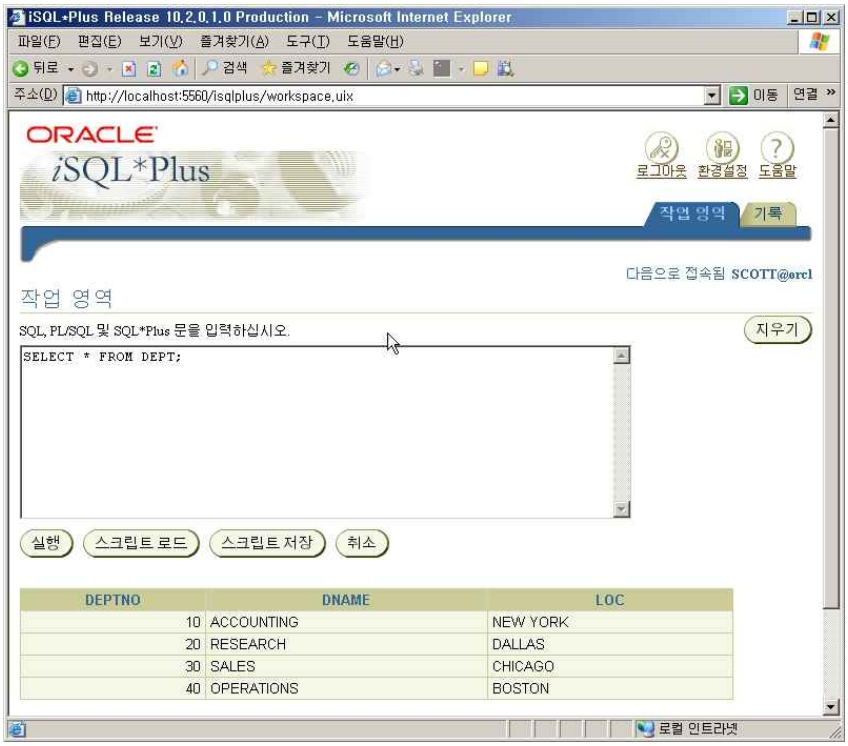


그림 1-13. iSQL*Plus 작업환경

실습에 사용할 계정 및 테이블 목록

이 책에서 실습을 위해 다루고 있는 대표적인 테이블은 다음과 같다.

■ 부서(DEPT) 테이블

부서 테이블은 부서 정보가 입력되어 있는 테이블로서 부서코드(DEPTNO), 부서명(DNAME), 부서위치(LOC)로 구성되어 있다.

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

그림 1-14. DEPT 테이블

■ 사원(EMP) 테이블

사원 테이블은 사원 정보가 입력되어 있는 테이블로서 사번(EMPNO), 사원명(ENAME), 업무(JOB), 관리자사번(MGR), 입사일(HIREDATE), 급여(SAL), 커미션(COMM), 부서코드(DEPTNO)로 구성되어 있다.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	80/12/17	800		20
7499	ALLEN	SALESMAN	7698	81/02/20	1600	300	30
7521	WARD	SALESMAN	7698	81/02/22	1250	500	30
7566	JONES	MANAGER	7839	81/04/02	2975		20
7654	MARTIN	SALESMAN	7698	81/09/28	1250	1400	30
7698	BLAKE	MANAGER	7839	81/05/01	2850		30
7782	CLARK	MANAGER	7839	81/06/09	2450		10
7788	SCOTT	ANALYST	7566	87/04/19	3000		20
7839	KING	PRESIDENT		81/11/17	5000		10
7844	TURNER	SALESMAN	7698	81/09/08	1500	0	30
7876	ADAMS	CLERK	7788	87/05/23	1100		20
7900	JAMES	CLERK	7698	81/12/03	950		30
7902	FORD	ANALYST	7566	81/12/03	3000		20
7934	MILLER	CLERK	7782	82/01/23	1300		10

그림 1-15. EMP 테이블

■ 급여등급(SALGRADE) 테이블

급여등급 테이블은 급여등급이 입력되어 있는 테이블로서 등급(GRADE), 하한값(LOSAL), 상한값(HISAL)으로 구성되어 있다.

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

그림 1-16. SALGRADE 테이블

복습

1. 관계형 데이터 모델의 기본적인 3가지 구성요소는 무엇인가?

2. 데이터 모델링의 4단계는 무엇인가?

3. 개체관계도(ERD)란 무엇인가?

4. 기본키(Primary Key)와 외래키(Foreign Key)를 각각 설명하시오.

Chapter 2. SELECT문 기초

이번 장에서는 테이블의 데이터를 검색하는데 사용되는 SELECT 문장을 살펴보고, SQL 문장과 SQL*Plus간의 동작원리 및 SQL*Plus 자체 명령어를 알아보도록 한다.

SELECT

테이블에서 데이터를 검색하기 위해서는 SELECT 문장을 사용하여야 하며 기본 문법은 다음과 같다.

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

SELECT는 데이터를 검색하는 문장으로 대상 테이블에 어떠한 조작도 수행하지 않는다. 먼저, Oracle 데이터베이스에 SQL*Plus를 이용하여 SCOTT 계정으로 접속한다. 데이터를 검색하기 위해서는 어떤 테이블들이 저장되어 있는지 확인해야 하는데 테이블 목록을 확인하는 명령어는 다음과 같다.

```
SQL> SELECT * FROM TAB;

TNAME                          TABTYPE  CLUSTERID
------------------------------ ------- ----------
BONUS                          TABLE
DEPT                           TABLE
EMP                            TABLE
SALGRADE                       TABLE

SQL>
```

출력 결과를 보면 SCOTT 계정이 사용할 수 있는 객체는 BONUS, DEPT, EMP, SALGRADE이며 모두 TABLE 임을 알 수 있다. 그렇다면 DEPT 테이블에는 어떤 컬럼이 있는지 확인해보자.

```
SQL> DESC DEPT
 이름                                      널?      유형
 ----------------------------------------- -------- ----------------------------
 DEPTNO                                    NOT NULL NUMBER(2)
 DNAME                                              VARCHAR2(14)
 LOC                                                VARCHAR2(13)
```

DEPT 테이블에는 DEPTNO, DNAME, LOC 컬럼이 있으며 해당 컬럼에 NULL 값이 입력될 수 있는지 여부와 각 컬럼의 데이터 타입이 나타나 있다. 여기서, 사용한 DESC 명령은 SELECT와 같은 표준 SQL 문장은 아니므로 다른 데이터베이스에서는 사용할 수 없다. DESC 명령은 SQL*Plus나 iSQL*Plus의 자체 명령어이다.

테이블내의 모든 데이터를 검색하려면 SELECT 뒤에 * 기호를 붙이고 FROM 뒤에 테이블을 기술해준다.

```
SQL> SELECT * FROM DEPT;

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

테이블내의 특정 컬럼만을 검색하려면 SELECT 뒤에 해당 컬럼을 차례로 기술한다. 컬럼을 나열할 때는 쉼표(,)로 구분해야만 한다.

```
SQL> SELECT EMPNO, ENAME, JOB, HIREDATE FROM EMP;

     EMPNO ENAME      JOB       HIREDATE
---------- ---------- --------- --------
      7369 SMITH      CLERK     80/12/17
      7499 ALLEN      SALESMAN  81/02/20
      7521 WARD       SALESMAN  81/02/22
        ...
      7900 JAMES      CLERK     81/12/03
      7902 FORD       ANALYST   81/12/03
      7934 MILLER     CLERK     82/01/23

14 개의 행이 선택되었습니다.
```

SQL 문장의 작성 지침

SQL 문장을 작성할 때, 다음과 같은 규칙과 지침을 준수하면, SQL 문장의 가독성(Readability)이 향상될 뿐 아니라 긴 문장의 디버깅 작업도 수월해진다.

· SQL 문장은 대소문자를 구별하지 않는다.
· SQL 문장은 여러 줄에 걸쳐서 작성할 수 있다.
· 키워드(SELECT, FROM 등)는 줄여 쓸 수 없고, 여러 줄에 걸쳐서 작성할 수 없다.
· 일반적으로 각 키워드는 별도 라인에 작성하도록 한다.
· 각 라인에 들여쓰기를 하면 이해하기가 쉽다.

컬럼의 정렬 방식

SELECT 문장의 실행 결과는 각 컬럼의 데이터 타입에 따라 정렬 형태가 달라진다.

■ iSQL*Plus
컬럼명은 가운데 정렬되며 대문자로 표시되는 반면, 문자 타입과 날짜 타입 컬럼의 값은 좌측 정렬되며 숫자 타입 컬럼의 값은 우측 정렬된다.

EMPNO	ENAME	HIREDATE	SAL
7369	SMITH	80/12/17	800
7499	ALLEN	81/02/20	1600
7521	WARD	81/02/22	1250
7566	JONES	81/04/02	2975
7654	MARTIN	81/09/28	1250
7698	BLAKE	81/05/01	2850
7782	CLARK	81/06/09	2450
7788	SCOTT	87/04/19	3000
7839	KING	81/11/17	5000
7844	TURNER	81/09/08	1500
7876	ADAMS	87/05/23	1100
7900	JAMES	81/12/03	950
7902	FORD	81/12/03	3000
7934	MILLER	82/01/23	1300

그림 2-1. 컬럼의 정렬 방식

■ SQL*Plus

마찬가지로 문자 타입과 날짜 타입 컬럼 값은 좌측 정렬되며 숫자 타입 컬럼 값은 우측 정렬된다. 컬럼명도 같은 방식으로 정렬된다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE, SAL FROM EMP;

     EMPNO ENAME      HIREDATE           SAL
---------- ---------- ---------- ----------
      7369 SMITH      80/12/17           800
      7499 ALLEN      81/02/20          1600
      7521 WARD       81/02/22          1250
        ...
      7900 JAMES      81/12/03           950
      7902 FORD       81/12/03          3000
      7934 MILLER     82/01/23          1300

14 개의 행이 선택되었습니다.
```

산술 연산자

SQL 문장내의 숫자 타입 및 날짜 타입 데이터에는 +, -, *, / 와 같은 산술연산자를 사용할 수도 있다.

```
SQL> SELECT EMPNO, ENAME, SAL * 1.1 FROM EMP;

     EMPNO ENAME      SAL*1.1
---------- ---------- ----------
      7369 SMITH            880
      7499 ALLEN           1760
      7521 WARD            1375
        ...
      7900 JAMES           1045
      7902 FORD            3300
      7934 MILLER          1430

14 개의 행이 선택되었습니다.
```

연산자의 우선순위는 *, / 가 +, - 에 우선하며, 우선순위가 동등할 때는 좌측 연산자부터 우측 연산자로 연산이 진행된다. 우선순위를 강제적으로 지정하려면 괄호를 사용하도록 한다. 아래 SQL 문장들은 연산의 우선순위가 다르기 때문에 그 결과도 서로 다름을 알 수 있다.

```
SQL> SELECT EMPNO, ENAME, 1.1 * SAL + 200 FROM EMP;

     EMPNO ENAME      1.1*SAL+200
---------- ---------- -----------
      7369 SMITH             1080
      7499 ALLEN             1960
        ...
SQL> SELECT EMPNO, ENAME, 1.1 * (SAL + 200) FROM EMP;

     EMPNO ENAME      1.1*(SAL+200)
---------- ---------- -------------
      7369 SMITH              1100
      7499 ALLEN              1980
        ...
```

NULL 값의 정의 및 처리

NULL 이란 이용 불가능한(Unavailable), 지정되지 않은(Unassigned), 알 수 없는(Unknown), 적용할 수 없는(Inapplicable) 값을 의미한다. 한 가지 기억해야 할 것은 NULL은 0(Zero) 또는 공백(Space)과는 다르다는 점이다. 아래 SQL 문장을 수행해보면 값이 표시되지 않는 데이터가 NULL이다.

```
SQL> SELECT EMPNO, ENAME, SAL, COMM FROM EMP;

     EMPNO ENAME             SAL       COMM
---------- ---------- ---------- ----------
      7369 SMITH             800
      7499 ALLEN            1600        300
      7521 WARD             1250        500
      7566 JONES            2975
      7654 MARTIN           1250       1400
        ...
      7902 FORD             3000
      7934 MILLER           1300
```

NULL이 산술연산에 포함되면 그 결과는 연산에 상관없이 무조건 NULL이 된다.

```
SQL> SELECT EMPNO, ENAME, COMM, COMM + 100 FROM EMP;

     EMPNO ENAME            COMM   COMM+100
---------- ---------- ---------- ----------
      7369 SMITH
      7499 ALLEN              300        400
      7521 WARD               500        600
      7566 JONES
        ...
      7902 FORD
      7934 MILLER
```

컬럼명에 별칭 사용

SELECT 문장의 실행 결과를 살펴보면 최상단의 컬럼명은 테이블내의 컬럼명과 동일하게 표시된다. 그러나 컬럼 별칭을 사용하면 최상단의 컬럼명을 별도로 지정한 별칭으로 표시되도록 할 수가 있다. 컬럼 별칭은 다음과 같은 특징을 갖는다.

- 결과 출력시 컬럼명을 변경한다.
- 산술 연산된 컬럼과 같이 사용하면 컬럼명을 보기 좋게 표현할 수 있다.
- SELECT 문장의 컬럼명 뒤에 AS라는 키워드를 붙이고 별칭을 작성하는데, AS는 생략되어도 상관 없다.
- 별칭에 공백이나 특수문자가 포함되거나 대소문자 구분이 필요한 경우 별칭에 " "를 붙인다.

컬럼 별칭을 사용하는 방법은 다음과 같다.

```
SQL> SELECT EMPNO AS 사번, ENAME AS 성명, SAL AS 급여 FROM EMP;

      사번 성명             급여
---------- ---------- ----------
      7369 SMITH             800
      7499 ALLEN            1600
      7521 WARD             1250
        ...
      7900 JAMES             950
      7902 FORD             3000
      7934 MILLER           1300

14 개의 행이 선택되었습니다.
```

AS 키워드는 생략 가능하며, 공백이 포함된 경우는 " "를 붙인다.

```
SQL> SELECT EMPNO AS "사 번", ENAME AS "성 명", SAL * 12 AS "연 봉" FROM EMP;

     사 번 성 명          연 봉
---------- ---------- ----------
      7369 SMITH            9600
      7499 ALLEN           19200
      7521 WARD            15000
        ...
      7900 JAMES           11400
      7902 FORD            36000
      7934 MILLER          15600

14 개의 행이 선택되었습니다.
```

연결 연산자(Concatenation Operator)

연결 연산자는 여러 개의 문자열을 한 개의 문자열로 결합시키는 연산자로서 다음과 같은

특징이 있다.

· 문자 타입 컬럼간 또는 문자 타입 컬럼과 문자열을 연결할 필요가 있을 때 사용한다.
· 컬럼간 또는 컬럼과 문자열 사이에 ||를 입력한다.
· 연결 연산자에 의한 연산 결과는 문자열이다.

```
SQL> SELECT ENAME||JOB FROM EMP;

ENAME||JOB
--------------------
SMITHCLERK
ALLENSALESMAN
WARDSALESMAN
...
JAMESCLERK
FORDANALYST
MILLERCLERK

14 개의 행이 선택되었습니다.
```

리터럴(Literal)

리터럴은 SELECT 문장에 포함된 컬럼명 또는 컬럼별칭 이외의 문자값, 숫자값, 날짜이며 다음과 같은 특징을 갖는다.

· 리터럴은 SELECT 문장 뒤에 기술되는 문자값, 숫자값, 날짜를 의미한다.
· 날짜와 문자 리터럴은 반드시 ‘ ’를 붙여야 한다.
· SELECT문에 포함된 각 리터럴 문자열은 결과 출력시 각각의 행에 한번씩 나타난다.

```
SQL> SELECT ENAME||'의 직급은 '||JOB||'이다' AS "사원별 직급" FROM EMP;

사원별 직급
--------------------------------
SMITH의 직급은 CLERK이다
ALLEN의 직급은 SALESMAN이다
WARD의 직급은 SALESMAN이다
...
JAMES의 직급은 CLERK이다
FORD의 직급은 ANALYST이다
MILLER의 직급은 CLERK이다

14 개의 행이 선택되었습니다.
```

DISTINCT를 이용한 중복 데이터 제거

다음과 같은 SQL 문장을 수행한 결과를 보면 해당 결과가 중복되어 있음을 알 수 있다.

```
SQL> SELECT JOB FROM EMP;

JOB
---------
CLERK
SALESMAN
SALESMAN
MANAGER
SALESMAN
MANAGER
MANAGER
ANALYST
PRESIDENT
SALESMAN
CLERK
CLERK
ANALYST
CLERK

14 개의 행이 선택되었습니다.
```

위 결과는 정상적인 것이지만 JOB 컬럼의 값이 중복되어 있어서 JOB의 종류를 쉽게 파악할 수가 없다. 이러한 경우에 DISTINCT 키워드를 사용하면 중복된 결과를 제거하고 출력하기 때문에 JOB의 종류를 쉽게 확인할 수 있다.

```
SQL> SELECT DISTINCT JOB FROM EMP;

JOB
---------
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

SQL과 SQL*Plus의 상호작용

SQL은 Oracle 서버와 직접 통신하는 명령어로서 사용자가 SQL 문장을 SQL*Plus에 입력하면 SQL 버퍼라는 메모리에 저장되고 새로운 SQL 문장이 입력될 때까지 버퍼에 남아 있게 된다.

SQL*Plus는 사용자가 입력한 SQL 문장을 인식하고 데이터베이스 서버에 해당 문장을 전송하여 사용자가 원하는 작업을 수행되도록 해주는 프로그램이다. 뿐만 아니라, SQL*Plus는 자체 명령어를 다수 포함하고 있다. (예, DESC)

다음은 SQL 문장과 SQL*Plus 명령어의 차이를 정리한 것이다.

표 2-1. SQL 문장과 SQL*Plus 명령어의 차이

SQL	SQL*Plus
데이터 접근을 위해 Oracle 서버와 통신하는 언어	SQL 문장을 인식하여 Oracle 서버에 보내주는 역할
ANSI 표준 SQL에 기초함	SQL 문장을 실행하기 위한 Oracle 인터페이스
데이터베이스 내의 데이터 또는 테이블을 변경함	데이터베이스내의 데이터 조작을 허용하지 않음
SQL 버퍼에 저장됨	한번에 한개의 라인에 입력하며 SQL 버퍼에 저장되지 않음
여러개의 라인을 연결하기 위한 별도의 문자가 필요 없음	"-" 기호를 사용하여 SQL*Plus 명령어의 여러 개의 라인에 걸쳐서 기술할 수 있음
축약될 수 없음	축약될 수 있음
문장 끝에 종료 문자가 필요함	명령어의 끝에 종료 문자가 불필요함
데이터 포맷팅을 위해 별도 기능을 사용	데이터의 포맷팅에 주로 사용

SQL*Plus 명령어

다음은 자주 사용되는 SQL*Plus 명령어이다.

■ 테이블의 구조 확인하기

테이블의 구조를 확인하는 명령어는 DESCRIBE이지만, 축약해서 DESC로 기술해도 무관하다.

```
SQL> DESC EMP
 이름                                      널?      유형
 ----------------------------------------- -------- ----------------
 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                              VARCHAR2(10)
 JOB                                                VARCHAR2(9)
 MGR                                                NUMBER(4)
 HIREDATE                                           DATE
 SAL                                                NUMBER(7,2)
 COMM                                               NUMBER(7,2)
 DEPTNO                                             NUMBER(2)
```

■ SQL*Plus 편집 명령

편집 명령은 버퍼내에 저장된 SQL 문장을 수정하는 명령어이다.

표 2-2. SQL*Plus 명령어 목록

명령	설명
A[PPEND] *text*	버퍼내의 마지막 라인의 끝에 *text*를 추가
C[HANGE] / *old* / *new*	현재 라인에서 *old* 문자열을 *new* 문자열로 교체
C[HANGE] / *text* /	현재 라인에서 *text* 문자열을 삭제
CL[LEAR] BUFF[ER]	버퍼내의 모든 내용을 삭제
DEL	현재 라인을 삭제
I[NPUT]	버퍼내의 현재 라인에 새로운 라인 입력
I[NPUT] *text*	현재 라인에서 *text*를 삽입
L[IST]	버퍼내의 모든 라인을 검색
L[IST] *n*	버퍼내의 *n*번째 라인을 검색
L[IST] *m n*	버퍼내의 *m*~*n*번째 라인을 검색
R[UN]	버퍼 내용을 실행
n	*n*번째 라인을 현재 라인으로 지정
n text	*n*번째 라인을 *text*로 교체
0 text	버퍼의 앞에 *text*를 삽입

아래는 SQL 버퍼내에 저장된 문장을 수정하여 재실행하는 예이다.

```
SQL> SELECT EMPNO, ENAME, JOB, SAL
  2  FROM EMP;

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7369 SMITH      CLERK            800
      7499 ALLEN      SALESMAN        1600
        ...
      7902 FORD       ANALYST         3000
      7934 MILLER     CLERK           1300

14 개의 행이 선택되었습니다.

SQL> L
  1  SELECT EMPNO, ENAME, JOB, SAL
  2* FROM EMP
SQL> 1
  1* SELECT EMPNO, ENAME, JOB, SAL
SQL> A ,COMM
  1* SELECT EMPNO, ENAME, JOB, SAL,COMM
SQL> R
  1  SELECT EMPNO, ENAME, JOB, SAL,COMM
  2* FROM EMP

     EMPNO ENAME      JOB              SAL       COMM
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK            800
      7499 ALLEN      SALESMAN        1600        300
        ...
      7902 FORD       ANALYST         3000
      7934 MILLER     CLERK           1300

14 개의 행이 선택되었습니다.
```

■ SQL*Plus 파일 명령

파일 명령은 SQL*Plus 버퍼 내용을 저장, 불러오기, 편집 등의 작업을 수행하는 명령어이다.

표 2-3. SQL*Plus 파일 명령

명령	설명
SAV[E] *filename*[.*ext*] [REP[LACE]|APP[END]]	버퍼의 내용을 외부 파일로 저장함. REPLACE는 지정된 파일명과 같은 파일이 외부에 존재하면 파일을 교체하며, APPEND는 기존의 파일 뒤에 내용을 추가함. 디폴트 확장자는 sql
GET *filename*[.*ext*]	외부 파일의 내용을 버퍼로 읽음
STA[RT] *filename*[.*ext*]	외부 파일을 실행
@*filename*	외부 파일을 실행(START 명령과 동일)
ED[IT]	버퍼를 수정하기 위해 지정된 편집기를 실행하고 버퍼의 내용을 읽음
ED[IT] [*filename*[.*ext*]]	외부 파일을 수정하기 위해 지정된 편집기를 실행
SPO[OL] [*filename*[.*ext*]] OFF|OUT	SQL*Plus 내의 모든 실행 결과를 외부 파일로 저장함. OFF는 저장을 종료하며, OUT은 마찬가지로 저장을 종료하고 프린터로 전송
EXIT	SQL*Plus를 종료

버퍼 내용을 저장하고 재실행하는 방법은 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB, SAL, COMM FROM EMP;

     EMPNO ENAME      JOB              SAL       COMM
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK            800
      7499 ALLEN      SALESMAN        1600        300
      7521 WARD       SALESMAN        1250        500
        ...
      7900 JAMES      CLERK            950
      7902 FORD       ANALYST         3000
      7934 MILLER     CLERK           1300

14 개의 행이 선택되었습니다.

SQL> SAVE C:WEMP.SQL
file C:WEMP.SQL(이)가 생성되었습니다
SQL> START C:WEMP.SQL

     EMPNO ENAME      JOB              SAL       COMM
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK            800
      7499 ALLEN      SALESMAN        1600        300
      7521 WARD       SALESMAN        1250        500
        ...
      7900 JAMES      CLERK            950
      7902 FORD       ANALYST         3000
      7934 MILLER     CLERK           1300

14 개의 행이 선택되었습니다.
```

화면의 내용을 모두 저장하는 방법은 다음과 같다.

```
SQL> SPOOL C:WEMP.TXT
SQL> SELECT EMPNO, ENAME, JOB, SAL FROM EMP;

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7369 SMITH      CLERK            800
      7499 ALLEN      SALESMAN        1600
      7521 WARD       SALESMAN        1250
      7566 JONES      MANAGER         2975
      7654 MARTIN     SALESMAN        1250
      7698 BLAKE      MANAGER         2850
      7782 CLARK      MANAGER         2450
      7788 SCOTT      ANALYST         3000
      7839 KING       PRESIDENT       5000
      7844 TURNER     SALESMAN        1500
      7876 ADAMS      CLERK           1100
      7900 JAMES      CLERK            950
      7902 FORD       ANALYST         3000
      7934 MILLER     CLERK           1300

14 개의 행이 선택되었습니다.

SQL> SPOOL OFF
```

저장된 EMP.TXT를 확인해보자

그림 2-2. 스풀된 내용 확인

iSQL*Plus

Oracle 9i에서부터 추가된 iSQL*Plus는 과거 SQL*Plus를 대체하는 새로운 도구이다. Oracle 데이터베이스와 함께 설치되는 Apache 웹 서버를 이용하면 사용자 컴퓨터에 SQL*Plus를 일일이 설치하지 않고 운영체제에 포함된 브라우저로 Oracle을 사용할 수 있다. iSQL*Plus는 SQL*Plus와는 달리 버퍼가 존재하지 않으므로 SQL*Plus의 편집명령은

사용할 수 없지만, 직접 iSQL*Plus 화면에서 SQL 문장을 수정 할 수 있다. 또한, SQL*Plus에서 사용하는 파일 명령어도 iSQL*Plus에서는 GUI 기반으로 제공되므로 단순히 해당 버튼을 클릭하여 SQL*Plus에서 수행하던 작업을 거의 유사하게 수행할 수 있다. iSQL*Plus가 GUI 환경으로 편리하기는 하지만 저자와 같이 텍스트 기반의 SQL*Plus 환경에 익숙한 작업자는 여전히 SQL*Plus를 선호하기 때문에 앞으로도 SQL*Plus는 계속 지원될 예정이다.

■ iSQL*Plus에서 스크립트 저장하기

1. iSQL*Plus에 접속하여, 다음과 같이 SQL 문장을 입력하고 **실행** 버튼을 클릭한다.

그림 2-3 iSQL*Plus 작업환경

2. **스크립트 저장** 버튼을 클릭하면, 파일 다운로드 대화상자가 나타나고 **저장** 버튼을 클릭하여 저장할 파일이름을 저장한다.

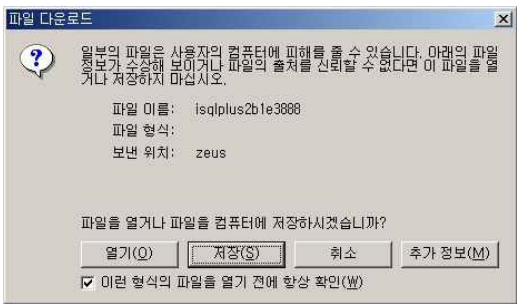


그림 2-4 파일 다운로드 대화상자

■ iSQL*Plus에서 스크립트 불러오기

1. iSQL*Plus에 접속하여, 불러올 파일 이름을 **파일 또는 URL**에 입력한다. 또한, **찾아보기** 버튼을 클릭하여 검색도 가능하다.

그림 2-5. 스크립트 파일 지정

2. **스크립트 로드** 버튼을 클릭하여 파일 내용을 명령문 입력 창으로 불러 올 수 있다.

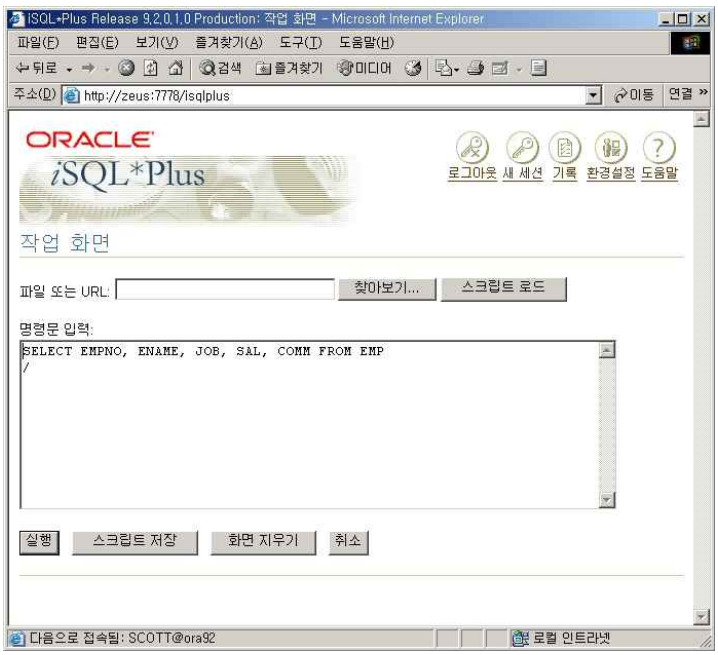


그림 2-6. 스크립트 로드

복습

1. 테이블의 목록을 확인할 수 있는 SQL 문장은 무엇인가?

2. 테이블의 구조를 확인할 수 있는 SQL*Plus 명령어는 무엇인가?

3. 사원 테이블에서 직원들의 연봉(SAL * 12)을 계산하여, 컬럼명은 “연 봉”으로 출력하는 SQL 문장을 작성하라.

4. NULL에 대한 설명 중 올바른 것을 모두 고르시오?
   a. NULL은 공백(space)을 의미한다.
   b. NULL과의 연산은 NULL이다.
   c. NULL은 숫자로 zero(0)을 의미한다.
   d. NULL은 Unknown, Inapplicable의 의미를 갖는다.

5. 사원 테이블을 이용하여 다음과 같은 결과를 얻을 수 있는 SQL 문장을 작성하라.

```
사원정보
------------------------------------------------------------------------
SMITH의 업무는 CLERK이고 급여는 800만원입니다
ALLEN의 업무는 SALESMAN이고 급여는 1600만원입니다
WARD의 업무는 SALESMAN이고 급여는 1250만원입니다
JONES의 업무는 MANAGER이고 급여는 2975만원입니다
MARTIN의 업무는 SALESMAN이고 급여는 1250만원입니다
BLAKE의 업무는 MANAGER이고 급여는 2850만원입니다
CLARK의 업무는 MANAGER이고 급여는 2450만원입니다
SCOTT의 업무는 ANALYST이고 급여는 3000만원입니다
KING의 업무는 PRESIDENT이고 급여는 5000만원입니다
TURNER의 업무는 SALESMAN이고 급여는 1500만원입니다
ADAMS의 업무는 CLERK이고 급여는 1100만원입니다
JAMES의 업무는 CLERK이고 급여는 950만원입니다
FORD의 업무는 ANALYST이고 급여는 3000만원입니다
MILLER의 업무는 CLERK이고 급여는 1300만원입니다

14 개의 행이 선택되었습니다.
```

Chapter 3. WHERE 및 ORDER BY

지난 장에서는 SELECT 문장을 이용하여 테이블의 모든 행을 검색할 수 있었다. 이번 장에서는 테이블 내의 모든 행을 검색하는 대신 검색 조건을 지정하여 사용자가 원하는 행들만 검색할 수 있는 방법을 살펴보고 출력 결과를 특정 컬럼 기준으로 정렬하는 방법을 알아본다.

WHERE

데이터를 검색할 때 WHERE 조건을 지정하면, 조건에 만족하는 행들만 검색된다.

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

WHERE 구문 뒤에는 원하는 검색 조건을 지정한다. 예를 들어, 사원 테이블에서 부서코드가 30번인 사원들의 사번, 사원명, 업무, 부서코드를 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB, DEPTNO
  2  FROM EMP
  3  WHERE DEPTNO = 30;

     EMPNO ENAME      JOB           DEPTNO
---------- ---------- --------- ----------
      7499 ALLEN      SALESMAN          30
      7521 WARD       SALESMAN          30
      7654 MARTIN     SALESMAN          30
      7698 BLAKE      MANAGER           30
      7844 TURNER     SALESMAN          30
      7900 JAMES      CLERK             30

6 개의 행이 선택되었습니다.
```

WHERE 절에서 문자열과 날짜를 조건에 기술할 때는 반드시 ' '를 붙여야하며, 대소문자 구분이 되기 때문에 주의하여야 한다. 날짜 타입 데이터는 반드시 RR/MM/DD 형식으로 기술하여야 날짜 데이터로 올바르게 변환된다.

사원 테이블에서 사원명이 'KING'인 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB, DEPTNO
  2  FROM EMP
  3  WHERE ENAME = 'KING';

     EMPNO ENAME      JOB           DEPTNO
---------- ---------- --------- ----------
      7839 KING       PRESIDENT         10
```

문자열은 대소문자 구별이 되므로 원하는 결과를 얻기 위해서는 정확히 기술하여야 한다.

```
SQL> SELECT EMPNO, ENAME, JOB, DEPTNO
  2  FROM EMP
  3  WHERE ENAME = 'King';

선택된 레코드가 없습니다.
```

사원 테이블에서 입사일이 1981년 11월 17인 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB, HIREDATE
  2  FROM EMP
  3  WHERE HIREDATE='81/11/17'

     EMPNO ENAME      JOB       HIREDATE
---------- ---------- --------- ----------
      7839 KING       PRESIDENT 81/11/17
```

비교 연산자

WHERE 조건에 사용할 수 있는 비교 연산자는 다음과 같다.

표 3-1 비교 연산자 1

연산자	의미
=	같다
>	보다 크다
>=	보다 크거나 같다
<	보다 작다
<=	보다 작거나 같다
<>	다르다

여기서, != 와 ^= 는 <> 와 동일한 의미를 같는다.

사원 테이블에서 급여가 1000 이하인 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE SAL <= 1000;

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800
      7900 JAMES             950
```

다음은 일반적인 연산자는 아니지만 자주 사용되는 연산자들이다.

표 3-2. 비교 연산자 2

연산자	의미
BETWEEN ... AND ...	두 값의 범위에 포함되는
IN (set)	괄호 안의 값과 일치하는
LIKE	문자의 조합이 같은
IS NULL	널 값

■ BETWEEN ... AND ...

사원 테이블에서 급여가 1000 이상이고 2000 이하인 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE SAL BETWEEN 1000 AND 2000;

     EMPNO ENAME             SAL
---------- ---------- ----------
      7499 ALLEN            1600
      7521 WARD             1250
      7654 MARTIN           1250
      7844 TURNER           1500
      7876 ADAMS            1100
      7934 MILLER           1300
```

BETWEEN을 사용 할 때 주의할 점은 하한값이 상한값보다 먼저 기술되어야 한다는 점과 하한 및 상한값이 모두 조건에 포함된다는 것이다. 위 문장은 다음 문장과 동일한 결과를 보여준다.

```
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE SAL >= 1000 AND SAL <= 2000;
```

■ IN (set)

사원 테이블에서 사번이 7876, 7844, 7839인 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB
  2  FROM EMP
  3  WHERE EMPNO IN (7839, 7844, 7876);

     EMPNO ENAME      JOB
---------- ---------- ---------
      7876 ADAMS      CLERK
      7844 TURNER     SALESMAN
      7839 KING       PRESIDENT
```

위 문장은 아래 문장과 동일한 결과를 보여준다.

```
SQL> SELECT EMPNO, ENAME, JOB
  2  FROM EMP
  3  WHERE EMPNO = 7839 OR EMPNO = 7844 OR EMPNO = 7876;
```

■ LIKE

검색하고자 하는 문자열을 정확히 알 수 없는 경우, LIKE 연산자와 패턴 매치 문자열을 이용하면 와일드카드(Wildcard) 탐색 방식을 수행 할 수 있다. 다음은 패턴 매치 문자열에서 사용되는 기호를 나타낸다.

표 3-3. 패턴 매치 문자열에서 사용되는 기호

기호	설명
%	0 글자 이상의 임의 문자를 대표한다.
_	1 글자의 임의 문자를 대표한다

사원 테이블에서 사원명이 'A'로 시작하는 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB
  2  FROM EMP
  3  WHERE ENAME LIKE 'A%';

     EMPNO ENAME      JOB
---------- ---------- ---------
      7499 ALLEN      SALESMAN
      7876 ADAMS      CLERK
```

사원 테이블에서 사원명이 'N'으로 끝나는 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB
  2  FROM EMP
  3  WHERE ENAME LIKE '%N';

     EMPNO ENAME      JOB
---------- ---------- ---------
      7499 ALLEN      SALESMAN
      7654 MARTIN     SALESMAN
```

사원 테이블에서 사원명에 'T'가 포함된 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB
  2  FROM EMP
  3  WHERE ENAME LIKE '%T%';

     EMPNO ENAME      JOB
---------- ---------- ---------
      7369 SMITH      CLERK
      7654 MARTIN     SALESMAN
      7788 SCOTT      ANALYST
      7844 TURNER     SALESMAN
```

패턴 매치열에 %와 _를 같이 사용할 수도 있다.

```
SQL> SELECT EMPNO, ENAME, JOB
  2  FROM EMP
  3  WHERE ENAME LIKE '_L%';

     EMPNO ENAME      JOB
---------- ---------- ---------
      7499 ALLEN      SALESMAN
      7698 BLAKE      MANAGER
      7782 CLARK      MANAGER
```

참고로 검색하고자 하는 컬럼에 패턴매치 기호 %, _가 포함되어 있고, 이러한 기호를 검

색하고자 하는 경우는 ESCAPE 옵션을 사용하면 된다. 예를 들어, JOB 컬럼의 데이터에 _이 포함되어 있다고 가정하고 _가 포함된 행을 검색하려면 다음과 같이 하면 된다.

```
SQL> SELECT EMPNO, ENAME, JOB
  2  FROM EMP
  3  WHERE ENAME LIKE '%W_%' ESCAPE 'W';
```

■ NULL

사원 테이블에서 커미션이 NULL인 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, COMM
  2  FROM EMP
  3  WHERE COMM IS NULL;

     EMPNO ENAME            COMM
---------- ---------- ----------
      7369 SMITH
      7566 JONES
      7698 BLAKE
      7782 CLARK
      7788 SCOTT
      7839 KING
      7876 ADAMS
      7900 JAMES
      7902 FORD
      7934 MILLER

10 개의 행이 선택되었습니다.
```

위와 같은 경우, 절대 아래와 같이 작성하면 원하는 결과를 얻을 수 없다.

```
SQL> SELECT EMPNO, ENAME, COMM
  2  FROM EMP
  3  WHERE COMM = NULL;

선택된 레코드가 없습니다.
```

사원 테이블에서 커미션이 NULL이 아닌 경우를 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, COMM
  2  FROM EMP
  3  WHERE COMM IS NOT NULL;

     EMPNO ENAME            COMM
---------- ---------- ----------
      7499 ALLEN             300
      7521 WARD              500
      7654 MARTIN           1400
      7844 TURNER              0
```

논리 연산자

논리 연산자는 WHERE 절에 부여할 조건이 여러 개인 경우, 조건을 결합시키는데 사용한

다. 논리 연산자의 종류는 다음과 같다.

표 3-4. 논리 연산자

연산자	의미
AND	두개의 조건이 TRUE이면 TRUE를 리턴
OR	두개의 조건중 하나의 조건이 TRUE이면 TRUE를 리턴
NOT	조건이 FALSE이면 TRUE를 리턴

사원 테이블에서 업무가 'SALESMAN'이고 급여가 1500이상인 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB, SAL
  2  FROM EMP
  3  WHERE JOB = 'SALESMAN'
  4  AND SAL >= 1500;

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7499 ALLEN      SALESMAN         1600
      7844 TURNER     SALESMAN         1500
```

사원 테이블에서 업무가 'SALESMAN' 또는 급여가 1500이상인 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB, SAL
  2  FROM EMP
  3  WHERE JOB = 'SALESMAN'
  4  OR SAL >= 1500

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7499 ALLEN      SALESMAN         1600
      7521 WARD       SALESMAN         1250
      7566 JONES      MANAGER          2975
      7654 MARTIN     SALESMAN         1250
      7698 BLAKE      MANAGER          2850
      7782 CLARK      MANAGER          2450
      7788 SCOTT      ANALYST          3000
      7839 KING       PRESIDENT        5000
      7844 TURNER     SALESMAN         1500
      7902 FORD       ANALYST          3000
```

사원 테이블에서 업무가 'SALESMAN', 'ANALYST', 'MANAGER' 가 아닌 사원을 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB, SAL
  2  FROM EMP
  3  WHERE JOB NOT IN ('SALESMAN', 'ANALYST', 'MANAGER');

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7369 SMITH      CLERK            800
      7839 KING       PRESIDENT       5000
      7876 ADAMS      CLERK           1100
      7900 JAMES      CLERK            950
      7934 MILLER     CLERK           1300
```

논리 연산자는 NOT, AND, OR 순의 우선순위가 있으며 우선순위를 임의로 지정하고자 하는 경우에는 괄호를 사용하면 된다. 다음 예제를 보고 어떤 조건이 먼저 적용될 것인지를 생각해보자.

```
SQL> SELECT EMPNO, ENAME, JOB, SAL
  2  FROM EMP
  3  WHERE JOB = 'SALESMAN'
  4  OR JOB = 'MANAGER'
  5  AND SAL >= 2000;

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7499 ALLEN      SALESMAN        1600
      7521 WARD       SALESMAN        1250
      7566 JONES      MANAGER         2975
      7654 MARTIN     SALESMAN        1250
      7698 BLAKE      MANAGER         2850
      7782 CLARK      MANAGER         2450
      7844 TURNER     SALESMAN        1500

7 개의 행이 선택되었습니다.
```

위의 경우는 논리 연산자의 우선순위에 따라 다음과 같이 연산된다.

```
SQL> SELECT EMPNO, ENAME, JOB, SAL
  2  FROM EMP
  3  WHERE JOB = 'SALESMAN'
  4  OR (JOB = 'MANAGER'
  5  AND SAL >= 2000);
```

만약, OR 조건부터 적용된 후 AND 조건이 적용되도록 하려면 다음과 같이 하면 된다.

```
SQL> SELECT EMPNO, ENAME, JOB, SAL
  2  FROM EMP
  3  WHERE (JOB = 'SALESMAN'
  4  OR JOB = 'MANAGER')
  5  AND SAL >= 2000;

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7566 JONES      MANAGER         2975
      7698 BLAKE      MANAGER         2850
      7782 CLARK      MANAGER         2450
```

ORDER BY

SELECT 문장에 의해 검색된 결과를 정렬하려면 ORDER BY 문장을 추가하면 된다.

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

ORDER BY 구문 뒤에는 정렬의 기준이 되는 컬럼을 기술하고 정렬방식을 지정하는데, 오름차순으로 정렬을 하려면 ASC, 내림차순으로 정렬을 하려면 DESC를 기술하면 된다. 지정하지 않으면 ASC가 생략된 것으로 오름차순으로 정렬된다.

사원 테이블의 사번, 사원명, 입사일을 입사일이 빠른 순서로 정렬하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE
  2  FROM EMP
  3  ORDER BY HIREDATE;

     EMPNO ENAME      HIREDATE
---------- ---------- --------
      7369 SMITH      80/12/17
      7499 ALLEN      81/02/20
      7521 WARD       81/02/22
      7566 JONES      81/04/02
        ...

14 개의 행이 선택되었습니다.
```

사원 테이블의 사번, 사원명, 입사일을 입사일이 늦은 순서로 정렬하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE
  2  FROM EMP
  3  ORDER BY HIREDATE DESC;

     EMPNO ENAME      HIREDATE
---------- ---------- --------
      7876 ADAMS      87/05/23
      7788 SCOTT      87/04/19
      7934 MILLER     82/01/23
      7900 JAMES      81/12/03
        ...

14 개의 행이 선택되었습니다.
```

컬럼 별칭을 ORDER BY 구문 뒤에 기술할 수도 있다.

```
SQL> SELECT EMPNO, ENAME, SAL * 12 ANNUAL
  2  FROM EMP
  3  ORDER BY ANNUAL;

     EMPNO ENAME          ANNUAL
---------- ---------- ----------
      7369 SMITH            9600
      7900 JAMES           11400
      7876 ADAMS           13200
      7521 WARD            15000
        ...

14 개의 행이 선택되었습니다.
```

또한, SELECT 구문 뒤의 컬럼 순번을 대신 기술할 수도 있다.

```
SQL> SELECT EMPNO, ENAME, SAL * 12 ANNUAL
  2  FROM EMP
  3  ORDER BY 3;

     EMPNO ENAME          ANNUAL
---------- ---------- ----------
      7369 SMITH            9600
      7900 JAMES           11400
      7876 ADAMS           13200
      7521 WARD            15000
        ...

14 개의 행이 선택되었습니다.
```

사원 테이블의 사번, 사원명, 급여를 급여가 높은 순서대로 정렬하고, 급여가 같은 사원의 경우에는 사번이 빠른 순서로 정렬하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  ORDER BY SAL DESC, EMPNO;

     EMPNO ENAME             SAL
---------- ---------- ----------
      7839 KING             5000
      7788 SCOTT            3000
      7902 FORD             3000
      7566 JONES            2975
      7698 BLAKE            2850
      7782 CLARK            2450
      7499 ALLEN            1600
      7844 TURNER           1500
      7934 MILLER           1300
      7521 WARD             1250
      7654 MARTIN           1250
      7876 ADAMS            1100
      7900 JAMES             950
      7369 SMITH             800

14 개의 행이 선택되었습니다.
```

위 문장에서 사번에는 ASC가 생략되었음을 주의해야 한다.

복습

1. 사원 테이블에서 입사일이 81년도인 직원의 사번, 사원명, 입사일, 업무, 급여를 검색하시오.

2. 사원 테이블에서 입사일이 81년이고 업무가 'SALESMAN'이 아닌 직원의 사번, 사원명, 입사일, 업무, 급여를 검색하시오.

3. 사원 테이블의 사번, 사원명, 입사일, 업무, 급여를 급여가 높은 순으로 정렬하고, 급여가 같으면 입사일이 빠른 사원으로 정렬하시오.

4. 사원 테이블에서 사원명의 세 번째 알파벳이 'N'인 사원의 사번, 사원명을 검색하시오.

5. 사원 테이블에서 연봉(SAL*12)이 35000 이상인 사번, 사원명, 연봉을 검색하시오.

Chapter 4. 단일행 함수

함수는 SQL 문장을 좀 더 강력하게 만들어 준다. 이 장에서는 함수 중에서 단일행 함수인 문자 함수, 숫자 함수, 날짜 함수, 데이터 타입 변환 함수에 대하여 살펴보고 이러한 함수들을 SELECT 문장에서 사용하는 방법을 알아본다.

함수의 종류

함수에는 단일행 함수(Single-row function)와 집계 함수라고 부르는 복수행 함수(Multiple-row function)의 두 종류가 있다. SQL 문장에서 단일행 함수는 모든 행에 대하여 각각 적용되어 행의 개수와 동일한 개수의 결과를 리턴해주는 반면, 집계 함수는 검색되는 모든 행에 한번만 적용되고 한건의 결과를 리턴한다. 이번 장에서는 단일행 함수에 대하여 설명하고 집계 함수는 추후에 살펴보기로 한다.

단일행 함수는 다음과 같은 기능이 있다.
· 데이터를 조작한다.
· 인자들을 입력받아 하나의 결과를 리턴한다.
· 각각의 행에 대하여 적용된다.
· 각 행별 하나의 결과를 리턴한다.
· 함수의 결과로서 데이터의 타입이 변경될 수도 있다.
· 함수의 중첩이 가능하다.
· 함수의 인자는 컬럼 또는 수식이 될 수 있다.

단일행 함수는 다음과 같은 종류가 있다.
· 문자 함수
· 숫자 함수
· 날짜 함수
· 변환 함수
· 일반 함수

문자 함수

문자 함수는 문자 타입의 컬럼 또는 값을 입력으로 받아서 그 결과로서 문자 타입의 값 또는 숫자 타입의 값을 리턴해주는 함수이다. 문자 함수는 대소문자 조작 함수와 문자열 조작 함수의 두 종류이며 간략하게 정리하면 다음과 같다.

표 4-1. 대소문자 조작 함수

함수	설명
LOWER(*column*| *expression*)	영문자를 소문자로 변환
UPPER(*column*| *expression*)	영문자를 대문자로 변환
INITCAP(*column*| *expression*)	영문자열의 첫 번째 문자를 대문자로 변환
CONCAT(*column1*| *expression1*, *column2*| *expression2*)	문자열을 결합, ||의 결과와 동등
SUBSTR(*column*| *expression*, *m* [,*n*])	입력된 문자열의 *m*번째 문자부터 *n*개의 문자열을 추출

■ LOWER

LOWER는 입력된 문자열을 소문자로 변환하여 리턴한다.

```
SQL> SELECT EMPNO, ENAME, LOWER(ENAME)
  2  FROM EMP
  3  WHERE EMPNO = 7369;

     EMPNO ENAME      LOWER(ENAM
---------- ---------- ----------
      7369 SMITH      smith
```

■ UPPER

UPPER는 입력된 문자열을 대문자로 변환하여 리턴한다.

```
SQL> SELECT EMPNO, ENAME, LOWER(ENAME), UPPER(ENAME)
  2  FROM EMP
  3  WHERE EMPNO = 7369;

     EMPNO ENAME      LOWER(ENAM UPPER(ENAM
---------- ---------- ---------- ----------
      7369 SMITH      smith      SMITH
```

단일행 함수는 SELECT 구문 뒤에만 기술할 수 있는 것이 아니라 WHERE, ORDER BY 등 구문에도 기술할 수 있다.

```
SQL> SELECT EMPNO, ENAME
  2  FROM EMP
  3  WHERE ENAME = UPPER('smith');

     EMPNO ENAME
---------- ----------
      7369 SMITH
```

■ INITCAP

INITCAP은 입력된 문자열의 첫 번째 문자를 대문자로 변환하여 리턴한다.

```
SQL> SELECT EMPNO, ENAME, INITCAP(ENAME)
  2  FROM EMP
  3  WHERE EMPNO = 7369;

     EMPNO ENAME      INITCAP(EN
---------- ---------- ----------
      7369 SMITH      Smith
```

■ CONCAT

입력된 문자열을 결합하여 리턴한다.

```
SQL> SELECT EMPNO, ENAME, JOB, CONCAT(ENAME, JOB)
  2  FROM EMP
  3  WHERE EMPNO = 7369;

     EMPNO ENAME      JOB       CONCAT(ENAME,JOB)
---------- ---------- --------- ------------------
      7369 SMITH      CLERK     SMITHCLERK
```

■ SUBSTR

입력된 문자열에서 지정된 문자열을 추출한다.

```
SQL> SELECT EMPNO, JOB, SUBSTR(JOB, 6, 3)
  2  FROM EMP
  3  WHERE EMPNO = 7499;

     EMPNO JOB       SUBSTR
---------- --------- ------
      7499 SALESMAN  MAN
```

표 4-2. 문자열 조작 함수

함수	설명
LENGTH(*column*| *expression*)	입력된 문자열의 전체 문자 개수를 리턴
INSTR(*column*| *expression*, '*string*', [,*m*], [*n*])	입력된 문자열의 *m*번째 문자부터 '*string*'이 *n*번째 나오는 위치를 리턴, *m*의 디폴트 값은 1
LPAD(*column*| *expression*, *n*, '*string*') RPAD(*column*| *expression*, *n*, '*string*')	전체 문자의 개수가 *n*개가 되도록 입력된 문자열의 왼쪽에 '*string*'을 추가 전체 문자의 개수가 *n*개가 되도록 입력된 문자열의 오른쪽에 '*string*'을 추가
TRIM(LEADING|TRAILING|BOTH *trim_character* FROM *trim_source*)	*trim_source*에서 *trim_character*를 제거
REPLACE(*text*, *search_string*, *replacement_string*)	*text*에서 *search_string*을 *replacement_string*으로 교체

■ LENGTH

입력된 문자열의 전체 문자 개수를 리턴한다.

```
SQL> SELECT EMPNO, ENAME, LENGTH(ENAME)
  2  FROM EMP
  3  WHERE EMPNO = 7499;

     EMPNO ENAME      LENGTH(ENAME)
---------- ---------- -------------
      7499 ALLEN                 5
```

■ INSTR

입력된 문자열에서 특정 문자열의 위치를 리턴한다.

```
SQL> SELECT EMPNO, JOB, INSTR(JOB, 'A', 1, 2)
  2  FROM EMP
  3  WHERE EMPNO = 7844;

    EMPNO JOB       INSTR(JOB,'A',1,2)
--------- --------- ------------------
     7844 SALESMAN                   7
```

■ LPAD / RPAD

주어진 문자의 좌측 / 우측에 특정 문자를 추가한다.

```
SQL> SELECT EMPNO, SAL, LPAD(SAL, 6, '*'), RPAD(SAL, 6, '*')
  2  FROM EMP;

    EMPNO        SAL LPAD(SAL,6,' RPAD(SAL,6,'
--------- ---------- ------------ ------------
     7369        800 ***800       800***
     7499       1600 **1600       1600**
     7521       1250 **1250       1250**
     7566       2975 **2975       2975**
       ...
```

■ TRIM

주어진 문자열의 좌측 / 우측에서 특정 문자를 제거한다.

```
SQL> SELECT EMPNO, JOB,
  2   TRIM(LEADING 'S' FROM JOB) AS LEADING,
  3   TRIM(TRAILING 'N' FROM JOB) AS TRAILING
  4  FROM EMP
  5  WHERE EMPNO = 7844

    EMPNO JOB       LEADING   TRAILING
--------- --------- --------- ---------
     7844 SALESMAN  ALESMAN   SALESMA
```

■ REPLACE

주어진 문자열에서 특정 문자열을 주어진 문자열로 교체한다.

```
SQL> SELECT EMPNO, JOB, REPLACE(JOB, 'MAN', 'PERSON')
  2  FROM EMP
  3  WHERE EMPNO = 7844;

    EMPNO JOB       REPLACE(JOB,'MAN','PERSON')
--------- --------- ---------------------------------------------------
     7844 SALESMAN  SALESPERSON
```

숫자 함수

숫자 함수는 입력으로 숫자 타입의 컬럼 또는 값을 받아서 결과로 숫자 타입의 값을 리턴하는 함수이다. 숫자 함수의 종류는 다음과 같다.

표 4-3. 숫자 함수의 종류

함수	설명
ROUND(column|expression, n)	입력된 숫자를 반올림하여 소수점 n자리로 리턴, n이 음수인 경우 정수 자리에서 반올림
TRUNC(column|expression, n)	입력된 숫자를 내림하여 소수점 n자리로 리턴
MOD(m, n)	m을 n으로 나눈 나머지 리턴

■ ROUND

ROUND는 입력 값을 지정된 자리에서 반올림하는 함수이다.

```
SQL> SELECT ROUND(45.923, 2), ROUND(45.923, 0),
  2  ROUND(45.923, -1)
  3  FROM DUAL;

ROUND(45.923,2) ROUND(45.923,0) ROUND(45.923,-1)
--------------- --------------- ----------------
          45.92              46               50
```

위에서 보면 SELECT 구문 뒤에 기술된 컬럼들을 연산하기 위해서는 FROM 구문이 불필요한 것처럼 보이지만 Oracle에서는 SELECT 문장에서 FROM이 누락되면 에러를 발생시키기 때문에 DUAL 테이블이라는 하나의 행과 하나의 컬럼을 갖는 더미(Dummy) 테이블을 사용한다.

■ TRUNC

TRUNC는 입력 값을 지정된 자리에서 내림하는 함수이다.

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

TRUNC(45.923,2) TRUNC(45.923) TRUNC(45.923,-2)
--------------- ------------- ----------------
          45.92            45                0
```

■ MOD

MOD는 입력 값을 지정된 숫자로 나눈 나머지를 구하는 함수이다.

```
SQL> SELECT MOD(100, 30) FROM DUAL;

MOD(100,30)
-----------
         10
```

날짜 함수

날짜 함수를 설명하기에 앞서 날짜 타입에 대하여 먼저 알아보자. Oracle 데이터베이스는 날짜 타입을 저장할 때, 내부적으로 세기, 년도, 월, 일, 시간, 분, 초를 모두 저장한다. 그러나 화면에 표시되는 날짜는 디폴트로 RR/MM/DD로 표시되므로 20세기 년도(1900년대)와

21세기 년도(2000년대)의 처리에 의문이 생길 수 있지만, Oracle 데이터베이스는 날짜 타입에 내부적으로 세기 정보를 저장하고 있고, 데이터를 저장하고자 할 때도 연도를 4자리로 입력할 수 있기 때문에 큰 문제가 되지 않는다.

현재, 데이터베이스 서버에 설정된 날짜를 살펴보면 다음과 같다.

```
SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
--------
04/06/16
```

참고로 현재 세션에서 날짜의 표시형식을 변경하는 방법은 다음과 같다. 이 설정은 현재 세션에서만 유효하므로 SQL*Plus를 종료하고 다시 시작하면 원래 설정으로 돌아간다.

```
SQL> ALTER SESSION
  2  SET NLS_DATE_FORMAT = 'YYYY/MM/DD HH24:MI:SS';

세션이 변경되었습니다.

SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
-------------------
2004/06/16 14:14:22
```

날짜 타입의 연산에는 주의할 필요가 있다. 다음은 날짜 타입의 연산에 대한 의미와 결과를 정리한 것이다.

표 4-4. 날짜 연산

연산	결과	설명
날짜 + 숫자	날짜	날짜에 일수를 더한다
날짜 - 숫자	날짜	날짜에서 일수를 뺀다
날짜 - 날짜	숫자(일수)	두 날짜의 차이(일수)를 계산한다
날짜 + 숫자/24	날짜	날짜에 시간을 더한다

사원 테이블에서 사원들이 입사후 몇주가 경과되었는지를 살펴보면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, (SYSDATE-HIREDATE)/7 "주"
  2  FROM EMP;

     EMPNO ENAME                 주
---------- ---------- ----------
      7369 SMITH      1226.08576
      7499 ALLEN      1216.80004
      7521 WARD       1216.51433
        ...
      7900 JAMES       1175.9429
      7902 FORD        1175.9429
      7934 MILLER     1168.65719

14 개의 행이 선택되었습니다.
```

날짜 함수의 종류는 다음과 같다.

표 4-5. 날짜 함수

함수	설명
MONTHS_BETWEEN(*date1*, *date2*)	*date1*과 *date2*의 차이를 월 단위로 계산한다
ADD_MONTHS(*date*, *n*)	*date*에 *n*개월을 더한다
NEXT_DAY(*date*, '*char*')	*date*이후 '*char*'로 지정된 요일의 날짜를 계산한다
LAST_DAY(*date*)	*date*의 해당 월에서 마지막 일을 계산한다
ROUND(*date*[, '*fmt*'])	*date*에서 '*fmt*'로 지정된 자리수로 반올림한다
TRUNC(*date*[, '*fmt*'])	*date*에서 '*fmt*'로 지정된 자리수로 내림한다

■ MONTHS_BETWEEN

두 날짜의 차이를 월 단위로 계산한다.

```
SQL> SELECT EMPNO, ENAME, MONTHS_BETWEEN(SYSDATE, HIREDATE)
  2  FROM EMP
  3  WHERE EMPNO = 7839;

     EMPNO ENAME      MONTHS_BETWEEN(SYSDATE,HIREDATE)
---------- ---------- --------------------------------
      7839 KING                            270.987419
```

■ ADD_MONTHS

날짜에 개월을 더한다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE, ADD_MONTHS(HIREDATE, 6)
  2  FROM EMP
  3  WHERE EMPNO = 7839;

     EMPNO ENAME      HIREDATE ADD_MONT
---------- ---------- -------- --------
      7839 KING       81/11/17 82/05/17
```

■ NEXT_DAY

입력된 날짜로부터 지정된 다음 요일의 날짜를 계산한다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE, NEXT_DAY(HIREDATE, '수')
  2  FROM EMP
  3  WHERE EMPNO = 7839;

     EMPNO ENAME      HIREDATE NEXT_DAY
---------- ---------- -------- --------
      7839 KING       81/11/17 81/11/18
```

■ LAST_DAY

입력된 날짜의 월에서 마지막 일을 계산한다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE, LAST_DAY(HIREDATE)
  2  FROM EMP
  3  WHERE EMPNO = 7839;

     EMPNO ENAME      HIREDATE LAST_DAY
---------- ---------- -------- --------
      7839 KING       81/11/17 81/11/30
```

■ ROUND

입력된 날짜를 반올림한다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE,
  2  ROUND(HIREDATE, 'MONTH'),
  3  ROUND(HIREDATE, 'YEAR')
  4  FROM EMP
  5  WHERE EMPNO = 7839;

     EMPNO ENAME      HIREDATE ROUND(HI ROUND(HI
---------- ---------- -------- -------- --------
      7839 KING       81/11/17 81/12/01 82/01/01
```

■ TRUNC

입력된 날짜를 내림한다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE,
  2  TRUNC(HIREDATE, 'MONTH'),
  3  TRUNC(HIREDATE, 'YEAR')
  4  FROM EMP
  5  WHERE EMPNO = 7839;

     EMPNO ENAME      HIREDATE TRUNC(HI TRUNC(HI
---------- ---------- -------- -------- --------
      7839 KING       81/11/17 81/11/01 81/01/01
```

변환 함수

변환 함수는 데이터 타입을 다른 데이터 타입으로 변환하는 함수이다. 데이터 타입의 변환은 Oracle 데이터베이스 내부에서 자동으로 진행되는 암시적인 데이터 타입 변환과 변환 함수를 사용하는 명시적 데이터 타입 변환으로 구분되며, 다시 암시적 데이터 타입 변환은 값을 저장할 때와 수식을 전개할 때의 두 가지 경우에 따라 변환 방식이 다르다.

먼저, 테이블 또는 변수에 값을 입력하거나 지정하는 경우 변환 방식은 다음과 같다. 예를 들어, 테이블내 문자 타입 컬럼에 숫자를 저장하면 숫자는 내부적으로 문자로 변환되어 저장된다.

표 4-6. 암시적 데이터 타입 변환 1

변환 전	변환 후
VARCHAR2 또는 CHAR	NUMBER
VARCHAR2 또는 CHAR	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

다음은 수식을 전개하거나 조건식에서 값을 비교할 때 변환 방식이다. 예를 들어, 테이블내 문자 타입 컬럼을 숫자와 비교하면 컬럼내 저장된 값은 숫자로 변환되어 비교된다.

표 4-7. 암시적 데이터 타입 변환 2

변환 전	변환 후
VARCHAR2 또는 CHAR	NUMBER
VARCHAR2 또는 CHAR	DATE

특히, 두 번째 암시적 데이터 타입 변환이 SELECT 문장의 WHERE 구문에서 발생하는 경우를 INTERNAL SUPPRESSING이라고 부르며 검색 효율을 저하시키는 원인이 될 수가 있다. 예를 들어, SELECT 문장에 WHERE ID = 4567 이라고 기술되어 있고, ID 컬럼에 인덱스가 생성되어 있다고 가정하자. 여기서 ID 컬럼의 데이터 타입이 VARCHAR2 타입이라면 조건문의 4567이 VARCHAR2 타입으로 변환되는 것이 아니라 ID 컬럼의 데이터 값이 NUMBER 타입으로 변환되기 때문에 ID 컬럼에 생성되어 있는 인덱스를 사용할 수 없으므로 검색 효율은 나빠지게 된다. 인덱스는 책의 찾아보기와 같은 것으로 데이터 검색의 효율을 향상시켜 주는 역할을 하는 것인데, 추후에 다루기로 한다.

명시적 데이터 타입 변환은 아래와 같은 전용 변환 함수를 사용하여 데이터 타입을 변경하는 방법이다.

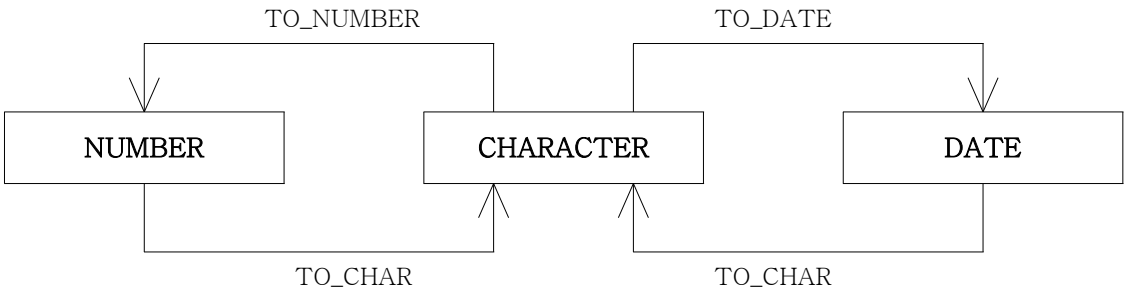


그림 4-1. 데이터 타입간 변환

■ TO_CHAR

TO_CHAR 함수는 NUMBER 또는 DATE 타입을 CHAR 타입으로 변환시켜 준다. 먼저, DATE 타입을 CHAR 타입으로 변환하기 위한 TO_CHAR 함수의 사용 방법은 다음과 같다.

```
TO_CHAR(date, 'format_model')
```

여기서 *format_model*은 아래 표와 같이 날짜, 시간, 기타 형식이 있으며 유효한 형식을 사용하여야 한다. 사용된 형식 앞에 *fm*을 붙이면 결과 값의 좌측부분에 있는 공백이나 0을 제거하여 표시할 수 있다.

표 4-8. *format_model* - 날짜 형식

형식	설명
YYYY	년도를 숫자로 표시
YEAR	년도를 영문으로 표시
MM	월을 숫자로 표시
MONTH	월을 영문으로 표시
MON	월을 영문 3자리로 축약해서 표시
DY	요일을 영문 3자리로 축약해서 표시, 한글 환경에서는 한글 1자로 표시
DAY	요일을 영문으로 표시, 한글 환경에서는 한글 3자로 표시
DD	일을 숫자로 표시

표 4-9. *format_model* - 시간 형식

형식	설명
AM 또는 PM	오전 또는 오후를 표시
A.M. 또는 P.M.	오전 또는 오후를 표시
HH 또는 HH12 또는 HH24	시간 또는 1~12시 또는 1~24시
MI	분(0~59)
SS	초(0~59)
SSSSS	자정 이후 초(0~86399)

표 4-10. *format_model* - 기타 형식

형식	설명
/ . ,	구분자 표시
"of the"	결과에 추가할 문자열
TH	서수 표시 (예, DDTH인 경우 4TH)
SP	영문 기수 표시 (예, DDSP인 경우 FOUR)
SPTH 또는 THSP	영문 서수 표시 (예, DDSPTH인 경우 FOURTH)

사용 예는 다음과 같다.

```
SQL> SELECT EMPNO, ENAME,
  2  TO_CHAR(HIREDATE, 'fmDD MONTH YYYY') "입사일"
  3  FROM EMP;

     EMPNO ENAME      입사일
---------- ---------- -----------------
      7369 SMITH      17 DECEMBER 1980
      7499 ALLEN      20 FEBRUARY 1981
      7521 WARD       22 FEBRUARY 1981
      7566 JONES      2 APRIL 1981
        ...
      7902 FORD       3 DECEMBER 1981
      7934 MILLER     23 JANUARY 1982
```

NUMBER 타입을 CHAR 타입으로 변환하기 위한 TO_CHAR 함수의 사용 방법은 다음과 같다.

```
TO_CHAR(number, 'format_model')
```

마찬가지로 적용가능한 *format_model*이 있으며 다음과 같다.

표 4-11. *format_model* - 숫자 형식

형식	설명
9	숫자로 표시
0	숫자의 앞부분을 0으로 표시
$	달러 표시
L	지역 화폐 단위 표시
.	소수점 표시
,	1000 단위 구분자 표시

사용 예는 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, TO_CHAR(SAL, 'L99,999.00') "급여"
  2  FROM EMP;

     EMPNO ENAME      급여
---------- ---------- --------------------
      7369 SMITH                ₩800.00
      7499 ALLEN              ₩1,600.00
      7521 WARD               ₩1,250.00
        ...
      7934 MILLER             ₩1,300.00
```

■ TO_NUMBER

TO_NUMBER 함수는 CHAR 타입을 NUMBER 타입으로 변환시켜 준다. TO_NUMBER 함수의 사용 방법은 다음과 같다.

```
TO_NUMBER(char[, 'format_model'])
```

사용 예는 다음과 같다.

```
SQL> SELECT TO_NUMBER('100')+10 FROM DUAL;

TO_NUMBER('100')+10
-------------------
                110
```

■ TO_DATE

TO_DATE 함수는 CHAR 타입을 DATE 타입으로 변환시켜 준다. TO_DATE 함수의 사용 방법은 다음과 같다.

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

사용 예는 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE
  2  FROM EMP
  3  WHERE HIREDATE=TO_DATE('DECEMBER 17, 1980', 'MONTH DD, YYYY');

     EMPNO ENAME      HIREDATE
---------- ---------- --------
      7369 SMITH      80/12/17
```

특히, TO_DATE 함수의 *format_model*에 fx 식별자를 사용하면 문자열과 *format_model*의 형식이 정확히 일치해야 되며 그렇지 않은 경우 에러가 발생된다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE
  2  FROM EMP
  3  WHERE HIREDATE=TO_DATE('DECEMBER 17,  1980', 'fxMONTH DD, YYYY');
WHERE HIREDATE=TO_DATE('DECEMBER 17,  1980', 'fxMONTH DD, YYYY')
                       *
3행에 오류:
ORA-01841: 년은 영이 아닌 -4713 과 +4713 사이의 값으로 지정해야 합니다
```

또한, *format_model*에 추가적으로 RR 날짜 포맷이 있으며, 이 기능은 연도를 두자리로 입력하는 경우 적절히 1900년대 또는 2000년대의 4자리 연도로 적절히 변환되도록 해 준다. 변환 규칙은 다음과 같다.

표 4-12. RR 날짜 포맷에 의한 연도 변환 규칙

		지정하고자 하는 두자리 연도	
		0~49	50~99
시스템의 두자리 연도	0~49	현재 세기의 연도로 변환	전 세기의 연도로 변환
	50~99	다음 세기의 연도로 변환	현재 세기의 연도로 변환

예를 들면 다음과 같다.

```
SQL> ALTER SESSION
  2  SET NLS_DATE_FORMAT = 'YYYY/MM/DD';

SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
----------
2004/06/17

SQL> SELECT TO_DATE('99', 'YY'), TO_DATE('01', 'YY') FROM DUAL;

TO_DATE('9 TO_DATE('0
---------- ----------
2099/06/01 2001/06/01

SQL> SELECT TO_DATE('99', 'RR'), TO_DATE('01', 'RR') FROM DUAL;

TO_DATE('9 TO_DATE('0
---------- ----------
1999/06/01 2001/06/01
```

일반 함수

일반 함수 중에 NULL 값 처리와 관련된 함수는 다음과 같다.

표 4-13. 일반 함수

함수	설명
NVL(*expr1*, *expr2*)	*expr1*이 NULL이면 *expr2*를 리턴
NVL2(*expr1*, *expr2*, *expr3*)	*expr1*이 NULL이 아니면 *expr2*, NULL이면 *expr3*를 리턴
NULLIF(*expr1*, *expr2*)	*expr1*가 *expr2*가 같으면 NULL, 다르면 *expr1*을 리턴
COALESCE(*expr1*, *expr2*, ... , *exprn*)	인자들 중에서 NULL이 아닌 첫 번째 인자를 리턴

■ NVL

함수의 첫 번째 인자가 NULL 이면 두 번째 인자를 리턴한다. NVL 함수는 NULL 값이 포함된 컬럼을 연산에 포함시키고자 할 때, 아주 유용한 함수이다.

```
SQL> SELECT EMPNO, ENAME, SAL, COMM, SAL*12+COMM, SAL*12+NVL(COMM, 0)
  2  FROM EMP;

     EMPNO ENAME             SAL       COMM SAL*12+COMM SAL*12+NVL(COMM,0)
---------- ---------- ---------- ---------- ----------- -------------------
      7369 SMITH             800                                      9600
      7499 ALLEN            1600        300       19500               19500
      7521 WARD             1250        500       15500               15500
      7566 JONES            2975                                     35700
        ...
      7934 MILLER           1300                                     15600
```

■ NVL2

함수의 첫 번째 인자가 NULL이 아니면 두 번째 인자로, NULL이면 세 번째 인자를 리턴한다.

```
SQL> SELECT EMPNO, ENAME, SAL, COMM, SAL+COMM, NVL2(COMM, SAL+COMM, SAL)
  2  FROM EMP;

     EMPNO ENAME             SAL       COMM   SAL+COMM NVL2(COMM,SAL+COMM,SAL)
---------- ---------- ---------- ---------- ---------- ----------------------
      7369 SMITH             800                                          800
      7499 ALLEN            1600        300       1900                   1900
      7521 WARD             1250        500       1750                   1750
      7566 JONES            2975                                         2975
        ...
      7934 MILLER           1300                                         1300
```

■ NULLIF

함수의 첫 번째 인자와 두 번째 인자가 같으면 NULL을, 다르면 첫 번째 인자를 리턴한다.

```
SQL> SELECT EMPNO, ENAME, JOB, NULLIF(LENGTH(ENAME), LENGTH(JOB))
  2  FROM EMP;

     EMPNO ENAME      JOB       NULLIF(LENGTH(ENAME),LENGTH(JOB))
---------- ---------- --------- ---------------------------------
      7369 SMITH      CLERK
      7499 ALLEN      SALESMAN                                  5
      7521 WARD       SALESMAN                                  4
        ...
      7900 JAMES      CLERK
      7902 FORD       ANALYST                                   4
      7934 MILLER     CLERK                                     6
```

■ COALESCE

함수의 인자들 중에 NULL이 아닌 최초의 인자를 리턴한다.

```
SQL> SELECT EMPNO, ENAME, SAL, COMM, COALESCE(SAL, COMM, 10)
  2  FROM EMP;

     EMPNO ENAME             SAL       COMM COALESCE(SAL,COMM,10)
---------- ---------- ---------- ---------- --------------------
      7369 SMITH             800                             800
      7499 ALLEN            1600        300                 1600
      7521 WARD             1250        500                 1250
        ...
      7934 MILLER           1300                            1300
```

■ CASE

CASE 함수는 SQL 문장내에서 IF-THEN-ELSE와 같은 흐름제어문의 역할을 한다. CASE 함수의 문법은 다음과 같다.

```
CASE expr WHEN comparison_expr1 THEN return_expr1
         [WHEN comparison_expr2 THEN return_expr2
          WHEN comparison_exprn THEN return_exprn
          ELSE else_expr]
END
```

*expr*이 *comparison_expr1*과 같으면 *return_expr1*, *comparison_expr2*와 같으면 *return_expr2*, *comparison_exprn*과 같으면 *return_exprn*을 리턴하고 그렇지 않으면 *else_expr*을 리턴한다. 사용방법은 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, SAL, JOB,
  2   CASE JOB WHEN 'ANALYST'    THEN SAL*1.1
  3            WHEN 'CLERK'      THEN SAL*1.2
  4            WHEN 'MANAGER'    THEN SAL*1.3
  5            WHEN 'PRESIDENT'  THEN SAL*1.4
  6            WHEN 'SALESMAN'   THEN SAL*1.5
  7            ELSE SAL
  8   END "급여"
  9  FROM EMP;

     EMPNO ENAME             SAL JOB              급여
---------- ---------- ---------- --------- ----------
      7369 SMITH             800 CLERK            960
      7499 ALLEN            1600 SALESMAN        2400
      7521 WARD             1250 SALESMAN        1875
      7566 JONES            2975 MANAGER       3867.5
      7654 MARTIN           1250 SALESMAN        1875
        ...
      7900 JAMES             950 CLERK           1140
      7902 FORD             3000 ANALYST         3300
      7934 MILLER           1300 CLERK           1560
```

■ DECODE

DECODE 함수도 CASE 함수와 마찬가지로 SQL 문장내에서 IF-THEN-ELSE와 같은 흐름제어문의 역할을 한다. DECODE 함수의 문법은 다음과 같다.

```
DECODE(col/expression, search1, result1
                  [, search2, result2, ... ,]
                  [, default])
```

*col1/expression*이 *search1*과 같으면 *result1*, *search2*와 같으면 *result2*를 리턴하고 그렇지 않으면 *default*를 리턴한다. 사용방법은 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, SAL, JOB,
  2   DECODE(JOB, 'ANALYST'   , SAL*1.1,
  3               'CLERK'     , SAL*1.2,
  4               'MANAGER'   , SAL*1.3,
  5               'PRESIDENT', SAL*1.4,
  6               'SALESMAN' , SAL*1.5, SAL) "급여"
  7  FROM EMP;

     EMPNO ENAME             SAL JOB              급여
---------- ---------- ---------- --------- ----------
      7369 SMITH             800 CLERK            960
      7499 ALLEN            1600 SALESMAN        2400
      7521 WARD             1250 SALESMAN        1875
      7566 JONES            2975 MANAGER       3867.5
      7654 MARTIN           1250 SALESMAN        1875
        ...
      7900 JAMES             950 CLERK           1140
      7902 FORD             3000 ANALYST         3300
      7934 MILLER           1300 CLERK           1560

14 개의 행이 선택되었습니다.
```

위에서 기술한 단일행 함수들은 함수의 중첩이 가능하며, 함수 중첩이 되어 있는 경우 연산의 순서는 가장 안쪽에 있는 함수부터 바깥 쪽 함수로 연산된다.

```
F3(F2(F1(col1, arg1), arg2), arg3)
```

```
SQL> SELECT EMPNO, ENAME, MGR,
  2  NVL(TO_CHAR(MGR), '상위 관리자 없음')
  3  FROM EMP
  4  WHERE MGR IS NULL;

     EMPNO ENAME      MGR NVL(TO_CHAR(MGR),'상위관리자없음')
---------- ---------- ---------- ------------------------------------
      7839 KING                  상위 관리자 없음
```

복습

1. 사원 테이블의 사원명에서 2번째 문자부터 3개의 문자를 추출하시오.

2. 사원 테이블에서 입사일이 12월인 사원의 사번, 사원명, 입사일을 검색하시오.

3. 다음과 같은 결과를 검색할 수 있는 SQL 문장을 작성하시오.

```
     EMPNO ENAME      급여
---------- ---------- --------------------
      7369 SMITH      *******800
      7499 ALLEN      ******1600
        ...
      7934 MILLER     ******1300
```

4. 다음과 같은 결과를 검색할 수 있는 SQL 문장을 작성하시오.

```
     EMPNO ENAME      입사일
---------- ---------- ----------
      7369 SMITH      1980-12-17
      7499 ALLEN      1981-02-20
        ...
      7934 MILLER     1982-01-23
```

5. 사원 테이블에서 급여에 따라 사번, 이름, 급여, 등급을 검색하는 SQL 문장을 작성하시오. (Hint : CASE 함수 사용)

급여	등급
0~1000	E
1001~2000	D
2001~3000	C
3001~4000	B
4001~5000	A

```
     EMPNO ENAME             SAL -
---------- ---------- ---------- -
      7369 SMITH             800 E
      7499 ALLEN            1600 D
        ...
      7934 MILLER           1300 D
```

Chapter 5. Join

이전 장까지는 한 개의 테이블만을 검색하는 경우를 살펴보았다. 이번 장에서는 검색하고자 하는 컬럼이 여러 개의 테이블에 존재하는 경우, 데이터를 검색할 수 있는 Join에 대하여 설명한다.

Join의 개념

여러 개의 테이블로부터 데이터를 검색하는 경우를 "테이블을 Join 한다"라고 부른다. 예를 들어, 다음 그림과 같이 사용자가 검색하고자 하는 컬럼이 EMP 테이블의 EMPNO, ENAME, DEPT 테이블의 DNAME이라고 가정했을 때, EMPNO가 7369인 SMITH의 DNAME을 검색하기 위해서는 EMP 테이블의 외래키(FK)인 DEPTNO 컬럼과 DEPT 테이블의 기본키(PK)인 DEPTNO가 일치되는 DNAME을 DEPT 테이블에서 검색해야 한다. 즉, SMITH의 DEPTNO는 20이므로, DEPT 테이블에서 DEPTNO가 20인 DNAME을 검색하여 SMITH의 부서명은 RESEARCH임을 알 수가 있게 된다. 테이블이 Join 되기 위해서는 반드시 FK와 FK가 참조하는 PK가 반드시 존재하여야 함을 알 수가 있다. 여기서, FK와 PK가 일치해야하는 조건을 Join 조건이라고 부르며 SELECT 문장의 WHERE절에 기술한다.

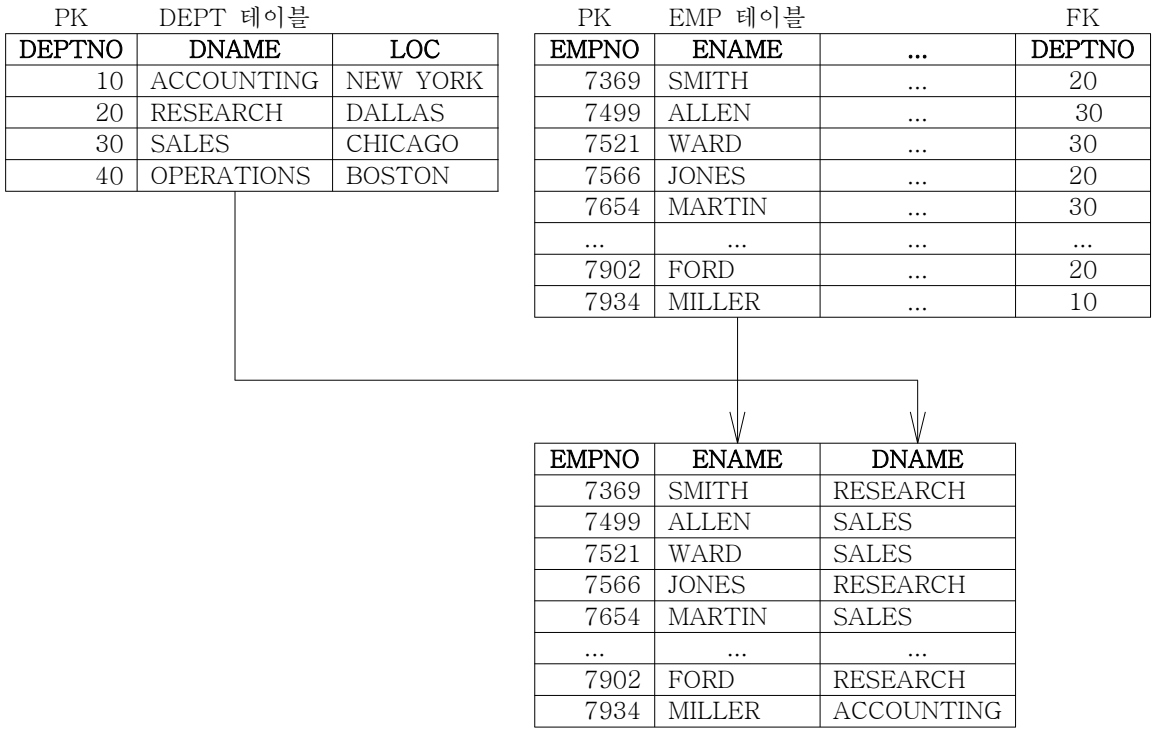


그림 5-1. Equi-Join

카르테시안 프로덕트(Cartesian Product)

카르테시안 프로덕트란 SELECT 문장에서 Join 조건을 생략하여 첫 번째 테이블의 모든 행들이 두 번째 테이블의 모든 행들과 Join되는 경우이다.

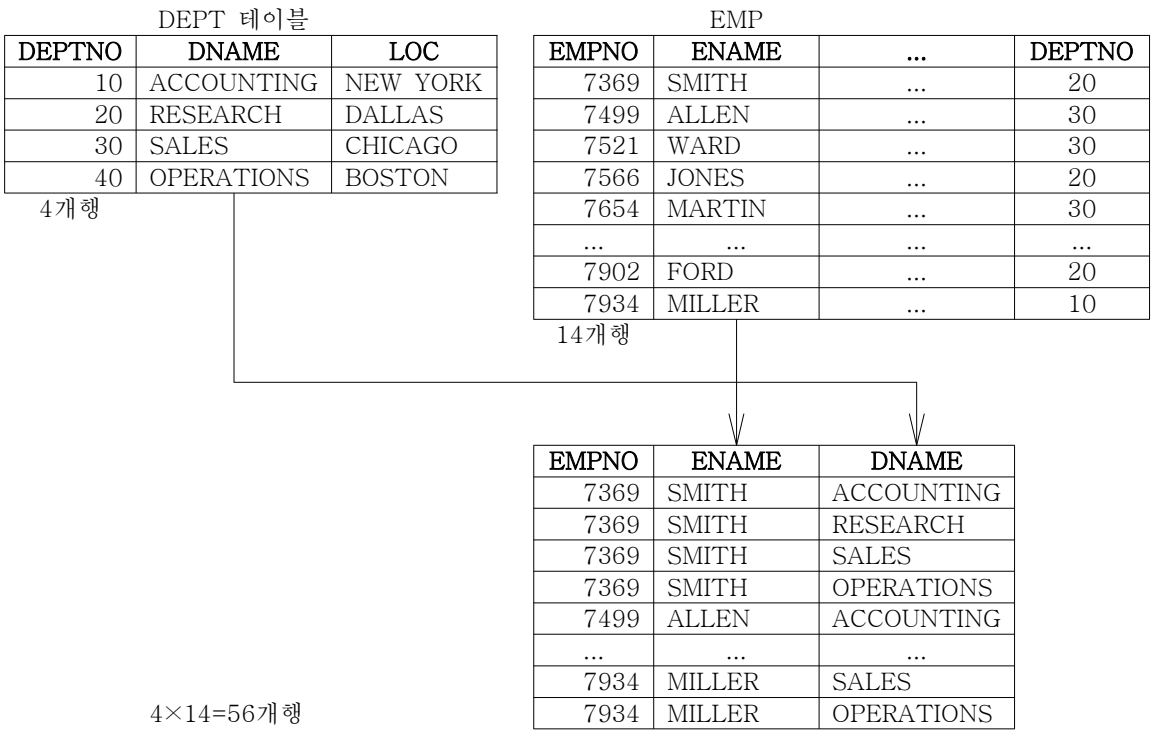


그림 5-2. 카르테시안 프로덕트

```
SQL> SELECT EMPNO, ENAME, DNAME
  2  FROM DEPT, EMP;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7369 SMITH      ACCOUNTING
      7369 SMITH      RESEARCH
      7369 SMITH      SALES
      7369 SMITH      OPERATIONS
      7499 ALLEN      ACCOUNTING
        ...
      7934 MILLER     OPERATIONS
```

Join에는 Equi-Join, Non-EquiJoin, Outer Join, Self Join이 있으며, 각각 살펴보도록 한다.

Equi-Join

Equi-Join이란 테이블을 Join할 때, FK와 FK가 참조하는 PK가 Equal 조건에 의해 정확히 일치하는 경우만 검색하는 Join 방식이다. 먼저, Equi-Join의 문법은 다음과 같다.

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

위 문법에서 보듯이 Join을 하기 위해서는 어떤 테이블에 어떤 컬럼이 있는지, 테이블간의 관계는 어떻게 되는지, FK와 PK의 컬럼명은 무엇인지를 정확히 알아야만 Join 문장을 오류없이 정확하게 기술할 수 있다. 그러나, 데이터베이스에는 수십~수백개의 테이블이 존재하기 때문에 이러한 정보를 일일이 검색하는 것은 불가능하다고 할 수 있다. 그러면 이러한 정보를 어떻게 하면 손쉽게 확인할 수 있을까? 해법은 1장에서 설명한 개체관계도(ERD)이다. 이렇듯 ERD는 데이터베이스 관리자나 개발자에게 여러모로 쓸모 있는 것임에 틀림없다.

실습에 사용할 DEPT, EMP, SALGRADE 테이블의 ERD는 다음 그림과 같다.

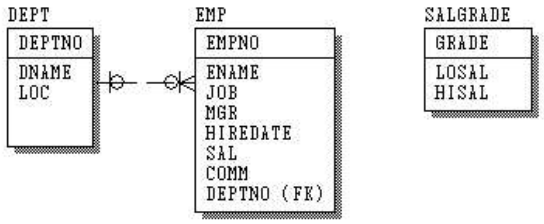


그림 5-3. 개체관계도(ERD)

그림의 ERD를 보면서 Join 문장을 작성해보자. DEPT, EMP 테이블에서 EMPNO, ENAME, DNAME을 검색하려면 ERD를 확인하면서 다음과 같은 절차에 의해 SQL 문장을 작성하면 된다.

(1) 검색하고자 하는 컬럼을 SELECT절 뒤에 기술한다.
(2) 검색하고자 하는 컬럼이 소속되어 있는 테이블들의 이름을 FROM절 뒤에 기술한다.
(3) DEPT 테이블과 EMP 테이블의 관계에서 실선으로 되어 있는 개체 쪽의 기본키 컬럼과 까마귀발 모양으로 되어 있는 개체 쪽의 외래키 컬럼을 WHERE절에 Equal 조건으로 기술한다.

```
SQL> SELECT EMP.EMPNO, EMP.ENAME, DEPT.DNAME
  2  FROM EMP, DEPT
  3  WHERE EMP.DEPTNO = DEPT.DEPTNO;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7369 SMITH      RESEARCH
      7499 ALLEN      SALES
      7521 WARD       SALES
        ...
      7934 MILLER     ACCOUNTING
```

만약, 위의 Join 결과에서 EMPNO가 7698인 행들만 검색하려면 WHERE절의 Join 조건에

검색 조건을 추가하면 된다.

```
SQL> SELECT EMP.EMPNO, EMP.ENAME, DEPT.DNAME
  2  FROM EMP, DEPT
  3  WHERE EMP.DEPTNO = DEPT.DEPTNO
  4  AND EMP.EMPNO = 7698;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7698 BLAKE      SALES
```

SELECT절 뒤의 컬럼명에는 해당 컬럼이 소속되어 있는 테이블명을 정확히 기술해주는 것이 바람직하지만 해당 컬럼명이 FROM절 뒤의 모든 테이블내에서 유일하다면 반드시 기술해 줄 필요는 없다.

```
SQL> SELECT EMPNO, ENAME, DNAME
  2  FROM EMP, DEPT
  3  WHERE EMP.DEPTNO = DEPT.DEPTNO
  4  AND EMP.EMPNO = 7698;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7698 BLAKE      SALES
```

그러나, SELECT절 뒤에 기술되는 컬럼명이 FROM절 뒤의 일부 테이블들에 중복되어 있다면 다음과 같은 오류 메시지가 발생하게 된다.

```
SQL> SELECT EMPNO, ENAME, DEPTNO, DNAME
  2  FROM EMP, DEPT
  3  WHERE EMP.DEPTNO = DEPT.DEPTNO
  4  AND EMP.EMPNO = 7698;
SELECT EMPNO, ENAME, DEPTNO, DNAME
                     *
1행에 오류:
ORA-00918: 열의 정의가 애매합니다
```

FROM절 뒤에 기술되는 테이블명이 길거나 너무 복잡한 경우에는 테이블 별칭을 사용하여 SQL 문장을 간략하게 작성할 수 있다. 테이블 별칭을 사용할 때는 다음의 작성지침을 준수하도록 한다.

- 테이블 별칭의 길이는 30문자까지 가능하지만 짧을수록 좋다.
- 테이블 별칭이 사용되면 SQL 문장 내에서 모든 테이블명은 지정된 테이블 별칭을 사용해야만 한다.
- 테이블 별칭은 기술된 SELECT 문장 내에서만 유효하다.

```
SQL> SELECT E.EMPNO, E.ENAME, D.DNAME
  2  FROM EMP E, DEPT D
  3  WHERE E.DEPTNO = D.DEPTNO
  4  AND E.EMPNO = 7698;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7698 BLAKE      SALES
```

이번에는 검색하려는 테이블의 개수가 2개 이상일 경우, Join하는 방법을 알아보자. ERD가 다음과 같다면, A1, B1, C1 컬럼을 검색할 수 있는 SQL 문장은 다음과 같다.

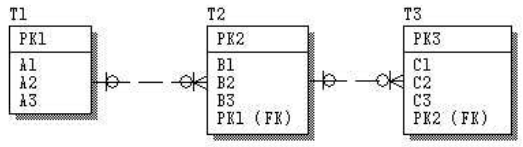


그림 5-4. Join 예제 1

```
SELECT T1.A1, T2.B1, T3.C1
FROM T1, T2, T3
WHERE T1.PK1 = T2.PK1
AND T2.PK2 = T3.PK2;
```

앞에서 설명한 바와 같이 SELECT절 뒤에는 검색하고자 하는 컬럼들을 나열하고, FROM절 뒤에는 해당 컬럼들이 포함된 테이블들을 기술한 뒤에 Join 조건을 차례로 작성한다. 위에서 보는 것처럼 Join할 테이블이 N개라면 Join 조건은 N-1개가 됨을 확인할 수 있다.

만약, 검색하고자 하는 컬럼이 A1, C1이라면 어떻게 Join 문장을 작성해야 되는지 생각해보자. 해당 컬럼이 포함되어 있는 테이블은 T1, T3이지만 직접적인 Join 조건이 존재하지 않으므로 이런 경우도 위와 마찬가지로 어쩔 수 없이 3개의 테이블을 모두 Join 해야 한다.

```
SELECT T1.A1, T3.C1
FROM T1, T2, T3
WHERE T1.PK1 = T2.PK1
AND T2.PK2 = T3.PK2;
```

이번에는 다음과 같은 ERD에서 A1, C1 컬럼을 검색하는 경우, Join 문장은 어떻게 작성되어야 하는지 생각해보자.

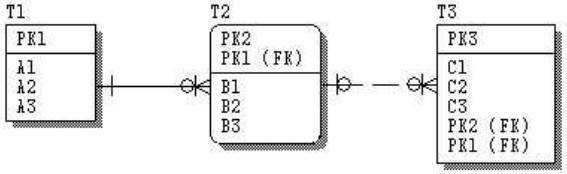


그림 5-5. Join 예제 2

ERD에서 보는 것처럼 A1, C1 컬럼이 포함되어 있는 T1, T3 테이블간에는 앞의 경우처럼 직접적인 관계가 존재하지 않아서 T1과 T3간에 Join 조건을 작성할 수 없을 것 같이 보이지만 T3 테이블에는 T1의 기본키인 PK1이 T3의 외래키로 추가되어 있기 때문에 직접 Join이 가능하다. 이와 같이 되는 이유는 T1과 T2간에 식별관계로 설정되어 T1의 기본키인 PK1이 T2의 외래키이면서 기본키에 참여했기 때문에 T2의 기본키(PK2, PK1)가 T3의 외래키로 추가되었기 때문이다.

```
SELECT T1.A1, T3.C1
FROM T1, T3
WHERE T1.PK1 = T3.PK1;
```

Non-EquiJoin

Non-EquiJoin은 테이블간에 Join 조건으로 Equal이 아닌 경우이다.

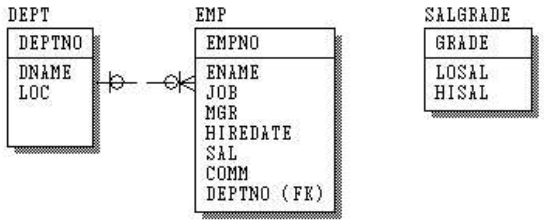


그림 5-6. 개체관계도(ERD)

실습 ERD에서 사원 테이블내의 급여가 급여등급 테이블내의 하한값과 상한값의 범위에 포함되는 경우 해당 등급를 검색하는 SQL 문장은 다음과 같다.

```
SQL> SELECT E.EMPNO, E.ENAME, E.SAL, S.GRADE
  2  FROM EMP E, SALGRADE S
  3  WHERE E.SAL BETWEEN S.LOSAL AND S.HISAL;

     EMPNO ENAME             SAL      GRADE
---------- ---------- ---------- ----------
      7369 SMITH             800          1
      7876 ADAMS            1100          1
      7900 JAMES             950          1
      7521 WARD             1250          2
        ...
      7902 FORD             3000          4
      7839 KING             5000          5
```

Outer-Join

앞에서 설명한 Equi-Join과 Non-EquiJoin의 경우 Join 조건이 반드시 만족하는 경우에만 해당 테이블의 행들이 검색된다. 즉, DEPT 테이블과 EMP 테이블을 Equi-Join한 결과를 살펴보면 DEPT 테이블에 저장되어 있는 DEPTNO가 40번인 OPERATIONS 부서는 절대로 검색되지 않는다. 그 이유는 EMP 테이블에 저장되어 있는 사원 데이터중에 DEPTNO가

40인 사원이 존재하지 않기 때문이다.

```
SQL> SELECT DEPTNO, DNAME
  2  FROM DEPT
  3  WHERE DEPTNO = 40;

    DEPTNO DNAME
---------- --------------
        40 OPERATIONS

SQL> SELECT EMPNO, ENAME, DEPTNO
  2  FROM EMP
  3  WHERE DEPTNO = 40;

선택된 레코드가 없습니다.
```

이런 경우, Outer-Join을 이용하면 40번 OPERATIONS 부서를 출력할 수 있다. 먼저, Outer-Join의 문법은 다음과 같다.

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

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

위에서 보는 것처럼 Join 조건에 (+) 기호를 포함하는데, Join 조건의 양단에 모두 사용할 수는 없다. (+) 기호를 붙이는 쪽은 Join 할 때, 일치되는 행이 존재하지 않는 쪽에 붙이도록 한다.

즉, DEPT와 EMP 테이블에서 DEPT 테이블의 DEPTNO가 40인 행과 일치되는 행이 EMP 테이블에는 존재하지 않으므로 SQL 문장은 다음과 같이 작성하면 된다.

```
SQL> SELECT EMP.EMPNO, EMP.ENAME, DEPT.DNAME
  2  FROM EMP, DEPT
  3  WHERE EMP.DEPTNO(+) = DEPT.DEPTNO;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7782 CLARK      ACCOUNTING
      7839 KING       ACCOUNTING
      7934 MILLER     ACCOUNTING
        ...
      7521 WARD       SALES
                      OPERATIONS
```

Self-Join

지금까지의 Join은 2개 이상의 테이블이 사용되었으나, Self-Join은 테이블 자신을 자신이 Join하는 방법이다. 실습 테이블인 EMP 테이블의 데이터를 살펴보면 MGR 컬럼이 있는데, 이 컬럼은 해당 사원의 담당 사수인 EMPNO를 참조하고 있다.

```
SQL> SELECT EMPNO, ENAME, MGR
  2  FROM EMP;

     EMPNO ENAME             MGR
---------- ---------- ----------
      7369 SMITH            7902
        ...
      7566 JONES            7839
        ...
      7839 KING
        ...
      7902 FORD             7566
      7934 MILLER           7782
```

결과에서 보듯이 SMITH의 사수는 7902번 FORD이고, FORD의 사수는 7566번 JONES, JONES의 사수는 7839번 KING, KING의 사수는 존재하지 않으므로 NULL을 테이블에 저장한 것이다. 그렇다면, 사원명과 사수명을 동시에 검색하는 방법을 생각해보자. 사원명과 사수명은 모두 ENAME 이므로, Join을 사용하지 않은 일반 SQL 문장으로는 절대 검색할 수 없다. 이런 경우는 테이블 별칭을 사용하여 EMP 테이블을 사원 테이블과 사수 테이블로 명명하여 Join 하면 된다. 즉, 사원 테이블의 MGR이 사수 테이블의 EMPNO를 Equal 조건으로 Join 하면 다음과 같이 검색할 수 있다.

```
SQL> SELECT W.ENAME "사원", M.ENAME "사수"
  2  FROM EMP W, EMP M
  3  WHERE W.MGR = M.EMPNO;

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

추가적으로 위의 결과에서 사원 컬럼에 KING을, 사수 컬럼에 NULL을 표시하려면 Outer-Join을 사용하면 된다.

```
SQL> SELECT W.ENAME "사원", M.ENAME "사수"
  2  FROM EMP W, EMP M
  3  WHERE W.MGR = M.EMPNO(+);

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

SQL : 1999 Join

지금까지 살펴본 Join 구문은 Oracle 8i와 이전 버전에서 사용하는 문법이었다. 그러나, Oracle 9i 버전부터는 SQL : 1999 표준과의 호환성을 위해 Join 문장이 다수 추가되었다.

SQL : 1999 호환을 위해 추가된 Join 문법은 다음과 같다.

```
SELECT table1.column, table2.column
FROM table1
[CROSS JOIN table2] |
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
  ON (table1.column_name = table2.column_name)] |
[LEFT|RIGHT|FULL OUTER JOIN table2
  ON (table1.column_name = table2.column_name)];
```

■ Cross Join

Cross Join은 카르테시안 프로덕트와 동일하다.

```
SQL> SELECT EMPNO, ENAME, DNAME
  2  FROM DEPT
  3  CROSS JOIN EMP;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7369 SMITH      ACCOUNTING
      7369 SMITH      RESEARCH
      7369 SMITH      SALES
      7369 SMITH      OPERATIONS
      7499 ALLEN      ACCOUNTING
        ...
      7934 MILLER     OPERATIONS
```

■ Natural Join

Natural Join은 Equi-Join과 동일하지만 기술하는 방법이 약간 다르다. 즉, Join 조건을 기술 할 필요가 없으며, FROM 절 뒤의 테이블내에서 동일한 이름을 같는 컬럼들을 찾아서 Equal 조건으로 Join 해준다.

```
SQL> SELECT EMPNO, ENAME, DNAME
  2  FROM DEPT
  3  NATURAL JOIN EMP;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7369 SMITH      RESEARCH
      7499 ALLEN      SALES
      7521 WARD       SALES
      7566 JONES      RESEARCH
        ...
      7934 MILLER     ACCOUNTING
```

■ USING 구문을 이용한 Join

Natural Join에서 Join 해야 할 테이블 간에 동일한 이름을 갖는 컬럼이 여러 개인 경우에는 원하는 컬럼으로 Join 할 수가 없다. 이러한 경우, Join 하고자 하는 컬럼을 USING 구문 뒤에 기술하면 원하는 결과를 검색 할 수 있다.

```
SQL> SELECT EMPNO, ENAME, DNAME
  2  FROM DEPT
  3  JOIN EMP
  4  USING (DEPTNO);

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7369 SMITH      RESEARCH
      7499 ALLEN      SALES
      7521 WARD       SALES
      7566 JONES      RESEARCH
        ...
      7934 MILLER     ACCOUNTING
```

한 가지 주의할 점은 USING 구문 뒤의 컬럼에는 테이블 이름 또는 테이블 별칭을 참조하면 에러가 발생할 수 있으니 주의하여야 하며, NATURAL JOIN과 USING 구문은 동시에 사용할 수 없다.

```
SQL> SELECT EMPNO, ENAME, DNAME
  2  FROM DEPT
  3  JOIN EMP
  4  USING (DEPTNO)
  5  WHERE DEPT.DEPTNO =10;
WHERE DEPT.DEPTNO =10
      *
5행에 오류:
ORA-25154: USING 절의 열 부분은 식별자를 가질 수 없음
```

■ ON 구문을 이용한 Join

Natural Join은 테이블간의 동일한 이름을 갖는 컬럼에 대하여 Equi-Join을 수행하기 때문에 Non-EquiJoin이나 PK와 FK 컬럼의 이름이 서로 다른 경우는 Natural Join을 사용할 수 없다. 이러한 경우, ON 구문을 사용하여 임의의 조건 및 컬럼을 지정할 수 있다. 또한, ON 구문을 사용하면 다른 검색 조건과 분리되어 SQL 문장을 이해하기가 쉬워진다.

```
SQL> SELECT EMPNO, ENAME, DNAME
  2  FROM DEPT JOIN EMP
  3  ON DEPT.DEPTNO = EMP.DEPTNO
  4  AND DEPT.DEPTNO = 10;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7782 CLARK      ACCOUNTING
      7839 KING       ACCOUNTING
      7934 MILLER     ACCOUNTING
```

다음과 같은 ERD에서 ON 구문으로 여러 개의 테이블을 Join하는 방법은 다음과 같다.

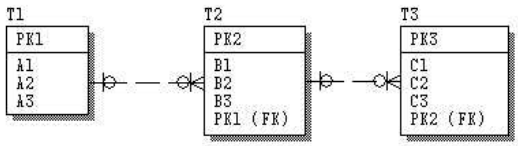


그림 5-4. Join 예제 3

```
SELECT T1.A1, T2.B1, T3.C1
FROM T1 JOIN T2
ON T1.PK1 = T2.PK1
JOIN T3
ON T2.PK2 = T3.PK2;
```

■ LEFT OUTER JOIN

LEFT OUTER JOIN은 LEFT OUTER JOIN 구문의 좌측에 기술한 테이블내의 모든 행들이 우측에 기술한 테이블내 행들과 일치 여부에 상관 없이 모두 출력된다.

```
SQL> SELECT EMPNO, ENAME, DNAME
  2  FROM DEPT LEFT OUTER JOIN EMP
  3  ON DEPT.DEPTNO = EMP.DEPTNO;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7369 SMITH      RESEARCH
      7499 ALLEN      SALES
      7521 WARD       SALES
        ...
      7934 MILLER     ACCOUNTING
                      OPERATIONS
```

위의 문장은 다음 문장과 동일하다.

```
SELECT EMPNO, ENAME, DNAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO(+);
```

■ RIGHT OUTER JOIN

RIGHT OUTER JOIN은 RIGHT OUTER JOIN 구문의 우측에 기술한 테이블내의 모든 행들이 좌측에 기술한 테이블내 행들과 일치 여부에 상관 없이 모두 출력된다. 실습을 위해서 먼저 EMP 테이블에 새로운 행을 입력한다.

```
SQL> INSERT INTO EMP
  2  VALUES(8000, 'KIM', 'MANAGER', 7934, '04/06/22', 2000, NULL, NULL);

1 개의 행이 만들어졌습니다.

SQL> SELECT EMPNO, ENAME, DNAME
  2  FROM DEPT RIGHT OUTER JOIN EMP
  3  ON DEPT.DEPTNO = EMP.DEPTNO;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7934 MILLER     ACCOUNTING
      7839 KING       ACCOUNTING
      7782 CLARK      ACCOUNTING
        ...
      7499 ALLEN      SALES
      8000 KIM
```

위의 문장은 다음 문장과 동일하다.

```
SELECT EMPNO, ENAME, DNAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO(+) = EMP.DEPTNO;
```

■ FULL OUTER JOIN

LEFT OUTER JOIN과 RIGHT OUTER JOIN을 결합한 것이다. 즉, FULL OUTER JOIN의 좌측과 우측에 기술된 테이블내의 모든 행들이 일치되는 것은 물론이고 일치되지 않는 행들도 모두 출력된다. 사용 방법은 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, DNAME
  2  FROM DEPT FULL OUTER JOIN EMP
  3  ON DEPT.DEPTNO=EMP.DEPTNO;

     EMPNO ENAME      DNAME
---------- ---------- --------------
      7369 SMITH      RESEARCH
      7499 ALLEN      SALES
      7521 WARD       SALES
      7566 JONES      RESEARCH
        ...
      7934 MILLER     ACCOUNTING
                      OPERATIONS
      8000 KIM
```

복습

1. 부서 테이블과 사원 테이블에서 사번, 사원명, 부서코드, 부서명을 검색하시오. 단, 출력시, 사원명을 기준으로 오름차순으로 정렬하시오.

2. 부서 테이블과 사원 테이블에서 사번, 사원명, 급여, 부서명을 검색하시오. 단, 급여가 2000 이상인 사원에 대하여 급여를 기준으로 내림차순으로 정렬하시오.

3. 부서 테이블과 사원 테이블에서 사번, 사원명, 업무, 급여, 부서명을 검색하시오. 단, 업무가 MANAGER이며 급여가 2500 이상인 사원에 대하여 사번을 기준으로 오름차순으로 정렬하시오.

4. 사원 테이블과 급여등급 테이블에서 사번, 사원명, 급여, 등급을 검색하시오. 단, 등급은 급여가 하한값과 상한값 범위에 포함되고 등급이 4이며 급여를 기준으로 내림차순으로 정렬하시오.

5. 부서 테이블, 사원 테이블, 급여등급 테이블에서 사번, 사원명, 부서명, 급여, 등급를 검색하시오. 단, 등급은 급여가 하한값과 상한값 범위에 포함되며 등급을 기준으로 내림차순으로 정렬하시오.

6. 사원 테이블에서 사원명과 해당 사원의 관리자명을 검색하시오.

7. 사원 테이블에서 사원명, 해당 사원의 관리자명, 해당 사원의 관리자의 관리자명을 검색하시오.

8. 7번 결과에서 상위 관리자가 없는 모든 사원의 이름도 사원명에 출력되도록 수정하시오.

Chapter 6. 그룹 함수

그룹 함수는 단일행 함수와는 달리 복수 개의 행을 입력으로 받아 한건의 결과를 리턴하는 함수이다. 이 장에서는 그룹 함수의 사용방법을 알아보고 GROUP BY 구문을 이용하여 테이블을 여러개로 그룹핑하는 방법, HAVING 구문을 이용하여 각 그룹을 제한하는 방법을 설명한다.

그룹 함수

그룹 함수란 다음 그림과 같이 복수 개의 행을 입력으로 받아 한건의 결과를 리턴하는 함수이다.

EMPNO	ENAME	...	SAL	DEPTNO
7369	SMITH	...	800	20
7499	ALLEN	...	1600	30
7521	WARD	...	1250	30
7566	JONES	...	2975	20
7654	MARTIN	...	1250	30
...	...	...	...	...
7902	FORD	...	3000	20
7934	MILLER	...	1300	10

SAL의 합

SUM(SAL)
29025

그림 6-1. 그룹 함수의 개념

그룹 함수는 다음과 같은 함수들이 있다.

표 6-1. 그룹 함수

함수	설명
AVG([DISTINCT|ALL]*n*)	*n*의 평균값, Null 무시
COUNT({*|[DISTINCT|ALL]*expr*})	행의 수, 여기서 *expr*은 Null이 아님(Null을 포함하여 전체 행의 수를 계산할 때는 *을 사용)
MAX([DISTINCT|ALL]*expr*)	*expr*의 최대값, Null 무시
MIN([DISTINCT|ALL]*expr*)	*expr*의 최소값, Null 무시
STDDEV([DISTINCT|ALL]*n*)	*n*의 표준편자, Null 무시
SUM([DISTINCT|ALL]*n*)	*n*의 합, Null 무시
VARIANCE([DISTINCT|ALL]*n*)	*n*의 분산, Null 무시

그룹 함수를 사용하는 경우 다음과 같은 작성 지침을 따르도록 한다.

- DISTINCT는 중복되지 않는 값들에 대해서 연산을 수행하는 반면 ALL은 모든 값에 대하여 연산을 수행한다. ALL이 디폴트이므로 매번 ALL을 기술할 필요는 없다.
- *expr*은 CHAR, VARCHAR2, NUMBER, DATE 타입이 될 수 있다.
- 모든 그룹 함수는 Null 값을 연산에 포함시키지 않으므로, NULL을 연산에 포함시키기 위해서는 NVL, NVL2, COALESCE와 같은 NULL 관련 단일행 함수를 사용해야 한다.

그룹 함수를 사용하는 방법은 다음과 같다.

```
SELECT [column,] group_function(column), ...
FROM table;
```

■ AVG, SUM

AVG 함수는 평균값을, SUM 함수는 합계를 리턴하는 함수이다.

```
SQL> SELECT SUM(SAL), AVG(SAL)
  2  FROM EMP;

  SUM(SAL)   AVG(SAL)
---------- ----------
     29025 2073.21429
```

■ MIN, MAX

MIN 함수는 최소값을, MAX 함수는 최대값을 리턴하는 함수이다.

```
SQL> SELECT MIN(SAL), MAX(SAL)
  2  FROM EMP;

  MIN(SAL)   MAX(SAL)
---------- ----------
       800       5000
```

■ COUNT

COUNT(*)는 테이블의 전체 행의 개수를, COUNT(*expr*)은 *expr*이 NULL이 아닌 값들의 개수를 리턴한다.

```
SQL> SELECT COUNT(*), COUNT(COMM)
  2  FROM EMP;

  COUNT(*) COUNT(COMM)
---------- -----------
        14           4
```

COUNT(DISTINCT *expr*)은 *expr*이 NULL이 아닌 값들에 대해서 중복된 값을 제거하고 값들의 개수를 리턴한다.

```
SQL> SELECT COUNT(DISTINCT JOB)
  2  FROM EMP;

COUNT(DISTINCTJOB)
------------------
                 5
```

지금까지 설명한 그룹 함수들은 입력 값으로 NULL이 들어오면 연산에 포함시키지 않는다. 그러면, 테이블내에 NULL이 포함된 컬럼에 대하여 그룹 함수를 사용했을 때 실수하기 쉬운 예를 몇가지 살펴보도록 하자.

EMP 테이블에는 전체 14개의 행이 입력되어 있지만 COMM 컬럼에 값이 입력되어 있는 행은 총 4개 밖에 되지 않는다. 이런 경우, EMP 테이블에서 COMM 컬럼의 평균을 계산해 보자.

```
SQL> SELECT SUM(COMM)/COUNT(*), AVG(COMM)
  2  FROM EMP;

SUM(COMM)/COUNT(*)  AVG(COMM)
------------------ ----------
        157.142857        550
```

SELECT절 뒤의 SUM(COMM)/COUNT(*)은 COMM에서 NULL을 제외한 합산 결과에 전체 행의 개수 즉, 사원 수로 나누어 평균을 계산한 것이며, AVG(COMM)은 COMM에서 NULL을 제외한 합산 결과에 Null이 아닌 값의 개수로 나누어 평균을 계산한 것이다. 즉, 첫 번째 컬럼은 (300+ 500+ 1400+ 0)/14, 두 번째 컬럼은 (300+ 500+ 1400+ 0)/4을 계산한 것이다. 그렇다면 어떤 결과가 과연 옳은 것인가를 생각해보자. 사용자의 관점에 따라 두 경우 모두 맞는다고 볼 수 있다. 그러므로, 사용자가 어떤 결과 값이 필요한지를 신중히 생각해보고 적절히 구현해야 한다. 첫 번째 경우는 NVL 함수를 사용하면 좀 더 간단하게 작성할 수 있다.

```
SQL> SELECT AVG(NVL(COMM, 0)), AVG(COMM)
  2  FROM EMP;

AVG(NVL(COMM,0))  AVG(COMM)
---------------- ----------
      157.142857        550
```

그렇다면 그룹 함수중에서 어떤 함수에 대해 위와 같은 경우를 고려해야 하는지 살펴보자. SUM, MAX, MIN은 해당 컬럼에 NULL 값이 있어도 그 결과에는 영향을 미치지 못함을 유추할 수 있다. 그러나, 위에서 예를 들었던 AVG, STDDEV, VARIANCE와 같이 연산과정에서 값의 개수를 이용하여 나눗셈 연산이 되는 경우는 모두 주의해야 할 필요가 있다.

```
SQL> SELECT STDDEV(NVL(COMM, 0)), STDDEV(COMM)
  2  FROM EMP;

STDDEV(NVL(COMM,0)) STDDEV(COMM)
------------------- ------------
         387.723704   602.771377

SQL> SELECT VARIANCE(NVL(COMM, 0)), VARIANCE(COMM)
  2  FROM EMP;

VARIANCE(NVL(COMM,0)) VARIANCE(COMM)
--------------------- --------------
            150329.67     363333.333
```

GROUP BY

앞 절에서는 주어진 테이블 전체에 대하여 그룹 함수를 이용하는 방법을 설명하였다. 그러

나, 실제 통계작업에 있어서는 전체 집계보다는 부분 집계를 압도적으로 많이 사용하게 된다. 예를 들어, 우리 회사 사원들의 부서별 평균 급여를 계산하거나 직급별 총 급여, 월별 매출액, 분기별 매출액 등등을 계산해야 할 일이 더 많다는 얘기이다. 부분 집계에 대한 개념은 다음 그림과 같다. 아래 그림은 부서별 총급여액을 계산한 것이다.

EMPNO	ENAME	...	SAL	DEPTNO
7782	CLARK	...	2450	10
7839	KING	...	5000	10
7934	MILLER	...	1300	10
7369	SMITH	...	800	20
7876	ADAMS	...	1100	20
7902	FORD	...	3000	20
7788	SCOTT	...	3000	20
7566	JONES	...	2975	20
7499	ALLEN	...	1600	30
7698	BLAKE	...	2850	30
7654	MARTIN	...	1250	30
7900	JAMES	...	950	30
7844	TURNER	...	1500	30
7521	WARD	...	1250	30

DEPTNO별 SAL의 합

DEPTNO	SUM(SAL)
10	8750
20	10875
30	9400

그림 6-2. GROUP BY

■ GROUP BY 작성 지침

이러한 부분집계를 수행하는데 유용한 것이 GROUP BY절이다. 먼저, GROUP BY를 사용하는 방법은 다음과 같다.

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

GROUP BY를 사용할 때는 다음의 작성 지침을 따르도록 한다.

- GROUP BY가 사용될 때, SELECT 절 뒤에 사용할 수 있는 컬럼은 GROUP BY 뒤에 기술된 컬럼 또는 그룹 함수가 적용된 컬럼이어야만 한다.
- WHERE 절을 사용하면 테이블에서 주어진 조건을 만족하는 결과 행들에 대해서만 GROUP BY에 적용 된다.
- GROUP BY는 WHERE 조건을 만족하는 결과 행들에 대해서 GROUP BY 뒤에 기술한 컬럼의 값이 같은 것들끼리 그룹으로 묶는다.
- GROUP BY 절 뒤에는 컬럼 별칭을 사용할 수 없다.
- GROUP BY를 사용하면 내부적으로 GROUP BY에 기술된 컬럼으로 오름차순 정렬된다.

■ GROUP BY의 이해

GROUP BY는 Oracle 입문자들이 SQL을 학습하는 과정에서 처음으로 만나게 되는 어

려운 부분 중의 하나이다. 물론, GROUP BY 문법을 완벽히 외어서 사용하는 개발자도 있겠지만 SQL 문법을 단순히 암기해서는 고급 SQL 문장을 구사할 수 없다. 그러므로 사고의 전환을 통해, 하나의 프로그램을 개발한다는 생각으로 SQL 문장을 작성해야 할 것이다.

먼저, GROUP BY의 이해를 돕기 위해 다음과 같은 시나리오를 생각해보자. 모 초등학교에서 학생들의 성적처리를 위해 데이터베이스를 구축하고 다음과 같은 간략한 테이블을 만들어 학생들의 시험 성적 결과를 모두 입력했다고 가정하자. (물론, 성적처리를 위한 실제 데이터베이스는 훨씬 더 복잡하다.) 이 초등학교는 전체 6개 학년, 10개반, 반별 정원은 30명이며 전교생이 월말고사를 치른 후, 그 시험 결과를 모두 저장하였다고 가정한다. 그렇다면 SUNGJEOK 테이블에는 1800개의 행이 입력되어 있을 것이다.

표 6-2. SUNGJEOK 테이블

HAK	BAN	BUN	JUMSOO
1	1	1	92
1	1	2	85
...	...	...	...
3	1	1	75
3	1	2	95
3	1	3	82
...	...	...	...
6	10	29	84
6	10	30	95

위와 같은 상황에서 다음과 같은 문제를 해결해보자.
· 전교생의 평균시험점수는?
· 전교생의 학년별 평균시험점수는? (예, 1학년 88.2점, 6학년 92.5점 등)
· 전교생의 반별 평균시험점수는? (예, 1학년 1반 92.3점, 6학년 10반 89.1점 등)

첫 번째 문제는 단순히 그룹함수만을 사용하면 해결할 수 있으며, 그 보다 나은 SQL문장은 없다.

```
SELECT AVG(JUMSOO) FROM SUNGJUK;
```

두 번째 문제는 학년별 평균시험점수를 계산하는 것이다. 이 경우는 WHERE 절만을 사용하는 방법과 이 장에서 설명하고 있는 GROUP BY를 사용하는 방법의 두가지 방법이 있다. 먼저, WHERE 절만을 사용해서 학년별 평균시험점수를 모두 계산해보자.

```
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 1;
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 2;
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 3;
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 4;
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 5;
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 6;
```

위에서 보는 바와 같이 WHERE 절의 조건만 조금씩 수정하여 총 6번의 SQL 문장을 작

성하면 원하는 결과를 얻을 수 있다. 그렇다면, 세 번째 문제도 WHERE 절만을 사용해서 반별 평균시험점수를 모두 계산할 수 있을 것이다.

```
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 1 AND BAN = 1;
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 1 AND BAN = 2;
...
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 3 AND BAN = 1;
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 3 AND BAN = 2;
...
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 6 AND BAN = 1;
SELECT AVG(JUMSOO) FROM SUNGJUK WHERE HAK = 6 AND BAN = 2;
```

특히, 위와 같이 WHERE 절만을 사용하는 경우 전체 60번의 SQL 문장을 작성하여야만 원하는 결과를 얻을 수 있게 되므로 현실적으로 사용하기에는 무리가 따른다. 그렇다면, 두 번째 문제와 세 번째 문제를 하나의 SQL 문장으로 해결할 수는 없는지 살펴보자. 다음 SQL 문장을 보면서 GROUP BY를 암기하는 것이 아니라 이해해보도록 하자.

```
SELECT HAK, AVG(JUMSOO) - ③
FROM SUNGJUK - ①
GROUP BY HAK; - ②
```

SQL 문장을 사용하지 않고 선생님 입장에서 초등학생들에게 직접 명령을 내려 학년별 평균점수를 계산한다고 가정해보자.

· 1단계 : 초등학생들을 모두 운동장에 집합시킨다. (①)
· 2단계 : 학년별로 줄을 세운다. 가장 좌측에 1학년부터 맨 우측에는 6학년. (②)
· 3단계 : 학년별 학생 대표에게 각 학년별 평균점수를 계산하도록 하고, 그 결과를 "학년, 평균점수" 형식으로 제출하도록 지시한다. (③)

위와 같은 시나리오와 SQL 문장이 정확히 일치되어 이해가 되었다면 반별 평균점수도 계산해보자.

· 1단계 : 초등학생들을 모두 운동장에 집합시킨다. (①)
· 2단계 : 반별로 줄을 세운다. 가장 좌측에 1학년 1반, 맨 우측에는 6학년 10반. (②)
· 3단계 : 각 반의 반장에게 각 반 평균점수를 계산하도록 하고, 그 결과를 "학년, 반, 평균점수" 형식으로 제출하도록 지시한다. (③)

```
SELECT HAK, BAN, AVG(JUMSOO) - ③
FROM SUNGJUK - ①
GROUP BY HAK, BAN; - ②
```

추가적으로 전교생 중에서 번호가 홀수인 학생들의 반별 평균점수를 계산해보자.

· 1단계 : 초등학생들을 모두 운동장에 집합시킨다. (①)
· 2단계 : 번호가 홀수인 학생들만 운동장에 남고, 나머지 학생은 교실로 들여보낸다.(②)
· 3단계 : 반별로 줄을 세운다. 가장 좌측에 1학년 1반, 우측에는 6학년 10반. (③)
· 4단계 : 각 반의 반장에게 각 반 평균점수를 계산하도록 하고, 그 결과를 "학년, 반, 평균점수" 형식으로 제출하도록 지시한다. (④)

```
SELECT HAK, BAN, AVG(JUMSOO) - ④
FROM SUNGJUK - ①
WHERE MOD(BUN, 2) = 1 - ②
GROUP BY HAK, BAN; - ③
```

마지막으로 위 결과에서 검색 결과를 평균점수가 높은 반부터 출력되도록 하려면 다음과 같이 ORDER BY를 추가해주면 된다.

```
SELECT HAK, BAN, AVG(JUMSOO)
FROM SUNGJUK
WHERE MOD(BUN, 2) = 1
GROUP BY HAK, BAN
ORDER BY AVG(JUMSOO) DESC;
```

■ GROUP BY 예제

실습 테이블을 이용하여 GROUP BY 문장을 연습해보자. 사원 테이블에서 부서별 평균 급여를 계산하면 다음과 같다.

```
SQL> SELECT DEPTNO, AVG(SAL)
  2  FROM EMP
  3  GROUP BY DEPTNO;

    DEPTNO   AVG(SAL)
---------- ----------
        10 2916.66667
        20       2175
        30 1566.66667
```

특히, 주의 할 점은 SELECT 뒤의 컬럼들은 GROUP BY에 기술된 컬럼이어야만 한다. 입문자들이 GROUP BY 사용에 있어서 가장 많은 오류를 경험하는 경우이다.

```
SQL> SELECT ENAME, AVG(SAL)
  2  FROM EMP
  3  GROUP BY DEPTNO;
SELECT ENAME, AVG(SAL)
       *
1행에 오류:
ORA-00979: GROUP BY 의 식이 없습니다
```

그러나, GROUP BY 뒤의 컬럼이 반드시 SELECT 뒤에 나와야 할 필요는 없다.

```
SQL> SELECT AVG(SAL)
  2  FROM EMP
  3  GROUP BY DEPTNO;

  AVG(SAL)
----------
2916.66667
      2175
1566.66667
```

또한, GROUP BY 절이 없이 SELECT 뒤에 일반 컬럼과 그룹함수가 사용된 컬럼이 동

시에 올 수 없다.

```
SQL> SELECT DEPTNO, AVG(SAL)
  2  FROM EMP;
SELECT DEPTNO, AVG(SAL)
       *
1행에 오류:
ORA-00937: 단일 그룹의 그룹 함수가 아닙니다
```

마지막으로 GROUP BY와 WHERE가 같이 쓰이는 SQL 문장에서 WHERE 절에는 그룹 함수가 사용된 컬럼이 올 수 없다. 이러한 경우, 그룹을 제한하기 위해서는 뒤에 설명할 HAVING 절을 사용해야한다.

```
SQL> SELECT DEPTNO, AVG(SAL)
  2  FROM EMP
  3  WHERE AVG(SAL) > 2000
  4  GROUP BY DEPTNO;
WHERE AVG(SAL) > 2000
      *
3행에 오류:
ORA-00934: 그룹 함수는 허가되지 않습니다
```

사원 테이블에서 부서, 업무별 급여 합계를 계산하면 다음과 같다.

```
SQL> SELECT DEPTNO, JOB, SUM(SAL)
  2  FROM EMP
  3  GROUP BY DEPTNO, JOB;

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

HAVING

HAVING은 GROUP BY에 의해 분류된 그룹들을 제한하기 위한 구문이다. 다음 그림은 부서별 급여의 합계가 9000이 넘는 경우만을 검색한 것이다.

EMPNO	ENAME	...	SAL	DEPTNO
7782	CLARK	...	2450	10
7839	KING	...	5000	10
7934	MILLER	...	1300	10
7369	SMITH	...	800	20
7876	ADAMS	...	1100	20
7902	FORD	...	3000	20
7788	SCOTT	...	3000	20
7566	JONES	...	2975	20
7499	ALLEN	...	1600	30
7698	BLAKE	...	2850	30
7654	MARTIN	...	1250	30
7900	JAMES	...	950	30
7844	TURNER	...	1500	30
7521	WARD	...	1250	30

DEPTNO별 SAL의
합이 9000보다 큰 경우

DEPTNO	SUM(SAL)
20	10875
30	9400

그림 6-3. HAVING

■ HAVING 작성 지침

HAVING을 사용하는 방법은 다음과 같다.

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

HAVING이 SELECT 문장에 포함되어 있으면 다음과 같은 절차에 의해 결과가 출력된다.

1. GROUP BY에 의해 테이블내의 행들이 주어진 컬럼별로 그룹화된다.
2. HAVING 뒤의 그룹함수가 각 그룹에 적용된다.
3. HAVING 뒤의 조건이 만족하는 그룹들만 출력된다.

■ HAVING 예제

사원 테이블에서 부서별 급여의 합계가 9000보다 큰 경우는 다음과 같다.

```
SQL> SELECT DEPTNO, SUM(SAL)
  2  FROM EMP
  3  GROUP BY DEPTNO
  4  HAVING SUM(SAL) > 9000;

    DEPTNO   SUM(SAL)
---------- ----------
        20      10875
        30       9400
```

사원 테이블에서 급여가 1000보다 큰 사원들에 대하여 부서별 SAL의 합계가 9000보다 큰 경우는 다음과 같다.

```
SQL> SELECT DEPTNO, SUM(SAL)
  2  FROM EMP
  3  WHERE SAL > 1000
  4  GROUP BY DEPTNO
  5  HAVING SUM(SAL) > 9000;

    DEPTNO   SUM(SAL)
---------- ----------
        20      10075
```

그룹함수도 다음과 같이 중첩될 수 있다.

```
SQL> SELECT MAX(SUM(SAL))
  2  FROM EMP
  3  GROUP BY DEPTNO;

MAX(SUM(SAL))
-------------
        10875
```

피벗 테이블

지금까지 GROUP BY를 이용한 부분 집계에 대하여 알아보았다. 예를 들어, 사원 테이블에서 부서, 업무별 급여의 합을 구하기 위해서는 아래와 같은 SQL 문장을 사용했다.

```
SQL> SELECT DEPTNO, JOB, SUM(SAL)
  2  FROM EMP
  3  GROUP BY DEPTNO, JOB;

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

위 결과를 보면, 원하는 데이터를 얻을 수는 있으나 우리가 실제적으로 사용하는 출력 양식은 위와 같은 형태가 아니라 다음과 같은 피벗 테이블의 형태이다.

표 6-3. 피벗 테이블 형식

	CLERK	MANAGER	PRESIDENT	ANALYST	SALESMAN
10	1300	2450	5000		
20	1900	2975		6000	
30	950	2850			5600

이와 같은 결과를 얻기 위해서는 DECODE 함수와 GROUP BY를 사용해야만 한다. 그렇다

면, 다음 절차에 따라 SQL 문장을 적절히 구성하여 위와 같은 피벗 테이블 형태의 결과를 만들어보자.

· 1단계 : 원본 테이블에서 피벗 테이블의 가로줄 컬럼을 DECODE를 이용하여 분류한다.

```
SQL> SELECT DEPTNO,
  2  DECODE(JOB, 'CLERK', SAL) CLERK,
  3  DECODE(JOB, 'MANAGER', SAL) MANAGER,
  4  DECODE(JOB, 'PRESIDENT', SAL) PRESIDENT,
  5  DECODE(JOB, 'ANALYST', SAL) ANALYST,
  6  DECODE(JOB, 'SALESMAN', SAL) SALESMAN
  7  FROM EMP;

    DEPTNO      CLERK    MANAGER  PRESIDENT    ANALYST   SALESMAN
---------- ---------- ---------- ---------- ---------- ----------
        20        800
        30                                              1600
        30                                              1250
        20                  2975
        30                                              1250
        30                  2850
        10                  2450
        20                                        3000
        10                            5000
        30                                              1500
        20       1100
        30        950
        20                                        3000
        10       1300
```

· 2단계 : GROUP BY를 이용하여 부서별로 그룹화한다.

```
SQL> SELECT DEPTNO,
  2  SUM(DECODE(JOB, 'CLERK', SAL)) CLERK,
  3  SUM(DECODE(JOB, 'MANAGER', SAL)) MANAGER,
  4  SUM(DECODE(JOB, 'PRESIDENT', SAL)) PRESIDENT,
  5  SUM(DECODE(JOB, 'ANALYST', SAL)) ANALYST,
  6  SUM(DECODE(JOB, 'SALESMAN', SAL)) SALESMAN
  7  FROM EMP
  8  GROUP BY DEPTNO;

    DEPTNO      CLERK    MANAGER  PRESIDENT    ANALYST   SALESMAN
---------- ---------- ---------- ---------- ---------- ----------
        10       1300       2450       5000
        20       1900       2975                  6000
        30        950       2850                            5600
```

참고로 위의 결과에서 맨 우측에 세로줄을 추가하여 부서별 합계를 구하면 다음과 같다.

```
SQL> SELECT DEPTNO,
  2  SUM(DECODE(JOB, 'CLERK', SAL)) CLERK,
  3  SUM(DECODE(JOB, 'MANAGER', SAL)) MANAGER,
  4  SUM(DECODE(JOB, 'PRESIDENT', SAL)) PRESIDENT,
  5  SUM(DECODE(JOB, 'ANALYST', SAL)) ANALYST,
  6  SUM(DECODE(JOB, 'SALESMAN', SAL)) SALESMAN,
  7  SUM(SAL) "계"
  8  FROM EMP
  9  GROUP BY DEPTNO;

    DEPTNO      CLERK    MANAGER  PRESIDENT    ANALYST   SALESMAN         계
---------- ---------- ---------- ---------- ---------- ---------- ----------
        10       1300       2450       5000                          8750
        20       1900       2975                  6000                10875
        30        950       2850                             5600       9400
```

마지막으로 맨 마지막줄에 JOB별 합계와 전체 급여 총합을 추가하면 다음과 같다.

```
SQL> SELECT DEPTNO,
  2  SUM(DECODE(JOB, 'CLERK', SAL)) CLERK,
  3  SUM(DECODE(JOB, 'MANAGER', SAL)) MANAGER,
  4  SUM(DECODE(JOB, 'PRESIDENT', SAL)) PRESIDENT,
  5  SUM(DECODE(JOB, 'ANALYST', SAL)) ANALYST,
  6  SUM(DECODE(JOB, 'SALESMAN', SAL)) SALESMAN,
  7  SUM(SAL) "계"
  8  FROM EMP
  9  GROUP BY ROLLUP(DEPTNO);

    DEPTNO      CLERK    MANAGER  PRESIDENT    ANALYST   SALESMAN         계
---------- ---------- ---------- ---------- ---------- ---------- ----------
        10       1300       2450       5000                          8750
        20       1900       2975                  6000                10875
        30        950       2850                             5600       9400
                 4150       8275       5000       6000       5600      29025
```

위 두가지 경우에 대한 분석은 독자들의 몫으로 남겨두도록 하겠다.

복습

1. 사원 테이블에서 최대 급여, 최소 급여, 전체 급여 합, 평균 급여를 검색하시오.

2. 사원 테이블에서 부서별 인원수를 검색하시오.

3. 사원 테이블에서 부서별 인원수가 6명 이상인 부서코드를 검색하시오.

4. 사원 테이블에서 급여가 높은 순서대로 등수를 부여하고자 한다. 다음과 같은 결과를 출력할 수 있는 SQL 문장을 작성하시오.
   (Hint : Self Join, Non-Equi Join, GROUP BY, COUNT 사용)

ENAME	등수
KING	1
FORD	2
SCOTT	2
JONES	4
BLAKE	5
CLARK	6
ALLEN	7
TURNER	8
MILLER	9
MARTIN	10
WARD	10
ADAMS	12
JAMES	13
SMITH	14

5. 사원 테이블로부터 부서명, 업무별 급여 합계를 계산하고자 한다. 다음과 같은 결과를 출력할 수 있는 SQL 문장을 작성하시오.

DNAME	CLERK	MANAGER	PRESIDENT	ANALYST	SALESMAN
ACCOUNTING	1300	2450	5000		
RESEARCH	1900	2975		6000	
SALES	950	2850			5600

Chapter 7. 서브 쿼리 I

서브 쿼리(Subquery)란 SQL 문장에 포함되어 있는 또 하나의 별도 SQL 문장이다. 이 장에서는 서브 쿼리의 정의와 서브 쿼리로 수행 할 수 있는 작업을 살펴보고, 서브쿼리의 종류와 단일행 및 복수행 서브쿼리를 작성하는 방법을 설명한다.

서브 쿼리의 정의

서브 쿼리는 SQL 문장내에 포함된 쿼리 문장으로서 여러 번의 쿼리 문장을 수행해야 얻을 수 있는 결과를 하나의 중첩된 SQL 문장으로 쉽게 얻을 수 있도록 해준다. 예를 들어, 사원 테이블에서 SCOTT보다 많은 급여를 받는 사원의 이름을 검색하기 위해서는 먼저 SCOTT가 받는 급여를 검색한 후, 검색된 급여보다 많은 급여를 받는 직원들을 검색하여야 한다.

```
SQL> SELECT SAL
  2  FROM EMP
  3  WHERE ENAME = 'SCOTT';

       SAL
----------
      3000

SQL> SELECT ENAME
  2  FROM EMP
  3  WHERE SAL > 3000;

ENAME
----------
KING
```

서브 쿼리를 사용하면 위와 같이 두 단계에 걸쳐서 수행되어야 할 작업을 한번에 수행할 수 있게 된다.

서브 쿼리의 사용방법

서브 쿼리의 사용방법은 다음과 같다.

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

위에서 바깥 쪽 쿼리를 메인 쿼리(Main query), 안쪽 쿼리를 서브 쿼리(Subquery)라고 부르며, 서브 쿼리가 먼저 실행되고 그 결과가 메인 쿼리에 전달되어 실행된다. 서브 쿼리는 SQL 문장의 WHERE절 뿐만 아니라 HAVING, FROM 절에도 포함될 수 있다. 위에서

*operator*는 단일행 연산자(>, =, >=, <, <>, <=)와 복수행 연산자(IN, ANY, ALL)로 분류할 수 있으며 SQL 문장내의 서브 쿼리가 단일행 서브 쿼리인 경우는 단일행 연산자를 사용하고 복수행 서브 쿼리인 경우는 복수행 연산자를 사용하여야만 한다.

서브 쿼리를 사용할 때는 다음과 같은 작성 지침을 준수하도록 한다.
- 서브 쿼리에는 괄호를 붙인다.
- 서브 쿼리가 WHERE 절에 포함 될 때는 비교 연산자의 오른쪽에 기술한다.
- Top-N 쿼리를 작성하는 경우 이외에는 ORDER BY 절에 서브 쿼리를 작성할 필요는 없다.
- 단일행 서브 쿼리에는 단일행 연산자를 사용하고 복수행 서브 쿼리인 경우는 복수행 연산자를 사용한다.

사원 테이블에서 SCOTT의 급여보다 많은 급여를 받는 사원의 이름을 검색하면 다음과 같다.

```
SQL> SELECT ENAME
  2  FROM EMP
  3  WHERE SAL > (SELECT SAL
  4               FROM EMP
  5               WHERE ENAME = 'SCOTT');

ENAME
----------
KING
```

서브 쿼리의 종류

서브 쿼리의 종류에는 단일행 서브 쿼리와 복수행 서브 쿼리가 있다. 이렇게 서브 쿼리를 구분하는 기준은 서브 쿼리가 리턴하는 행의 개수에 따라 다른데, 서브 쿼리가 한개의 행을 리턴하면 단일행 서브 쿼리라 하고 두개 이상의 행을 리턴하면 복수행 서브 쿼리라고 부른다.

단일행 서브 쿼리

단일행 서브 쿼리는 오직 하나의 행만을 리턴하며, 반드시 단일행 연산자(=, >, <=, <, >=, <>)를 사용해야만 한다.

사원 테이블에서 사번이 7844인 사원의 업무와 동일한 업무를 수행하는 사원의 사번, 이름, 업무를 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB
  2  FROM EMP
  3  WHERE JOB = (SELECT JOB
  4               FROM EMP
  5               WHERE EMPNO = 7844);

     EMPNO ENAME      JOB
---------- ---------- ---------
      7499 ALLEN      SALESMAN
      7521 WARD       SALESMAN
      7654 MARTIN     SALESMAN
      7844 TURNER     SALESMAN
```

사원 테이블에서 사번이 7521번인 사원과 같은 업무를 수행하고 있으며, 사번이 7900번인 사원의 급여보다 많은 급여를 받는 사원의 이름, 업무, 급여는 다음과 같다.

```
SQL> SELECT ENAME, JOB, SAL
  2  FROM EMP
  3  WHERE JOB = (SELECT JOB
  4               FROM EMP
  5               WHERE EMPNO = 7521)
  6  AND SAL > (SELECT SAL
  7             FROM EMP
  8             WHERE EMPNO = 7900);

ENAME      JOB              SAL
---------- --------- ----------
ALLEN      SALESMAN         1600
WARD       SALESMAN         1250
MARTIN     SALESMAN         1250
TURNER     SALESMAN         1500
```

사원 테이블에서 가장 적은 급여를 받는 사원의 사번, 이름, 급여를 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE SAL = (SELECT MIN(SAL)
  4               FROM EMP);

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800
```

서브 쿼리는 WHERE 절 뿐만 아니라 HAVING 절에도 사용할 수 있다.

```
SQL> SELECT DEPTNO, MIN(SAL)
  2  FROM EMP
  3  GROUP BY DEPTNO
  4  HAVING MIN(SAL) > (SELECT MIN(SAL)
  5                     FROM EMP
  6                     WHERE DEPTNO = 20);

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

그러나, 다음의 예는 2개 이상의 행이 리턴되는 복수행 서브 쿼리에 단일행 연산자를 사용하였기 때문에 오류가 발생한 경우이다.

```
SQL> SELECT ENAME
  2  FROM EMP
  3  WHERE SAL = (SELECT MIN(SAL)
  4               FROM EMP
  5               GROUP BY DEPTNO);
WHERE SAL = (SELECT MIN(SAL)
             *
3행에 오류:
ORA-01427: 단일 행 부속 질의에 2개 이상의 행이 리턴되었습니다
```

또한, 서브 쿼리가 NULL을 리턴 하는 경우에는 상황에 따라 원치 않는 결과를 얻을 수도 있다. 아래 서브 쿼리는 NULL을 메인 쿼리의 WHERE 절로 리턴하기 때문에 EMP 테이블의 JOB 컬럼에 NULL이 있든 없든 상관 없이 WHERE 절의 비교 결과가 NULL이 되어 어떤 결과도 출력할 수가 없게 된다. 그러므로, 서브 쿼리의 결과로 혹시 NULL이 나올 가능성이 있는지도 사전에 확인 할 필요가 있다.

```
SQL> SELECT ENAME
  2  FROM EMP
  3  WHERE JOB = (SELECT JOB
  4               FROM EMP
  5               WHERE ENAME = 'KIM');

선택된 레코드가 없습니다.
```

복수행 서브 쿼리

복수행 서브 쿼리는 두개 이상의 행을 리턴하며, 반드시 복수행(IN, ANY, ALL) 연산자를 사용해야만 한다.

■ IN

IN은 WHERE 조건에서 사용하는 일반 비교 연산자와 동일하다. 예를 들어, 사원 테이블에서 부서별 최소 급여를 받는 사원의 사번, 이름, 급여, 부서코드를 검색하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, SAL, DEPTNO
  2  FROM EMP
  3  WHERE SAL IN (SELECT MIN(SAL)
  4                FROM EMP
  5                GROUP BY DEPTNO);

     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7369 SMITH             800         20
      7900 JAMES             950         30
      7934 MILLER           1300         10
```

위 문장은 다음 문장과 동일하다.

```
SQL> SELECT EMPNO, ENAME, SAL, DEPTNO
  2  FROM EMP
  3  WHERE SAL IN (800, 950, 1300);
```

■ ALL

ALL은 복수행 서브 쿼리의 결과가 메인 쿼리의 WHERE 절에서 부등호 조건으로 비교될 때 사용되며, 서브 쿼리에서 리턴된 모든 결과값이 WHERE 절에서 모두 비교된다. ALL 연산자를 이해하기 위해 다음과 같은 예를 살펴보도록 하자.

사원 테이블에서 업무가 MANAGER인 사원의 급여보다 적은 급여를 받는 사원들의 이름을 검색하고자 한다. 먼저, 업무가 MANAGER인 사원의 급여를 검색하면 다음과 같다.

```
SQL> SELECT SAL
  2  FROM EMP
  3  WHERE JOB = 'MANAGER';

       SAL
----------
      2975
      2850
      2450
```

여기서, 업무가 MANAGER인 사원들의 최소 급여보다 급여가 적은 사원을 검색하려면, 다음과 같이 검색하여야 한다.

```
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE SAL < 2450;

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800
      7499 ALLEN            1600
      7521 WARD             1250
      7654 MARTIN           1250
      7844 TURNER           1500
      7876 ADAMS            1100
      7900 JAMES             950
      7934 MILLER           1300
```

위와 같은 절차를 서브 쿼리로 작성하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE SAL < ALL (SELECT SAL
  4                   FROM EMP
  5                   WHERE JOB='MANAGER');

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800
      7499 ALLEN            1600
      7521 WARD             1250
      7654 MARTIN           1250
      7844 TURNER           1500
      7876 ADAMS            1100
      7900 JAMES             950
      7934 MILLER           1300
```

■ ANY

ANY는 복수행 서브 쿼리의 결과가 메인 쿼리의 WHERE 절에서 부등호 조건으로 비교될 때 사용되며, 서브 쿼리에서 리턴된 각각의 결과값이 WHERE 절에서 비교된다. 마찬가지로 ANY 연산자를 이해하기 위해 다음과 같은 예를 살펴본다.

사원 테이블에서 업무가 MANAGER인 사원의 급여보다 많은 급여를 받는 사원들의 이름을 검색하고자 한다. 먼저, 업무가 MANAGER인 사원의 급여를 검색하면 다음과 같다.

```
SQL> SELECT SAL
  2  FROM EMP
  3  WHERE JOB = 'MANAGER';

       SAL
----------
      2975
      2850
      2450
```

여기서, 업무가 MANAGER인 사원들의 최소 급여보다 급여가 많은 사원을 검색하면 다음과 같이 검색하여야 한다.

```
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE SAL > 2450;

     EMPNO ENAME             SAL
---------- ---------- ----------
      7566 JONES            2975
      7698 BLAKE            2850
      7788 SCOTT            3000
      7839 KING             5000
      7902 FORD             3000
```

위와 같은 절차를 서브 쿼리로 작성하면 다음과 같다.

```
SQL>  SELECT EMPNO, ENAME, SAL
  2   FROM EMP
  3   WHERE SAL > ANY (SELECT SAL
  4                    FROM EMP
  5                    WHERE JOB='MANAGER');

     EMPNO ENAME             SAL
---------- ---------- ----------
      7566 JONES            2975
      7698 BLAKE            2850
      7788 SCOTT            3000
      7839 KING             5000
      7902 FORD             3000
```

ANY와 ALL 연산자를 정리하면 다음과 같다.

· < ANY : < 서브쿼리의 리턴값 중 최대값
· > ANY : > 서브쿼리의 리턴값 중 최소값
· < ALL : < 서브쿼리의 리턴값 중 최소값
· > ALL : > 서브쿼리의 리턴값 중 최대값

복습

1. 사원 테이블에서 BLAKE 보다 급여가 많은 사원들의 사번, 이름, 급여를 검색하시오.

2. 사원 테이블에서 MILLER보다 늦게 입사한 사원의 사번, 이름, 입사일을 검색하시오.

3. 사원 테이블에서 사원 전체 평균 급여보다 급여가 많은 사원들의 사번, 이름, 급여를 검색하시오.

4. 사원 테이블에서 CLARK와 같은 부서이며, 사번이 7698인 직원의 급여보다 많은 급여를 받는 사원들의 사번, 이름, 급여를 검색하시오.

5. 사원 테이블에서 부서별 최대 급여를 받는 사원들의 사번, 이름, 부서코드, 급여를 검색하시오.

Chapter 8. SQL*Plus 변수와 환경설정

이 장에서는 SQL*Plus와 iSQL*Plus에서 변수의 사용방법 및 사용자 고유의 환경을 설정하는 방법을 알아본다. 또한, 데이터베이스 검색 결과를 알아 보기 쉽도록 포맷팅 해주는 명령과 스크립트 파일의 작성 방법을 설명한다.

대체 변수

대체 변수(Substitution variable)는 SQL*Plus와 iSQL*Plus에서 사용할 수 있는 변수이다. 예를 들어, 사원 테이블에서 사번을 조건으로 급여를 검색하는 SQL 문장을 자주 작성해야 한다고 가정하자. 매번 같은 사원의 급여를 검색하는 일보다는 상황에 따라 매번 다른 사원의 급여를 검색하는 일이 더 많을 것이다. 이런 경우에 매번 새로운 SQL을 작성하는 것보다는 검색할 사번을 변수로 지정하여 SQL 문을 작성해두고, 조건에 따라 다른 사번을 변수에 저장한 후 기존에 작성해 두었던 SQL 문을 수정 없이 실행하는 편이 효율적일 것이다. 특히, SQL*Plus나 iSQL*Plus에서 대체 변수를 사용하면 변수가 사용된 SQL 문장이 실행될 때마다 사용자에게 변수의 값을 입력하도록 요청한다.

대체 변수는 대표적으로 다음과 같은 경우에 사용한다.
· 값들을 임시 저장하고자 할 때
· SQL 문장들 간에 값들을 전달하고자 할 때

■ & 대체 변수

변수명 앞에 &를 붙이면 해당 변수가 기술된 SQL 문장을 실행할 때마다 사용자에게 해당 변수의 값을 지정하도록 요구한다. 다음은 SQL*Plus에서 & 대체 변수를 사용한 예이다.

```
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE EMPNO = &EMPNUM;
empnum의 값을 입력하십시오: 7369
구    3: WHERE EMPNO = &EMPNUM
신    3: WHERE EMPNO = 7369

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800
```

물론, iSQL*Plus에서도 & 대체 변수를 사용할 수 있다. 먼저, iSQL*Plus에서 SQL 문장을 작성하고 **확인**을 클릭한 후, 대체 변수 값을 입력한다.

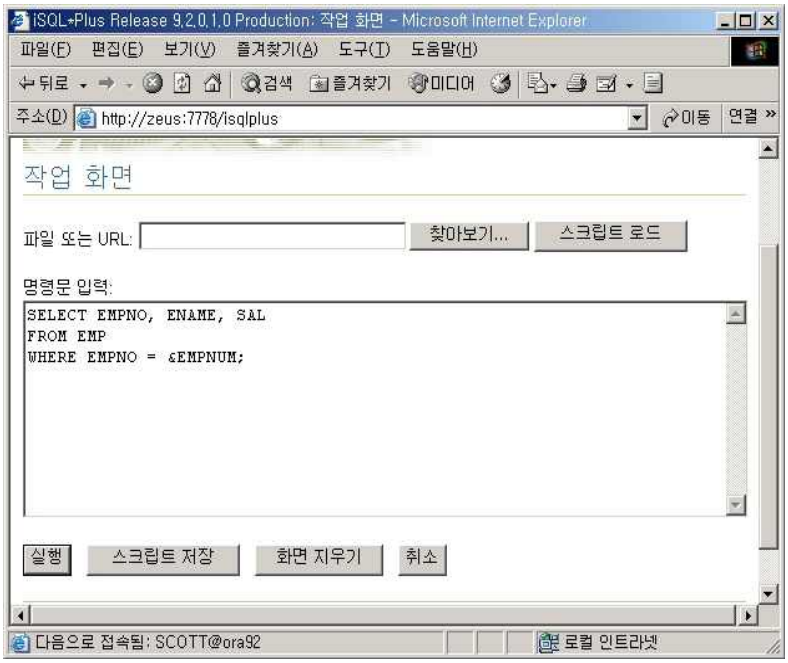


그림 8-1. iSQL*Plus 작업화면

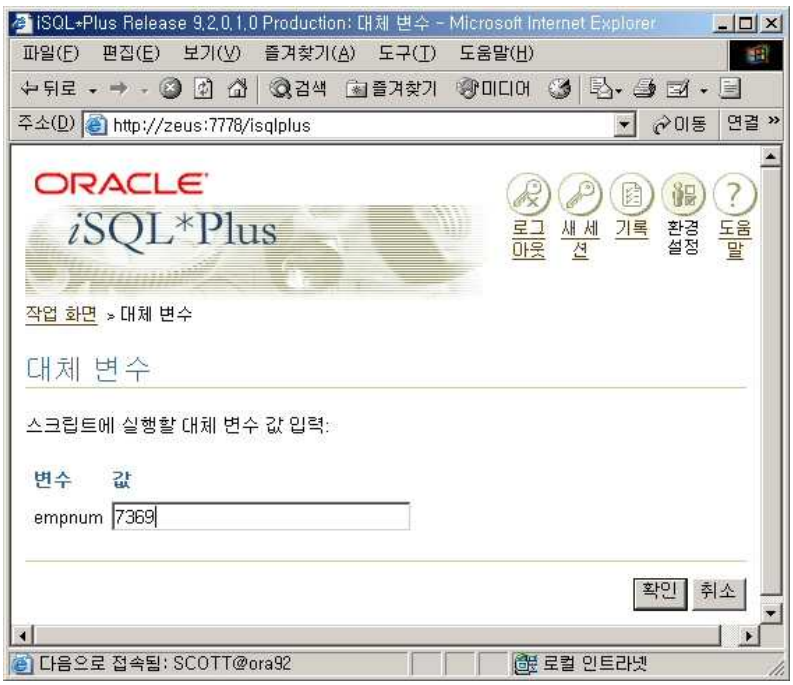


그림 8-2. iSQL*Plus 대체 변수 값 입력

검색 결과는 다음과 같이 나타난다.

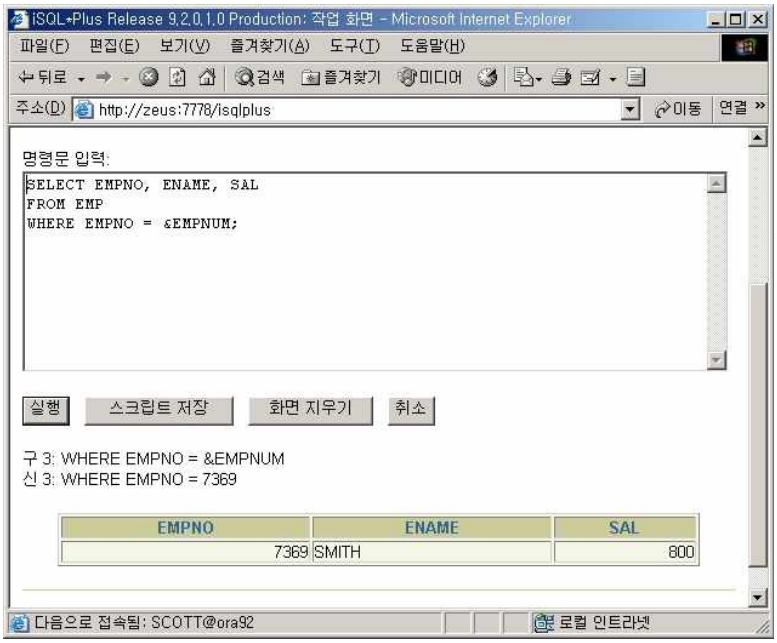


그림 8-3. iSQL*Plus 검색 결과

만약에 대체 변수에 문자 타입이나 날짜 타입의 값을 입력하려면 ‘ ’를 붙여준다.

```
SQL>SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE ENAME = '&EMPNAME';
empname의 값을 입력하십시오: KING
구    3: WHERE ENAME = '&EMPNAME'
신    3: WHERE ENAME = 'KING'

     EMPNO ENAME             SAL
---------- ---------- ----------
      7839 KING             5000
```

대체 변수는 다음과 같은 문장에 사용할 수 있다.

· WHERE 절
· ORDER BY 절
· 컬럼명
· 테이블명
· SELECT 절 전체

사용 예는 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, JOB, &COLUMN_NAME
  2  FROM EMP
  3  WHERE &CONDITION
  4  ORDER BY &ORDER_COLUMN;
column_name의 값을 입력하십시오: SAL
구   1: SELECT EMPNO, ENAME, JOB, &COLUMN_NAME
신   1: SELECT EMPNO, ENAME, JOB, SAL
condition의 값을 입력하십시오: SAL > 2000
구   3: WHERE &CONDITION
신   3: WHERE SAL > 2000
order_column의 값을 입력하십시오: ENAME DESC
구   4: ORDER BY &ORDER_COLUMN
신   4: ORDER BY ENAME DESC

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7788 SCOTT      ANALYST         3000
      7839 KING       PRESIDENT       5000
      7566 JONES      MANAGER         2975
      7902 FORD       ANALYST         3000
      7782 CLARK      MANAGER         2450
      7698 BLAKE      MANAGER         2850
```

■ DEFINE, UNDEFINE

대체 변수는 사용되기 전에 미리 선언 할 수 있다. 대체 변수를 선언하는 방법은 다음과 같다.

```
DEFINE variable = value
```

위와 같이 대체 변수를 선언하면 CHAR 타입의 변수가 생성되고 값이 지정 된다. 만약, 변수 값에 여백을 포함하려면 ' '를 붙여주면 되며, 선언 된 대체 변수는 해당 변수가 선언 된 세션내에서만 사용가능하다.

다음은 대체 변수를 선언하고, 선언 된 대체 변수를 확인하는 예이다.

```
SQL> DEFINE job_title = salesman
SQL> DEFINE job_title
DEFINE JOB_TITLE       = "salesman" (CHAR)
```

선언 된 변수를 해제하는 방법은 다음과 같다.

```
SQL> UNDEFINE job_title
SQL> DEFINE job_title
SP2-0135: 기호 job_title은 UNDEFINED
```

DEFINE 명령을 이용하여 대체 변수를 선언한 후, 선언된 대체 변수를 SQL 문장에서 사용하는 경우 해당 변수가 해제되기 전까지는 사용자가 변수 값을 입력할 수 없음을 주의하여야 한다.

```
SQL> DEFINE empnum = 7369
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE EMPNO = &empnum;
구   3: WHERE EMPNO = &empnum
신   3: WHERE EMPNO = 7369

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800
```

■ && 대체 변수

대체 변수 앞에 &&를 붙이면 사용자가 SQL 문장을 실행할 때마다 매번 변수값을 요구하지 않는다. 즉, 해당 대체 변수가 DEFINE에 의해 선언되어 있지 않다면 SQL 문장의 최초 실행시에만 변수값을 받아 들이고 그 다음부터는 변수값을 요구하지 않는다.

```
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE EMPNO = &&EMPNUM;
empnum의 값을 입력하십시오: 7369
구   3: WHERE EMPNO = &&EMPNUM
신   3: WHERE EMPNO = 7369

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800

SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE EMPNO = &&EMPNUM;
구   3: WHERE EMPNO = &&EMPNUM
신   3: WHERE EMPNO = 7369

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800
```

대체 변수 사용시, 입력 값의 확인 메시지를 보이지 않게 하려면 VERIFY 명령을 사용한다.

```
SQL> SET VERIFY OFF
SQL> SELECT EMPNO, ENAME, SAL
  2  FROM EMP
  3  WHERE SAL > &EMPSAL;
empsal의 값을 입력하십시오: 2000

     EMPNO ENAME             SAL
---------- ---------- ----------
      7566 JONES            2975
      7698 BLAKE            2850
        ...
      7902 FORD             3000
```

다시 원래대로 메시지가 보이도록 하려면 SET VERIFY ON 명령을 사용하면 된다.

환경 설정

SQL*Plus 및 iSQL*Plus의 환경 설정을 변경하기 위해서는 SET 명령을 이용해 관련 시스템 변수의 설정값을 수정해야 한다.

```
SET system_variable value
```

SHOW 명령을 이용하여 시스템 변수의 설정값을 확인할 수 있다.

```
SHOW system_variable
```

다음은 시스템 변수의 설정값을 수정하고 확인하는 예이다.

```
SQL> SHOW PAGESIZE
pagesize 14
SQL> SET PAGESIZE 30
SQL> SHOW PAGESIZE
pagesize 30
```

또한, SQL*Plus 메뉴의 **옵션-환경**을 선택하면 시스템 변수를 설정할 수 있는 **환경** 대화상자가 표시된다.

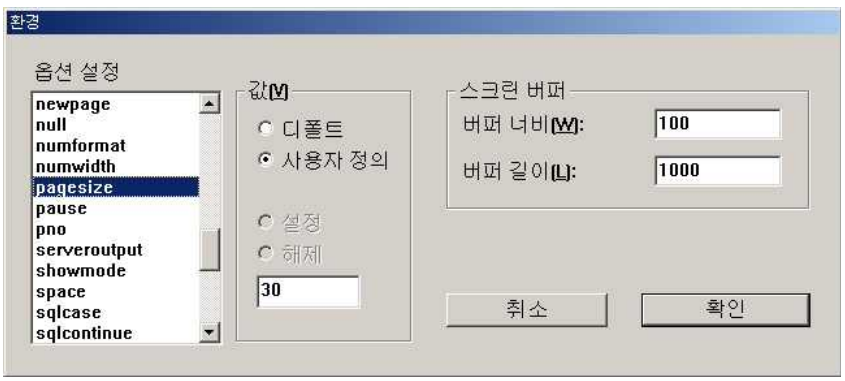


그림 8-4. 시스템 변수 설정

iSQL*Plus에서는 우측 상단의 **환경설정** 링크를 클릭하면 **환경설정** 페이지가 표시되고 **시스템 변수 설정**의 **이동** 버튼을 클릭한다. **시스템 변수** 페이지에서 빠른 링크를 이용하여 시스템 변수를 입력 또는 선택한 후 **이동** 버튼을 클릭하거나 마우스로 스크롤하여 해당 시스템 변수를 변경할 수 있는 항목이 있는 곳으로 이동한다.

그림 8-5. 환경 설정 페이지

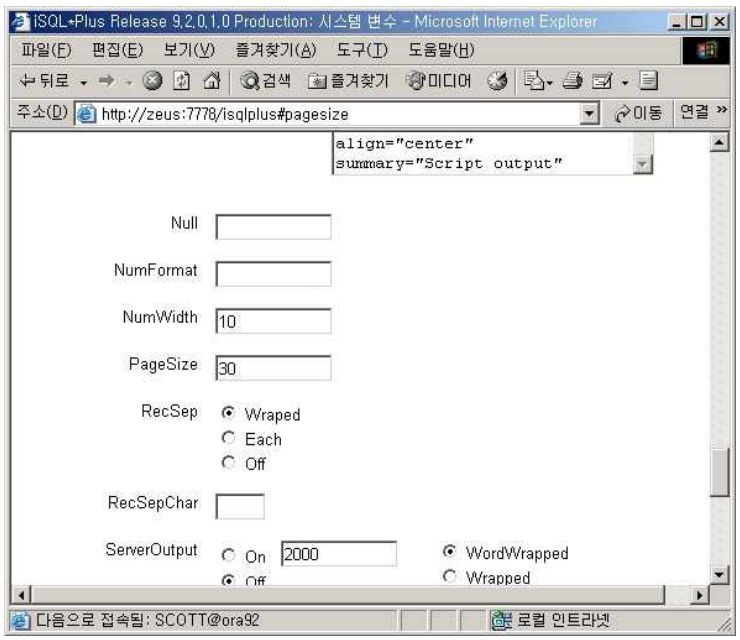


그림 8-6. 시스템 변수 설정

시스템 변수의 개수는 상당히 많지만 실제 사용하는 변수는 그리 많지 않다. 아래 표는 활용도가 높은 시스템 변수의 예이다.

표 8-1. 대표적인 시스템 변수

시스템 변수	설명
LINESIZE	검색 결과의 출력 너비를 지정
PAGESIZE	검색 결과의 한 페이지에 출력될 라인 수를 지정
HEADING	검색 결과에서 컬럼명의 표시 여부
FEEDBACK	SQL 문장 실행 후, 결과 메시지 표시 여부
SQLPROMPT	프롬프트 지정

포맷 명령

SQL*Plus 및 iSQL*Plus의 포맷 명령은 SQL 문장에 의한 검색 결과를 보고서로 작성할 수 있도록 도와주는 기능이다. 포맷 명령어는 다음과 같은 것들이 있다.

표 8-2. 포맷 명령

포맷 명령	설명
COL[UMN] [*column option*]	컬럼 포맷을 설정
TTI[TLE] [*text*|OFF|ON]	검색 결과의 각 페이지 최상단에 표시될 제목
BTI[TLE] [*text*|OFF|ON]	검색 결과의 각 페이지 최하단에 표시될 제목
BRE[AK] [ON *report_element*]	중복된 값의 출력을 억제

위 포맷 명령어 또한 대체 변수처럼 해당 세션에서만 유효하다.

■ COLUMN

COLUMN 명령을 사용하는 방법은 다음과 같다.

```
COL[UMN] [{column|alias} [option]]
```

위에서 *option*으로 지정할 수 있는 항목은 다음과 같다.

표 8-3. COLUMN 명령 *option*

Option	설명
CLE[AR]	컬럼 포맷을 삭제
HEA[DING] *text*	컬럼명을 *text*로 지정(컬럼명의 |는 두줄에 걸쳐서 출력)
FOR[MAT] *format*	컬럼 데이터의 출력 형식
NOPRI[NT]	컬럼을 출력하지 않음
NUL[L] *text*	컬럼값이 NULL이면 *text* 출력
PRI[NT]	컬럼을 출력

지정 가능한 컬럼 *format*은 다음과 같다.

표 8-4. 컬럼 *format*

항목	설명	예	결과
9	숫자 1자리	999999	1234
0	숫자 1자리. 숫자가 지정되지 않으면 0으로 출력	099999	001234
$	달러 표시	$9999	$1234
L	지역 화폐단위	L9999	₩1234
.	소수점	9999.99	1234.00
,	천단위 구분자	9,999	1,234

COLUMN 명령을 이용하여 검색 결과를 보기 좋게 수정해보자. 포맷 명령을 사용하기전의 출력결과가 다음과 같다고 가정한다.

```
SQL> SELECT EMPNO, ENAME, JOB, MGR, SAL FROM EMP;

     EMPNO ENAME      JOB              MGR        SAL
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK           7902        800
      7499 ALLEN      SALESMAN        7698       1600
      7521 WARD       SALESMAN        7698       1250
      7566 JONES      MANAGER         7839       2975
      7654 MARTIN     SALESMAN        7698       1250
      7698 BLAKE      MANAGER         7839       2850
      7782 CLARK      MANAGER         7839       2450
      7788 SCOTT      ANALYST         7566       3000
      7839 KING       PRESIDENT                  5000
      7844 TURNER     SALESMAN        7698       1500
      7876 ADAMS      CLERK           7788       1100
      7900 JAMES      CLERK           7698        950
      7902 FORD       ANALYST         7566       3000
      7934 MILLER     CLERK           7782       1300
```

위 검색 결과를 다음과 같이 변경해보자.

· EMPNO 컬럼의 컬럼명을 '사번', ENAME 컬럼은 '사원명', JOB 컬럼은 '업무', MGR 컬럼은 '관리자사번', SAL 컬럼은 '급여'로 변경한다. 단, MGR 컬럼의 컬럼명은 '관리자'와 '사번'으로 두줄에 표현되도록 한다.
· EMPNO 컬럼의 폭은 5문자로 줄인다.
· MGR 컬럼의 폭은 5문자로 줄인다.
· ENAME 컬럼의 폭은 7문자로 줄인다.
· ENAME 컬럼은 우측 정렬한다.
· SAL 컬럼은 천단위마다 ,를 붙인다.
· MGR 컬럼의 값이 NULL인 경우는 '사장'으로 출력한다.

```
SQL> COLUMN EMPNO HEADING '사번'
SQL> COLUMN ENAME HEADING '사원명'
SQL> COLUMN JOB HEADING '업무'
SQL> COLUMN MGR HEADING '관리자|사번'
SQL> COLUMN SAL HEADING '급여'
SQL> COLUMN EMPNO FORMAT 99999
SQL> COLUMN MGR FORMAT 99999
SQL> COLUMN ENAME FORMAT A7
SQL> COLUMN ENAME JUSTIFY RIGHT
SQL> COLUMN SAL FORMAT 999,999
SQL> COLUMN MGR NULL '사장'
SQL> SELECT EMPNO, ENAME, JOB, MGR, SAL FROM EMP;

                          관리자
사번    사원명 업무       사번       급여
------ ------- --------- ------ --------
  7369 SMITH   CLERK       7902      800
  7499 ALLEN   SALESMAN    7698    1,600
  7521 WARD    SALESMAN    7698    1,250
  7566 JONES   MANAGER     7839    2,975
  7902 FORD    ANALYST     7566    3,000
         ...
  7934 MILLER  CLERK       7782    1,300
```

■ BREAK

BREAK 명령은 중복값의 출력을 억제하는 명령이다.

```
BREAK ON column[|alias|row]
```

BREAK 명령은 대상이 되는 컬럼이 ORDER BY에 의해 정렬된 후 검색해야만 의미가 있다.

```
SQL> BREAK ON JOB
SQL> SELECT EMPNO, ENAME, JOB
  2  FROM EMP
  3  ORDER BY JOB;

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

■ TTITLE, BTITLE

TTITLE, BTITLE은 각각 페이지의 최상단, 최하단에 출력할 제목을 지정하는 것이다.

```
TTI[TLE]|BTI[TLE] [text|OFF|ON]
```

```
SQL> TTITLE '부서 리스트'
SQL> BTITLE '대외비'
SQL> SELECT * FROM DEPT;

토 Jun 19
                                부서 리스트

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

                                   대외비
```

스크립트 파일

스크립트 파일은 복잡한 SQL 문장의 외부의 텍스트 파일로 저장한 것이다. 외부에 저장된 스크립트 파일은 SQL*Plus 및 iSQL*Plus에서 언제든지 불러들여 실행시킬 수 있다. 다음은 스크립트 파일을 작성하는 방법이다.

1. SQL SELECT 문장을 작성하고 올바르게 실행되는지 테스트한다.
2. 작성된 SELECT 문장을 스크립트 파일로 저장한다.
3. 스크립트 파일을 일반 편집기로 불러들인다.
4. SELECT 문장의 앞 부분에 포맷 명령을 추가한다.
5. SELECT 문장의 끝에 ; 또는 /이 있는지 확인한다.
6. SELECT 문장의 뒷 부분에 앞에서 지정한 포맷을 삭제하는 포맷 명령을 작성한다.
7. 스크립트 파일을 저장한다.
8. SQL*Plus 또는 iSQL*Plus에서 스크립트 파일을 실행한다.

복습

iSQL*Plus에서 다음과 같은 보고서를 생성하는 스크립트를 작성하시오.

Chapter 9. 데이터 조작

이번 장에서는 DML 문장을 이용하여 테이블에 새로운 행을 입력하고 기존에 입력되어 있는 행을 수정 및 삭제하는 방법을 알아본다. 또한, Oracle 9i 버전부터 새롭게 추가된 행의 머지(Merge) 구문과 트랜잭션을 제어하는 방법을 설명한다.

데이터 조작 언어(DML)

데이터 조작 언어(DML : Data Manipulation Language)란 테이블에 새로운 행을 입력하고, 기존에 입력된 행을 수정 및 삭제하는 명령이다. 여기서, 한개 이상의 DML 문장들은 하나의 트랜잭션(Transaction)으로 구성되는데, 트랜잭션은 데이터베이스의 논리적 작업 단위를 의미한다. 예를 들어, 은행에서의 계좌이체 작업을 생각해보자. 즉, A 구좌에서 B 구좌로의 계좌이체 작업은 전체 3가지 작업으로 완료 된다. 첫 번째는 A 구좌의 잔고를 감소시키고, 두 번째는 A 구좌의 감소된 잔고만큼 B 구좌의 잔고를 증가시킨 다음, 마지막으로 계좌이체 작업을 기록하는 것이다. 만약에 이 세 가지 작업 중에 하나의 작업이라도 실패하게 되었다면 이러한 모든 계좌 이체 작업은 전부 취소되어야 한다. 즉, 이 세 가지 데이터 조작 작업은 하나의 논리적 작업 단위로서 트랜잭션으로 처리되어야 한다는 것이다.

INSERT

INSERT는 테이블에 새로운 행을 입력하는 문장이다. INSERT 구문의 문법은 다음과 같다.

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

INSERT 구문은 한번에 하나의 행만을 입력할 수 있다.

```
SQL> INSERT INTO DEPT(DEPTNO, DNAME, LOC)
  2  VALUES(90, '인사과', '서울');

1 개의 행이 만들어졌습니다.
```

위 문장에서 테이블명 뒤에 컬럼들을 기술하지 않으면 테이블내의 컬럼 순서(DESC 명령으로 출력되는 컬럼 순서)에 일치하도록 입력할 값들을 기술해야한다. 즉, 위 문장은 아래 문장과 동일하다.

```
INSERT INTO DEPT VALUES(90, '인사과', '서울');
```

입력할 값이 문자 타입이나 날짜 타입인 경우는 ‘ ’를 붙여준다.

■ NULL 값 입력

NULL 값을 갖는 행을 입력하는 방법은 두 가지 방법이 있으며, 첫 번째는 NULL 값이

입력될 컬럼을 INSERT 문장에 기술하지 않는 방법이다.

```
SQL> INSERT INTO DEPT(DEPTNO, DNAME)
  2  VALUES(91, '총무과');

1 개의 행이 만들어졌습니다.
```

두 번째 방법은 NULL 값 대신에 NULL이라는 키워드를 기술하는 방법이다.

```
SQL> INSERT INTO DEPT
  2  VALUES(92, '경리과', NULL);

1 개의 행이 만들어졌습니다.
```

■ 특수 값 입력

INSERT 문장에 SYSDATE, USER와 같은 특수 함수를 이용하면 테이블에 시스템의 날짜 또는 현재 사용자의 계정을 입력 할 수 있다.

```
SQL> INSERT INTO EMP(EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO)
  2  VALUES(9000, USER, '연구원', 7839, SYSDATE, 5000, NULL, 90);

1 개의 행이 만들어졌습니다.
```

■ 날짜 입력

테이블에 날짜 타입을 입력할 때는 INSERT 문장에서 날짜를 RR/MM/DD 형식으로 입력하여야 날짜 타입으로 인식되어 저장되지만, DD-MON-RR과 같은 형식으로 입력하면 문자 타입으로 인식되어 올바르게 저장되지 않는다. 이러한 경우, TO_DATE 함수를 사용하여 날짜 타입으로 변환해야만 한다.

```
SQL> INSERT INTO EMP(EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO)
  2  VALUES(9001, '홍길동', '상담원', 7839, TO_DATE('19-JUN-04', 'DD-MON-RR'),
  3  3000, NULL, 30);

1 개의 행이 만들어졌습니다.
```

■ 대체 변수를 이용한 입력

INSERT 문장을 반복해서 사용하는 경우, 대체 변수를 사용하면 좀 더 편리하게 데이터 입력 작업을 수행 할 수 있다.

```
SQL> INSERT INTO DEPT
  2  VALUES(&DEPT_NO, '&DEPT_DNAME', '&DEPT_LOC');
dept_no의 값을 입력하십시오: 93
dept_dname의 값을 입력하십시오: 홍보과
dept_loc의 값을 입력하십시오: 대전
구   2: VALUES(&DEPT_NO, '&DEPT_DNAME', '&DEPT_LOC')
신   2: VALUES(93, '홍보과', '대전')

1 개의 행이 만들어졌습니다.
```

■ 다른 테이블로부터 행 복사

INSERT 문장은 한번에 하나의 행만 입력할 수 있으나, INSERT INTO ... SELECT 문

장을 이용하면 다른 테이블에 저장된 복수개의 행들을 한번에 입력할 수 있다.

다음과 같이 실습용 테이블을 생성한 후, 사원 테이블에서 부서코드가 10번인 사원 데이터들을 실습용 테이블 EMP10에 입력해보자.

```
SQL> CREATE TABLE EMP10
  2  (EMPNO NUMBER(4),
  3  ENAME VARCHAR2(10),
  4  JOB VARCHAR2(10),
  5  SAL NUMBER(7,2));

테이블이 생성되었습니다.

SQL> INSERT INTO EMP10(EMPNO, ENAME, JOB, SAL)
  2  SELECT EMPNO, ENAME, JOB, SAL
  3  FROM EMP
  4  WHERE DEPTNO = 10;

3 개의 행이 만들어졌습니다.
```

■ INSERT 문장에서 서브 쿼리 사용

INSERT 문장에서 테이블명 대신 서브 쿼리를 사용할 수도 있다.

```
SQL> INSERT INTO
  2  (SELECT EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO
  3   FROM EMP
  4   WHERE DEPTNO = 30)
  5  VALUES (9002, '콩쥐', '사무원', 7934, '04/01/01', 3500, NULL, 30);

1 개의 행이 만들어졌습니다.
```

위 문장에서 서브 쿼리에 WITH CHECK OPTION을 사용하면 서브 쿼리의 WHERE 조건에 만족하는 행만을 입력할 수 있다. 다음과 같이 WHERE 절의 조건과 입력 데이터가 일치 되지 않으면 오류가 발생한다.

```
SQL> INSERT INTO
  2  (SELECT EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO
  3   FROM EMP
  4   WHERE DEPTNO = 30 WITH CHECK OPTION)
  5  VALUES (9003, '팥쥐', '보조원', 7934, '04/04/05', 1500, NULL, 40);
FROM EMP
     *
3행에 오류:
ORA-01402: 뷰의 WITH CHECK OPTION의 조건에 위배 됩니다
```

UPDATE

UPDATE는 테이블에 저장되어 있는 행들을 변경하는 문장이다. UPDATE 구문의 문법은 다음과 같다.

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

UPDATE 구문은 한번에 여러 개의 행을 변경 할 수 있다.

```
SQL> UPDATE DEPT
  2  SET LOC = '부산'
  3  WHERE DEPTNO = 90;

1 행이 갱신되었습니다.
```

위 문장에서 WHERE 절이 생략되면 테이블 내의 모든 행이 변경되므로 주의할 필요가 있다.

■ 서브 쿼리를 이용한 복수 컬럼 변경

UPDATE 문장에서 여러 개의 컬럼을 한번에 변경 할 수 있으며, 서브 쿼리가 포함 될 수도 있다.

```
SQL> UPDATE EMP
  2  SET JOB = (SELECT JOB
  3             FROM EMP
  4             WHERE EMPNO = 7900),
  5      SAL = (SELECT SAL
  6             FROM EMP
  7             WHERE EMPNO = 7844)
  8  WHERE EMPNO = 9000;

1 행이 갱신되었습니다.
```

■ 다른 테이블을 기반으로 테이블의 행 변경

UPDATE 문장의 서브 쿼리에서 참조하는 테이블과 변경하고자 하는 테이블이 다를 수도 있다.

```
SQL> UPDATE EMP10
  2  SET JOB = (SELECT JOB
  3             FROM EMP
  4             WHERE EMPNO = 7844)
  5  WHERE EMPNO = (SELECT EMPNO
  6                 FROM EMP
  7                 WHERE ENAME = 'MILLER');

1 행이 갱신되었습니다.
```

■ 행 변경 오류 : 무결성 제약조건 위반

다른 테이블을 참조하는 테이블의 행을 변경하는 경우, 외래키 제약조건에 의해 오류가 발생 될 수 있다. 예를 들어, 다음 UPDATE 문장은 사번이 9001번인 사원의 부서코드를 50번으로 변경하는 문장인데, 부서 테이블에는 50번 부서가 존재하지 않기 때문에 외래키 제약조건에 위배되어 해당 행이 변경되지 않는다.

```
SQL> UPDATE EMP
  2  SET DEPTNO = 50
  3  WHERE EMPNO = 9001;
UPDATE EMP
*
1행에 오류:
ORA-02291: 무결성 제약조건(SCOTT.FK_DEPTNO)이 위배되었습니다- 부모 키가 없습니
다
```

DELETE

DELETE는 테이블에 저장되어 있는 행들을 삭제하는 문장이다. DELETE 구문의 문법은 다음과 같다.

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

DELETE 구문은 한번에 여러 개의 행을 삭제 할 수 있다.

```
SQL> DELETE FROM DEPT
  2  WHERE DEPTNO = 93;

1 행이 삭제되었습니다.
```

위 문장에서 WHERE 절이 생략되면 테이블 내의 모든 행이 삭제되므로 주의할 필요가 있다.

■ 다른 테이블을 기반으로 테이블의 행 삭제

DELETE 문장의 서브 쿼리에서 참조하는 테이블과 삭제하고자 하는 테이블이 다를 수도 있다.

```
SQL> DELETE FROM EMP
  2  WHERE DEPTNO = (SELECT DEPTNO
  3                  FROM DEPT
  4                  WHERE DNAME = '인사과');

1 행이 삭제되었습니다.
```

■ 행 삭제 오류 : 무결성 제약조건 위반

다른 테이블에 의해 참조되는 테이블의 행을 삭제하는 경우, 외래키 제약조건에 의해 오류가 발생 할 수 있다. 예를 들어, 다음 DELETE 문장은 부서 테이블에서 부서코드가 10번인 행을 삭제하는 문장인데, 10번 부서는 사원 테이블에서 사원들이 참조하고 있는 부서이므로 외래키 제약조건에 위배되어 해당 행이 삭제되지 않는다.

```
SQL> DELETE FROM DEPT
  2  WHERE DEPTNO = 10;
DELETE FROM DEPT
*
1행에 오류:
ORA-02292: 무결성 제약조건(SCOTT.FK_DEPTNO)이 위배되었습니다- 자식 레코드가 발
견되었습니다
```

DEFAULT

DEFAULT는 INSERT와 UPDATE 문장에서 입력 또는 변경하고자 하는 값 대신에 기술할 수 있는 키워드이다. DEFAULT가 사용되면 변경하고자 하는 테이블의 컬럼에 지정된 디폴트 값으로 입력 또는 변경된다. DEFAULT는 SQL : 1999 표준과의 호환을 위해서 Oracle 9i에서 새롭게 추가된 키워드이다. 만약, 테이블의 컬럼에 디폴트가 지정되어 있지 않은 경우에 DEFAULT 키워드를 사용하면 NULL 값이 적용된다. 다음과 같이 실습용 테이블을 만들고 DEFAULT를 이용하여 데이터를 입력해보자

```
SQL> CREATE TABLE TEST1
  2  (NO NUMBER(4),
  3  NAME VARCHAR2(10) DEFAULT 'UNKNOWN',
  4  AGE NUMBER(4) DEFAULT 0);

테이블이 생성되었습니다.

SQL> INSERT INTO TEST1 VALUES(1, '길동', DEFAULT);

1 개의 행이 만들어졌습니다.

SQL> SELECT * FROM TEST1;

        NO NAME              AGE
---------- ---------- ----------
         1 길동                0
```

UPDATE 문장에도 DEFAULT를 사용할 수 있다.

```
SQL> UPDATE TEST1 SET NAME = DEFAULT WHERE NO = 1;

1 행이 갱신되었습니다.

SQL> SELECT * FROM TEST1;

        NO NAME              AGE
---------- ---------- ----------
         1 UNKNOWN             0
```

MERGE

MERGE는 조건에 따라 데이터를 입력하거나 변경 할 수 있는 문장이다. 즉, 입력할 데이터가 대상 테이블에 존재하지 않으면 INSERT가 수행되며, 동일한 데이터가 대상 테이블에

존재하면 UPDATE가 수행된다. MERGE 문장의 문법은 다음과 같다.

```
MERGE INTO table_name table_alias
  USING (table|view|sub_query) alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
    col1 = col1_val,
    col2 = col2_val
  WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```

EMP10 테이블의 데이터를 EMP 테이블의 데이터로 갱신해보자. 즉, EMP 테이블에 저장된 행이 EMP10 테이블에 존재하면 EMP10 테이블의 해당 행을 EMP 테이블에 저장된 행으로 UPDATE하고, 존재하지 않으면 EMP10 테이블에 해당 행을 INSERT하는 것이다.

```
SQL> MERGE INTO EMP10 N
  2     USING EMP O
  3     ON (N.EMPNO = O.EMPNO)
  4     WHEN MATCHED THEN
  5       UPDATE SET
  6       N.ENAME = O.ENAME,
  7       N.JOB = O.JOB,
  8       N.SAL = O.SAL
  9     WHEN NOT MATCHED THEN
 10       INSERT (N.EMPNO, N.ENAME, N.JOB, N.SAL)
 11       VALUES (O.EMPNO, O.ENAME, O.JOB, O.SAL);

16 행이 병합되었습니다.
```

위의 예를 보면 EMP10 테이블과 EMP 테이블에서 동일한 행의 존재 여부는 두 테이블의 EMPNO 컬럼에 동일한 값이 존재하는지 여부로 판단한다. 즉, 동일한 사번이 있으면 EMP 테이블의 데이터로 EMP10 테이블을 변경하고, 동일한 사번이 없으면 EMP 테이블의 데이터를 입력한다.

트랜잭션(Transaction)

Oracle 데이터베이스는 트랜잭션에 의해 데이터의 일관성이 보장된다. 트랜잭션은 데이터 변경시 사용자에게 유연성 및 제어성을 부여해주고 사용자 프로세스 또는 시스템 오류가 발생 했을 때, 데이터의 일관성을 지켜주는 역할을 한다. 트랜잭션은 데이터의 일관성 있는 변경을 수행하는 여러 개의 DML 문장으로 구성된다. 예를 들어, 은행에서 사용자 구좌 간에 발생하는 계좌이체는 차변과 대변 값이 정확하게 일치해야하기 때문에 각 계좌에서 발생하는 작업은 모두 성공하거나 실패 되도록 하여야 한다.

트랜잭션의 타입은 다음과 같다.

표 9-1. 트랜잭션의 타입

타입	설명
DML(Data Manipulation Language)	데이터베이스의 논리적 작업 단위로서 여러 개의 DML 문장으로 구성
DDL(Data Definition Language)	하나의 DDL 문장으로 구성
DCL(Data Control Language)	하나의 DCL 문장으로 구성

그렇다면 트랜잭션이 시작되는 시점과 종료되는 시점을 알아보자. 트랜잭션은 첫 번째 DML 문장이 실행되면 시작되고 다음과 같은 상황에서 트랜잭션은 종료된다.

· 사용자가 COMMIT 또는 ROLLBACK 명령을 실행한 경우
· CREATE 명령과 같은 DDL 문장을 실행한 경우
· DCL 문장을 실행한 경우
· 사용자가 SQL*Plus 또는 iSQL*Plus를 종료한 경우
· 하드웨어 고장 또는 시스템 오류

사용자가 COMMIT 또는 ROLLBACK 명령을 사용하여 트랜잭션을 직접 제어함으로써 다음과 같은 장점을 얻을 수 있다.

· 데이터의 일관성을 보장해준다.
· 데이터의 변경사항을 데이터베이스에 영구히 반영하기 전에 데이터 변경사항을 미리 볼 수 있다.
· 논리적으로 연관된 작업들을 그룹화 할 수 있다.

■ 트랜잭션 제어 명령
트랜잭션을 제어하는 명령은 다음과 같다.

표 9-2. 트랜잭션 제어 명령

명령	설명
COMMIT	현재 진행 중인 트랜잭션을 종료하며 모든 변경사항을 데이터베이스에 영구히 반영
SAVEPOINT *name*	현재 진행 중인 트랜잭션의 중간에 저장점 *name*을 표시
ROLLBACK	현재 진행 중인 트랜잭션을 종료하며 모든 변경사항을 취소
ROLLBACK TO *name*	현재 진행 중인 트랜잭션의 저장점으로 복귀하고 저장점 이후의 모든 데이터 변경사항을 취소

트랜잭션을 제어하는 명령 중에 SAVEPOINT란 다음 그림과 같이 트랜잭션이 진행중일 때, 임의 지점에 저장점을 기록함으로써 ROLLBACK TO 명령으로 언제든지 사용자가 원하는 저장점으로 데이터를 복원할 수 있도록 해준다. 여기서, ROLLBACK TO 명령은 진행 중인 트랜잭션의 저장점으로만 복원하는 것이지 트랜잭션을 종료시키지 않음을 주의해야 한다. 최종적으로 트랜잭션을 종료하려면 COMMIT 또는 ROLLBACK 명령을 사용해야 한다. 또한, 동일한 트랜잭션내에서 두 개 이상의 저장점을 지정하는 경우에 저장점의 이름이 중복되면 이전의 지정된 저장점은 삭제된다.

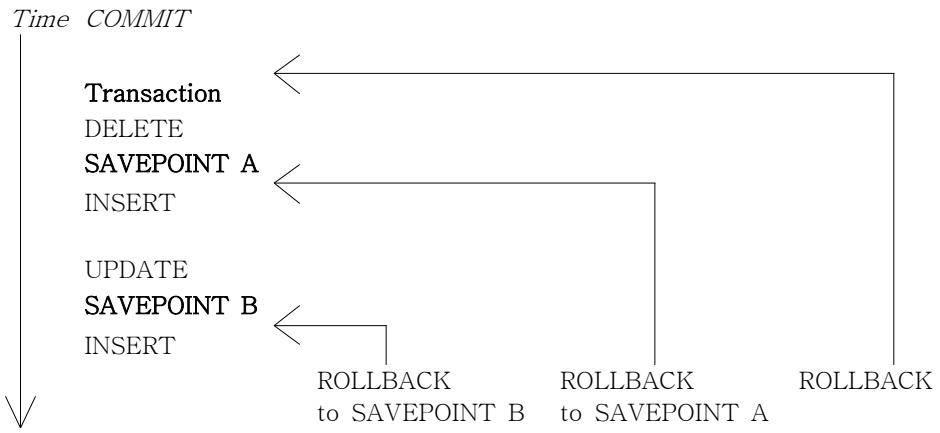


그림 9-1. 트랜잭션 제어 명령

■ 암시적 트랜잭션 처리(Implicit transaction processing)
암시적 트랜잭션에는 자동 커밋(Automatic commit)과 자동 롤백(Automatic rollback)이 있으며 다음과 같이 분류할 수 있다.

표 9-3. 암시적 트랜잭션 처리의 종류

종류	상황
Automatic commit	DDL 또는 DCL 문장의 실행시 SQL*Plus 또는 iSQL*Plus에서 COMMIT 또는 ROLLBACK 명령을 사용하지 않고 정상 종료
Automatic rollback	SQL*Plus 또는 iSQL*Plus에서 비정상 종료 또는 시스템 오류

위에서 SQL*Plus 또는 iSQL*Plus의 정상종료라 함은 SQL*Plus에서 EXIT 명령을 입력하거나 iSQL*Plus에서 로그아웃 링크를 클릭하여 종료하는 것을 의미한다.

■ 트랜잭션 종료 전의 데이터 상태
트랜잭션내의 모든 데이터 변경사항은 트랜잭션이 종료되기 이전까지는 임시적이다. COMMIT 또는 ROLLBACK 명령을 발생하기 전까지의 데이터 상태는 다음과 같다.

- 데이터 변경 작업은 주로 데이터베이스 버퍼에서 수행되므로, 데이터의 변경전 데이터는 복구 될 수 있다.
- 현재 사용자는 SELECT 문장을 이용하여 데이터 변경 후의 결과를 확인 할 수 있다.
- 다른 사용자들은 현재 사용자에 의해 변경된 데이터 결과를 확인할 수 없다. Oracle 데이터베이스는 다른 사용자들에게 해당 데이터의 가장 최근의 커밋된 결과를 보여줌으로써 데이터의 일관성을 보장한다.
- 변경된 행은 잠금(Lock)이 걸리며, 다른 사용자들은 해당 행들을 변경 할 수 없다.

COMMIT

COMMIT은 모든 데이터 변경사항을 데이터베이스에 영구히 반영시키는 명령으로 다음과

같은 작업이 수행된다.

· 데이터의 변경사항은 모두 데이터베이스에 기록된다.
· 변경 전의 데이터는 모두 잃게 된다.
· 모든 사용자들이 트랜잭션 종료 후의 결과를 볼 수 있다.
· 트랜잭션이 진행 중이었던 행들에 대한 잠금이 모두 해소되며, 다른 사용자에 의해서 변경이 가능해진다.
· 모든 SAVEPOINT는 삭제된다.

COMMIT 명령을 사용하는 방법은 다음과 같다.

```
SQL> INSERT INTO EMP
  2  VALUES (9003, '팥쥐', '상담원', 7698, '04/03/01', 3100, NULL, 30);

1 개의 행이 만들어졌습니다.

SQL> DELETE FROM DEPT
  2  WHERE DEPTNO = 92;

1 행이 삭제되었습니다.

SQL> COMMIT;

커밋이 완료되었습니다.
```

ROLLBACK

ROLLBACK은 모든 데이터 변경사항을 취소하는 명령어로 다음과 같은 작업이 수행된다.
· 데이터의 모든 변경 사항이 취소 된다.
· 변경 전의 데이터가 복원 된다.
· 트랜잭션이 진행 중이었던 행들에 대한 잠금이 모두 해소 된다,

ROLLBACK 명령을 사용하는 방법은 다음과 같다.

```
SQL> DELETE FROM EMP;

17 행이 삭제되었습니다.

SQL> ROLLBACK;

롤백이 완료되었습니다.
```

■ 문장 수준의 롤백
만약, DML 문장에 오류가 감지되면 트랜잭션의 일부분이 취소될 수 있다. 트랜잭션이 진행되는 중간에 단일 DML 문장이 실패하게 되면 해당 문장은 문장 수준의 롤백에 의해 취소되지만 해당 문장 이전의 DML 문장에 의해 발생된 변경사항은 취소되지 않는다.

한 가지 주의할 점으로 Oracle은 DDL 문장의 실행 전과 실행 후에 암시적 커밋을 발생시키는데, 해당 DDL 문장이 성공하지 못하더라고 해당 DDL 문장 전에 커밋이 발생하기 때문에 이전의 모든 데이터 변경사항은 롤백 되지 못한다.

읽기 일관성(Read Consistency)

데이터의 읽기 일관성이 필요한 이유는 다음과 같다.

- 데이터를 검색하는 사용자들과 변경하는 사용자들 사이에 일관적인 관점을 제공한다. 즉, 다른 사용자들이 변경 중인 데이터를 볼 수 없게 한다.
- 데이터를 변경하는 사용자들 사이에 일관적인 데이터베이스 변경 방법을 제공함으로써 동일한 데이터의 동시에 변경함으로써 발생 할 수 있는 혼란을 방지한다.

결과적으로 읽기 일관성의 목적은 각각의 사용자들에게 다른 사용자들의 DML 작업이 시작되기 이전의 데이터 즉, 가장 최근에 커밋 된 데이터를 보여주는 것이다.

■ 읽기 일관성의 구현 원리

읽기 일관성은 Oracle에 의해 자동으로 관리되며, 데이터의 변경 이전 값을 UNDO 세그먼트에 저장함으로써 구현된다. 즉, 데이터베이스에 INSERT, UPDATE, DELETE 문장이 실행되면 Oracle 데이터베이스는 변경 전 데이터를 UNDO 세그먼트에 임시적으로 저장한다. 이러한 상황에서 해당 데이터를 변경한 사용자를 제외한 모든 사용자들은 UNDO 세그먼트의 변경 전 데이터를 보게 된다.

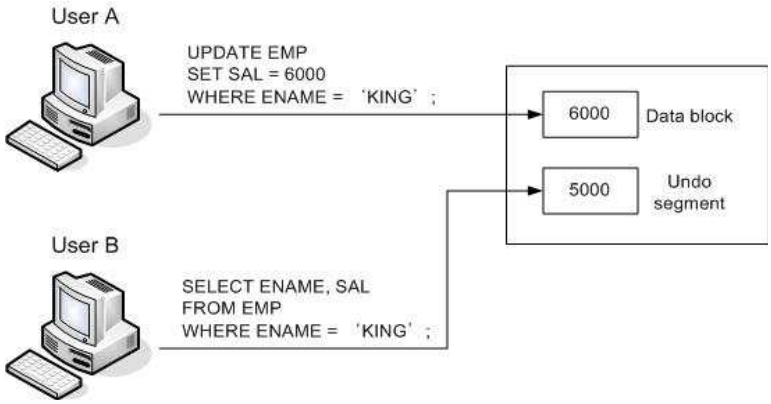


그림 9-2. 읽기 일관성의 원리

트랜잭션이 종료되기 전까지는 해당 데이터를 변경 중인 사용자만 변경 후의 결과를 보게 되며 다른 모든 사용자들은 모두 UNDO 세그먼트의 변경 전 데이터를 보게 된다. 즉, 이러한 메커니즘에 의해 사용자들은 가장 최근의 커밋된 데이터만을 보게 됨으로써 읽기 일관성이 유지된다.

DML 문장이 커밋되면 모든 변경사항은 데이터베이스에 영구히 반영되므로 모든 사용자들은 변경 후 데이터를 볼 수 있게 되며 UNDO 세그먼트에서 변경 전 데이터가 저장된 공간은 다른 트랜잭션을 위해 재활용 된다.

만약, 트랜잭션이 취소되면 모든 변경사항은 다음과 같은 작업에 의해 취소된다.
· UNDO 세그먼트내의 변경 전 데이터는 다시 테이블에 기록된다.
· 모든 사용자는 트랜잭션이 시작되기 이전의 데이터를 보게 된다.

잠금(Lock)

잠금이란 동일한 데이터를 변경하고 있는 트랜잭션 간에 발생 할 수 있는 데이터의 혼란을 사전에 예방하기 위한 메커니즘으로 Oracle 데이터베이스에 의해서 자동으로 수행되며 사용자의 개입이 불필요하다. 여기서, Oracle 데이터베이스에 의해 자동으로 진행되는 잠금을 암시적 잠금(Implicit locking)이라고 하며 SELECT를 제외한 모든 문장에서 잠금이 발생한다. 또한, 사용자가 직접 데이터에 잠금을 설정하는 것을 명시적 잠금(Explicit locking)이라고 부른다.

■ 암시적 잠금의 종류
DML 작업을 수행하면 다음과 같은 잠금이 발생한다.

· 공유 잠금(Share lock)은 DML 문장이 진행중인 테이블에 설정되고, 모든 트랜잭션들은 동일한 데이터에 공유 잠금을 설정할 수 있다.
· 배타 잠금(Exclusive lock)은 DML 문장이 변경 중인 각각의 행에 대하여 설정되고, 트랜잭션이 종료되기 전까지 다른 트랜잭션에 의해서 데이터가 변경되는 것을 방지한다. 또한, 배타 잠금은 같은 데이터를 여러 사용자가 동시에 변경 할 수 없도록 하고 커밋되지 않은 데이터가 다른 사용자에 의해 다시 변경되는 것을 방지한다.
· DDL 잠금은 테이블과 같은 데이터베이스 객체를 변경할 때 발생한다.

복습

1. 부서 테이블에 부서코드가 99, 부서명은 '관리과', 위치는 '대구'인 행을 입력하시오.

2. 부서 테이블에 99번 부서의 부서명을 '회계과'로 변경하시오.

3. 부서 테이블의 99번 부서 행을 삭제하시오.

4. 트랜잭션이 자동 커밋되는 시점을 모두 고르시오.
   a. DDL 문장의 실행
   b. DCL 문장의 실행
   c. 시스템 오류
   d. SQL*Plus의 정상 종료

5. 현재 진행 중인 트랜잭션을 종료하고 모든 데이터 변경사항을 취소하는 명령어는 ( )이며, 데이터의 변경사항을 데이터베이스에 영구히 반영하는 명령어는 ( )이다.

Chapter 10. 테이블 생성 및 관리

이번 장에서는 DDL(Data definition language) 문장을 이용하여 주요 데이터베이스 객체인 테이블의 생성에 대하여 알아보고, 테이블의 변경, 이름 바꾸기, 잘라내기 등의 테이블 관리 방법을 설명한다.

데이터베이스 객체의 종류

Oracle에서 지원하는 데이터베이스 객체는 다음과 같은 것들이 있다.

표 10-1. 데이터베이스 객체

객체명	설명
테이블(Table)	기본적인 저장 단위로 행과 컬럼으로 구성
뷰(View)	한개 이상의 테이블의 논리적인 부분 집합을 표시
시퀀스(Sequence)	숫자 값 생성기
인덱스(Index)	데이터 검색 성능 향상
동의어(Synonym)	객체에 대한 별칭

Oracle의 테이블 구조는 다음과 같은 특징을 갖는다.

- 테이블은 사용자들이 데이터베이스를 사용 중에 언제라도 생성할 수 있다.
- 테이블의 저장 용량을 지정 할 필요는 없지만 테이블에 얼마나 많은 데이터가 저장 될 것인가를 예측하는 것도 중요하다.
- 테이블 구조는 데이터베이스가 온라인 상태에서도 변경이 가능하다.

테이블 생성

테이블 생성시 테이블 및 컬럼명은 다음과 같은 명명 규칙을 준수하도록 한다.

- 테이블 및 컬럼명은 문자로 시작하며 1~30 문자 이내로 작성한다.
- 테이블 및 컬럼명은 A~Z, a~z, 0~9, _, $, #로 작성한다. 물론, 한글로 작성할 수도 있으나 권장하지 않는다.
- 동일한 사용자의 다른 객체와 이름이 중복되지 않도록 한다.
- Oracle에 의해 예약되어 있는 키워드는 사용 할 수 없다.

테이블명은 대소문자를 구분하지 않는다. 예를 들어, EMP, eMP, eMp는 동일한 테이블로 인식된다.

■ CREATE TABLE

테이블을 생성하려면 CREATE TABLE 명령을 사용해야 하며, 이 문장은 DDL 명령 중

하나이다. 이 문장을 실행하면 데이터베이스에 곧바로 적용되어 테이블이 생성되며, 관련 정보가 데이터 딕셔너리(Data dictionary)라는 객체 관련 정보가 저장되어 있는 테이블에 기록된다.

테이블을 생성하기 위해서는 CREATE TABLE 권한과 테이블을 생성 할 저장공간을 부여 받아야 한다. 이러한 권한들은 DCL(Data control language)에 의해 사용자에게 부여되고 회수 될 수 있다.

테이블을 생성하는 명령어는 다음과 같다.

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

■ 다른 사용자의 테이블 검색

위 문장에서 스키마(schema)는 객체의 모음을 의미하며, 스키마 객체는 데이터베이스내 데이터를 직접 참조하는 논리적 구조로서 테이블, 뷰, 동의어, 시퀀스, 저장 프로시져, 인덱스, 클러스터, 데이터베이스 링크를 포함한다.

만약, 다른 사용자가 소유하고 있는 테이블을 검색하려면 테이블 이름 앞에 소유자의 이름을 반드시 붙여주어야 한다. 예를 들어, KIM으로 명명된 스키마가 존재하고 KIM이 EMP 테이블을 소유하고 있다면 해당 테이블은 다음과 같이 검색하여야만 한다.

```
SELECT * FROM KIM.EMP;
```

■ DEFAULT 옵션

테이블 작성시 해당 컬럼에 DEFAULT 옵션을 사용하여 디폴트 값을 지정할 수 있는데, 해당 테이블에 행을 입력할 때, 해당 컬럼에 값을 지정하지 않은 경우 자동으로 디폴트 값이 입력되어 NULL 값이 저장되는 것을 방지하기 위한 것이다. 또한, INSERT 문장에 DEFAULT 키워드를 지정하여 디폴트 값이 저장되도록 할 수 있다.

```
... HIREDATE DATE DEFAULT SYSDATE, ...
```

디폴트 값으로 리터럴 및 SQL 함수를 지정 할 수 있으며, 다른 컬럼의 이름 또는 가상 컬럼(Pseudocolumn)은 지정 할 수 없다. 물론, 컬럼의 데이터 타입과 디폴트 값의 타입은 일치되어야 한다.

■ 테이블 만들기

다음과 같이 테이블을 생성하고 올바르게 생성되었는지 확인해보자.

```
SQL> CREATE TABLE BUSE0
  2  (DEPTNO NUMBER(2),
  3  DNAME VARCHAR2(14),
  4  LOC VARCHAR2(13));

테이블이 생성되었습니다.

SQL> DESC BUSE0
 이름                                      널?      유형
 ----------------------------------------- -------- ----------------------------
 DEPTNO                                             NUMBER(2)
 DNAME                                              VARCHAR2(14)
 LOC                                                VARCHAR2(13)
```

CREATE 명령은 DDL 명령으로 실행된 후, 자동 커밋 된다.

■ 테이블의 종류

Oracle 서버 내에는 EMP와 같이 사용자가 생성한 사용자 테이블(User table)과 Oracle 서버가 생성하고 관리하는 테이블이 있는데, 이를 데이터 딕셔너리(Data dictionary)라고 부르며 데이터베이스 관련 정보가 저장되어 있다. 데이터 딕셔너리 테이블은 SYS가 소유하고 있으며, 사용자들이 알아보기 어려운 형태로 저장되어 있기 때문에 직접 검색하기는 어렵다. 그래서, Oracle 서버는 이러한 테이블들을 알아보기 쉽도록 뷰로 만들어 제공하는데, 이러한 뷰를 데이터 딕셔너리 뷰라고 부르며 이러한 뷰를 통해서 Oracle 서버의 사용자, 사용자에게 부여된 권한, 데이터베이스 객체 이름, 테이블 제약조건, 감사 정보 등을 검색할 수 있다. 데이터 딕셔너리 뷰에는 다음의 4가지 종류가 있다.

표 10-2. 데이터 딕셔너리 뷰의 종류

접두사	설명
USER_	사용자가 소유한 객체와 관련된 정보를 포함하고 있는 뷰
ALL_	사용자가 접근 가능한 객체와 관련된 정보를 포함하고 있는 뷰
DBA_	DBA 롤을 부여 받은 사용자들만 접근할 수 있는 제한적인 뷰
V$	데이터베이스 서버 성능, 메모리, 잠금 등의 정보를 포함하고 있는 동적 성능 뷰

데이터 딕셔너리를 검색하는 방법은 다음과 같다. 먼저 사용자가 소유하고 있는 테이블의 이름을 검색하면 다음과 같다.

```
SQL> SELECT TABLE_NAME
  2  FROM USER_TABLES;

TABLE_NAME
------------------------------
BONUS
BUSE0
DEPT
...
TEST1
```

사용자가 소유하고 있는 객체의 종류를 검색하면 다음과 같다.

```
SQL> SELECT DISTINCT OBJECT_TYPE
  2  FROM USER_OBJECTS;

OBJECT_TYPE
------------------
INDEX
TABLE
```

사용자가 소유하고 있는 테이블, 뷰, 동의어, 시퀀스를 검색하면 다음과 같다.

```
SQL> SELECT *
  2  FROM USER_CATALOG;

TABLE_NAME                     TABLE_TYPE
------------------------------ -----------
BONUS                          TABLE
BUSEO                          TABLE
DEPT                           TABLE
...
TEST1                          TABLE
```

사용자들에 의해 자주 검색되는 데이터 딕셔너리는 다음과 같다.

· USER_TABLES

· USER_OBJECTS

· USER_CATALOG

USER_CATALOG는 CAT라는 동의어가 생성되어 있으므로, 동의어를 이용하여 검색이 가능하다.

```
SQL> SELECT *
  2  FROM CAT;

TABLE_NAME                     TABLE_TYPE
------------------------------ -----------
BONUS                          TABLE
BUSEO                          TABLE
DEPT                           TABLE
...
TEST1                          TABLE
```

데이터 타입의 종류

테이블 생성시 컬럼에 지정 할 수 있는 데이터 타입은 다음과 같다.

표 10-3. 데이터 타입

데이터 타입	설명
VARCHAR2(*size*)	가변 길이 문자열 데이터. (최소 길이 1, 최대 길이 4000 바이트)
CHAR[(*size*)]	고정 길이 문자열 데이터. (디폴트 및 최소 길이 1, 최대 길이 2000 바이트)
NUMBER(*p*, *s*)	가변 길이 숫자 데이터. (전체 자리수 *p*, 소수점 자리수 *s*. 전체 자리수는 1~38, 소수점 자리수는 -84~127)
DATE	날짜 및 시간 데이터. (B.C 4712년 1월 1일~ A.D 9999년 12월 31일)
LONG	가변 길이 문자열 데이터(2GB)
CLOB	문자 데이터(4GB)
RAW(*size*)	이진 데이터. (최대 2000 바이트)
LONG RAW	가변 길이 이진 데이터(2GB)
BLOB	이진 데이터(4GB)
BFILE	외부 파일에 저장된 이진 데이터(4GB)
ROWID	시스템에서 테이블내의 행들을 유일하게 식별할 수 있는 64 비트 숫자

LONG 타입과 관련하여 주의할 사항은 다음과 같다.

· LONG 타입의 컬럼은 서브쿼리에 의해 테이블을 생성할 때 복사 되지 않는다.
· LONG 타입의 컬럼은 GROUP BY 또는 ORDER BY에 포함 될 수 없다.
· LONG 타입의 컬럼은 테이블에 오직 1개만 사용 할 수 있다.
· LONG 타입의 컬럼에는 제약조건을 정의 할 수 없다.
· LONG 타입의 컬럼보다는 CLOB 타입의 컬럼을 사용하도록 한다.

추가된 날짜 타입

Oracle 9i에서는 아래와 같은 새로운 날짜 타입이 추가 되었다.

표 10-4 추가된 날짜 타입

날짜 데이터 타입	설명
TIMESTAMP	DATE 타입을 확장한 것으로 10^{-9} 초까지 지정 가능
INTERVAL YEAR TO MONTH	년, 월 단위로 시간을 저장(예, 1년 2개월). 즉, 두개의 날짜 타입 데이터 간의 간격을 저장하는데 사용
INTERVAL DAY TO SECOND	일, 시, 분, 초 단위로 시간을 저장(예, 1일 2시간 3분 4초). 즉, 두개의 날짜 타입 데이터 간의 간격을 저장하는데 사용

■ TIMESTAMP

TIMESTAMP 타입은 DATE 타입이 확장된 것으로 DATE 타입의 년, 월, 일, 시, 분, 초뿐만 아니라 10^{-9} 초까지 지정가능한 타입이다. TIMESTAMP 타입을 지정하는 방법은 다음과 같다.

```
TIMESTAMP[(fractional_seconds_precision)]
```

*fractional_seconds_precision*은 초의 소수점 자리 수를 의미하며, 디폴트 값은 소수점 6자리이다.

```
SQL> CREATE TABLE NEW_EMP
  2  (EMP_ID NUMBER,
  3  EMP_NAME VARCHAR2(15),
  4  START_DATE TIMESTAMP(7));

테이블이 생성되었습니다.

SQL> INSERT INTO NEW_EMP
  2  VALUES(1, '홍길동', SYSDATE);

1 개의 행이 만들어졌습니다.

SQL> INSERT INTO NEW_EMP
  2  VALUES(2, '콩쥐', '04/06/21 22:40:30.1234567');

1 개의 행이 만들어졌습니다.

SQL> SELECT * FROM NEW_EMP;

    EMP_ID EMP_NAME        START_DATE
---------- --------------- --------------------------
         1 홍길동          04/06/21 21:33:22.0000000
         2 콩쥐            04/06/21 22:40:30.1234567
```

■ TIMESTAMP WITH TIME ZONE

TIMESTAMP WITH TIME ZONE 타입은 TIMESTAMP 타입이 확장 된 것으로 현재 지역시간과 그리니치 표준시(GMT : Greenwich Mean Time)간의 시차를 추가로 저장한다.

```
TIMESTAMP[(fractional_seconds_precision)] WITH TIME ZONE
```

즉, 다국적 기업에서 사용하는 데이터베이스에 날짜 및 시간을 저장하는 경우, 현지 국가의 시간으로 입력된 데이터는 지역별 시차로 인하여 사용자에게 혼란을 일으킬 수 있으므로 TIMESTAMP WITH TIME ZONE 타입을 사용하여 현재 지역시간에 GMT와 현재 지역시간과의 시차를 추가로 저장하면 해결 할 수 있다.

```
SQL> CREATE TABLE NEW_EMP2
  2  (EMP_ID NUMBER,
  3  EMP_NAME VARCHAR2(15),
  4  START_DATE TIMESTAMP WITH TIME ZONE);

테이블이 생성되었습니다.

SQL> INSERT INTO NEW_EMP2
  2  VALUES(1, '홍길동', SYSDATE);

1 개의 행이 만들어졌습니다.

SQL> SELECT * FROM NEW_EMP2;

    EMP_ID EMP_NAME        START_DATE
---------- --------------- -------------------------------
         1 홍길동          04/06/21 21:52:08.000000 +09:00
```

위 결과에서 보면 홍길동의 입사일은 2004년 6월 21일 21시 52분 8초이며, GMT를 기준으로 9시간이 빠름을 보여준다.

■ TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP WITH LOCAL TIME ZONE 타입은 현재 지역 시간을 기준으로 변환되어 저장되므로 시차가 추가로 저장되지 않는다.

```
TIMESTAMP[(fractional_seconds_precision)] WITH LOCAL TIME ZONE
```

```
SQL> CREATE TABLE NEW_EMP3
  2  (EMP_ID NUMBER,
  3  EMP_NAME VARCHAR2(15),
  4  START_DATE TIMESTAMP WITH LOCAL TIME ZONE);

테이블이 생성되었습니다.

SQL> INSERT INTO NEW_EMP3
  2  VALUES(1, '홍길동', SYSDATE);

1 개의 행이 만들어졌습니다.

SQL> SELECT * FROM NEW_EMP3;

    EMP_ID EMP_NAME        START_DATE
---------- --------------- -----------------------
         1 홍길동          04/06/21 22:01:49.000000
```

■ INTERVAL YEAR TO MONTH

INTERVAL YEAR TO MONTH 타입은 2년 6개월과 같이 년과 월 단위로 날짜 간격을 저장한다.

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

여기서, *year_precision*은 년도를 표시하는 자리수를 의미하며, 디폴트는 2이다.

INTERVAL YEAR TO MONTH 타입의 데이터 값은 다음과 같이 표현된다.
예) INTERVAL YEAR '123-2' YEAR(3) TO MONTH : 123년 2개월
예) INTERVAL '123' YEAR(3) : 123년
예) INTERVAL '300' MONTH(3) : 300개월

```
SQL> CREATE TABLE TIME1
  2  (DURATION INTERVAL YEAR(3) TO MONTH);

테이블이 생성되었습니다.

SQL> INSERT INTO TIME1 VALUES (INTERVAL '10' MONTH(2));

1 개의 행이 만들어졌습니다.

SQL> SELECT SYSDATE, SYSDATE+DURATION FROM TIME1;

SYSDATE  SYSDATE+
-------- --------
04/06/21 05/04/21
```

■ INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND 타입은 10일 1시간 30분 20초와 같이 년, 월, 시, 분, 초 단위로 시간 간격을 저장한다.

```
 INTERVAL DAY [(day_precision)]
 TO SECOND [(fractional_seconds_precision)]
```

여기서, day_precision은 일수를 표시하는 자리수를 의미하며, 디폴트는 2이고, fractional_seconds_precision은 소수점 자리수를 의미하며, 디폴트는 6이다.

INTERVAL DAY TO SECOND 타입의 데이터 값은 다음과 같이 표현된다.
예) INTERVAL '4 5:12:10.222' DAY TO SECOND(3) : 4일 5시간 12분 10.222초
예) INTERVAL '4 5:12' DAY TO MINUTE : 4일 5시간 12분
예) INTERVAL '400 5' DAY(3) TO HOUR : 400일 5시간
예) INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)
: 11시간 12분 10.222222초

```
SQL> CREATE TABLE TIME2
  2  (DURATION INTERVAL DAY(3) TO SECOND);

테이블이 생성되었습니다.

SQL> INSERT INTO TIME2 VALUES(INTERVAL '60' DAY(2));

1 개의 행이 만들어졌습니다.

SQL> SELECT SYSDATE, SYSDATE+DURATION FROM TIME2;

SYSDATE  SYSDATE+
-------- --------
04/06/21 04/08/20
```

서브 쿼리를 이용한 테이블 생성

CREATE TABLE 명령과 서브 쿼리를 결합하면 테이블을 생성하는 동시에 데이터를 입력할 수 있다.

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

위에서 지정된 컬럼의 개수와 서브 쿼리에서 리턴 된 컬럼의 개수가 일치해야하며 컬럼명과 디폴트 값을 이용하여 컬럼을 정의 할 수 있다.

서브 쿼리를 이용하여 테이블 생성시 주의 할 점은 다음과 같다.

- 지정된 컬럼명으로 테이블이 생성되고 서브 쿼리의 SELECT 문장에 의해 리턴된 행들이 테이블에 입력된다.
- 컬럼을 정의할 때 컬럼명과 디폴트 값만 지정할 수 있다.
- 컬럼을 지정할 때 컬럼의 개수와 서브 쿼리인 SELECT 문장의 컬럼 개수가 일치해야 한다.
- 제약조건은 생성된 테이블에 만들어지지 않으며, 오직 컬럼의 데이터 타입만 동일하게 생성된다.

```
SQL> CREATE TABLE EMP_YEAR
  2  AS SELECT EMPNO, ENAME, SAL*12+NVL(COMM, 0) ANNSAL, HIREDATE
  3  FROM EMP
  4  WHERE DEPTNO = 10;

테이블이 생성되었습니다.

SQL> DESC EMP_YEAR
 이름                                      널?      유형
 ----------------------------------------- -------- ------------------
 EMPNO                                              NUMBER(4)
 ENAME                                              VARCHAR2(10)
 ANNSAL                                             NUMBER
 HIREDATE                                           DATE

SQL> SELECT * FROM EMP_YEAR;

     EMPNO ENAME          ANNSAL HIREDATE
---------- ---------- ---------- --------
      7782 CLARK           29400 81/06/09
      7839 KING            60000 81/11/17
      7934 MILLER          15600 82/01/23
```

위 문장의 서브 쿼리에서 SAL*12+NVL(COMM, 0) 컬럼은 테이블의 컬럼명으로 부적합하기 때문에 컬럼 별칭을 반드시 지정하여야 한다.

테이블 변경

ALTER TABLE 명령을 이용하면 다음과 같은 작업을 수행 할 수 있다.

- 새로운 컬럼의 추가

· 컬럼 변경
· 신규 컬럼의 디폴트 값 지정
· 컬럼 삭제

■ 컬럼 추가

테이블에 컬럼을 추가하는 명령은 다음과 같다.

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

앞서 생성한 EMP_YEAR 테이블에 JOB 컬럼을 추가하면 다음과 같다.

```
SQL> ALTER TABLE EMP_YEAR
  2  ADD (JOB VARCHAR2(10));

테이블이 변경되었습니다.

SQL> DESC EMP_YEAR
 이름                                      널?      유형
 ----------------------------------------- -------- ----------------
 EMPNO                                              NUMBER(4)
 ENAME                                              VARCHAR2(10)
 ANNSAL                                             NUMBER
 HIREDATE                                           DATE
 JOB                                                VARCHAR2(10)
```

■ 컬럼 변경

테이블의 컬럼을 변경하는 명령어는 다음과 같다.

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

컬럼을 변경 할 때는 다음 사항을 주의하여야 한다.

· 숫자 컬럼의 전체 자리수를 증가 시킬 수 있다.
· 문자 컬럼의 전체 길이를 증가 시킬 수 있다.
· 컬럼의 길이를 축소시킬 수 있으나, 모든 행의 해당 컬럼 값이 전부 NULL 또는 행이 아직 입력되지 않은 경우에만 가능하다.
· 모든 행의 해당 컬럼 값이 NULL인 경우에만 데이터 타입을 변경 할 수 있다.
· CHAR 타입을 VARCHAR2 타입으로, VARCHAR2 타입을 CHAR 타입으로 변경이 가능하지만 해당 테이블에 행이 아직 입력되지 않았거나, 모든 행의 해당 컬럼 값이 전부 NULL인 경우만 가능하다.
· 디폴트 값을 변경하면 변경 이후부터 입력되는 행에 대해서만 적용 된다.

```
SQL> ALTER TABLE EMP_YEAR
  2  MODIFY (JOB VARCHAR2(20));

테이블이 변경되었습니다.

SQL> DESC EMP_YEAR
 이름                                      널?      유형
 ----------------------------------------- -------- ----------------
 EMPNO                                              NUMBER(4)
 ENAME                                              VARCHAR2(10)
 ANNSAL                                             NUMBER
 HIREDATE                                           DATE
 JOB                                                VARCHAR2(20)
```

■ 컬럼명 변경

테이블의 컬럼명을 변경하는 명령어는 다음과 같다.

```
ALTER TABLE table
RENAME COLUMN old_column TO new_column;
```

JOB 컬럼을 WORK로 변경해본다.

```
SQL> ALTER TABLE EMP_YEAR
  2  RENAME COLUMN JOB TO WORK;

테이블이 변경되었습니다.
```

■ 컬럼 삭제

테이블의 컬럼을 삭제하는 명령어는 다음과 같다.

```
ALTER TABLE table
DROP (column );
```

컬럼을 삭제하는 경우, 다음 사항을 주의하도록 한다.

· 컬럼은 값의 존재 유무에 상관 없이 삭제된다.
· ALTER TABLE 명령을 이용하여 한번에 하나의 컬럼만 삭제 할 수 있다.
· 테이블에는 최소한 하나의 컬럼이 남아 있어야 한다.
· 컬럼이 삭제되면 복구가 불가능하다.

```
SQL> ALTER TABLE EMP_YEAR
  2  DROP (JOB);

테이블이 변경되었습니다.

SQL> DESC EMP_YEAR
 이름                                      널?      유형
 ----------------------------------------- -------- ----------------
 EMPNO                                              NUMBER(4)
 ENAME                                              VARCHAR2(10)
 ANNSAL                                             NUMBER
 HIREDATE                                           DATE
```

위에서 ALTER TABLE ... DROP 명령에 의해 컬럼을 삭제하는 대신에 SET UNUSED 옵션을 사용하여 해당 컬럼을 사용하지 않는 것으로 설정 할 수 있다. 이렇게 되면 사용자는 해당 컬럼을 검색 할 수 없을뿐더러 DESC 명령으로도 확인이 불가능하기 때문에 해당 컬럼이 삭제된 것처럼 보여진다.

또한, ALTER TABLE ... DROP 명령은 테이블의 모든 행에 접근하여야 하므로 시간이 많이 소요되며 데이터베이스에 많은 부담을 주는 작업이므로 업무시간에는 해당 컬럼을 SET UNUSED 명령으로 숨겨두었다가 데이터베이스의 부담이 적은 시간에 해당 컬럼을 삭제하는 것이 바람직하다고 볼 수 있다. SET UNUSED 명령으로 표시해둔 컬럼의 갯수는 USER_UNUSED_COL_TABS 데이터 딕셔너리 뷰에서 확인 가능하다.

```
SQL> ALTER TABLE EMP_YEAR
  2  SET UNUSED (HIREDATE);

테이블이 변경되었습니다.

SQL> SELECT * FROM USER_UNUSED_COL_TABS;

TABLE_NAME                       COUNT
------------------------------ ----------
EMP_YEAR                              1

SQL> ALTER TABLE EMP_YEAR
  2  DROP UNUSED COLUMNS;

테이블이 변경되었습니다.
```

테이블 삭제

테이블 삭제는 데이터베이스에서 해당 테이블의 정의를 제거하는 것이다. 테이블을 삭제하면 테이블에 입력된 모든 행들과 관련 인덱스가 모두 삭제된다. 테이블을 삭제하는 명령어는 다음과 같다.

```
DROP TABLE table
```

테이블을 삭제하는 경우, 주의할 사항은 다음과 같다.

- 테이블의 모든 행들은 삭제된다.
- 뷰와 동의어는 삭제되지 않지만 유효하지 않다.
- 진행중인 트랜잭션은 커밋된다.
- 테이블의 소유자 또는 DROP ANY TABLE 권한을 가진 소유자만이 테이블을 삭제 할 수 있다.

```
SQL> DROP TABLE EMP_YEAR;

테이블이 삭제되었습니다.
```

기타 테이블 관련 명령어

■ 테이블 이름 변경

RENAME 명령을 이용하면 테이블, 뷰, 시퀀스, 동의어의 이름을 변경 할 수 있다. 해당 객체의 소유자만이 객체의 이름을 변경할 수 있다.

```
RENAME old_name TO new_name;
```

■ 테이블 잘라내기

DDL 문장인 TRUNCATE 명령을 사용하면 테이블의 모든 행들을 삭제 할 수 있으며 테이블이 사용하고 있던 저장 공간을 해제하여 다른 테이블들이 사용할 수 있도록 해준다. 반면, DELETE 명령은 저장 공간을 해제하지 않는다.

```
TRUNCATE TABLE table;
```

또한, TRUNCATE 명령어는 DELETE 문장과는 달리 롤백이 불가능하지만 다음과 같은 이유로 인하여 DELETE 명령보다 수행 속도가 빠르다.

· TRUNCATE 명령은 DDL 명령이므로 롤백 정보를 생성하지 않는다.
· TRUNCATE 명령은 삭제 트리거를 발생시키지 않는다.
· 외래키 제약조건에 의해 참조되는 테이블은 TRUNCATE 명령으로 테이블을 잘라낼 수 없으므로, 먼저 제약조건을 취소해야 한다.

복습

1. 다음과 같은 테이블을 작성하시오.
   테이블명 : JUSO

컬럼명	데이터 타입
NO	NUMBER(3)
NAME	VARCHAR2(10)
ADDR	VARCHAR2(20)
EMAIL	VARCHAR2(5)

2. 전화번호(PHONE) 컬럼을 VARCHAR2(10) 타입으로 추가하시오.

3. 이메일(EMAIL) 컬럼을 VARCHAR2(20) 타입으로 변경하시오.

4. 주소(ADDR) 컬럼을 삭제하시오.

5. 테이블을 삭제하시오.

Chapter 11. 제약조건

제약조건은 데이터의 무결성(Integrity)을 지켜주는 중요한 역할을 수행한다. 이번 장에서는 제약조건의 정의와 종류를 살펴보고 이러한 제약조건을 생성 및 관리하는 방법을 알아본다.

제약조건의 정의

Oracle 서버는 제약조건을 이용하여 적절치 않은 데이터가 테이블에 입력되는 것을 방지하며 다음과 같은 장점이 있다.

- 테이블에 행이 입력, 갱신, 삭제 될 때마다 지정된 제약 조건을 반드시 만족해야하므로 데이터의 무결성을 강화시켜준다.
- 다른 테이블에 의해 참조 되고 있는 테이블이 삭제 되는 것을 방지한다.

제약조건을 정의 할 때는 다음 사항을 참고하도록 한다.

- 제약조건에 적절한 이름을 지정하도록 한다. 그렇지 않으면 Oracle 서버가 SYS_C*n* 형식으로 임의 지정하기 때문에 제약조건의 이름으로 어떤 종류의 제약조건인지 식별하기가 어려워진다.
- 제약조건은 테이블의 생성과 동시에 작성되도록 할 수 있으며, 테이블이 생성된 이후에도 추가 할 수 있다.
- 제약조건은 테이블 수준 또는 컬럼 수준에서 정의 할 수 있다.
- 제약조건은 USER_CONSTRAINTS 데이터 딕셔너리를 이용하여 검색 할 수 있다.

제약조건을 정의하는 문법은 다음과 같다.

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

제약조건을 컬럼 수준에서 정의하려면 *column_constraint*에 제약조건을 기술한다. 컬럼 수준의 제약조건은 한 개의 컬럼에 한 개의 제약조건만 정의 할 수 있으며, 모든 제약조건 타입을 기술 할 수 있다. 작성하는 방법은 다음과 같이 테이블에 대한 컬럼을 정의 한 후, 제약조건을 기술한다.

```
column [CONSTRAINT constraint_name] constraint_type,
```

제약조건을 테이블 수준에서 정의하려면 *table_constraint*에 제약조건을 기술해주며 한 개 이상의 컬럼에 한 개의 제약조건을 정의 할 수 있고, NOT NULL 제약조건을 제외한 모든 제약조건 타입을 기술 할 수 있다. 작성하는 방법은 테이블의 컬럼 정의와 별도로 기술한

다.

```
 column, ...
   [CONSTRAINT constraint_name] constraint_type
   (column, ...),
```

NOT NULL

NOT NULL 제약조건은 해당 컬럼에 NULL 값이 입력되지 않도록 제한하는 것으로, NOT NULL 제약조건이 정의되지 않은 컬럼은 디폴트로 NULL 값의 저장이 허용된다. NOT NULL 제약조건은 컬럼 수준에서만 지정가능하며 사용방법은 다음과 같다.

```
SQL> CREATE TABLE SAWON (
  2  S_NO NUMBER(4),
  3  S_NAME VARCHAR2(10) NOT NULL,
  4  S_HIREDATE DATE CONSTRAINT SAWON_S_HIREDATE_NN NOT NULL);

테이블이 생성되었습니다.

SQL> INSERT INTO SAWON
  2  VALUES(1, '길동', NULL);
INSERT INTO SAWON
*
1행에 오류:
ORA-01400: NULL을 ("SCOTT"."SAWON"."S_HIREDATE") 안에 삽입할 수 없습니다
```

위에서 SAWON 테이블의 S_NAME 컬럼과 S_HIREDATE 컬럼에 NOT NULL 제약조건이 지정되었으며, 해당 컬럼에 NULL 값을 입력하려고 시도하면 제약조건에 위배되어 오류가 발생한다. S_NAME 컬럼과 같이 제약조건 선언시 제약조건 이름을 지정하지 않으면 Oracle 서버가 임의로 제약조건 이름을 정의하게 된다. 제약조건을 검색하는 방법은 다음과 같다.

```
SQL> SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE
  2  FROM USER_CONSTRAINTS
  3  WHERE TABLE_NAME = 'SAWON';

CONSTRAINT_NAME                C
------------------------------ -
SAWON_S_HIREDATE_NN            C
SYS_C003005                    C
```

UNIQUE

UNIQUE 제약조건은 컬럼에 중복된 값이 입력되지 않도록 제한하는 것으로, 한 개 이상의 컬럼으로 구성 할 수 있다. UNIQUE 제약조건이 지정된 컬럼에 별도의 NOT NULL 제약조건을 지정하지 않는 한, NULL 값이 중복되어 입력 될 수 있으며, 한 개의 컬럼으로 UNIQUE 제약조건을 구성 할 때는 컬럼 수준 및 테이블 수준에서 제약조건의 정의가 가능하지만, 두 개 이상의 컬럼으로 구성하는 경우는 테이블 수준에서만 정의 할 수 있다. UNIQUE 제약조건이 지정된 컬럼을 고유키라고도 부르며 사용방법은 다음과 같다.

```
SQL> CREATE TABLE SAWON (
  2  S_NO NUMBER(4),
  3  S_NAME VARCHAR2(10),
  4  S_SAL NUMBER(10),
  5  S_EMAIL VARCHAR2(20) CONSTRAINT SAWON_S_EMAIL_UK UNIQUE);

테이블이 생성되었습니다.

SQL> INSERT INTO SAWON VALUES (1, '길동', 1000, 'GDH@XYZ.COM');

1 개의 행이 만들어졌습니다.

SQL> INSERT INTO SAWON VALUES (2, '콩쥐', 2000, 'GDH@XYZ.COM');
INSERT INTO SAWON VALUES (2, '콩쥐', 2000, 'GDH@XYZ.COM')
*
1행에 오류:
ORA-00001: 무결성 제약 조건(SCOTT.SAWON_S_EMAIL_UK)에 위배됩니다

SQL> INSERT INTO SAWON VALUES (3, '팥쥐', 1500, NULL);

1 개의 행이 만들어졌습니다.
```

위의 UNIQUE 제약조건을 테이블 수준에서 정의하면 다음과 같다.

```
CREATE TABLE SAWON (
  S_NO NUMBER(4),
  S_NAME VARCHAR2(10),
  S_SAL NUMBER(10),
  S_EMAIL VARCHAR2(20),
  CONSTRAINT SAWON_S_EMAIL_UK UNIQUE (S_EMAIL));
```

두 개 이상의 컬럼으로 UNIQUE 제약조건을 구성 할 때는 테이블 수준에서 정의하여야만 한다.

```
CREATE TABLE SAWON (
  S_NO NUMBER(4),
  S_NAME VARCHAR2(10),
  S_SAL NUMBER(10),
  S_COMM NUMBER(10),
  CONSTRAINT SAWON_UK UNIQUE (S_NO, S_NAME));
```

또한, UNIQUE 제약조건을 지정하면 해당 컬럼에 UNIQUE INDEX가 생성된다.

PRIMARY KEY

PRIMARY KEY 제약조건은 테이블에 기본키를 작성하는 것으로 한 개의 테이블에는 오직 한 개의 기본키를 만들 수 있다. PRIMARY KEY 제약조건은 해당 컬럼에 중복된 값과 NULL 값이 입력되지 않도록 제한하며, 한 개 이상의 컬럼으로 PRIMARY KEY 제약조건을 구성 할 수 있다. 한 개의 컬럼으로 PRIMARY KEY 제약조건을 구성하는 경우에는 컬럼 수준 및 테이블 수준에서 제약조건의 정의가 가능하지만, 두 개 이상의 컬럼으로 PRIMARY KEY 제약조건을 구성하는 경우 반드시 테이블 수준에서 제약조건을 정의해야 한다.

```
SQL> CREATE TABLE SAWON (
  2  S_NO NUMBER(4) CONSTRAINT SAWON_S_NO_PK PRIMARY KEY,
  3  S_NAME VARCHAR2(10),
  4  S_SAL NUMBER(10));

테이블이 생성되었습니다.

SQL> INSERT INTO SAWON VALUES (1, '길동', 1000);

1 개의 행이 만들어졌습니다.

SQL> INSERT INTO SAWON VALUES (1, '콩쥐', 2000);
INSERT INTO SAWON VALUES (1, '콩쥐', 2000)
*
1행에 오류:
ORA-00001: 무결성 제약 조건(SCOTT.SAWON_S_NO_PK)에 위배됩니다

SQL> INSERT INTO SAWON VALUES (NULL, '팥쥐', 3000);
INSERT INTO SAWON VALUES (NULL, '팥쥐', 3000)
*
1행에 오류:
ORA-01400: NULL을 ("SCOTT"."SAWON"."S_NO") 안에 삽입할 수 없습니다
```

위의 PRIMARY KEY 제약조건을 테이블 수준에서 정의하면 다음과 같다.

```
CREATE TABLE SAWON (
  S_NO NUMBER(4),
  S_NAME VARCHAR2(10),
  S_SAL NUMBER(10),
  CONSTRAINT SAWON_S_NO_PK PRIMARY KEY (S_NO));
```

두 개 이상의 컬럼으로 PRIMARY KEY 제약조건을 구성하려면 테이블 수준에서 정의하여야만 한다.

```
CREATE TABLE SAWON (
  S_NO NUMBER(4),
  S_NAME VARCHAR2(10),
  S_SAL NUMBER(10),
  CONSTRAINT SAWON_PK PRIMARY KEY (S_NO, S_NAME));
```

또한, PRIMARY KEY 제약조건을 지정하면 UNIQUE 제약조건과 마찬가지로 해당 컬럼에 UNIQUE INDEX가 생성된다.

FOREIGN KEY

FOREIGN KEY 제약조건은 참조 무결성 제약조건이라고 부르며 외래키 컬럼에 지정되어 반드시 다른 테이블의 기본키나 고유키 컬럼의 값을 참조하도록 제한하는 것이다. FOREIGN KEY 제약조건이 지정된 컬럼에 별도의 NOT NULL 제약조건이 지정되지 않으면 NULL 값도 입력 가능하다. 한 개의 컬럼으로 FOREIGN KEY 제약조건을 구성하는 경우에는 컬럼 수준 및 테이블 수준에서 제약조건의 정의가 가능하지만, 두 개 이상의 컬럼으로 FOREIGN KEY 제약조건을 구성하는 경우 반드시 테이블 수준에서 제약조건을 정의해

야 한다.

FOREIGN KEY 제약조건을 설정하기에 앞서 외래키가 참조할 기본키가 있는 테이블을 먼저 작성해보자

```
SQL> CREATE TABLE BUSE0(
  2  B_NO NUMBER(4) CONSTRAINT BUSE0_B_NO_PK PRIMARY KEY,
  3  B_NAME VARCHAR2(10),
  4  B_LOC VARCHAR2(10));

테이블이 생성되었습니다.

SQL> INSERT INTO BUSE0 VALUES (100, '인사과', '서울');

1 개의 행이 만들어졌습니다.

SQL> INSERT INTO BUSE0 VALUES (200, '총무과', '대전');

1 개의 행이 만들어졌습니다.

SQL> INSERT INTO BUSE0 VALUES (300, '경리과', '부산');

1 개의 행이 만들어졌습니다.
```

부서 테이블을 참조하는 사원 테이블의 외래키에 FOREIGN KEY 제약조건을 지정하여 데이터를 입력하면 다음과 같다.

```
SQL> CREATE TABLE SAWON(
  2  S_NO NUMBER(4) CONSTRAINT SAWON_S_NO_PK PRIMARY KEY,
  3  S_NAME VARCHAR2(10),
  4  S_SAL NUMBER(5),
  5  B_NO NUMBER(4) CONSTRAINT SAWON_B_NO_FK REFERENCES BUSE0(B_NO));

테이블이 생성되었습니다.

SQL> INSERT INTO SAWON VALUES (1, '길동', 1000, 100);

1 개의 행이 만들어졌습니다.

SQL> INSERT INTO SAWON VALUES (2, '콩쥐', 2000, 150);
INSERT INTO SAWON VALUES (2, '콩쥐', 2000, 150)
*
1행에 오류:
ORA-02291: 무결성 제약조건(SCOTT.SAWON_B_NO_FK)이 위배되었습니다- 부모 키가
없습니다

SQL> INSERT INTO SAWON VALUES (3, '팥쥐', 3000, NULL);

1 개의 행이 만들어졌습니다.
```

위의 제약조건을 테이블 수준에서 정의하면 다음과 같다.

```
CREATE TABLE SAWON(
  S_NO NUMBER(4) CONSTRAINT SAWON_S_NO_PK PRIMARY KEY,
  S_NAME VARCHAR(10),
  S_SAL NUMBER(5),
  B_NO NUMBER(4),
  CONSTRAINT SAWON_B_NO_FK FOREIGN KEY (B_NO) REFERENCES BUSEO(B_NO));
```

위에서 보는 것과 같이 FOREIGN KEY 구문은 제약조건을 컬럼 수준에서 정의하는 경우에는 기술 할 필요가 없으며, 테이블 수준에서 정의하는 경우에만 사용한다. REFERENCES 구문 뒤에는 참조할 테이블과 컬럼을 기술해주며 다음과 같은 옵션을 추가 할 수 있다.

- ON DELETE CASCADE : FOREIGN KEY 제약조건에 의해 참조되는 테이블(부모 테이블)의 행이 삭제되면, 해당 행을 참조하는 테이블(자식 테이블)의 행도 삭제되도록 한다.
- ON DELETE SET NULL : FOREIGN KEY 제약조건에 의해 참조되는 테이블(부모 테이블)의 행이 삭제되면, 해당 행을 참조하는 테이블(자식 테이블)의 외래키 컬럼을 NULL로 변경한다.

CHECK

CHECK 제약조건은 해당 컬럼의 값이 반드시 만족해야 될 조건을 지정하는 것으로 제약조건에 CURRVAL, NEXTVAL, LEVEL, ROWNUM과 같은 가상 컬럼(Pseudocolumn)과 SYSDATE, UID, USER, USERENV 함수를 호출 할 수 없으며 다른 행의 값도 참조 할 수 없다. CHECK 제약조건은 컬럼 수준 및 테이블 수준에서 정의 할 수 있다.

```
SQL> CREATE TABLE SAWON (
  2  S_NO NUMBER(4),
  3  S_NAME VARCHAR2(10),
  4  S_SAL NUMBER(10) CONSTRAINT SAWON_S_SAL_CK CHECK (S_SAL > 0));

테이블이 생성되었습니다.

SQL> INSERT INTO SAWON VALUES (1, '길동', 1000);

1 개의 행이 만들어졌습니다.

SQL> INSERT INTO SAWON VALUES (2, '콩쥐', -1000);
INSERT INTO SAWON VALUES (2, '콩쥐', -1000)
*
1행에 오류:
ORA-02290: 체크 제약조건(SCOTT.SAWON_S_SAL_CK)이 위배되었습니다
```

위 제약조건을 테이블 수준에서 정의하면 다음과 같다.

```
CREATE TABLE SAWON (
  S_NO NUMBER(4),
  S_NAME VARCHAR2(10),
  S_SAL NUMBER(10),
  CONSTRAINT SAWON_S_SAL_CK CHECK (S_SAL > 0));
```

제약조건의 관리

ALTER TABLE 명령을 이용하면 기존의 테이블에 제약조건을 추가 할 수 있으며, 기존 제약조건을 활성화하거나 비활성화 할 수 있다.

■ 제약조건의 추가

기존 테이블에 제약조건을 추가하는 명령은 다음과 같다. 단, NOT NULL 제약조건의 경우는 컬럼 변경 명령인 ALTER TABLE ... MODIFY 명령을 사용해야만 한다.

```
ALTER TABLE table
ADD [CONSTRAINT constraint] type (column);
```

기존 테이블에 제약조건을 설정하는 방법은 다음과 같다. SAWON 테이블의 S_MGR 컬럼 값이 S_NO 컬럼에 존재하는 값을 참조하도록 FOREIGN KEY 제약조건을 추가하였다.

```
SQL> CREATE TABLE SAWON (
  2  S_NO NUMBER(4) CONSTRAINT SAWON_S_NO_PK PRIMARY KEY,
  3  S_NAME VARCHAR2(10),
  4  S_MGR NUMBER(4));

테이블이 생성되었습니다.

SQL> ALTER TABLE SAWON
  2  ADD CONSTRAINT SAWON_S_MGR_FK FOREIGN KEY (S_MGR)
  3  REFERENCES SAWON(S_NO);

테이블이 변경되었습니다.
```

■ 제약조건의 삭제

기존 테이블에서 제약조건을 삭제하는 방법은 다음과 같다.

```
SQL> ALTER TABLE SAWON
  2  DROP CONSTRAINT SAWON_S_MGR_FK;

테이블이 변경되었습니다.
```

PRIMARY KEY 제약조건을 삭제했을 때, 해당 기본키를 참조하는 FOREIGN KEY 제약조건을 동시에 삭제하는 명령은 다음과 같다.

```
SQL> ALTER TABLE SAWON
  2  DROP PRIMARY KEY CASCADE;

테이블이 변경되었습니다.
```

■ 제약조건 비활성화

ALTER TABLE 명령의 DISABLE 구문을 이용하면 기존 제약조건을 잠시 비활성화 할 수 있다. 비활성화 할 제약조건에 CASCADE 옵션을 추가하면 해당 제약조건과 관련된 제약조건을 모두 비활성화 할 수 있다.

```
SQL> CREATE TABLE SAWON (
  2  S_NO NUMBER(4) CONSTRAINT SAWON_S_NO_PK PRIMARY KEY,
  3  S_NAME VARCHAR2(10),
  4  S_MGR NUMBER(4));

테이블이 생성되었습니다.

SQL> ALTER TABLE SAWON
  2  DISABLE CONSTRAINT SAWON_S_NO_PK;

테이블이 변경되었습니다.
```

PRIMARY KEY 제약조건과 UNIQUE 제약조건을 비활성화하면 해당 제약조건에 생성된 인덱스는 자동으로 삭제된다.

■ 제약조건 활성화

비활성화 된 제약조건은 ALTER TABLE 명령의 ENABLE 구문으로 다시 활성화 할 수 있다. 제약조건이 활성화 되면 해당 테이블에 입력된 모든 행에 적용되며 반드시 모든 데이터는 제약조건에 만족해야한다.

```
SQL> ALTER TABLE SAWON
  2  ENABLE CONSTRAINT SAWON_S_NO_PK;

테이블이 변경되었습니다.
```

만약, CASCADE 옵션으로 비활성화된 관련 제약조건은 ENABLE 명령으로도 활성화되지 않음을 주의해야 한다. 즉, PRIMARY KEY 제약조건을 CASCADE 옵션으로 비활성화 하면 해당 기본키를 참조하는 FOREIGN KEY 제약조건도 비활성화 되지만 이후, PRIMARY KEY 제약조건을 활성화하면 해당 FOREIGN KEY 제약조건은 활성화되지 않는다. 그리고, PRIAMRY KEY 제약조건과 UNIQUE 제약조건이 활성화되면 삭제된 인덱스가 다시 생성된다.

■ CASCADE CONSTRAINTS

기존 테이블에서 컬럼을 삭제하는 ALTER TABLE ... DROP 명령에 CASCADE CONSTRAINTS 구문을 추가하면 PRIMARY KEY 또는 UNIQUE 제약조건이 지정된 컬럼을 삭제했을 때, 해당 컬럼을 참조하는 모든 제약조건을 삭제 할 수 있다. 또한, 삭제된 컬럼이 포함되어 있는 모든 제약조건도 삭제된다.

```
SQL> CREATE TABLE TEST1 (
  2  PK NUMBER PRIMARY KEY,
  3  FK NUMBER,
  4  C1 NUMBER,
  5  C2 NUMBER,
  6  CONSTRAINT FK_CONSTRAINT FOREIGN KEY (FK) REFERENCES TEST1,
  7  CONSTRAINT CK1 CHECK (PK > 0 AND C1 > 0),
  8  CONSTRAINT CK2 CHECK (C2 > 0));

테이블이 생성되었습니다.

SQL> ALTER TABLE TEST1 DROP (PK);
ALTER TABLE TEST1 DROP (PK)
                        *
1행에 오류:
ORA-12992: 부모 키 열을 삭제할 수 없습니다


SQL> ALTER TABLE TEST1 DROP (C1);
ALTER TABLE TEST1 DROP (C1)
                        *
1행에 오류:
ORA-12991: 열이 다중-열 제약 조건에 참조되었습니다
```

위에서 PK 컬럼을 CASCADE CONSTRAINTS 구문을 이용하여 삭제하면, PRIMARY KEY 제약조건과 해당 컬럼을 참조하는 FOREIGN KEY 제약조건 및 CK1 CHECK 제약조건이 동시에 삭제된다.

```
SQL> ALTER TABLE TEST1 DROP (PK) CASCADE CONSTRAINTS;

테이블이 변경되었습니다.
```

만약, CASCADE CONSTRAINTS 구문을 사용하지 않으려면 관련 제약조건들이 지정된 모든 컬럼을 동시에 삭제하면 된다.

```
SQL> ALTER TABLE TEST1 DROP (PK, FK, C1);

테이블이 변경되었습니다.
```

제약조건의 확인

USER_CONSTRAINTS 뷰를 이용하면 제약조건의 정의와 이름을 아래와 같이 확인 할 수 있다.

```
SQL> SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, SEARCH_CONDITION
  2  FROM USER_CONSTRAINTS
  3  WHERE TABLE_NAME = 'EMP';

CONSTRAINT_NAME                C SEARCH_CON
------------------------------ - ----------
PK_EMP                         P
FK_DEPTNO                      R
```

위 결과에서 CONSTRAINT_TYPE 컬럼의 값이 C인 경우는 CHECK 또는 NOT NULL 제약조건, P인 경우는 PRIMARY KEY, R인 경우는 FOREIGN KEY, U인 경우는 UNIQUE 제약조건을 의미한다.

USER_CONS_COLUMNS 뷰를 이용하면 제약조건이 지정된 컬럼명도 확인 할 수 있다.

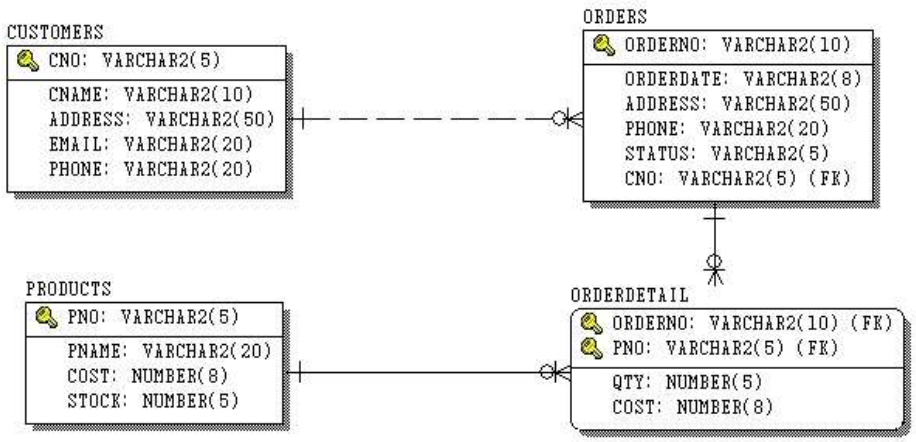
```
SQL> SELECT CONSTRAINT_NAME, COLUMN_NAME
  2  FROM USER_CONS_COLUMNS
  3  WHERE TABLE_NAME = 'EMP';

CONSTRAINT_NAME                COLUMN_NAME
------------------------------ --------------------
FK_DEPTNO                      DEPTNO
PK_EMP                         EMPNO
```

저자가 제약조건에 대하여 강의를 하다보면 외래키와 FOREIGN KEY 제약조건을 혼동하는 수강생이 다소 많은 편이다. 즉, 외래키와 FOREIGN KEY 제약조건을 동일한 것으로 생각하고 외래키를 만들기 위해서는 반드시 FOREIGN KEY 제약조건을 지정해야 하는 것으로 알고 있는데, FOREIGN KEY 제약조건은 외래키가 자신의 역할을 충실히 수행 할 수 있도록 지켜주는 역할을 하는 것으로 외래키가 사용자에 의해서 적절히 관리 될 수 있다면 반드시 FOREIGN KEY 제약조건을 지정 할 필요는 없다. 테이블에 제약조건이 많으면 많을수록 행의 입력, 변경, 삭제시 모든 제약조건을 확인하게 되므로 데이터베이스의 성능은 상대적으로 저하되게 된다.

복습

다음 ERD를 보고 테이블을 작성하시오. 단, 제약조건은 PRIMARY KEY, FOREIGN KEY만 사용하시오.



Chapter 12. 뷰(View)의 작성

이번 장에서는 뷰(View)를 작성하고, 뷰를 통하여 데이터를 검색, 입력, 변경, 삭제하는 방법을 알아본다. 또한, FROM 절 뒤에 기술되는 서브쿼리로서 인라인(Inline) 뷰를 작성하고 TOP-N 분석을 수행하는 방법을 설명한다.

뷰의 용도 및 종류

테이블에 생성된 뷰는 데이터의 논리적 부분집합으로 표현 할 수 있으며, 기존 테이블 또는 뷰에 기초한 논리적 테이블이다. 뷰는 내부에 데이터를 저장하고 있지 않지만 기존 테이블에 대한 윈도우와 같아서 데이터를 검색하거나 변경 할 수 있다. 뷰에서 참조하는 테이블을 기본 테이블이라고 하며, 뷰에는 SELECT 문장이 저장되어 있다.

뷰를 사용하는 목적은 다음과 같다.
- 뷰는 지정된 컬럼 및 행만을 보여주기 때문에 데이터의 접근을 제한 할 수 있다.
- 복잡한 쿼리 문장을 뷰에 저장함으로서 쿼리 문장을 단순하게 만들 수 있다. 예를 들어, 복잡한 조인 문장을 뷰에 저장함으로서 검색시 매번 조인 문장을 작성 할 필요가 없어진다.
- 뷰는 임의 사용자 및 애플리케이션 프로그램에 대하여 데이터 독립성을 제공한다. 한 개의 뷰는 여러 개의 테이블로부터 데이터를 검색하는데 사용 될 수 있다.

뷰는 뷰 내에 정의된 SELECT 문장이 검색하는 테이블의 개수에 따라 단순 뷰와 복합 뷰로 분류된다.

표 12-1. 뷰의 종류

특징	단순 뷰	복합 뷰
테이블 갯수	1개	1개 이상
함수 포함 여부	포함하지 않음	포함
그룹 연산 포함 여부	포함하지 않음	포함
뷰를 통한 DML 작업 가능 여부	가능	부분적으로 가능

뷰의 작성 및 검색

뷰를 작성하는 CREATE VIEW 문장내에는 서브쿼리가 포함되며 문법은 다음과 같다.

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
      [(alias[, alias] ...)]
AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY [CONSTRAINT constraint]];
```

뷰를 작성 할 때는 다음 사항에 주의하여야 한다.

· 뷰에 포함되는 서브쿼리는 조인, 집계 연산, 서브쿼리가 포함된 복잡한 SELECT 문장이 정의 될 수 있다.
· 뷰에 포함되는 서브쿼리에는 ORDER BY 절을 사용 할 수 없다. ORDER BY를 사용하려면 검색시 뷰에 기술하도록 한다.
· WITH CHECK OPTION으로 생성된 뷰에 제약조건 이름을 지정하지 않으면 SYS_C*n* 형식으로 저장된다.
· 뷰를 삭제한 후 다시 생성하는 대신 OR REPLACE 옵션으로 뷰를 변경하면, 기존의 객체 권한을 재부여 할 필요가 없다.

부서번호가 10번인 직원들의 정보를 검색하는 EMPVU10을 작성하면 다음과 같다.

```
SQL> CREATE VIEW EMPVU10
  2  AS SELECT EMPNO, ENAME, SAL
  3  FROM EMP
  4  WHERE DEPTNO = 10;

뷰가 생성되었습니다.

SQL> SELECT * FROM EMPVU10;

     EMPNO ENAME             SAL
---------- ---------- ----------
      7782 CLARK            2450
      7839 KING             5000
      7934 MILLER           1300
```

만약에 Oracle 10g를 사용 중이라면 다음과 같이 뷰를 생성하기 전에 SCOTT 사용자에게 CREATE VIEW 권한을 미리 부여해주어야 한다.

```
SQL> CONNECT / AS SYSDBA
연결되었습니다.
SQL> GRANT CREATE VIEW TO SCOTT;

권한이 부여되었습니다.
SQL> CONNECT SCOTT/TIGER
연결되었습니다.
```

뷰의 구조를 확인하면 다음과 같다.

```
SQL> DESC EMPVU10
 이름                                      널?      유형
 ----------------------------------------- -------- ----------------------------
 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                              VARCHAR2(10)
 SAL                                                NUMBER(7,2)
```

컬럼 별칭을 이용하여 뷰를 생성하면, 뷰를 검색 할 때 지정된 컬럼 별칭을 사용해야 한다.

```
SQL> CREATE VIEW SALVU10
  2  AS SELECT EMPNO ID, ENAME NAME, SAL*12 YEAR_SAL
  3  FROM EMP
  4  WHERE DEPTNO = 10;

뷰가 생성되었습니다.

SQL> SELECT * FROM SALVU10;

        ID NAME         YEAR_SAL
---------- ---------- ----------
      7782 CLARK           29400
      7839 KING            60000
      7934 MILLER          15600
```

위 문장은 아래 문장과 동일한 뷰를 생성한다.

```
CREATE VIEW SALVU10 (ID, NAME, YEAR_SAL)
AS SELECT EMPNO, ENAME, SAL*12
FROM EMP
WHERE DEPTNO = 10;

뷰가 생성되었습니다.
```

뷰를 검색하는 방법은 테이블을 검색하는 방법과 동일하며 내부적인 처리과정은 다음과 같다.

1. USER_VIEWS 딕셔너리 테이블로부터 뷰의 정의를 검색한다.
2. 뷰가 참조하는 테이블에 대한 접근 권한을 확인한다.
3. 뷰 쿼리가 기본 테이블을 참조하는 쿼리로 변환된다. 즉, 기본 테이블로부터 데이터가 검색, 변경이 이루어진다.

뷰를 이용한 검색은 다음과 같다.

```
SQL> SELECT * FROM SALVU10
  2  WHERE YEAR_SAL > 20000;

        ID NAME         YEAR_SAL
---------- ---------- ----------
      7782 CLARK           29400
      7839 KING            60000
```

뷰를 변경하려면 CREATE OR REPLACE VIEW 명령을 사용하며 컬럼 별칭을 추가할 수 있다.

```
SQL> CREATE OR REPLACE VIEW EMPVU10 (ID, NAME, JOB, SALARY)
  2  AS SELECT EMPNO, ENAME, JOB, SAL
  3  FROM EMP
  4  WHERE DEPTNO = 10;

뷰가 생성되었습니다.
```

집계 함수나 조인 문장이 포함된 복합 뷰를 생성하는 방법은 다음과 같다.

```
SQL> CREATE VIEW DEPT_AVG_VU (NAME, AVG_SAL)
  2  AS SELECT D.DNAME, AVG(E.SAL)
  3  FROM DEPT D, EMP E
  4  WHERE D.DEPTNO = E.DEPTNO
  5  GROUP BY D.DNAME;

뷰가 생성되었습니다.

SQL> SELECT * FROM DEPT_AVG_VU;

NAME              AVG_SAL
-------------- ----------
ACCOUNTING     2916.66667
RESEARCH             2175
SALES          2111.11111
```

뷰를 이용한 데이터 변경

단순 뷰에는 DML 작업이 가능하지만 아래와 같은 경우에는 DML 작업이 불가능하다.

■ 삭제가 불가능한 경우
서브쿼리에 다음과 같은 함수 또는 구문이 사용되면 뷰를 통하여 행의 삭제가 불가능하다.
· 집계 함수
· GROUP BY 구문
· DISTINCT 키워드
· ROWNUM과 같은 가상 컬럼

■ 변경이 불가능한 경우
서브쿼리에 다음과 같은 함수 또는 구문이 사용되면 뷰를 통하여 행의 변경이 불가능하다.
· 집계 함수
· GROUP BY 구문
· DISTINCT 키워드
· ROWNUM과 같은 가상 컬럼
· 표현식으로 정의된 컬럼 (예, SAL*12)

■ 입력이 불가능한 경우
서브쿼리에 다음과 같은 함수 또는 구문이 사용되면 뷰를 통하여 행의 변경이 불가능하다.
· 집계 함수
· GROUP BY 구문
· DISTINCT 키워드
· ROWNUM과 같은 가상 컬럼
· 표현식으로 정의된 컬럼

· 기본 테이블에 뷰에서 선택되지 않은 NOT NULL 컬럼이 있는 경우

WITH CHECK OPTION

뷰에 정의된 쿼리의 WHERE 절에 WITH CHECK OPTION 옵션을 사용하면 WHERE 조건에 만족하는 데이터만이 INSERT, UPDATE 작업을 수행 할 수 있다. 아래와 같이 변경하고자하는 데이터가 WHERE 조건에 만족하지 않으면 오류를 발생시킨다.

```
SQL> CREATE OR REPLACE VIEW EMPVU10
  2  AS SELECT *
  3  FROM EMP
  4  WHERE DEPTNO = 10
  5  WITH CHECK OPTION CONSTRAINT EMPVU10_CK;

뷰가 생성되었습니다.

SQL> UPDATE EMPVU10 SET DEPTNO = 20 WHERE ENAME = 'MILLER';
UPDATE EMPVU10 SET DEPTNO = 20 WHERE ENAME = 'MILLER'
       *
1행에 오류:
ORA-01402: 뷰의 WITH CHECK OPTION의 조건에 위배 됩니다
```

WITH READ ONLY

뷰에 정의된 쿼리의 WHERE 절에 WITH READ ONLY 옵션을 추가하면 뷰를 통한 DML 작업이 불가능하다. 만약, DML 작업을 수행하려고 하면 Oracle 서버는 오류를 발생시킨다.

```
SQL> CREATE OR REPLACE VIEW EMPVU10
  2  AS SELECT *
  3  FROM EMP
  4  WHERE DEPTNO = 10
  5  WITH READ ONLY;

뷰가 생성되었습니다.

SQL> DELETE FROM EMPVU10
  2  WHERE ENAME = 'SMITH';
DELETE FROM EMPVU10
            *
1행에 오류:
ORA-01752: 뷰으로 부터 정확하게 하나의 키-보전된 테이블 없이 삭제할 수 없습니다
```

뷰의 삭제

뷰를 삭제하는 방법은 다음과 같다.

```
DROP VIEW view
```

```
SQL> DROP VIEW EMPVU10;

뷰가 삭제되었습니다.
```

인라인 뷰

FROM 절의 서브쿼리를 인라인 뷰라고 하며 별칭을 부여 할 수 있으며, 서브쿼리의 결과는 메인 쿼리에서 참조된다. 다음은 인라인 뷰를 이용하여 부서별 최대 급여를 출력하는 SQL 문장이다.

```
SQL> SELECT D.DNAME, E.MAXSAL
  2  FROM DEPT D, (SELECT DEPTNO, MAX(SAL) MAXSAL
  3                FROM EMP
  4                GROUP BY DEPTNO) E
  5  WHERE D.DEPTNO = E.DEPTNO;

DNAME              MAXSAL
-------------- ----------
ACCOUNTING           5000
RESEARCH             3000
SALES                3500
```

TOP-N 분석

TOP-N 쿼리란 컬럼 값 중에서 가장 큰 값 또는 가장 작은 값 *n* 개를 질의하는 경우에 사용되는 SQL 문장이다. 예를 들면, 급여가 가장 높은 5명의 직원을 검색하거나 입사일이 가장 늦은 5명의 직원을 검색하는 경우이다. TOP-N 쿼리의 형식은 다음과 같다.

```
SELECT [column_list], ROWNUM
FROM (SELECT [column_list]
      FROM table
      ORDER BY Top-N_column)
WHERE ROWNUM <= N;
```

위에서 보듯이 인라인 뷰에 검색하고자 하는 컬럼을 ORDER BY 구문을 이용하여 정렬한 다음, 메인 쿼리의 결과에서 필요한 행의 개수 만큼 행을 출력하면 된다. 쿼리의 결과를 임의 개수만 출력하려면 가상 컬럼인 ROWNUM을 사용하면 되는데, ROWNUM 컬럼은 테이블에 실제 존재하지는 않지만 쿼리 실행 결과를 출력할 때 출력순서대로 행의 번호를 붙여주는 컬럼이다.

사원 중에서 급여를 가장 많이 받는 5명의 직원을 출력하면 다음과 같다.

```
SQL> SELECT ROWNUM, ENAME, SAL
  2  FROM (SELECT ENAME, SAL
  3        FROM EMP
  4        ORDER BY SAL DESC)
  5  WHERE ROWNUM <= 5;

    ROWNUM ENAME             SAL
---------- ---------- ----------
         1 KING             5000
         2 SCOTT            3000
         3 FORD             3000
         4 JONES            2975
         5 BLAKE            2850
```

복습

1. 부서명과 사원명을 출력하는 뷰 DNAME_ENAME_VU를 작성하시오.

2. 사원명과 사수명을 출력하는 뷰 WORKER_MANAGER_VU를 작성하시오.

3. 사원 테이블에서 사번, 사원명, 입사일을 입사일이 늦은 사원 순으로 정렬하시오.

4. 사원 테이블에서 사번, 사원명, 입사일을 입사일이 늦은 사원 5명을 출력하시오.

5. 사원 테이블에서 사번, 사원명, 입사일을 입사일이 6번째로 늦은 사원부터 10번째 사원까지 출력하시오.

Chapter 13. 데이터베이스 객체

이번 장에서는 테이블과 뷰를 제외한 나머지 데이터베이스 객체인 시퀀스(Sequence), 인덱스(Index), 동의어(Synonym)에 대하여 설명한다.

시퀀스

시퀀스는 여러 사용자들이 공유하는 데이터베이스 객체로서 호출 될 때마다 중복되지 않은 고유한 숫자를 리턴하는 객체이다. 그러므로, 시퀀스는 기본키 컬럼에 사용할 값을 발생시키는데 주로 사용된다. 이와 같이, 번호를 중복되지 않게 취득하는 것을 보통 채번이라고 부르는데 반드시 시퀀스를 사용해야 하는 것은 아니며 애플리케이션 코드에 의해서도 구현될 수 있다. 그러나 시퀀스를 이용하면 한 번 호출시 여러 개의 시퀀스 번호를 메모리에 미리 캐시에 저장해두고 사용할 수 있으므로 작업 속도를 향상 시킬 수 있다.

■ 시퀀스 생성

시퀀스 번호를 자동으로 생성해주는 시퀀스를 작성하는 명령은 다음과 같다.

```
CREATE SEQUENCE sequence
       [INCREMENT BY n]
       [START WITH n]
       [{MAXVALUE n | NOMAXVALUE}]
       [{MINVALUE n | NOMINVALUE}]
       [{CYCLE | NOCYCLE)]
       [{CACHE n | NOCACHE}];
```

위에서 START WITH에 시퀀스 번호의 초기값을 입력하고, INCREMENT BY에 의해 시퀀스 번호의 증분을 기술한다. MAXVALUE와 MINVALUE에는 시퀀스 번호의 최대값 및 최소값을 입력한다. 만약, NOMAXVALUE로 지정된 경우 증분이 (+)이면 10^{27}, (-)이면 -1이 되며 NOMINVALUE로 지정된 경우 증분이 (+)이면 1, (-)이면 10^{26}이 된다.

시퀀스 작성시 CYCLE을 지정하면 시퀀스 번호가 최대값에 도달한 후 순환되므로. 시퀀스 번호를 기본키에 사용해서는 안된다.

다음은 EMP 테이블의 기본키 할당을 위한 시퀀스를 작성한 것이다.

```
SQL> CREATE SEQUENCE EMP_EMPNO_SEQ
  2  INCREMENT BY 1
  3  START WITH 8000
  4  MAXVALUE 9999
  5  NOCACHE
  6  NOCYCLE;

주문번호가 생성되었습니다.
```

만약, Oracle 10g를 사용 중이라면 시퀀스를 생성하기 전에 SCOTT에게 CREATE

SEQUENCE 권한을 부여해주어야 한다.

```
SQL> CONNECT / AS SYSDBA
연결되었습니다.
SQL> GRANT CREATE SEQUENCE TO SCOTT;

권한이 부여되었습니다.
SQL> CONNECT SCOTT/TIGER
연결되었습니다.
```

작성된 시퀀스는 USER_SEQUENCES 데이터 딕셔너리에서 확인 할 수 있다.

```
SQL> SELECT SEQUENCE_NAME, MIN_VALUE, MAX_VALUE,
  2  INCREMENT_BY, LAST_NUMBER
  3  FROM USER_SEQUENCES;

SEQUENCE_NAME                  MIN_VALUE  MAX_VALUE INCREMENT_BY LAST_NUMBER
------------------------------ ---------- ---------- ------------ -----------
EMP_EMPNO_SEQ                           1       9999            1        8000
```

시퀀스 작성시 NOCACHE를 지정하면 LAST_NUMBER컬럼에 다음 발생할 시퀀스 번호가 표시된다.

■ NEXTVAL, CURRVAL 가상 컬럼

시퀀스를 작성한 후에는 NEXTVAL, CURRVAL 가상 컬럼을 이용하여 시퀀스 번호를 검색할 수 있다. NEXTVAL 가상 컬럼은 지정된 시퀀스에서 순차적인 시퀀스 번호를 추출할 때 사용하는데, 시퀀스명.NEXTVAL에 의해서 시퀀스 번호가 추출되면 해당 시퀀스의 CURRVAL에 추출된 시퀀스 번호가 저장된다. CURRVAL 가상 컬럼은 사용자가 방금 추출한 시퀀스 번호를 참조하는데 사용되므로, 반드시 NEXTVAL에 의해서 시퀀스 번호를 추출한 후 사용해야한다.

NEXTVAL과 CURRVAL 가상 컬럼을 사용 할 수 있는 SQL 문장은 다음과 같다.
· 서브쿼리가 아닌 SELECT 문장의 컬럼
· INSERT 문장내 SELECT 문장의 컬럼
· INSERT 문장의 VALUES 절
· UPDATE 문장의 SET 절

다음은 NEXTVAL과 CURRVAL 가상 컬럼을 사용 할 수 없는 SQL 문장이다.
· 뷰 안에 포함된 SELECT 문장의 컬럼
· DISTINCT가 포함된 SELECT 문장
· GROUP BY, HAVING, ORDER BY 절이 포함된 SELECT 문장
· SELECT, DELETE, UPDATE 문장내 서브쿼리
· CREATE TABLE 또는 ALTER TABLE 문장에서 DEFAULT 값

■ 시퀀스의 사용

앞 절에서 생성한 EMP_EMPNO_SEQ를 이용하여 사원 테이블에 신규 데이터를 입력하

면 다음과 같다.

```
SQL> INSERT INTO EMP
  2  VALUES (EMP_EMPNO_SEQ.NEXTVAL, '길동', '홍보', NULL,
  3  SYSDATE, 3000, 300, 40);

1 개의 행이 만들어졌습니다.
```

EMP_EMPNO_SEQ 시퀀스에서 현재 시퀀스 번호를 확인하면 다음과 같다.

```
SQL> SELECT EMP_EMPNO_SEQ.CURRVAL FROM DUAL;

   CURRVAL
----------
      8000
```

시퀀스 생성시 CACHE 옵션을 사용하면 시퀀스 번호의 추출을 좀 더 빠르게 진행 할 수 있다. NEXTVAL에 의해 최초로 시퀀스 번호를 추출하면 CACHE 옵션에 기술된 숫자만큼의 시퀀스 번호가 추출되어 메모리에 캐시되며 이후, 다시 NEXTVAL에 의해 시퀀스 번호를 추출하면 캐시된 시퀀스 번호를 하나씩 리턴해준다. 이 후, 캐시된 시퀀스 번호가 모두 소비되면 다음번 NEXTVAL에 의해 다시 시퀀스 번호들이 메모리에 캐시된다.

시퀀스에 의해 추출되는 번호는 순차적이지만, 시퀀스 번호는 COMMIT 또는 ROLLBACK 명령의 영향을 받지 않기 때문에 시퀀스 번호에는 빈 번호들이 발생 할 수 있다. 즉, INSERT 문장에 의해 시퀀스 번호를 테이블에 입력한 다음, 해당 문장이 롤백되었다면 해당 시퀀스 번호를 영원히 잃어버리게 된다. 또한, 시퀀스 번호를 캐시하여 사용하는 경우, 시스템 오류에 의해 데이터베이스가 종료 되면 미리 캐시해둔 시퀀스 번호들도 모두 잃게 된다.

■ 시퀀스 변경

시퀀스의 증분, 최대값, 최소값, 순환여부, 캐시여부를 변경하는 방법은 다음과 같다.

```
SQL> ALTER SEQUENCE EMP_EMPNO_SEQ
  2  INCREMENT BY 2
  3  MAXVALUE 9000
  4  NOCACHE
  5  NOCYCLE;

주문번호가 변경되었습니다.
```

시퀀스 변경시 주의사항은 다음과 같다.

· 시퀀스를 변경하려면 해당 시퀀스의 소유자이거나 ALTER SEQUENCE 권한을 부여 받아야 한다.
· 시퀀스가 변경되면 다음번 시퀀스 번호 추출부터 변경사항이 적용된다.
· START WITH 옵션은 변경 할 수 없으며, 시퀀스를 삭제하고 재생성해야만 한다.
· MAXVALUE 값은 현재 시퀀스 번호보다 큰 번호로 지정해야 한다.

■ 시퀀스 삭제

데이터 딕셔너리에서 시퀀스를 삭제하는 방법은 다음과 같으며, 해당 시퀀스의 소유자이거나 DROP ANY SEQUENCE 권한을 부여받아야 한다.

```
SQL> DROP SEQUENCE EMP_EMPNO_SEQ;

주문번호가 삭제되었습니다.
```

인덱스

인덱스는 테이블에서 행을 검색할 때 검색 속도를 높이기 위해 Oracle 서버가 사용하는 스키마 객체이다. 테이블에 인덱스를 생성하지 않으면 데이터를 검색 할 때 테이블의 모든 데이터를 읽어 검색 조건에 의해 데이터가 선별되지만, 인덱스를 사용하면 조건에 맞는 데이터를 해당 테이블에 직접 접근하여 읽어오기 때문에 디스크의 I/O를 급격히 감소시킬 수 있다. 또한, 해당 테이블과 논리적으로 독립적이며 Oracle 서버에 의해 자동으로 사용 및 관리된다. 반면, 테이블을 삭제하면 관련 인덱스도 삭제된다.

인덱스는 Oracle 서버에 의해 자동으로 생성되기도 하며, 사용자의 요구에 의해 직접 생성할 수도 있다. 테이블에 PRIMARY KEY 제약조건 또는 UNIQUE 제약조건을 지정하면 해당 컬럼에는 고유 인덱스가 생성되며, 사용자가 필요에 따라 특정 컬럼에 별도의 인덱스를 생성하여 검색 속도를 높일 수 있다. 제약조건 생성시에 자동으로 생성되는 인덱스의 이름은 제약조건의 이름과 동일하게 부여된다.

참고로, FOREIGN KEY 제약조건이 지정된 컬럼에 인덱스를 생성해두면 조인시 검색속도가 향상되므로, 외래키 컬럼에는 반드시 인덱스를 생성하도록 한다.

■ 인덱스 생성

인덱스는 한 개 컬럼 또는 두 개 이상의 컬럼을 결합하여 생성 할 수 있다. 컬럼이 두 개 이상 결합되어 생성된 인덱스를 결합 인덱스라고 부른다.

```
CREATE INDEX index
ON table (column[, column] ...);
```

사원 테이블에서 사원명을 조건으로 검색하는 경우가 많다면 해당 컬럼에 인덱스를 생성하여 검색 속도를 향상 시킬 수 있다.

```
SQL> CREATE INDEX EMP_ENAME_IDX
  2  ON EMP(ENAME);

인덱스가 생성되었습니다.
```

■ 인덱스를 생성할 컬럼의 선정

인덱스가 많을수록 검색속도가 반드시 향상되는 것은 아니다. 또한, 인덱스가 생성된 테

이블에 DML 작업을 수행하면 관련 인덱스도 변경되어야 하므로 오히려 속도가 저하된다. 그러므로 인덱스의 수는 최소한으로 유지하면서 최대한 활용될 수 있도록 인덱스를 적절한 컬럼에 생성하도록 하여야 한다.

일반적으로 인덱스를 작성해야 경우는 다음과 같다.
· 값의 범위가 넓은 컬럼. 즉, 컬럼내의 값이 다양할수록 좋다.
· NULL 값이 많은 컬럼. NULL 값은 인덱스에 포함되지 않기 때문에 인덱스의 크기를 줄일 수 있다.
· WHERE 절이나 조인 조건에 사용되는 컬럼
· 테이블이 크고 대부분의 쿼리 문장이 테이블내 전체 데이터의 약 2~4% 이내를 검색하는 경우

다음은 인덱스를 작성할 필요가 없는 경우로서 인덱스를 생성하면 오히려 검색 속도가 늦어질 수도 있다.
· 테이블이 작은 경우
· 쿼리 문장의 조건에 자주 사용되지 않는 컬럼
· 대부분의 쿼리 문장이 테이블내 전체 데이터의 약 2~4% 이상을 검색하는 경우
· 테이블이 자주 변경되는 경우
· 인덱스가 작성된 컬럼이 쿼리 문장의 조건에서 표현식에 포함된 경우

■ 인덱스 확인

USER_INDEXES 데이터 딕셔너리와 USER_IND_COLUMNS 데이터 딕셔너리를 이용하면 인덱스와 관련된 정보를 확인 할 수 있다.

```
SQL> SELECT IC.INDEX_NAME, IC.COLUMN_NAME,
  2  IC.COLUMN_POSITION, IX.UNIQUENESS
  3  FROM USER_INDEXES IX, USER_IND_COLUMNS IC
  4  WHERE IC.INDEX_NAME = IX.INDEX_NAME
  5  AND IC.TABLE_NAME = 'EMP';

INDEX_NAME           COLUMN_NAME                    COLUMN_POSITION UNIQUENES
-------------------- ------------------------------ --------------- ---------
PK_EMP               EMPNO                                        1 UNIQUE
EMP_ENAME_IDX        ENAME                                        1 NONUNIQUE
```

■ 함수 기반 인덱스

인덱스가 생성된 컬럼이 함수에 입력되어 사용되거나 연산에 사용되면 절대 인덱스를 사용할 수 없다. 그래서 함수 기반 인덱스는 WHERE 조건의 함수 또는 연산식 자체를 인덱스로 생성한 것이다. 함수 기반 인덱스를 생성하려면 QUERY REWRITE 권한을 부여받아야 한다.

```
SQL> CONNECT / AS SYSDBA
연결되었습니다.
SQL> GRANT QUERY REWRITE TO SCOTT;

권한이 부여되었습니다.

SQL> CONNECT SCOTT/TIGER
연결되었습니다.
SQL> CREATE INDEX LOWER_JOB_IDX
  2  ON EMP(LOWER(JOB));

인덱스가 생성되었습니다.

SQL> SELECT * FROM EMP WHERE LOWER(JOB) = 'manager';
```

■ 인덱스 삭제

인덱스를 삭제하는 방법은 다음과 같다. 인덱스를 삭제하려면 인덱스의 소유자이거나 DROP ANY INDEX 권한을 부여받아야 한다.

```
SQL> DROP INDEX LOWER_JOB_IDX;

인덱스가 삭제되었습니다.
```

참고로, 테이블을 삭제하면 관련 인덱스와 제약조건은 자동으로 삭제되지만 뷰와 시퀀스는 삭제되지 않는다.

동의어

동의어는 객체에 대한 별칭이다. 예를 들어, 다른 사용자가 소유한 테이블을 검색하기 위해서는 검색 할 테이블의 앞에 소유자의 이름을 붙여주어야 하지만 동의어를 사용하면 좀 더 간단하게 해당 테이블을 검색 할 수 있다.

■ 동의어 생성

동의어를 생성하는 명령어는 다음과 같다.

```
CREATE [PUBLIC] SYNONYM synonym
FOR object;
```

EMP 테이블에 대한 동의어를 생성하면, 동의어를 통하여 검색이 가능하다.

```
SQL> CREATE SYNONYM E FOR EMP;

동의어가 생성되었습니다.

SQL> SELECT EMPNO, ENAME FROM E
  2  WHERE ENAME = 'SMITH';

     EMPNO ENAME
---------- ----------
      7369 SMITH
```

만약, Oracle 10g를 사용중이라면 동의어를 생성하기 전에 다음과 같이 CREATE SYNONYM 권한을 부여해주어야 한다.

```
SQL> CONNECT / AS SYSDBA
연결되었습니다.
SQL> GRANT CREATE SYNONYM TO SCOTT;

권한이 부여되었습니다.
SQL> CONNECT SCOTT/TIGER
연결되었습니다.
```

동의어 작성시 PUBLIC 옵션을 추가하면 다른 사용자들도 같이 사용 할 수 있는 공용 동의어가 생성된다. 그러나, DBA 권한을 갖는 관리자만이 공용 동의어를 만들 수 있으며 다른 사용자들이 사용하기 위해서는 공용 동의어에 의해 참조되는 테이블에 접근 할 수 있는 권한을 부여 받아야 한다.

```
SQL> CONNECT / AS SYSDBA;
연결되었습니다.
SQL> CREATE PUBLIC SYNONYM HR_EMP FOR HR.EMPLOYEES;

동의어가 생성되었습니다.

SQL> GRANT SELECT ON HR.EMPLOYEES TO SCOTT;

권한이 부여되었습니다.

SQL> CONNECT SCOTT/TIGER
연결되었습니다.

SQL> SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME FROM HR_EMP
  2  WHERE FIRST_NAME = 'Steven';

EMPLOYEE_ID FIRST_NAME           LAST_NAME
----------- -------------------- -------------------------
        100 Steven               King
        128 Steven               Markle
```

■ 동의어 삭제

동의어를 삭제하는 방법은 다음과 같다. 공용 동의어는 관리자만이 삭제할 수 있다.

```
SQL> DROP SYNONYM E;

동의어가 삭제되었습니다.

SQL> CONNECT / AS SYSDBA
연결되었습니다.
SQL> DROP PUBLIC SYNONYM HR_EMP;

동의어가 삭제되었습니다.
```

복습

1. 초기값 1, 증분 1, 최소값 1, 최대값 9999인 TEST_SEQ를 생성하시오. 단, 시퀀스 번호는 순환되지 않으며, 시퀀스 번호 추출시 한번에 10개씩 메모리에 캐시하도록 한다.

2. 사원 테이블의 외래키인 부서코드 컬럼에 인덱스를 작성하시오.

3. 사원 테이블의 사원명 컬럼과 업무 컬럼에 결합 인덱스를 작성하시오.

4. 부서 테이블에 대한 동의어 D를 생성하시오.

5. 사원 테이블에 대한 공용 동의어 E를 생성하시오.

Chapter 14. 사용자 접근 제어

이번 장에서는 사용자를 생성하고 롤(Role)을 이용하여 사용자에게 여러 가지 권한(Privilege)을 효과적으로 부여 및 관리하는 방법을 설명한다. 권한 부여 및 회수를 위한 CREATE 및 REVOKE 문장을 살펴보고 원격 데이터베이스 연결을 위한 데이터베이스 링크의 작성 및 사용방법을 알아본다.

Oracle 데이터베이스의 보안

다중 사용자 환경에서 개별 사용자들은 데이터베이스 접근 및 사용에 있어서 적절한 보안을 유지하여야만 한다. 이를 위하여 Oracle 서버에서는 다음과 같은 작업을 수행 할 수 있다.

· 데이터베이스 접근 제어
· 데이터베이스의 특정 객체에 대한 접근 권한 부여
· Oracle 데이터 딕셔너리로부터 부여하거나 부여 받은 권한 확인
· 데이터베이스 객체에 대한 동의어 작성

데이터베이스 보안은 시스템 보안과 데이터 보안으로 분류 할 수 있다. 시스템 보안은 사용자 계정 생성, 암호 변경, 디스크 공간 할당, 시스템 작업 등과 같이 시스템 수준에서의 데이터베이스 접근 및 사용을 관리하는 것이며 데이터베이스 보안은 데이터베이스 객체에 대한 사용자들의 접근 및 사용을 관리하는 것이다.

사용자 생성

데이터베이스 관리자(DBA)는 CREATE USER 명령을 이용하여 사용자를 생성 할 수 있다. CREATE USER 명령의 문법은 다음과 같다.

```
CREATE USER user
IDENTIFIED BY password;
```

계정은 TOM, 암호는 JERRY인 사용자를 생성하는 방법은 다음과 같다.

```
SQL> CONNECT / AS SYSDBA
연결되었습니다.
SQL> CREATE USER TOM
  2  IDENTIFIED BY JERRY;

사용자가 생성되었습니다.
```

위에서 생성된 TOM은 아직 아무런 권한이 부여되지 않았기 때문에 어떠한 작업도 불가능하다.

TOM 사용자의 암호를 TIGER로 변경하는 방법은 다음과 같다.

```
SQL> ALTER USER TOM
  2  IDENTIFIED BY TIGER;

사용자가 변경되었습니다.
```

권한

권한이란 특별한 SQL 문장을 실행 할 수 있는 권리를 의미한다. 데이터베이스 관리자는 사용자들에게 데이터베이스와 데이터베이스 객체에 접근 할 수 있는 권한을 부여 할 수 있는 고급 사용자이며, 일반 사용자들은 데이터베이스에 접근 할 수 있는 시스템 권한과 데이터베이스 객체의 내용에 접근할 수 있는 객체 권한을 부여 받아야 한다. 또한, 사용자는 다른 사용자 또는 롤(Role)에게 권한을 부여 할 수 있는 권한을 부여 받을 수도 있다. 롤이란 관련 권한들의 논리적 집합이다.

참고로 스키마(Schema)란 테이블, 뷰, 시퀀스와 같은 객체들의 모음이다. 스키마는 데이터베이스 사용자가 소유하며 사용자 이름과 동일한 이름을 갖는다.

시스템 권한

Oracle에서 사용가능한 시스템 권한은 약 100여개 이상이며, 일반적으로 데이터베이스 관리자에 의해 부여된다. 다음은 데이터베이스 관리자가 가지고 있는 일반적인 시스템 권한이다.

표 14-1. 일반적인 데이터베이스 관리자의 시스템 권한

시스템 권한	수행 가능한 작업
CREATE USER	사용자 생성
DROP USER	사용자 삭제
DROP ANY TABLE	모든 스키마에서 테이블 삭제 가능
BACKUP ANY TABLE	모든 스키마에서 Exp 유틸리티를 이용하여 백업 가능
SELECT ANY TABLE	모든 스키마에서 테이블, 뷰, 스냅샷을 검색 가능
CREATE ANY TABLE	모든 스키마에서 테이블 생성 가능

사용자가 생성되면 데이터베이스 관리자는 특정한 시스템 권한을 사용자에게 부여해야 한다. 권한을 부여하는 명령은 다음과 같다.

```
GRANT privilege [, privilege ...]
TO user [, user | role | PUBLIC ...];
```

위에서 PUBLIC은 모든 사용자에게 지정된 권한을 부여하는 것이며, 다음은 일반 사용자에게 부여하는 일반적인 시스템 권한이다.

표 14-2. 일반적인 사용자에게 부여하는 시스템 권한

시스템 권한	수행 가능한 작업
CREATE SESSION	데이터베이스 연결
CREATE TABLE	사용자 스키마에 테이블 생성
CREATE SEQUENCE	사용자 스키마에 시퀀스 생성
CREATE VIEW	사용자 스키마에 뷰 생성
CREATE PROCEDURE	사용자 스키마에 저장 프로시져, 함수, 패키지 생성

TOM 사용자에게 시스템 권한을 부여하면 다음과 같다.

```
SQL> GRANT CREATE SESSION, CREATE TABLE,
  2  CREATE SEQUENCE, CREATE VIEW
  3  TO TOM;

권한이 부여되었습니다.
```

객체 권한

객체 권한은 특정 테이블, 뷰, 시퀀스, 프로시저 등에 특별한 작업을 수행 할 수 있는 권리이다. 각 객체는 아래 표와 같이 각각 부여 가능한 권리들의 집합을 가지고 있다. 예를 들어, 시퀀스에는 ALTER와 SELECT 할 수 있는 권한만 부여 가능하다.

표 14-3. 객체 권한

객체 권한	테이블	뷰	시퀀스	프로시저
ALTER	√		√	
DELETE	√	√		
EXECUTE				√
INDEX	√			
INSERT	√	√		
REFERENCES	√	√		
SELECT	√	√	√	
UPDATE	√	√		

위에서 보는 것과 같이 객체 권한은 객체에 따라 부여 할 수 있는 권한이 다르다. 사용자는 일반적으로 자신의 스키마에 저장된 모든 객체에 대하여 모든 권한을 부여 받기 때문에 다른 사용자 또는 롤에게 자신이 소유한 권한을 부여 할 수 있다. 객체 권한을 부여하는 명령은 다음과 같다.

```
GRANT object_priv [(columns)]
ON object
TO {user | role | PUBLIC}
[WITH GRANT OPTION];
```

객체 권한을 부여할 때 WITH GRANT OPTION을 사용하면 권한을 부여 받은 사람이 받은 권한을 다른 사용자에게 다시 부여 할 수 있다.

SCOTT가 자신의 EMP 테이블을 SELECT 할 수 있는 권한을 TOM에게 부여하는 방법은

다음과 같다.

```
SQL> CONNECT SCOTT/TIGER
연결되었습니다.
SQL> GRANT SELECT ON EMP TO TOM;

권한이 부여되었습니다.

SQL> CONNECT TOM/TIGER
연결되었습니다.
SQL> SELECT ENAME, JOB FROM SCOTT.EMP
  2  WHERE ENAME = 'SMITH';

ENAME      JOB
---------- ---------
SMITH      CLERK
```

DEPT 테이블의 특정 컬럼을 UPDATE 할 수 있는 권한을 TOM에게 부여하면 다음과 같다.

```
SQL> CONNECT SCOTT/TIGER
연결되었습니다.
SQL> GRANT UPDATE (DNAME, LOC) ON DEPT TO TOM;

권한이 부여되었습니다.

SQL> CONNECT TOM/TIGER
연결되었습니다.

SQL> UPDATE SCOTT.DEPT
  2  SET LOC='서울'
  3  WHERE DEPTNO = 10;

1 행이 갱신되었습니다.
```

WITH GRANT OPTION을 사용하면 부여된 권한을 받은 사용자가 해당 권한을 다른 사용자에게 부여 할 수 있다.

```
SQL> CONNECT SCOTT/TIGER
연결되었습니다.
SQL> GRANT SELECT, INSERT
  2  ON DEPT
  3  TO TOM
  4  WITH GRANT OPTION;

권한이 부여되었습니다.

SQL> CONNECT TOM/TIGER
연결되었습니다.
SQL> GRANT SELECT
  2  ON SCOTT.DEPT
  3  TO PUBLIC;

권한이 부여되었습니다.
```

부여된 권한의 확인

사용자에게 부여된 권한을 확인 할 수 있는 데이터 딕셔너리는 다음과 같다.

표 14-4. 권한 관련 데이터 딕셔너리

데이터 딕셔너리	설명
ROLE_SYS_PRIVS	롤에 부여된 시스템 권한
ROLE_TAB_PRIVS	롤에 부여된 테이블 권한
USER_ROLE_PRIVS	사용자가 접근 가능한 롤
USER_TAB_PRIVS_MADE	사용자가 부여한 객체 권한
USER_TAB_PRIVS_RECD	사용자에게 부여된 객체 권한
USER_COL_PRIVS_MADE	사용자가 부여한 컬럼에 대한 객체 권한
USER_COL_PRIVS_RECD	사용자에게 부여된 컬럼에 대한 객체 권한
USER_SYS_PRIVS	사용자에게 부여된 시스템 권한

SCOTT가 부여한 권한과 TOM이 부여 받은 권한을 확인하는 방법은 다음과 같다.

```
SQL> CONNECT SCOTT/TIGER
연결되었습니다.
SQL> SELECT GRANTEE, TABLE_NAME, GRANTOR, PRIVILEGE
  2  FROM USER_TAB_PRIVS_MADE;

GRANTEE    TABLE_NAME           GRANTOR    PRIVILEGE
---------- -------------------- ---------- --------------------
TOM        DEPT                 SCOTT      INSERT
TOM        DEPT                 SCOTT      SELECT
PUBLIC     DEPT                 TOM        SELECT
TOM        EMP                  SCOTT      SELECT

SQL> CONNECT TOM/TIGER
연결되었습니다.
SQL> SELECT OWNER, TABLE_NAME, GRANTOR, PRIVILEGE
  2  FROM USER_TAB_PRIVS_RECD;

OWNER      TABLE_NAME GRANTOR    PRIVILEGE
---------- ---------- ---------- --------------------
SCOTT      DEPT       SCOTT      INSERT
SCOTT      DEPT       SCOTT      SELECT
SCOTT      EMP        SCOTT      SELECT
```

객체 권한의 회수

객체 권한을 회수하면 WITH GRANT OPTION에 의해 다른 사람에게 부여된 권한도 모두 회수된다. 예를 들어, A 사용자가 B 사용자에게 특정 테이블의 SELECT 권한을 WITH GRANT OPTION으로 부여하고, B 사용자가 부여 받은 권한을 다시 WITH GRANT OPTION으로 C 사용자에게 부여한 후에 C 사용자가 부여 받은 권한을 다시 D 사용자에게 부여했다면 A 사용자가 B 사용자에게 부여된 권한을 회수하면 C, D 사용자에게 부여된 모든 권한도 회수된다. 부여된 객체 권한을 회수하는 명령은 다음과 같다.

```
REVOKE {privilege [, privilege ...] | ALL}
ON object
FROM {user [, user ...] | role | PUBLIC}
[CASCADE CONSTRAINTS];
```

위에서 CASCADE CONSTRAINTS 옵션을 추가하면 REFERENCES 권한에 의해 객체에 부여된 참조 무결성 제약조건도 삭제한다.

SCOTT가 TOM에게 부여한 부서 테이블의 SELECT, INSERT 권한을 회수하는 방법은 다음과 같다.

```
SQL> REVOKE SELECT, INSERT ON DEPT
  2  FROM TOM;

권한이 취소되었습니다.
```

롤(Role)

롤이란 사용자에게 부여 할 관련 권한들의 논리적인 집합이다. 롤을 사용하면 권한의 부여, 회수 작업을 단순화 할 수 있다. 사용자는 여러 개의 롤을 할당 받을 수 있으며, 여러 사용자는 같은 롤을 할당 받을 수도 있다.

먼저, 데이터베이스 관리자는 롤을 생성하고, 관련 권한들을 롤에 할당한 다음 사용자에게 해당 롤을 할당하면 된다.

```
SQL> CONNECT / AS SYSDBA
연결되었습니다.
SQL> CREATE ROLE MANAGER;

롤이 생성되었습니다.

SQL> GRANT CREATE TABLE, CREATE VIEW TO MANAGER;

권한이 부여되었습니다.

SQL> GRANT MANAGER TO TOM;

권한이 부여되었습니다.
```

데이터베이스 링크

데이터베이스 링크는 로컬 Oracle 데이터베이스 서버에서 원격 Oracle 데이터베이스 서버로의 단방향 통신 경로를 정의하는 포인터이다. 링크 포인터는 데이터 딕셔너리에 저장되며, 저장된 링크 포인터를 사용하기 위해서는 해당 데이터 딕셔너리가 있는 로컬 데이터베이스에 먼저 연결해야 한다.

데이터베이스 링크는 단방향이므로 A 데이터베이스에 연결된 사용자는 저장된 링크를 사용

하여 B 데이터베이스에 연결 할 수 있지만 B 데이터베이스에 연결된 사용자는 A 데이터베이스에 연결 할 수 없다. 만약, B 데이터베이스에 연결된 사용자가 A 데이터베이스에 연결하려면 B 데이터베이스에 A 데이터베이스로의 링크가 데이터 딕셔너리에 저장되어 있어야 한다.

데이터베이스 링크는 로컬 사용자가 원격 데이터베이스의 데이터에 접근 할 수 있도록 해주며, 이러한 연결을 수행하기 위해서는 각 데이터베이스가 고유한 전역 데이터베이스 이름을 갖고 있어야 한다. 데이터베이스 링크를 사용함으로서 얻을 수 있는 가장 큰 장점은 로컬 데이터베이스의 사용자들이 원격 데이터베이스내 객체에 접근하면 모든 권한은 해당 객체의 소유자 권한으로 한정된다는 점이다. 데이터베이스 링크는 데이터베이스 관리자가 생성하며 USER_DB_LINKS 데이터 딕셔너리를 이용하여 확인할 수 있다.

데이터베이스 링크를 생성하기에 앞서 TNSNAME.ORA 파일에 원격 데이터베이스의 접속 연결자를 추가한다. 여기서, 원격 Oracle 서버는 192.168.0.100, 서비스명은 ora92, 접속 연결자는 MYORA92이다.

```
# TNSNAMES.ORA Network Configuration File:
# C:WoracleWora92WnetworkWadminWtnsnames.ora
# Generated by Oracle configuration tools.

MYORA92 =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.0.100)(PORT = 1521))
    )
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = ora92)
    )
  )
```

데이터베이스 링크를 작성한다.

```
SQL> CONNECT / AS SYSDBA
연결되었습니다.
SQL> CREATE PUBLIC DATABASE LINK LINK_TEST
  2  CONNECT TO SCOTT IDENTIFIED BY TIGER
  3  USING 'MYORA92';

데이타베이스 링크가 생성되었습니다.

SQL> SELECT * FROM DEPT@LINK_TEST;

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

복습

1. 계정이 KIM, 암호가 LION인 사용자 계정을 작성하시오.

2. KIM에게 CREATE TABLE과 CREATE SESSION 권한을 부여하시오.

3. KIM에게 SCOTT의 DEPT, EMP 테이블의 SELECT 권한을 부여하시오.

4. KIM에게 SCOTT의 EMP 테이블에 SAL, COMM 컬럼을 UPDATE 할 수 있는 권한을 부여하시오.

5. KIM에게 부여된 EMP 테이블의 UPDATE 권한을 회수하시오.

Chapter 15. 집합 연산자

이번 장에서는 집합 연산자에 대하여 알아보고 집합 연산자를 이용하여 여러 개의 쿼리 문장을 하나의 쿼리 문장으로 결합하는 방법을 설명한다.

집합 연산자의 종류

집합 연산자는 두 개의 쿼리 문장을 결합하며 다음과 같은 연산을 할 수 있다.

표 15-1. 집합 연산자

연산자	결과 값
UNION	각 쿼리에 의해서 선택 된 결과 중에 중복 행을 제거하고 출력
UNION ALL	각 쿼리에 의해서 선택 된 모든 결과를 출력, 중복 허용
INTERSECT	두 개의 쿼리를 모두 만족하는 결과 중에 중복 행을 제거하고 출력
MINUS	첫 번째 쿼리에 의해서 선택 된 결과에서 두 번째 쿼리에 의해서 선택 된 결과를 제거하고 출력

집합 연산자의 사용방법을 학습하기 위해 다음과 같은 임시 테이블을 작성하자.

```
CREATE TABLE A(VAL CHAR);
INSERT INTO A VALUES ('A');
INSERT INTO A VALUES ('B');
INSERT INTO A VALUES ('C');
INSERT INTO A VALUES ('D');
INSERT INTO A VALUES ('E');

CREATE TABLE B(NUM NUMBER, VAL CHAR);
INSERT INTO B VALUES (1, 'C');
INSERT INTO B VALUES (2, 'D');
INSERT INTO B VALUES (3, 'E');
INSERT INTO B VALUES (4, 'F');
INSERT INTO B VALUES (5, 'G');
```

UNION

UNION 연산자는 각각의 쿼리에 의해서 선택된 모든 행들에 대하여 중복을 제거하고 출력한다. UNION 연산자를 사용할 때 다음과 같은 사항을 주의해야 한다.

- 각각의 SELECT 문장에 의해서 선택된 컬럼의 개수와 데이터 타입은 반드시 일치해야 하지만, 컬럼명은 같을 필요가 없다.
- UNION 연산자는 선택된 모든 컬럼들에 대하여 적용된다.
- 각각의 쿼리 결과에서 중복을 제거하는 과정에 NULL 값은 무시된다.
- IN 연산자가 UNION 연산자보다 우선순위가 높다.
- 기본적으로 SELECT 문장의 첫 번째 컬럼 기준으로 오름차순 정렬된다.

다음은 A 테이블과 B 테이블을 UNION 연산한 결과이다.

```
SQL> SELECT VAL FROM A
  2  UNION
  3  SELECT VAL FROM B;

V
-
A
B
C
D
E
F
G

7 개의 행이 선택되었습니다.
```

UNION ALL

UNION ALL 연산자는 각각의 쿼리에 의해서 선택된 모든 행들을 출력한다. UNION ALL 연산자를 사용 할 때 다음과 같은 사항을 주의해야 한다.

· UNION과는 달리 중복된 행을 제거하지 않으며, 디폴트로 출력결과를 정렬하지 않는다.
· DISTINCT 키워드는 사용 할 필요가 없다.

다음은 A 테이블과 B 테이블을 UNION ALL 연산한 결과이다.

```
SQL> SELECT VAL FROM A
  2  UNION ALL
  3  SELECT VAL FROM B;

V
-
A
B
C
D
E
C
D
E
F
G

10 개의 행이 선택되었습니다.
```

INTERSECT

INTERSECT 연산자는 각각의 쿼리에 공통적으로 포함되어 있는 행들을 출력한다. INTERSECT 연산자를 사용 할 때 다음과 같은 사항을 주의해야 한다.

· 각각의 SELECT 문장에 의해서 선택된 컬럼의 개수와 데이터 타입은 반드시 일치해야 하지만, 컬럼명은 같을 필요가 없다.
· 각각의 쿼리 작성 순서는 연산 결과를 변경하지 않는다.
· INTERSECT 연산자는 NULL 값을 무시하지 않는다.

다음은 A 테이블과 B 테이블을 INTERSECT 연산한 결과이다.

```
SQL> SELECT VAL FROM A
  2  INTERSECT
  3  SELECT VAL FROM B;

V
-
C
D
E
```

MINUS

MINUS 연산자는 첫 번째 쿼리 결과에서 두 번째 쿼리 결과에 포함된 행들을 제거하고 출력한다. MINUS 연산자를 사용 할 때 다음과 같은 사항을 주의해야 한다.

· 각각의 SELECT 문장에 의해서 선택된 컬럼의 개수와 데이터 타입은 반드시 일치해야 하지만, 컬럼명은 같을 필요가 없다.

다음은 A 테이블과 B 테이블을 MINUS 연산한 결과이다.

```
SQL> SELECT VAL FROM A
  2  MINUS
  3  SELECT VAL FROM B;

V
-
A
B
```

집합 연산자 사용 지침

집합 연산자를 사용할 때는 각각의 SELECT 문장의 컬럼의 개수와 데이터 타입이 정확히 일치해야 하며, 집합 연산자의 처리 순서를 변경하려면 괄호를 이용하면 된다. 집합 연산자가 사용된 쿼리 문장에서 ORDER BY 절을 사용할 때는 문장의 제일 마지막에 기술하며, 첫 번째 쿼리 문장의 컬럼명, 별칭, 또는 컬럼 순번을 지정 할 수 있다.

Oracle 서버에서 집합 연산자는 다음과 같은 특징을 갖는다.
· Oracle에서 집합 연산자는 UNION ALL 연산을 제외한 모든 연산에서 중복된 행을 자동

으로 제거한다.

· 첫 번째 쿼리의 컬럼명이 결과에 표시된다.

· UNION ALL 연산을 제외하고 모든 출력은 기본적으로 오름차순 정렬된다.

집합 연산자를 사용 할 때, 각각의 쿼리 문장에서 컬럼의 개수와 데이터 타입이 불일치 되는 경우 컬럼을 추가하거나 데이터 타입 변환 함수를 사용하여 일치시키면 된다.

```
SQL> SELECT VAL, NULL FROM A
  2  UNION
  3  SELECT VAL, NUM FROM B;

V       NULL
- ----------
A
B
C          1
C
D          2
D
E          3
E
F          4
G          5

10 개의 행이 선택되었습니다.
```

집합 연산자는 기본적으로 첫 번째 쿼리의 첫 번째 컬럼을 기준으로 오름차순 정렬되지만, 각 쿼리 결과별로 정렬하려면 별도의 컬럼을 추가하고 ORDER BY에 기술해주면 된다.

```
SQL> COL C NOPRINT
SQL> SELECT VAL, 1 C FROM A
  2  UNION
  3  SELECT VAL, 2 FROM B
  4  ORDER BY 2;

V
-
A
B
C
D
E
C
D
E
F
G

10 개의 행이 선택되었습니다.
```

복습

1. 다음 연산자 중 출력결과가 정렬되지 않는 집합 연산자는?
   a. UNION
   b. UNION ALL
   c. INTERSECT
   d. MINUS

2. 다음 연산자 중 쿼리의 작성 순서가 결과에 영향을 미치는 집합 연산자는?
   a. UNION
   b. UNION ALL
   c. INTERSECT
   d. MINUS

3. 집합 연산자를 사용 할 때 주의해야 할 사항이 아닌 것은?
   a. 각각의 쿼리에서 컬럼의 개수는 일치해야 한다.
   b. 각각의 쿼리에서 컬럼의 데이터 타입은 일치해야 한다.
   c. 각각의 쿼리에서 컬럼명은 일치해야 한다.

Chapter 16. Oracle 9i 날짜 함수

이번 장에서는 Oracle 9i에서 새롭게 지원하는 날짜 함수에 대하여 자세히 알아본다.

표준시간대(Time Zone)

Oracle 9i부터 날짜 데이터 타입에 표준시간대를 입력 할 수 있는 데이터 타입과 이러한 데이터 타입을 조작 할 수 있는 함수들이 추가되었다. 추가된 데이터 타입과 함수를 이해하기 위해서는 표준시간대와 그리니치 표준시(GMT : Greenwich Mean Time)에 대한 개념을 알아야 한다. 우리가 살고 있는 지구는 전 세계를 24개의 시간대로 나누고 영국의 그리니치 천문대를 기준 표준시간대로 지정하여 다른 표준시간대들에 의해 참조된다. 예를 들어, 한국은 그리니치 표준시를 기준으로 +9:00 시간 빠르다.

TIMESTAMP 데이터 타입

Oracle 9i 이전에서는 날짜를 저장하기 위한 데이터 타입으로 DATE 타입이 있지만 Oracle 9i 부터는 표준시간대와 정밀한 시간(10^{-9}초)을 저장 할 수 있는 TIMESTAMP 데이터 타입이 추가되었다.

DATE 타입에 새롭게 추가된 데이터 타입은 다음과 같다.

표 16-1. 추가된 날짜 타입

데이터 타입	표준 시간대	정밀도(초)
DATE	저장되지 않음	초
TIMESTAMP (*fractional_seconds_precision*)	저장되지 않음	fractional_seconds_precision는 초의 소수점 자리수를 지정하며, 지정가능한 값은 0~9이며 디폴트는 6
TIMESTAMP (*fractional_seconds_precision*) WITH TIME ZONE	TIMESTAMP에 지역시간과 그리니치 표준시와의 시간차가 저장	〃
TIMESTAMP (*fractional_seconds_precision*) WITH LOCAL TIME ZONE	TIMESTAMP WITH TIME ZONE에 다음과 같은 사항을 제외하고 저장 - 날짜를 저장 할 때 데이터베이스 표준시간대를 기준으로 변환 저장 - 날짜가 검색될 때 세션의 표준시간대로 변환	〃

위에서 TIMESTAMP WITH LOCAL TIME ZONE 타입에 날짜가 저장되면 데이터베이스에 지정된 표준시간대로 변환되어 저장된다. 예를 들어, 서울(+9:00)에서 뉴욕(-5:00)에 있는 데이터베이스의 TIMESTAMP WITH LOCAL TIME ZONE으로 정의된 컬럼에 '2004-6-27 20:00'를 저장하면 뉴욕의 표준시간대를 기준으로 '2004-6-27 6:00'가 저장된다. 이 후, 서울에서 입력된 데이터를 검색하면 '2004-6-27 20:00'가 검색되며, 뉴욕에서

검색하면 '2004-6-27 6:00'가 검색된다. 또한, Oracle 9i는 날짜 데이터 타입에 섬머타임(Daylight Savings Time)을 지원해주므로 섬머타임제가 시작되면 시간이 1시간 앞당겨지고 섬머타임제가 종료되면 원래 시간으로 설정된다.

TZ_OFFSET

TZ_OFFSET 함수는 입력된 표준시간대에 대하여 그리니치 표준시와의 시차를 리턴하는 함수이다. 예를 들어, TZ_OFFSET 함수가 +9:00를 리턴하면 입력된 표준시간대가 그리니치 표준시를 기준으로 9시간 빠르다는 것을 의미한다.

```
SQL> SELECT TZ_OFFSET('Asia/Seoul') FROM DUAL;

TZ_OFFS
-------
+09:00

SQL> SELECT TZ_OFFSET('Europe/London') FROM DUAL;

TZ_OFFS
-------
+01:00
```

TZ_OFFSET의 입력 값으로 사용 할 수 있는 표준시간대 이름을 확인하려면 V$TIMEZONE_NAMES를 검색하면 된다.

```
SQL> SELECT TZNAME FROM V$TIMEZONE_NAMES
  2  WHERE UPPER(TZNAME) LIKE '%SEOUL%';

TZNAME
--------------------
Asia/Seoul
Asia/Seoul
Asia/Seoul
```

CURRENT_DATE

CURRENT_DATE 함수는 세션의 표준시간대에서 현재 날짜를 리턴한다. 그러므로, 세션의 표준시간대가 변경되면 CURRENT_DATE 함수의 결과는 달라지게 된다. 현재 세션에서의 표준시간대와 날짜를 확인해보자. 먼저, 정확한 시간을 표시하기 위해 ALTER SESSION 명령으로 날짜 표시 형식을 변경한다.

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

세션이 변경되었습니다.

SQL> SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;

SESSIONTIMEZONE                CURRENT_DATE
------------------------------ -------------------
+09:00                         2004-06-27 22:03:51
```

세션의 표준시간대를 변경하여 현재 날짜를 확인해보자. 표준시간대가 변경되었으므로 CURRENT_DATE 함수의 결과값도 변경되었다.

```
SQL> ALTER SESSION
  2  SET TIME_ZONE = '+5:0';

세션이 변경되었습니다.

SQL> SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;

SESSIONTIMEZONE                CURRENT_DATE
------------------------------ -------------------
+05:00                         2004-06-27 18:05:26
```

참고로 SYSDATE 함수는 표준시간대 설정과는 무관한 결과를 리턴한다.

```
SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
-------------------
2004-06-27 22:06:42
```

CURRENT_TIMESTAMP

CURRENT_TIMESTAMP 함수도 CURRENT_DATE 함수와 마찬가지로 표준시간대의 설정에 따라 그 결과도 달라진다. 반면, CURRENT_TIMESTAMP 함수는 CURRENT_DATE 함수에 비해 좀 더 정밀한 시간을 표시한다. 즉, TIMESTAMP WITH TIME ZONE 데이터 타입으로 출력된다.

```
SQL> SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;

SESSIONTIMEZONE                CURRENT_TIMESTAMP
------------------------------ ---------------------------------
+05:00                         04/06/27 18:11:45.218000 +05:00
```

마찬가지로 표준시간대를 변경하면 리턴 값도 달라진다.

```
SQL> ALTER SESSION
  2  SET TIME_ZONE = '+9:0';

세션이 변경되었습니다.

SQL> SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;

SESSIONTIMEZONE                CURRENT_TIMESTAMP
------------------------------ ---------------------------------
+09:00                         04/06/27 22:13:07.747000 +09:00
```

LOCALTIMESTAMP

LOCALTIMESTAMP 함수는 CURRENT_TIMESTAMP와 마찬가지로 세션의 표준시간대에서 현재 날짜를 리턴한다. 그러나 LOCALTIMESTAMP 함수의 리턴값은 TIMESTAMP 타입으로 출력되며, CURRENT_TIMESTAMP 함수의 리턴값은 TIMESTAMP WITH TIME ZONE 타입으로 출력된다.

```
SQL> SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;

CURRENT_TIMESTAMP                         LOCALTIMESTAMP
----------------------------------------- --------------------------
04/06/27 22:26:18.664000 +09:00           04/06/27 22:26:18.664000

SQL> ALTER SESSION
  2  SET TIME_ZONE = '+5:0';

세션이 변경되었습니다.

SQL> SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;

CURRENT_TIMESTAMP                         LOCALTIMESTAMP
----------------------------------------- --------------------------
04/06/27 18:26:59.122000 +05:00           04/06/27 18:26:59.122000
```

DBTIMEZONE, SESSIONTIMEZONE

DBTIMEZONE 함수는 디폴트 데이터베이스 표준시간대로서 운영체제의 표준시간대와 동일하다. 디폴트 데이터베이스 표준시간대는 CREATE DATABASE 문장의 SET TIME_ZONE 구문으로 지정가능하며, 세션에서는 ALTER SESSION 명령에 의해 변경이 가능하다.

```
SQL> SELECT DBTIMEZONE FROM DUAL;

DBTIME
------
-07:00
```

SESSIONTIMEZONE 함수는 세션의 표준시간대를 리턴한다.

```
SQL> SELECT SESSIONTIMEZONE FROM DUAL;

SESSIONTIMEZONE
-------------------------------------------
+09:00
```

EXTRACT

EXTRACT는 입력값인 날짜 데이터 타입에서 지정된 값을 추출하는 함수이다. EXTRACT의 문법은 다음과 같다.

```
 SELECT EXTRACT ([YEAR] [MONTH] [DAY] [HOUR] [MINUTE] [SECOND]
                 [TIMEZONE_HOUR] [TIMEZONE_MINUTE]
                 [TIMEZONE_REGION] [TIMEZONE_ABBR]
 FROM [datetime_value_expression]
      [interval_value_expression]);
```

SYSDATE 함수의 결과값으로부터 년도를 추출하는 방법은 다음과 같다.

```
SQL> SELECT EXTRACT (YEAR FROM SYSDATE) FROM DUAL;

EXTRACT(YEARFROMSYSDATE)
------------------------
                    2004
```

사원 테이블에서 10번 부서에 근무하는 직원의 입사월을 출력하면 다음과 같다.

```
SQL> SELECT ENAME, HIREDATE, EXTRACT (MONTH FROM HIREDATE)
  2  FROM EMP
  3  WHERE DEPTNO = 10;

ENAME      HIREDATE EXTRACT(MONTHFROMHIREDATE)
---------- -------- --------------------------
CLARK      81/06/09                          6
KING       81/11/17                         11
MILLER     82/01/23                          1
```

FROM_TZ

FROM_TZ 함수는 TIMESTAMP 타입의 데이터를 TIMESTAMP WITH TIME ZONE 타입의 데이터로 변환해준다. FROM_TZ 함수의 문법은 다음과 같다.

```
 FROM_TZ(TIMESTAMP timestamp_value, time_zone_value)
```

TIMESTAMP 데이터인 '2004-06-27 20:00:00'를 TIMESTAMP WITH TIME ZONE 타입의 데이터로 변환하면 다음과 같다.

```
SQL> SELECT FROM_TZ(TIMESTAMP '2004-06-27 20:00:00', '+5:0') FROM DUAL;

FROM_TZ(TIMESTAMP'2004-06-2720:00:00','+5:0')
---------------------------------------------------------------------------
04/06/27 20:00:00.000000000 +05:00

SQL> SELECT FROM_TZ(TIMESTAMP '2004-06-27 20:00:00', 'Asia/Seoul') FROM DUAL;

FROM_TZ(TIMESTAMP'2004-06-2720:00:00','ASIA/SEOUL')
---------------------------------------------------------------------------
04/06/27 20:00:00.000000000 ASIA/SEOUL
```

TO_TIMESTAMP, TO_TIMESTAMP_TZ

TO_TIMESTAMP와 TO_TIMESTAMP_TZ 함수는 문자열을 각각 TIMESTAMP 타입 데이터와 TIMESTAMP WITH TIME ZONE 타입 데이터로 변환해준다.

문자열 '2004-06-27 20:00:00'를 TIMESTAMP 타입 데이터로 변환하면 다음과 같다.

```
SQL> SELECT TO_TIMESTAMP ('2004-06-27 20:00:00', 'YYYY-MM-DD HH24:MI:SS')
  2  FROM DUAL;

TO_TIMESTAMP('2004-06-2720:00:00','YYYY-MM-DDHH24:MI:SS')
---------------------------------------------------------------------------
04/06/27 20:00:00.000000000
```

문자열 '2004-06-27 20:00:00'를 TIMESTAMP WITH TIME ZONE 타입의 데이터로 변환하면 다음과 같다.

```
SQL> SELECT TO_TIMESTAMP_TZ ('2004-06-27 20:00:00 +9:00',
  2  'YYYY-MM-DD HH24:MI:SS TZH:TZM')
  3  FROM DUAL;

TO_TIMESTAMP_TZ('2004-06-2720:00:00+9:00','YYYY-MM-DDHH24:MI:SSTZH:TZM')
---------------------------------------------------------------------------
04/06/27 20:00:00.000000000 +09:00
```

TO_YMINTERVAL

TO_YMINTERVAL 함수는 문자열을 INTERVAL YEAR TO MONTH 타입 데이터로 변환해준다.

사원 테이블에서 10번 부서에 근무하는 직원의 입사일과 입사일로부터 1년 2개월 후의 날짜를 출력하면 다음과 같다.

```
SQL> SELECT ENAME, HIREDATE, HIREDATE+TO_YMINTERVAL('01-02') HIREDATE2
  2  FROM EMP
  3  WHERE DEPTNO = 10;

ENAME      HIREDATE HIREDATE
---------- -------- --------
CLARK      81/06/09 82/08/09
KING       81/11/17 83/01/17
MILLER     82/01/23 83/03/23
```

복습

1. 'Canada/Yukon'의 표준시간대를 검색하시오.

2. 현재 세션의 표준시간대를 검색하시오.

3. 사원 테이블에서 SMITH의 입사년도, 입사월, 입사일을 다음과 같이 출력하는 SQL 문장을 작성하시오.

```
ENAME      년도       월         일
---------- ---------- ---------- ----------
SMITH      1980       12         17
```

4. 세션의 날짜 표시 형식을 'YYYY-MM-DD HH24:MI:SS'로 변경하시오.

5. 세션의 현재 날짜를 출력하시오. 단, 표준시간대도 표시하시오.

Chapter 17. ROLLUP과 CUBE

부분 집계에 사용되는 GROUP BY 절에 ROLLUP과 CUBE를 사용하면 다양한 통계 자료를 출력 할 수 있다. 예를 들어, 사원 테이블에서 업무별 평균 급여, 부서별 평균 급여, 업무 및 부서별 평균 급여 등을 검색하려면, 각각의 결과를 별도의 쿼리 문장으로 작성해야 하지만 ROLLUP과 CUBE 옵션을 사용하면 한번에 모든 결과를 얻어낼 수 있다. 이번 장에서는 GROUP BY 절의 ROLLUP과 CUBE에 대해서 설명한다.

ROLLUP

ROLLUP은 GROUP BY에 의해서 출력되는 부분집계 결과에 누적된 부분 집계를 추가해준다. ROLLUP을 사용한 GROUP BY의 문법은 다음과 같다.

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

ROLLUP은 뒤에 기술된 컬럼들을 좌측 컬럼에서 우측 컬럼으로 진행하면서 그룹화하고, 각각의 그룹에 대하여 GROUP BY에 의해 부분집계를 수행한다. 예를 들면, 사원 테이블의 부서번호 및 업무별 부분집계에 ROLLUP을 추가하면 결과는 다음과 같다.

```
SQL> SELECT DEPTNO, JOB, SUM(SAL)
  2  FROM EMP
  3  GROUP BY ROLLUP(DEPTNO, JOB);

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        10                 8750        ← 10번 부서의 급여 합계
        20 CLERK            800
        20 ANALYST         3000
        20 MANAGER         2975
        20                 6775        ← 20번 부서의 급여 합계
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600
        30                 9400        ← 30번 부서의 급여 합계
                          24925        ← 전체 부서의 급여 합계

13 개의 행이 선택되었습니다.
```

결과를 살펴보면 부서번호 및 업무별 급여 합계에 부서별 급여 합계와 전체 급여 합계 결과가 추가되었다. 즉, ROLLUP(DEPTNO, JOB)은 (DEPTNO, JOB), (DEPTNO), () 그룹으로 분류되어 각각의 그룹이 GROUP BY에 의해 부분 집계 된다. ROLLUP에 의해 생성되는 그

룹의 수는 ROLLUP에 기술된 컬럼의 개수가 *n*이면 *n*+1개의 그룹이 만들어진다.

```
SELECT DEPTNO, JOB, SUM(SAL)
FROM EMP
GROUP BY DEPTNO, JOB
UNION ALL
SELECT DEPTNO, NULL, SUM(SAL)
FROM EMP
GROUP BY DEPTNO
UNION ALL
SELECT NULL, NULL, SUM(SAL)
FROM EMP
GROUP BY ();
```

CUBE

CUBE도 ROLLUP과 마찬가지로 GROUP BY에 의해서 출력되는 부분집계 결과에 누적된 부분 집계를 추가해준다. CUBE를 사용한 GROUP BY의 문법은 다음과 같다.

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

CUBE는 뒤에 기술된 컬럼들에 대한 모든 조합을 그룹화하고, 각각의 그룹에 대하여 GROUP BY에 의해 부분집계를 수행한다. 예를 들면, 사원 테이블의 부서번호 및 업무별 부분집계에 CUBE를 추가하면 결과는 다음과 같다.

```
SQL> SELECT DEPTNO, JOB, SUM(SAL)
  2  FROM EMP
  3  GROUP BY CUBE(DEPTNO, JOB);

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
                          24925        ← 전체       부서의 급여 합계
           CLERK           3050        ← CLERK      업무의 급여 합계
           ANALYST         3000        ← ANALYST    업무의 급여 합계
           MANAGER         8275        ← MANAGER    업무의 급여 합계
           SALESMAN        5600        ← SALESMAN   업무의 급여 합계
           PRESIDENT       5000        ← PRESIDENT 업무의 급여 합계
        10                 8750        ← 10번       부서의 급여 합계
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        20                 6775        ← 20번       부서의 급여 합계
        20 CLERK            800
        20 ANALYST         3000
        20 MANAGER         2975
        30                 9400        ← 30번       부서의 급여 합계
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600

18 개의 행이 선택되었습니다.
```

결과를 살펴보면 부서번호 및 업무별 급여 합계에 부서별 급여 합계, 업무별 급여 합계와 전체 급여 합계 결과가 추가되었다. 즉, CUBE(DEPTNO, JOB)은 (DEPTNO, JOB), (DEPTNO), (JOB), () 그룹으로 분류되어 각각의 그룹이 GROUP BY에 의해 부분 집계 된다. CUBE에 의해 생성되는 그룹의 수는 CUBE에 기술된 컬럼의 개수가 n이면 2^n개의 그룹이 만들어진다.

```
SELECT DEPTNO, JOB, SUM(SAL)
FROM EMP
GROUP BY DEPTNO, JOB
UNION ALL
SELECT DEPTNO, NULL, SUM(SAL)
FROM EMP
GROUP BY DEPTNO
UNION ALL
SELECT NULL, JOB, SUM(SAL)
FROM EMP
GROUP BY JOB
UNION ALL
SELECT NULL, NULL, SUM(SAL)
FROM EMP;
```

GROUPING 함수

GROUPING 함수는 ROLLUP과 CUBE를 사용했을 때, 각각의 행들이 부분집계에 참여했는지를 표시해준다. GROUPING 함수를 사용하는 SQL 문장의 문법은 다음과 같다.

```
SELECT [column,] group_function(column) ... ,
        GROUPING(expr)
FROM table
[WHERE condition]
[GROUP BY [ROLLUP][CUBE] group_by_expression]
[HAVING having_expression]
[ORDER BY column];
```

GROUPING 함수는 반드시 ROLLUP 또는 CUBE와 함께 사용하여야 하며, GROUPING 함수의 결과를 보면 출력 결과에서 부분집계의 누적집계를 쉽게 찾을 수 있다. 또한, GROUPING 함수를 사용할 때, 테이블에 저장된 NULL과 ROLLUP 또는 CUBE에 의해서 생성된 NULL값을 구별할 수 있어야 한다.

GROUPING 함수는 0 또는 1을 리턴하며 각각의 의미는 다음과 같다.

· 0을 리턴하는 경우
  - 해당 행이 집계 연산에 포함되었음.
  - 컬럼에 표시되는 NULL은 해당 컬럼에 저장된 NULL 값을 의미함.

· 1을 리턴하는 경우
  - 해당 행이 집계 연산에서 제외됨.
  - 컬럼에 표시되는 NULL은 ROLLUP 또는 CUBE가 추가한 값을 의미함.

다음은 GROUPING 함수를 사용하는 방법이다.

```
SQL> SELECT DEPTNO, JOB, SUM(SAL),
  2  GROUPING(DEPTNO),
  3  GROUPING(JOB)
  4  FROM EMP
  5  GROUP BY ROLLUP(DEPTNO, JOB);

    DEPTNO JOB         SUM(SAL) GROUPING(DEPTNO) GROUPING(JOB)
---------- --------- ---------- ---------------- -------------
        10 CLERK           1300                0             0
        10 MANAGER         2450                0             0
        10 PRESIDENT       5000                0             0
        10                 8750                0             1
        20 CLERK            800                0             0
        20 ANALYST         3000                0             0
        20 MANAGER         2975                0             0
        20                 6775                0             1
        30 CLERK            950                0             0
        30 MANAGER         2850                0             0
        30 SALESMAN        5600                0             0
        30                 9400                0             1
                          24925                1             1

13 개의 행이 선택되었습니다.
```

위의 결과를 보면 GROUPING 함수의 결과에 1이 표시된 부분은 부분집계의 누적치이며, 해당 컬럼에는 NULL이 표시된다.

GROUPING SETS

GROUPING SETS는 GROUP BY를 확장한 것으로서 ROLLUP 또는 CUBE가 지정된 원칙에 따라 컬럼들을 그룹화하는 반면 GROUPING SETS을 이용하면 사용자가 원하는 컬럼들로 구성된 그룹을 만들 수 있다. Oracle 서버는 GROUPING SETS에 정의된 컬럼들을 그룹화하고 각 그룹에 대하여 GROUP BY를 수행 한 후, 그 결과를 UNION ALL 연산하게 된다. GROUPING SETS는 테이블을 한번만 검색하므로 GROUPING SETS에 많은 컬럼들을 기술 할수록 처리 효율은 향상된다. GROUPING SETS를 사용하는 방법은 다음과 같다.

```
SQL> SELECT DEPTNO, JOB, MGR, SUM(SAL)
  2  FROM EMP
  3  GROUP BY GROUPING SETS
  4  ((DEPTNO, JOB, MGR),
  5  (DEPTNO, MGR), (JOB, MGR));

    DEPTNO JOB              MGR   SUM(SAL)
---------- --------- ---------- ----------
        10 PRESIDENT                  5000
        10 CLERK            7782      1300
        10 MANAGER          7839      2450
        20 ANALYST          7566      3000
        20 MANAGER          7839      2975
        20 CLERK            7902       800
        30 CLERK            7698       950
        30 SALESMAN         7698      5600
        30 MANAGER          7839      2850
        10                            5000
        10                  7782      1300
        10                  7839      2450
        20                  7566      3000
        20                  7839      2975
        20                  7902       800
        30                  7698      6550
        30                  7839      2850
           CLERK            7698       950
           CLERK            7782      1300
           CLERK            7902       800
           ANALYST          7566      3000
           MANAGER          7839      8275
           SALESMAN         7698      5600
           PRESIDENT                  5000

24 개의 행이 선택되었습니다.
```

위 문장은 다음 문장과 동일하다.

```
SELECT DEPTNO, JOB, MGR, SUM(SAL)
FROM EMP
GROUP BY DEPTNO, JOB, MGR
UNION ALL
SELECT DEPTNO, NULL, MGR, SUM(SAL)
FROM EMP
GROUP BY DEPTNO, MGR
UNION ALL
SELECT NULL, JOB, MGR, SUM(SAL)
FROM EMP
GROUP BY JOB, MGR;
```

그러면, 다음과 같은 CUBE 문장과 ROLLUP 문장을 GROUPING SETS로 변환해보자.

CUBE(a, b, c)	GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())
ROLLUP(a, b, c)	GROUPING SETS ((a, b, c), (a, b), (a), ())

복합 컬럼의 처리

복합 컬럼이란 ROLLUP, CUBE에 기술된 일부 컬럼이 ()에 의해 하나의 단위로 처리되는 컬럼이다. 예를 들면, ROLLUP (a, (b, c), d)에서 (b, c)는 하나의 컬럼으로 처리되기 때문에 GROUP BY에 의해 처리되는 그룹은 (a, b, c, d), (a, b, c), (a), ()이 된다. 복합 컬럼을 사용한 예는 다음과 같다.

```
SQL> SELECT DEPTNO, JOB, MGR, SUM(SAL)
  2  FROM EMP
  3  GROUP BY ROLLUP (DEPTNO, (JOB, MGR));

    DEPTNO JOB              MGR   SUM(SAL)
---------- --------- ---------- ----------
        10 CLERK            7782       1300
        10 MANAGER          7839       2450
        10 PRESIDENT                   5000
        10                             8750
        20 CLERK            7902        800
        20 ANALYST          7566       3000
        20 MANAGER          7839       2975
        20                             6775
        30 CLERK            7698        950
        30 MANAGER          7839       2850
        30 SALESMAN         7698       5600
        30                             9400
                                      24925

13 개의 행이 선택되었습니다.
```

위 문장은 아래 문장과 동일하다.

```
SELECT DEPTNO, JOB, MGR, SUM(SAL)
FROM EMP
GROUP BY DEPTNO, JOB, MGR
UNION ALL
SELECT DEPTNO, NULL, NULL, SUM(SAL)
FROM EMP
GROUP BY DEPTNO
UNION ALL
SELECT NULL, NULL, NULL, SUM(SAL)
FROM EMP
GROUP BY ();
```

그러면, 이와 같은 방식으로 다음과 같은 경우는 어떤 그룹들이 생성 될 것인지를 생각해보자.

GROUPING SETS	GROUP BY
GROUP BY GROUPING SETS (a, b, c)	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY c
GROUP BY GROUPING SETS (a, b, (b, c))	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY b, c
GROUP BY GROUPING SETS ((a, b, c))	GROUP BY a, b, c
GROUP BY GROUPING SETS (a, (b), ())	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY ()
GROUP BY GROUPING SETS (a, ROLLUP (b, c))	GROUP BY a UNION ALL GROUP BY ROLLUP (b, c)

연결된 그룹의 처리

연결된 그룹이란 GROUP BY 뒤에 ROLLUP, CUBE, GROUPING SETS가 콤마로 분리되어 같이 기술된 것이다. 이런 경우에 각각의 그룹 집단이 다른 그룹 집단과 서로 조합되어 전개된다. 예를 들어, GROUP BY GROUPING SETS (a, b), GROUPING SETS (c, d)에 의해 만들어지는 그룹은 (a, c), (a, d), (b, c), (b, d)가 된다. 결합된 그룹을 사용하는 방법은 다음과 같다.

```
SQL> SELECT DEPTNO, JOB, MGR, SUM(SAL)
  2  FROM EMP
  3  GROUP BY DEPTNO, ROLLUP(JOB), CUBE(MGR);

    DEPTNO JOB              MGR   SUM(SAL)
---------- --------- ---------- ----------
        10 PRESIDENT                  5000
        10 CLERK            7782      1300
        10 MANAGER          7839      2450
        20 ANALYST          7566      3000
        20 MANAGER          7839      2975
        20 CLERK            7902       800
        30 CLERK            7698       950
        30 SALESMAN         7698      5600
        30 MANAGER          7839      2850
        10                            5000
        10                  7782      1300
        10                  7839      2450
        20                  7566      3000
        20                  7839      2975
        20                  7902       800
        30                  7698      6550
        30                  7839      2850
        10 CLERK                      1300
        10 MANAGER                    2450
        10 PRESIDENT                  5000
        10                            8750
        20 CLERK                       800
        20 ANALYST                    3000
        20 MANAGER                    2975
        20                            6775
        30 CLERK                       950
        30 MANAGER                    2850
        30 SALESMAN                   5600
        30                            9400

29 개의 행이 선택되었습니다.
```

위 문장은 아래 문장과 동일하다.

```
SELECT DEPTNO, JOB, MGR, SUM(SAL)
FROM EMP
GROUP BY DEPTNO, JOB, MGR
UNION ALL
SELECT DEPTNO, JOB, NULL, SUM(SAL)
FROM EMP
GROUP BY DEPTNO, JOB
UNION ALL
SELECT DEPTNO, NULL, MGR, SUM(SAL)
FROM EMP
GROUP BY DEPTNO, MGR
UNION ALL
SELECT DEPTNO, NULL, NULL, SUM(SAL)
FROM EMP
GROUP BY DEPTNO;
```

복습

1. ROLLUP (a, b)에 의해서 만들어지는 그룹을 기술하시오.

2. CUBE (a, b)에 의해서 만들어지는 그룹을 기술하시오.

3. ROLLUP (a, (b, c, d), e)에 의해서 만들어지는 그룹을 기술하시오.

4. CUBE ((a, b), (c, d), e)에 의해서 만들어지는 그룹을 기술하시오.

5. 사원 테이블에서 업무 및 부서코드별 평균 급여, 업무별 평균 급여, 전체 평균 급여를 출력하는 SQL 문장을 작성하시오.

```
JOB           DEPTNO   AVG(SAL)
--------- ---------- ----------
CLERK             10       1300
CLERK             20        800
CLERK             30        950
CLERK                1016.66667
ANALYST           20       3000
ANALYST                    3000
MANAGER           10       2450
MANAGER           20       2975
MANAGER           30       2850
MANAGER              2758.33333
SALESMAN          30       1400
SALESMAN                   1400
PRESIDENT         10       5000
PRESIDENT                  5000
                     2077.08333

15 개의 행이 선택되었습니다.
```

Chapter 18. 서브쿼리 II

이번 장에서는 앞서 설명한 기본적인 서브쿼리의 사용방법에 이어 서브쿼리의 고급 활용 기법인 복수 컬럼 서브쿼리, 스칼라 서브쿼리, 상관관계 서브쿼리와 Oracle 9i부터 추가된 WITH 구문에 대하여 설명한다.

복수 컬럼 서브쿼리

복수 컬럼 서브쿼리란 서브쿼리에서 리턴되는 행들이 두개 이상의 컬럼을 가지는 경우이다. 즉, 다음과 같은 형식의 서브쿼리를 복수 컬럼 서브쿼리라고 한다.

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

복수 컬럼 서브쿼리는 비교조건의 형태의 따라 Pairwise 비교와 Nonpairwise 비교로 구분된다. Pairwise 비교조건은 다음과 같이 사원 테이블에서 사번이 7369 또는 7499번인 직원과 직급 및 부서코드가 동일한 사원을 검색하는 경우이다.

```
SQL> SELECT EMPNO, JOB, DEPTNO
  2  FROM EMP
  3  WHERE (JOB, DEPTNO) IN
  4                       (SELECT JOB, DEPTNO
  5                        FROM EMP
  6                        WHERE EMPNO IN (7369, 7499))
  7  AND EMPNO NOT IN (7369, 7499);

     EMPNO JOB           DEPTNO
---------- --------- ----------
      7876 CLERK             20
      7521 SALESMAN          30
      7654 SALESMAN          30
      7844 SALESMAN          30
```

반면, Nonpairwise 비교조건은 다음과 같이 사원 테이블에서 사번이 7369 또는 7499번인 직원과 부서코드가 같거나 직급이 같은 사원을 검색하는 경우이다.

```
SQL> SELECT EMPNO, JOB, DEPTNO
  2  FROM EMP
  3  WHERE JOB IN
  4             (SELECT JOB
  5              FROM EMP
  6              WHERE EMPNO IN (7369, 7499))
  7  AND DEPTNO IN
  8             (SELECT DEPTNO
  9              FROM EMP
 10              WHERE EMPNO IN (7369, 7499))
 11  AND EMPNO NOT IN (7369, 7499);

     EMPNO JOB           DEPTNO
---------- --------- ----------
      7876 CLERK             20
      7900 CLERK             30
      7521 SALESMAN          30
      7654 SALESMAN          30
      7844 SALESMAN          30
```

스칼라 서브쿼리

서브쿼리가 1개의 행에 1개의 컬럼값을 리턴하는 경우를 스칼라 서브쿼리라고 한다. 스칼라 서브쿼리는 Oracle 8i부터 지원되었지만 SELECT 문장과 INSERT 문장의 VALUES 구문에서와 같이 제한된 경우에서만 사용 할 수 있었다. 그러나 Oracle 9i부터는 스칼라 서브쿼리의 적용 범위가 훨씬 확대 되었으며, 다음과 같은 경우에도 사용 할 수 있게 되었다.

· DECODE와 CASE 함수내의 조건식 또는 연산식
· GROUP BY를 제외한 SELECT 문장의 모든 구문
· UPDATE 문장에서 SET 및 WHERE 구문에서 연산자의 좌변

반면, 다음과 같은 경우에는 스칼라 서브쿼리를 사용 할 수 없다.
· 컬럼에 대한 디폴트 값
· DML 명령의 RETURNING 구문
· 함수기반 인덱스
· GROUP BY 절, CHECK 제약조건, WHEN 조건
· HAVING 절
· START WITH, CONNECT BY 절
· CREATE PROFILE과 같이 쿼리와 관련 없는 문장

스칼라 서브쿼리를 CASE 함수에 사용하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME,
  2  (CASE
  3   WHEN DEPTNO = (SELECT DEPTNO FROM DEPT
  4                  WHERE LOC = 'CHICAGO')
  5   THEN '뉴욕' ELSE '기타' END) LOCATION
  6  FROM EMP;

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

14 개의 행이 선택되었습니다.
```

스칼라 서브쿼리를 ORDER BY 구문에 사용하면 다음과 같다.

```
SQL> SELECT EMPNO, ENAME
  2  FROM EMP E
  3  ORDER BY (SELECT DNAME
  4            FROM DEPT D
  5            WHERE E.DEPTNO = D.DEPTNO);

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

14 개의 행이 선택되었습니다.
```

상관관계 서브쿼리

상관관계 서브쿼리는 서브쿼리가 메인쿼리의 컬럼을 참조하도록 되어 있는 서브쿼리로서 단

독으로 실행이 불가능하기 때문에 메인쿼리에서 조건에 맞는 행을 차례로 서브쿼리에 넘겨주면 서브쿼리에서 참조하는 메인쿼리의 컬럼이 상수로 교체되어 서브쿼리가 실행되고 그 결과가 메인쿼리에 전달되어 처리된다. 즉, 상관관계 서브쿼리는 메인쿼리가 넘겨주는 행의 개수 만큼 반복 실행되게 된다.

상관관계 서브쿼리의 작성 문법은 다음과 같다.

```
SELECT column1, column2, ...
FROM table1 outer
WHERE column1 operator
              (SELECT column1, column2
               FROM table2
               WHERE expr1 = outer.expr2);
```

다음은 사원 테이블에서 자기 부서의 평균 급여보다 많은 급여를 받는 사원을 검색하기 위해 상관관계 서브쿼리를 사용한 것이다.

```
SQL> SELECT ENAME, SAL
  2  FROM EMP E
  3  WHERE SAL > (SELECT AVG(SAL)
  4               FROM EMP
  5               WHERE E.DEPTNO = DEPTNO);

ENAME             SAL
---------- ----------
ALLEN            1600
JONES            2975
BLAKE            2850
KING             5000
FORD             3000
SCOTT            3000

6 개의 행이 선택되었습니다.
```

위의 쿼리 문장에서는 메인쿼리의 첫 번째 행이 서브쿼리로 전달되어 E.DEPTNO가 상수로 교체되면 서브쿼리가 실행된 후, 그 결과가 다시 메인쿼리로 전달되어 처리된다.

EXISTS

EXISTS는 서브쿼리의 결과가 한 개의 행이라도 존재하면 조건은 참이 되며 반대로 한 개의 행도 존재하지 않으면 조건은 거짓이 된다. 또한, EXISTS는 서브쿼리에서 첫 번째 행이 검색되면 두 번째 행을 검색하지 않고 바로 메인 쿼리에 참을 리턴하기 때문에 검색 속도도 상당히 빠른 편이다. 다음은 사원 테이블에서 부하 직원을 한 사람이상 데리고 있는 사원을 검색하는 방법이다.

```
SQL> SELECT EMPNO, ENAME
  2      FROM EMP E
  3      WHERE EXISTS (SELECT 'X'
  4                    FROM EMP
  5                    WHERE E.EMPNO = MGR);

     EMPNO ENAME
---------- ----------
      7566 JONES
      7698 BLAKE
      7782 CLARK
      7839 KING
      7902 FORD
      7788 SCOTT

6 개의 행이 선택되었습니다.
```

위 문장은 IN을 사용하여 일반 서브쿼리 문장으로 변환이 가능하다.

```
SELECT EMPNO, ENAME
FROM EMP
WHERE EMPNO IN (SELECT MGR
                FROM EMP
                WHERE MGR IS NOT NULL);
```

부서 테이블에서 사원들이 배정되지 않은 부서명을 검색하려면 NOT EXISTS를 사용하면 된다.

```
SQL> SELECT DEPTNO, DNAME
  2   FROM DEPT D
  3   WHERE NOT EXISTS (SELECT 'X'
  4                     FROM EMP
  5                     WHERE DEPTNO = D.DEPTNO);

    DEPTNO DNAME
---------- --------------
        40 OPERATIONS
```

마찬가지로, 위의 문장을 IN으로 표현하면 다음과 같다.

```
SELECT DEPTNO, DNAME
FROM DEPT
WHERE DEPTNO NOT IN (SELECT DEPTNO
                     FROM EMP);
```

상관관계 UPDATE

UPDATE 명령에도 상관관계 서브쿼리를 사용하여 다른 테이블의 행을 기반으로 테이블의 행을 변경 할 수 있다. 상관관계 UPDATE 명령의 문법은 다음과 같다.

```
UPDATE table1 alias1
SET column = (SELECT expression
              FROM table2 alias2
              WHERE alias1.column = alias2.column);
```

상관관계 UPDATE 명령을 연습하기 위해 다음과 같이 테이블에 DNAME 컬럼을 추가한다.

```
ALTER TABLE EMP
ADD DNAME VARCHAR2(14);
```

부서 테이블을 기반으로 사원 테이블의 DNAME 컬럼에 부서명을 입력하는 방법은 다음과 같다.

```
SQL> UPDATE EMP E
  2  SET E.DNAME = (SELECT DNAME
  3                 FROM DEPT
  4                 WHERE DEPTNO = E.DEPTNO);

14 행이 갱신되었습니다.
```

상관관계 DELETE

DELETE 명령에도 상관관계 서브쿼리를 사용하여 다른 테이블의 행을 기반으로 테이블의 행을 삭제 할 수 있다. 상관관계 DELETE 명령의 문법은 다음과 같다.

```
DELETE FROM table1 alias1
WHERE column operator
             (SELECT expression
              FROM table2 alias2
              WHERE alias1.column = alias2.column);
```

상관관계 DELETE 명령을 연습하기 위해 다음과 같이 테이블을 추가한다. 추가되는 테이블에는 사원 테이블에서 삭제할 직원의 사번이 입력되어 있다고 가정한다.

```
CREATE TABLE DEL_LIST
(EMPNO NUMBER(4));

INSERT INTO DEL_LIST VALUES(7934);
INSERT INTO DEL_LIST VALUES(7788);
```

사원 테이블에서 DEL_LIST 테이블에 입력되어 있는 사원들을 삭제하는 방법은 다음과 같다.

```
SQL> DELETE FROM EMP E
  2  WHERE E.EMPNO = (SELECT EMPNO
  3                   FROM DEL_LIST
  4                   WHERE EMPNO = E.EMPNO);

2 행이 삭제되었습니다.
```

WITH

WITH는 쿼리 문장안에 반복해서 포함되는 서브쿼리를 블럭으로 선언하여 문장의 다양한 위치에서 활용하는 방법이다. WITH 구문에 기술되어 있는 서브쿼리의 실행 결과는 사용자의 임시 테이블스페이스에 저장되며 쿼리의 수행속도를 향상시켜 준다.

다음은 전체 부서별 총 급여의 평균보다 높은 부서의 총 급여를 검색하는 방법이다.

```
SQL> WITH
  2  DEPT_SAL AS (
  3       SELECT D.DNAME, SUM(E.SAL) AS SAL_SUM
  4       FROM DEPT D, EMP E
  5       WHERE D.DEPTNO = E.DEPTNO
  6       GROUP BY D.DNAME),
  7  DEPT_AVG AS (
  8       SELECT AVG(SAL_SUM) AS SAL_AVG
  9       FROM DEPT_SAL)
 10  SELECT *
 11  FROM DEPT_SAL
 12  WHERE SAL_SUM > (SELECT SAL_AVG
 13                   FROM DEPT_AVG)
 14  ORDER BY DNAME;

DNAME          SAL_SUM
-------------- ----------
RESEARCH            10875
```

WITH에 기술된 첫 번째 서브쿼리는 부서별 총 급여를 산출하여 DEPT_SAL이라는 별칭이 부여되고, 두 번째 서브쿼리는 첫 번째 서브쿼리에서 생성된 DEPT_SAL을 이용하여 부서별 총 급여의 평균을 산출하여 DEPT_AVG라는 별칭이 부여되었다. 이 후, 두 개의 서브쿼리 결과를 이용하여 최종 결과를 산출하게 된다.

복습

1. 다음 JOIN 문장을 상관관계 서브쿼리로 변환하시오.

```
SELECT E.ENAME
FROM DEPT D, EMP E
WHERE D.DEPTNO = E.DEPTNO;
```

2. 사원 테이블에서 같은 업무를 수행하는 직원들 중, 해당 업무별 평균 급여보다 많은 급여를 받는 직원의 사원명, 업무, 급여를 검색하시오.(상관관계 서브쿼리로 작성할 것)

3. 사원 테이블에서 같은 부서에 근무하는 직원들 중, 해당 부서별 최대 급여를 받는 직원의 사원명, 부서번호, 급여를 검색하시오.(상관관계 서브쿼리로 작성할 것)

[4-6] 다음과 같은 MASTER 테이블과 TEMP 테이블을 작성하시오

```
CREATE TABLE MASTER
(ID NUMBER(3),
 NAME VARCHAR2(10));

INSERT INTO MASTER VALUES (1, 'NORWAY');
INSERT INTO MASTER VALUES (2, 'SPAIN');
INSERT INTO MASTER VALUES (3, 'SWEDEN');
INSERT INTO MASTER VALUES (4, 'ASIA');
INSERT INTO MASTER VALUES (5, 'CHINA');
INSERT INTO MASTER VALUES (6, 'INDIA');

CREATE TABLE TEMP
(ID NUMBER(3),
 NAME VARCHAR2(10));

INSERT INTO TEMP VALUES (1, 'DENMARK');
INSERT INTO TEMP VALUES (2,  NULL);
INSERT INTO TEMP VALUES (3, 'FRANCE');
INSERT INTO TEMP VALUES (4,  NULL);
INSERT INTO TEMP VALUES (5, 'GERMANY');
INSERT INTO TEMP VALUES (7, 'BRAZIL');
INSERT INTO TEMP VALUES (8, 'CANADA');
```

MASTER 테이블에는 데이터의 원본이며, TEMP 테이블은 MASTER 테이블에 변경할 데이터가 저장되는 테이블이다. TEMP 테이블을 기반으로 MASTER 테이블을 변경하는 규칙은 다음과 같다.

- MASTER.ID = TEMP.ID 이면 MASTER 테이블에서 해당 행을 변경
- MASTER.ID에 TEMP.ID가 없는 경우는 MASTER 테이블에서 해당 행을 추가
- TEMP.NAME이 NULL인 경우는 MASTER 테이블에서 해당 행을 삭제

4. TEMP 테이블을 기반으로 MASTER 테이블의 해당 행을 변경하시오. 변경 후의 MASTER 테이블은 다음과 같아야 한다.

```
        ID NAME
---------- ----------
         1 DENMARK
         2 SPAIN
         3 FRANCE
         4 ASIA
         5 GERMANY
         6 INDIA
```

5. TEMP 테이블을 기반으로 MASTER 테이블의 해당 행을 추가하시오. 변경 후의 MASTER 테이블은 다음과 같아야 한다.

```
        ID NAME
---------- ----------
         1 DENMARK
         2 SPAIN
         3 FRANCE
         4 ASIA
         5 GERMANY
         6 INDIA
         7 BRAZIL
         8 CANADA
```

6. TEMP 테이블을 기반으로 MASTER 테이블의 해당 행을 삭제하시오. 변경 후의 MASTER 테이블은 다음과 같아야 한다.

```
        ID NAME
---------- ----------
         1 DENMARK
         3 FRANCE
         5 GERMANY
         6 INDIA
         7 BRAZIL
         8 CANADA
```

Chapter 19. 계층형 쿼리

이번 장에서는 계층형 쿼리에 대한 개념과 계층형 쿼리를 이용하여 트리 구조의 보고서를 작성하는 방법을 알아본다. 또한, 트리 구조에서 특정 노드(Node) 또는 브랜치(Branch)를 제거하고 검색하는 방법을 설명한다.

트리 구조

트리 구조를 설명하기에 앞서 사원 테이블의 데이터를 확인해보자.

```
SQL> SELECT * FROM EMP;

 EMPNO ENAME    JOB           MGR HIREDATE    SAL    COMM DEPTNO
------ -------- ---------- ------ -------- ------ ------ ------
  7369 SMITH    CLERK        7902 80/12/17    800             20
  7499 ALLEN    SALESMAN     7698 81/02/20   1600    300      30
  7521 WARD     SALESMAN     7698 81/02/22   1250    500      30
  7566 JONES    MANAGER      7839 81/04/02   2975             20
  7654 MARTIN   SALESMAN     7698 81/09/28   1250   1400      30
  7698 BLAKE    MANAGER      7839 81/05/01   2850             30
  7782 CLARK    MANAGER      7839 81/06/09   2450             10
  7788 SCOTT    ANALYST      7566 87/04/19   3000             20
  7839 KING     PRESIDENT         81/11/17   5000             10
  7844 TURNER   SALESMAN     7698 81/09/08   1500      0      30
  7876 ADAMS    CLERK        7788 87/05/23   1100             20
  7900 JAMES    CLERK        7698 81/12/03    950             30
  7902 FORD     ANALYST      7566 81/12/03   3000             20
  7934 MILLER   CLERK        7782 82/01/23   1300             10
```

앞선 내용에서 사원 테이블의 MGR 컬럼은 관리자사번이라고 설명하였다. 이 관리자사번은 사원들의 상하 수직관계를 데이터베이스에 저장하기 위해 추가한 컬럼인데 의미하는 바는 다음과 같다. 즉, 사번이 7369인 SMITH의 관리자는 사번이 7902번이며, 7902번 FORD의 관리자는 사번이 7566번이고, 7566번 JONES의 관리자 사번이 7839인 KING을 저장한 것이다. 이러한 관계를 우리가 평소에 많이 볼 수 있는 트리 구조로 표현하면 다음과 같다.

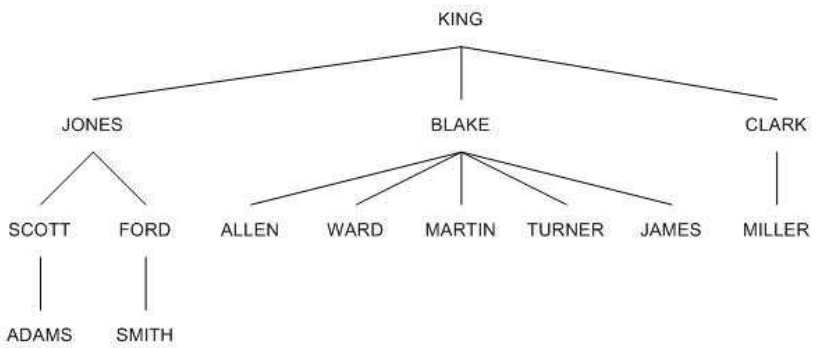


그림 19-1. 트리 구조

계층형 쿼리의 작성 방법

위 그림과 같은 트리 구조의 데이터를 검색하는 방법이 계층형 쿼리이며, 문법은 다음과 같다.

```
SELECT [LEVEL], column, expr ...
FROM table
[WHERE condition(s)]
[START WITH condition(s)]
[CONNECT BY PRIOR condition(s)];
```

여기서, LEVEL은 계층형 쿼리에서 출력되는 가상 컬럼으로 1은 루트, 2는 루트의 자식, 3은 루트의 자식의 자식을 표시한다. START WITH는 트리 구조의 루트에 위치할 행을 지정하는 것으로 계층형 질의가 시작되는 행을 의미한다. CONNECT BY에는 트리 구조에서 부모 행과 자식 행 사이의 관계를 조건으로 기술하는데 PRIOR 뒤에 기술된 컬럼은 계층형 질의를 진행하면서 바로 전에 읽은 행을 의미한다.

먼저, 계층형 쿼리를 작성하려면 트리 구조의 최상단, 즉 루트에 위치할 행을 검색 할 수 있는 조건을 START WITH에 기술한다.

```
START WITH column1 = value
```

예를 들어, 사원 테이블에서 루트에 위치할 사원의 이름이 'KING'이라면 다음과 같이 기술한다.

```
START WITH ENAME = 'KING'
```

다음은 계층형 쿼리의 진행 방향을 기술하는데 트리 구조에서 START WITH로 지정한 행부터 하향식(Top-down)으로 전개하려면 부모 컬럼을 *column1*, 자식 컬럼을 *column2*를 기술하고, 상향식(Bottom-up)으로 전개하려면 자식 컬럼을 *column1*, 부모 컬럼을 *column2*에 기술한다.

```
CONNECT BY PRIOR column1 = column2
```

예를 들어, 사원 테이블에서 트리 구조의 루트에 위치한 'KING'으로부터 하향식 전개하려면 다음과 같이 기술해야 한다.

```
CONNECT BY PRIOR EMPNO = MGR
```

사원 테이블에서 사원명이 'KING'인 직원부터 하향식 전개하는 방법은 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, MGR
  2  FROM EMP
  3  START WITH ENAME = 'KING'
  4  CONNECT BY PRIOR EMPNO = MGR;

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

반대로 사원 테이블에서 사원명이 'SMITH'인 직원부터 상향식 전개하는 방법은 다음과 같다.

```
SQL> SELECT EMPNO,ENAME, MGR
  2  FROM EMP
  3  START WITH ENAME = 'SMITH'
  4  CONNECT BY PRIOR MGR = EMPNO;

 EMPNO ENAME      MGR
------ ------- ------
  7369 SMITH     7902
  7902 FORD      7566
  7566 JONES     7839
  7839 KING
```

LEVEL

계층형 쿼리에서 사용 할 수 있는 가상 컬럼 LEVEL은 트리 구조에서 계층을 나타낸다. 즉, LEVEL 1은 루트, 2는 루트의 자식, 3은 루트의 자식의 자식을 의미한다. 예를 들어, 앞 그림에서 'KING'은 LEVEL=1, 'JOHN'은 LEVEL=2, 'FORD'는 LEVEL=3, 'SMITH'는 LEVEL=4이다.

계층형 쿼리에서 가상 컬럼인 LEVEL과 LPAD 함수를 사용하면 좀 더 보기 쉬운 보고서를 작성 할 수 있다. 즉, LEVEL 값에 따라 LPAD 함수를 이용하여 출력될 행의 앞에 들여쓰기를 하면 계층형 쿼리 결과를 알아보기 쉬운 형태로 만들 수 있다.

```
SQL> SELECT LPAD(ENAME, LENGTH(ENAME)+(LEVEL*2)-2, ' ') NAME
  2  FROM EMP
  3  START WITH ENAME = 'KING'
  4  CONNECT BY PRIOR EMPNO = MGR;

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

트리 구조의 노드(Node) 및 브랜치(Branch) 제거

'KING'으로부터 하향식 전개된 계층형 쿼리 결과에서 'FORD'를 제외하고 출력하려면 WHERE 절에서 제거할 노드를 제외 할 수 있는 조건을 기술해주면 된다.

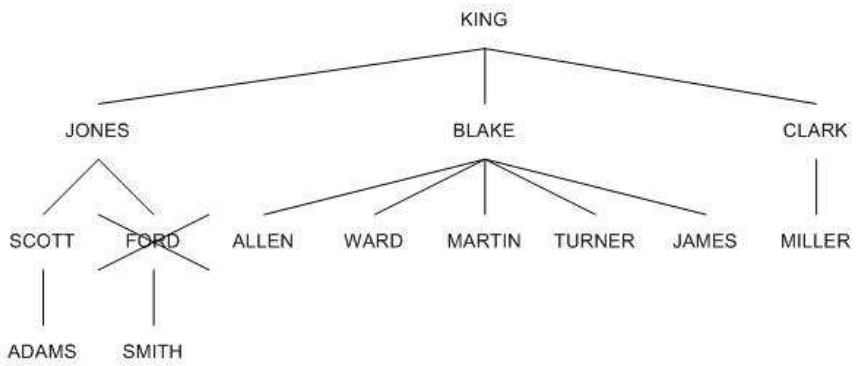


그림 19-2. 노드 제거

```
SQL> SELECT LPAD(ENAME, LENGTH(ENAME)+(LEVEL*2)-2, ' ') NAME
  2  FROM EMP
  3  WHERE ENAME != 'FORD'
  4  START WITH ENAME = 'KING'
  5  CONNECT BY PRIOR EMPNO = MGR;

NAME
--------------------------------------------------------------------
KING
  JONES
    SCOTT
      ADAMS
      SMITH
  BLAKE
    ALLEN
    WARD
    MARTIN
    TURNER
    JAMES
  CLARK
    MILLER
```

또한, 아래 그림과 같이 트리 구조에서 'FORD'의 브랜치를 제거하고 계층형 쿼리를 진행하고자 하는 경우에는 CONNECT BY 절 뒤에 제거할 브랜치의 최고 관리자를 제외 할 수 있는 조건을 기술하면 된다.

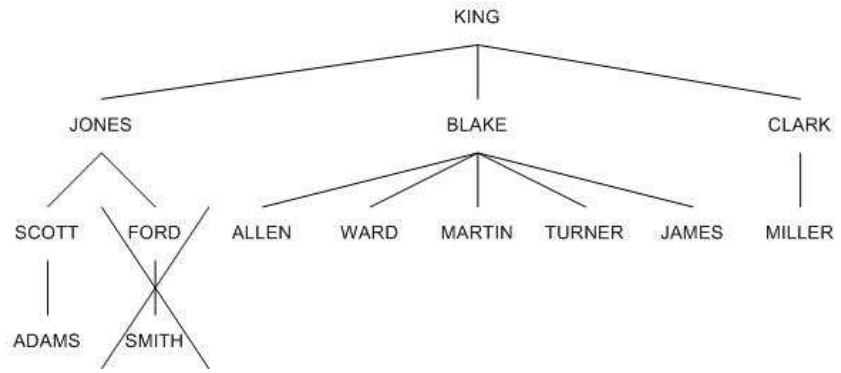


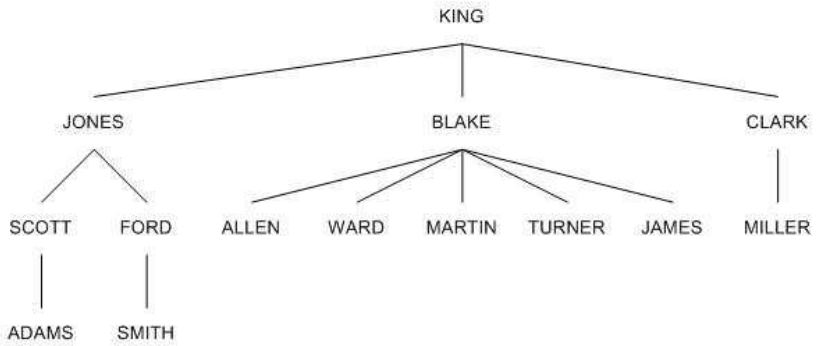
그림 19-3. 브랜치 제거

```
SQL> SELECT LPAD(ENAME, LENGTH(ENAME)+(LEVEL*2)-2, ' ') NAME
  2  FROM EMP
  3  START WITH ENAME = 'KING'
  4  CONNECT BY PRIOR EMPNO = MGR
  5  AND ENAME != 'FORD';

NAME
--------------------------------------------------------------------
KING
  JONES
    SCOTT
      ADAMS
  BLAKE
    ALLEN
    WARD
    MARTIN
    TURNER
    JAMES
  CLARK
    MILLER
```

복습

사원 테이블과 아래 트리 구조를 참고하여 적절한 쿼리 문장을 작성하시오.



1. 계층형 쿼리를 이용하여 'JONES'의 부하직원을 모두 출력하시오.

2. 계층형 쿼리를 이용하여 'ADAMS'의 상관을 모두 출력하시오.

3. 계층형 쿼리를 이용하여 'KING'의 부하직원을 출력하시오. 단, 'BLAKE'와 'BLAKE'의 부하직원들은 제외 할 것.

Chapter 20. Oracle 9i에서 향상된 DML과 DDL

이번 장에서는 Oracle 9i부터 추가된 다중 테이블 INSERT와 EXTERNAL TABLE에 대하여 설명한다. 다중 테이블 INSERT는 INSERT INTO ... SELECT문을 향상시킨 것으로 이전의 INSERT 문장은 한 번에 한 개의 테이블에 행을 입력 할 수 있었지만 다중 테이블 INSERT는 동시에 여러 개의 테이블에 행을 입력 할 수 있다. EXTERNAL TABLE은 데이터베이스 외부의 텍스트 파일을 테이블처럼 선언하여 검색하는 방법이다.

다중 테이블 INSERT

다중 테이블 INSERT 문장은 서브쿼리의 결과를 1개 이상의 테이블에 입력 할 수 있기 때문에 기존의 데이터베이스에 구축된 많은 데이터를 데이터웨어 하우스로 구축하는 업무에 매우 유용하다. 다중 테이블 INSERT 문장의 종류는 다음과 같다.

- 무조건 INSERT
- 조건부 INSERT ALL
- 조건부 INSERT FIRST
- 피벗팅(Pivoting) INSERT

다중 테이블 INSERT 문장의 문법은 다음과 같다.

```
INSERT [ALL] [conditional_insert_clause]
[insert_into_clause values_clause] (subquery)
```

여기서, *conditional_insert_clause*는 다음과 같다.

```
[ALL] [FIRST]
[WHEN condition THEN] [insert_into_clause values_clause]
[ELSE] [insert_into_clause values_clause]
```

무조건 INSERT

무조건 INSERT는 INSERT 뒤에 ALL을 기술하고 *insert_into_clause* 절에 서브쿼리에서 리턴된 행들을 입력할 테이블과 컬럼명을 필요한 만큼 기술하면 된다. Oracle 서버는 서브쿼리에서 리턴된 행들을 INTO 문장에 기술된 모든 테이블에 각각 입력한다.

무조건 INSERT 문장을 실행하기에 앞서 다음과 같은 임시테이블을 생성한다.

```
SQL> CREATE TABLE T1
  2  AS SELECT EMPNO, ENAME, HIREDATE
  3  FROM EMP
  4  WHERE 0 = 1;

테이블이 생성되었습니다.

SQL> CREATE TABLE T2
  2  AS SELECT EMPNO, ENAME, SAL
  3  FROM EMP
  4  WHERE 0 = 1;

테이블이 생성되었습니다.
```

사원 테이블에서 10번 부서에 근무하는 사원의 사번, 사원명, 입사일은 T1 테이블, 사번, 사원명, 급여는 T2 테이블에 입력하는 방법은 다음과 같다.

```
SQL> INSERT ALL
  2    INTO T1 VALUES (EMPNO, ENAME, HIREDATE)
  3    INTO T2 VALUES (EMPNO, ENAME, SAL)
  4    SELECT EMPNO, ENAME, HIREDATE, SAL
  5    FROM EMP
  6    WHERE DEPTNO = 10;
```

조건부 INSERT ALL

조건부 INSERT ALL은 *conditional_insert_clause*에 ALL을 기술하고 *insert_into_clause*에는 WHEN *condition*이 만족하는 경우에 입력 될 테이블과 컬럼명을 작성한다. Oracle 서버는 서브쿼리에서 리턴된 행들을 WHEN 조건과 비교하여 만족하는 경우 해당 테이블에 입력한다. 만약, INSERT ALL 문장에 ELSE 조건이 포함되어 있으면 어떤 조건에도 만족하지 않는 행들을 ELSE 조건에 기술된 테이블에 저장한다.

사원 테이블에서 입사일이 1981년 이후인 직원의 사번, 사원명, 입사일을 T2, 급여가 3000이 넘는 직원의 사번, 사원명, 급여를 T2 테이블에 입력하는 방법은 다음과 같다.

```
SQL> INSERT ALL
  2    WHEN HIREDATE >= '81/01/01' THEN
  3      INTO T1 VALUES (EMPNO, ENAME, HIREDATE)
  4    WHEN SAL > 3000 THEN
  5      INTO T2 VALUES (EMPNO, ENAME, SAL)
  6    SELECT EMPNO, ENAME, HIREDATE, SAL
  7    FROM EMP;
```

위의 조건부 INSERT ALL에서 주의할 사항은 입사일이 1981년 이후이고 급여가 3000이 초과하는 사원의 데이터는 T1과 T2 테이블에 모두 저장된다는 사실이다.

조건부 INSERT FIRST

조건부 INSERT FIRST는 조건부 INSERT ALL과 사용방법이 유사하다. 조건부 INSERT FIRST는 서브쿼리에서 리턴된 행을 INSERT FIRST 문장에서 기술된 WHEN 조건과 차례로 비교하면서 조건에 만족하면 해당 테이블에 입력하고 INSERT FIRST 문장을 종료한다. 만약, INSERT FIRST 문장에 ELSE 조건이 포함되어 있으면 어떤 조건에도 만족하지 않는 행들은 ELSE 조건에 기술된 INSERT 문장에 의해 해당 테이블에 저장된다.

조건부 INSERT FIRST 문장을 실행하기에 앞서 다음과 같은 임시테이블을 생성한다.

```
SQL> CREATE TABLE SAL1000
  2  AS SELECT EMPNO, ENAME, SAL FROM EMP
  3  WHERE 0 = 1;

테이블이 생성되었습니다.

SQL> CREATE TABLE SAL3000
  2  AS SELECT EMPNO, ENAME, SAL FROM EMP
  3  WHERE 0 = 1;

테이블이 생성되었습니다.

SQL> CREATE TABLE SALMAX
  2  AS SELECT EMPNO, ENAME, SAL FROM EMP
  3  WHERE 0 = 1;

테이블이 생성되었습니다.
```

사원 테이블에서 직원의 급여가 1000 이하이면 SAL1000, 3000 이하이면 SAL3000, 3000이 초과되면 SALMAX 테이블에 사번, 사원명, 급여를 입력하는 방법은 다음과 같다.

```
SQL> INSERT FIRST
  2    WHEN SAL <= 1000 THEN
  3      INTO SAL1000 VALUES (EMPNO, ENAME, SAL)
  4    WHEN SAL <= 3000 THEN
  5      INTO SAL3000 VALUES (EMPNO, ENAME, SAL)
  6    ELSE
  7      INTO SALMAX VALUES (EMPNO, ENAME, SAL)
  8    SELECT EMPNO, ENAME, SAL FROM EMP;
```

피벗팅(Pivoting) INSERT

피벗팅 INSERT는 한 개의 행을 한 개의 테이블내 여러 개의 행으로 입력하는 방법으로서 비관계형 데이터베이스의 테이블을 관계형 데이터베이스의 테이블로 변환하는데 유용하다. 사용 방법은 무조건 INSERT ALL 문장과 동일하지만 INTO 절에는 모두 같은 테이블을 기술하는 것이 다르다.

피벗팅 INSERT 문장을 실행하기에 앞서 다음과 같은 임시테이블을 생성한다.

```
CREATE TABLE SALES_DATA
(EMPNO NUMBER(4),
SPRING_SALE NUMBER(9),
SUMMER_SALE NUMBER(9),
AUTUMN_SALE NUMBER(9),
WINTER_SALE NUMBER(9));

INSERT INTO SALES_DATA VALUES (100, 3000, 5000, 6000, 7000);
INSERT INTO SALES_DATA VALUES (200, 4000, 2000, 3000, 5000);
INSERT INTO SALES_DATA VALUES (300, 6000, 3000, 4000, 4000);
INSERT INTO SALES_DATA VALUES (400, 3000, 1000, 5000, 2000);

CREATE TABLE SALES_SEASON
(EMPNO NUMBER(4),
SEASON_CODE CHAR,
SALES NUMBER(9));
```

SALES_DATA 테이블은 각 사원들의 계절별 판매액을 저장한 테이블인데, 계절별로 별도 컬럼이 있으며, 각 컬럼에 계절별 판매액이 저장되어 있다. 만약, 계절별로 컬럼을 두지 않고 SALES_SEASON 테이블과 같이 한 개의 컬럼을 가진 테이블을 만들어 데이터를 이관하고자 한다면, 피벗팅 INSERT 문장을 사용하여 수월하게 데이터를 이관 할 수 있다.

```
SQL> INSERT ALL
  2  INTO SALES_SEASON VALUES (EMPNO, '1', SPRING_SALE)
  3  INTO SALES_SEASON VALUES (EMPNO, '2', SUMMER_SALE)
  4  INTO SALES_SEASON VALUES (EMPNO, '3', AUTUMN_SALE)
  5  INTO SALES_SEASON VALUES (EMPNO, '4', WINTER_SALE)
  6  SELECT EMPNO, SPRING_SALE, SUMMER_SALE, AUTUMN_SALE, WINTER_SALE
  7  FROM SALES_DATA;
```

EXTERNAL TABLE

EXTERNAL TABLE은 데이터베이스에는 테이블에 대한 정의만 저장되고 실제 데이터는 데이터베이스 외부에 저장되는 읽기 전용 테이블로서 데이터베이스에 저장 시킬 필요 없이 직접 검색 또는 조인이 가능하다. 그러나 EXTERNAL TABLE은 읽기 전용으로 DML 작업이 불가능하며 인덱스를 생성할 수도 없다.

EXTERNAL TABLE을 사용하는 방법은 다음과 같다. 먼저, 외부 파일이 저장되어 있는 디렉터리 명으로 디렉터리 객체를 생성한다. 디렉터리 객체를 생성하기 위해서는 CREATE ANY DIRECTORY 권한을 부여 받아야 한다. CREATE ANY DIRECTORY 권한을 부여 받았으며 외부 파일이 C:₩ORACLE 디렉터리에 저장되어 있다면 디렉터리 객체를 만드는 방법은 다음과 같다.

```
SQL> CONNECT / AS SYSDBA
연결되었습니다.
SQL> GRANT CREATE ANY DIRECTORY TO SCOTT;

권한이 부여되었습니다.

SQL> CONNECT SCOTT/TIGER
연결되었습니다.
SQL> CREATE DIRECTORY EMP_DIR AS 'C:WORACLE';

디렉토리가 생성되었습니다.
```

C:₩ORACLE 디렉터리의 외부 파일 EMP.TXT의 내용은 다음과 같다.

```
7369,SMITH,CLERK,7902,80/12/17
7499,ALLEN,SALESMAN,7698,81/02/20
7521,WARD,SALESMAN,7698,81/02/22
7566,JONES,MANAGER,7839,81/04/02
7654,MARTIN,SALESMAN,7698,81/09/28
7698,BLAKE,MANAGER,7839,81/05/01
7782,CLARK,MANAGER,7839,81/06/09
7788,SCOTT,ANALYST,7566,87/04/19
7839,KING,PRESIDENT,,81/11/17
7844,TURNER,SALESMAN,7698,81/09/08
7876,ADAMS,CLERK,7788,87/05/23
7900,JAMES,CLERK,7698,81/12/03
7902,FORD,ANALYST,7566,81/12/03
7934,MILLER,CLERK,7782,82/01/23
```

외부 파일을 기반으로 EXTERNAL TABLE을 생성하는 방법은 다음과 같다.

```
SQL> CREATE TABLE EXTEMP (
  2  EMPNO NUMBER, ENAME VARCHAR2(10), JOB VARCHAR2(10), MGR NUMBER, HIREDATE DATE)
  3  ORGANIZATION EXTERNAL
  4  (TYPE ORACLE_LOADER
  5  DEFAULT DIRECTORY EMP_DIR
  6  ACCESS PARAMETERS
  7  (RECORDS DELIMITED BY NEWLINE
  8  BADFILE 'BAD_EMP'
  9  LOGFILE 'LOG_EMP'
 10  FIELDS TERMINATED BY ','
 11  (EMPNO CHAR,
 12  ENAME CHAR,
 13  JOB CHAR,
 14  MGR CHAR,
 15  HIREDATE CHAR DATE_FORMAT DATE MASK "YY/MM/DD"))
 16  LOCATION('EMP.TXT'))
 17  PARALLEL 5
 18  REJECT LIMIT 200;

테이블이 생성되었습니다.
```

위에서 EXTERNAL TABLE을 정의 할 때는 CREATE TABLE에 ORGANIZATION EXTERNAL을 지정하여 데이터가 데이터베이스 외부에 저장되어 있으며 읽기 전용이라는 것을 데이터베이스에 알려주어야 한다. TYPE절 뒤에는 외부 파일 접근을 위한 드라이버

이름을 기술하는데 Oracle은 SQL*Loader 유틸리티 기반의 ORACLE_LOADER 드라이버와 Import/Export 유틸리티 기반의 ORACLE_INTERNAL 드라이버를 제공하며, 별도로 기술하지 않으면 디폴트로 ORACLE_LOADER 드라이버가 사용된다. DEFAULT DIRECTORY는 외부 파일이 저장되어 있는 디렉터리 위치를 기술한 디렉터리 객체의 이름을 지정하고 ACCESS PARAMETERS에는 외부 파일의 형식을 기술한다. LOCATION에는 외부 파일의 이름을 지정하지만 반드시 기술 할 필요는 없으며 REJECT LIMIT는 외부 파일을 읽으면서 발생 할 수 있는 오류의 최대 갯수를 적어주면 된다.

CREATE TABLE 문장에서 CREATE INDEX 사용

Oracle 서버는 테이블에 PRIMARY KEY 제약조건을 설정하면 자동으로 제약조건의 이름과 동일한 고유 인덱스가 생성된다. Oracle 9i에서는 PRIMARY KEY 제약조건과 함께 CREATE INDEX 문장을 사용하면 인덱스 이름을 사용자 임의로 지정 할 수 있다.

```
SQL> CREATE TABLE NEW_EMP
  2  (EMPNO NUMBER(4)
  3  PRIMARY KEY USING INDEX
  4  (CREATE INDEX EMP_EMPNO_IDX ON
  5  NEW_EMP(EMPNO)),
  6  ENAME VARCHAR2(10),
  7  JOB VARCHAR2(10));

테이블이 생성되었습니다.

SQL> SELECT INDEX_NAME, TABLE_NAME
  2  FROM USER_INDEXES
  3  WHERE TABLE_NAME = 'NEW_EMP';

INDEX_NAME                     TABLE_NAME
------------------------------ ------------------------------
EMP_EMPNO_IDX                  NEW_EMP
```

복습

1. 다음과 같은 임시 테이블을 생성하고, 사원 테이블의 사원정보를 업무별로 ANALYST, CLERK, MANAGER, PRESIDENT, SALESMAN 테이블에 각각 입력하시오

```
CREATE TABLE ANALYST AS SELECT * FROM EMP WHERE 0 = 1;
CREATE TABLE CLERK AS SELECT * FROM EMP WHERE 0 = 1;
CREATE TABLE MANAGER AS SELECT * FROM EMP WHERE 0 = 1;
CREATE TABLE PRESIDENT AS SELECT * FROM EMP WHERE 0 = 1;
CREATE TABLE SALESMAN AS SELECT * FROM EMP WHERE 0 = 1;
```

2. 다음과 같은 임시 테이블을 생성하고, 부서번호가 10이고 급여가 2000이 초과되는 직원의 사원정보는 T10, 부서번호가 20이고 급여가 1000 미만인 직원의 사원정보는 T20, 부서번호가 30이고 급여가 2000에서 3000사이인 직원의 사원정보는 T30 테이블에 각각 입력하시오.

```
CREATE TABLE T10 AS SELECT * FROM EMP WHERE 0 = 1;
CREATE TABLE T20 AS SELECT * FROM EMP WHERE 0 = 1;
CREATE TABLE T30 AS SELECT * FROM EMP WHERE 0 = 1;
```

Chapter 21. 분석 함수

이번 장에서는 Oracle 8.1.6 버전부터 제공되는 분석함수에 대하여 설명한다. 분석함수를 사용하면 복잡한 쿼리를 간단하게 기술 할 수 있으며, 수행 속도 또한 향상된다.

랭킹(Ranking) 함수

랭킹과 관련된 함수들은 지정된 기준에 따라 해당 행의 순위를 계산하며, 사용 방법은 다음과 같다.

```
analytic_function() OVER (
        [PARTITION BY <value expression1>][,....]
        ORDER BY <value expression2>[collate clause][ASC|DESC]
        [NULLS FIRST|NULLS LAST][,...])
```

여기서,
· OVER : 해당 결과 집합을 이용해 동작하는 함수라는 의미이다.
· PARTITION BY : 결과 집합이 *value expression1*에 지정된 값으로 분할된다.
· ORDER BY : PARTITION BY에 의해 분할된 각 파티션을 *value expression2*에 지정된 컬럼으로 정렬한다.
· NULLS FIRST|NULLS LAST : NULL이 포함된 행이 PARTITION내에서 가장 먼저 위치 할 것인지 가장 마지막에 위치 할 것인지를 지정

■ RANK

RANK 함수는 각 파티션 내에서 ORDER BY 뒤에 지정된 컬럼을 기준으로 정렬한 뒤, 해당 행에 대한 순위를 계산한다. 사원 테이블에서 입사일이 빠른 순서대로 순위를 계산하는 방법은 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE, RANK() OVER (ORDER BY HIREDATE) RANK
  2  FROM EMP;

     EMPNO ENAME      HIREDATE       RANK
---------- ---------- -------- ----------
      7369 SMITH      80/12/17          1
      7499 ALLEN      81/02/20          2
      7521 WARD       81/02/22          3
      7566 JONES      81/04/02          4
      7698 BLAKE      81/05/01          5
      7782 CLARK      81/06/09          6
      7844 TURNER     81/09/08          7
      7654 MARTIN     81/09/28          8
      7839 KING       81/11/17          9
      7900 JAMES      81/12/03         10
      7902 FORD       81/12/03         10
      7934 MILLER     82/01/23         12
      7788 SCOTT      87/04/19         13
      7876 ADAMS      87/05/23         14
```

위의 경우에서 OVER에 PARTITION BY가 생략되었으며 테이블의 모든 행이 하나의 파티션으로 지정된다.

이번에는 각 부서에 대하여 사원들의 급여에 따라 순위를 계산한다. 부서별로 파티션이 되고, 해당 파티션별로 순위가 계산된다.

```
SQL> SELECT EMPNO, ENAME, DEPTNO, SAL,
  2  RANK() OVER (PARTITION BY DEPTNO ORDER BY SAL DESC) RANK
  3  FROM EMP;

     EMPNO ENAME          DEPTNO        SAL       RANK
---------- ---------- ---------- ---------- ----------
      7839 KING               10       5000          1
      7782 CLARK              10       2450          2
      7934 MILLER             10       1300          3
      7788 SCOTT              20       3000          1
      7902 FORD               20       3000          1
      7566 JONES              20       2975          3
      7876 ADAMS              20       1100          4
      7369 SMITH              20        800          5
      7698 BLAKE              30       2850          1
      7499 ALLEN              30       1600          2
      7844 TURNER             30       1500          3
      7521 WARD               30       1250          4
      7654 MARTIN             30       1250          4
      7900 JAMES              30        950          6
```

■ DENSE_RANK

RANK는 동일한 순위가 존재하면 그 다음 순위가 존재하지 않는다. 즉, RANK의 경우, 2등이 2건이면 3등은 존재 할 수 없으며, 그 다음 순위는 4등이 되는 반면, DENSE_RANK는 동일한 순위가 있더라도, 순위가 차례대로 계산된다.

```
SQL> SELECT EMPNO, ENAME, SAL,
  2  RANK() OVER (ORDER BY SAL DESC) RANK,
  3  DENSE_RANK() OVER (ORDER BY SAL DESC) DENSE_RANK
  4  FROM EMP;

     EMPNO ENAME          SAL       RANK DENSE_RANK
---------- ---------- ---------- ---------- ----------
      7839 KING           5000          1          1
      7788 SCOTT          3000          2          2
      7902 FORD           3000          2          2
      7566 JONES          2975          4          3
      7698 BLAKE          2850          5          4
      7782 CLARK          2450          6          5
      7499 ALLEN          1600          7          6
      7844 TURNER         1500          8          7
      7934 MILLER         1300          9          8
      7521 WARD           1250         10          9
      7654 MARTIN         1250         10          9
      7876 ADAMS          1100         12         10
      7900 JAMES           950         13         11
      7369 SMITH           800         14         12
```

■ CUME_DIST

CUME_DIST는 RANK 함수와 같이 파티션 내에서 ORDER BY에 지정된 컬럼을 기준으로 순위를 계산하지만, 최대값 1을 기준으로 순위를 0~1사이의 값으로 표시한다.

```
SQL> SELECT EMPNO, ENAME, SAL,
  2  RANK() OVER (ORDER BY SAL DESC) RANK,
  3  CUME_DIST() OVER (ORDER BY SAL DESC) CUME_DIST
  4  FROM EMP;

     EMPNO ENAME             SAL       RANK  CUME_DIST
---------- ---------- ---------- ---------- ----------
      7839 KING             5000          1 .071428571  <-  1건/14건
      7788 SCOTT            3000          2 .214285714
      7902 FORD             3000          2 .214285714
      7566 JONES            2975          4 .285714286
      7698 BLAKE            2850          5 .357142857
      7782 CLARK            2450          6 .428571429
      7499 ALLEN            1600          7         .5
      7844 TURNER           1500          8 .571428571
      7934 MILLER           1300          9 .642857143
      7521 WARD             1250         10 .785714286
      7654 MARTIN           1250         10 .785714286
      7876 ADAMS            1100         12 .857142857
      7900 JAMES             950         13 .928571429
      7369 SMITH             800         14          1  <- 14건/14건
```

■ PERCENT_RANK

PERCENT_RANK는 CUME_DIST와 유사하지만, 계산 방법은 약간 다르다.

PERCENT_RANK = (파티션 내 자신의 RANK-1)/(파티션 내의 행 개수-1)

```
SQL> SELECT EMPNO, ENAME, SAL,
  2  RANK() OVER (ORDER BY SAL DESC) RANK,
  3  PERCENT_RANK() OVER (ORDER BY SAL DESC) PERCENT_RANK
  4  FROM EMP;

     EMPNO ENAME             SAL       RANK PERCENT_RANK
---------- ---------- ---------- ---------- ------------
      7839 KING             5000          1            0 <-  (1-1)/(14-1)
      7788 SCOTT            3000          2   .076923077
      7902 FORD             3000          2   .076923077
      7566 JONES            2975          4   .230769231
      7698 BLAKE            2850          5   .307692308
      7782 CLARK            2450          6   .384615385
      7499 ALLEN            1600          7   .461538462
      7844 TURNER           1500          8   .538461538
      7934 MILLER           1300          9   .615384615
      7521 WARD             1250         10   .692307692
      7654 MARTIN           1250         10   .692307692
      7876 ADAMS            1100         12   .846153846
      7900 JAMES             950         13   .923076923
      7369 SMITH             800         14            1 <- (14-1)/(14-1)
```

■ NTILE(N)

파티션 내의 행들을 N개로 분류하여 행의 위치를 표시한다.

```
SQL> SELECT EMPNO, ENAME, SAL,
  2  NTILE(7) OVER (ORDER BY SAL DESC) NTILE
  3  FROM EMP;

     EMPNO ENAME             SAL      NTILE
---------- ---------- ---------- ----------
      7839 KING             5000          1
      7788 SCOTT            3000          1
      7902 FORD             3000          2
      7566 JONES            2975          2
      7698 BLAKE            2850          3
      7782 CLARK            2450          3
      7499 ALLEN            1600          4
      7844 TURNER           1500          4
      7934 MILLER           1300          5
      7521 WARD             1250          5
      7654 MARTIN           1250          6
      7876 ADAMS            1100          6
      7900 JAMES             950          7
      7369 SMITH             800          7
```

■ ROW_NUMBER

파티션 내에 행들에 대하여 차례로 순번을 표시한다. 가상 컬럼은 ROWNUM과 유사하다.

```
SQL> SELECT EMPNO, ENAME, SAL,
  2  ROW_NUMBER() OVER (ORDER BY SAL DESC) ROW_NUMBER
  3  FROM EMP;

     EMPNO ENAME             SAL ROW_NUMBER
---------- ---------- ---------- ----------
      7839 KING             5000          1
      7788 SCOTT            3000          2
      7902 FORD             3000          3
      7566 JONES            2975          4
      7698 BLAKE            2850          5
      7782 CLARK            2450          6
      7499 ALLEN            1600          7
      7844 TURNER           1500          8
      7934 MILLER           1300          9
      7521 WARD             1250         10
      7654 MARTIN           1250         11
      7876 ADAMS            1100         12
      7900 JAMES             950         13
      7369 SMITH             800         14
```

윈도우(Windowing) 함수

윈도우 함수란 기존의 집계 함수에 대하여 다음과 같은 기능을 지원하는 함수이다.

```
{SUM|AVG|MAX|MIN|COUNT|STDDEV|VARIANCE|FIRST_VALUE|LAST_VALUE}
        ({<value expression1>|*}) OVER
                ([PARTITION BY <value expression2>[,...]]
                ORDER BY <value expression3>[collate clause]
                [ASC|DESC][NULLS FIRST|NULLS LAST][,...]
                ROWS|RANGE
                {{UNBOUNDED PRECEDING|<value expression4> PRECEDING}
                |BETWEEN
                {UNBOUNDED PRECEDING|<value expression5> PRECEDING}
                AND {CURRENT ROW|<value expression6> FOLLOWING}}
```

여기서,

· OVER : 결과 집합을 이용해 동작하는 함수라는 의미이다.
· PARTITION BY : 결과 집합이 *value expression1*에 지정된 값으로 분할된다.
· ORDER BY : 각 파티션 내의 결과가 *value expression2*에 지정된 값으로 정렬된다.
· NULLS FIRST|NULLS LAST : NULL이 포함된 행이 파티션 내에서 가장 먼저 위치 할 것인지 가장 마지막에 위치 할 것인지를 지정
· ROWS|RANGE : 연산 할 행들을 파티션 내에서 물리적(ROWS) 순서를 이용하여 선택 할 것인지 논리적(RANGE) 순서를 이용하여 선택 할 것인지를 결정
· BETWEEN ... AND ... : 파티션 내에서 연산 할 행들의 범위를 결정
· UNBOUNDED PRECEDING : 파티션 내에서 지정된 행 이전의 모든 행을 포함
· UNBOUNDED FOLLOWING : 파티션 내에서 지정된 행 이후의 모든 행을 포함
· CURRENT ROW : 파티션 내에서 현재 행을 시작 행 또는 마지막 행으로 이용 할 때 사용

다음과 같은 다양한 예제를 통해 윈도우 함수의 사용법에 대해서 살펴본다.

먼저, 사원을 입사일로 정렬하고, 각 사원의 급여와 누적 급여를 출력하는 방법은 다음과 같다. UNBOUNDED PRECEDING은 현재 행과 이전 행을 대상으로 집계 함수를 적용한다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE, SAL,
  2  SUM(SAL) OVER (ORDER BY HIREDATE ROWS UNBOUNDED PRECEDING) CUM_SAL
  3  FROM EMP;

     EMPNO ENAME      HIREDATE        SAL    CUM_SAL
---------- ---------- -------- ---------- ----------
      7369 SMITH      80/12/17        800        800
      7499 ALLEN      81/02/20       1600       2400
      7521 WARD       81/02/22       1250       3650
      7566 JONES      81/04/02       2975       6625
      7698 BLAKE      81/05/01       2850       9475
      7782 CLARK      81/06/09       2450      11925
      7844 TURNER     81/09/08       1500      13425
      7654 MARTIN     81/09/28       1250      14675
      7839 KING       81/11/17       5000      19675
      7900 JAMES      81/12/03        950      20625
      7902 FORD       81/12/03       3000      23625
      7934 MILLER     82/01/23       1300      24925
      7788 SCOTT      87/04/19       3000      27925
      7876 ADAMS      87/05/23       1100      29025
```

PARTITION BY를 사용하면 부서별 누적 급여를 출력 할 수도 있다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE, SAL, DEPTNO,
  2  SUM(SAL) OVER (PARTITION BY DEPTNO ORDER BY HIREDATE
  3                 ROWS UNBOUNDED PRECEDING) CUM_SAL
  4  FROM EMP;

     EMPNO ENAME      HIREDATE        SAL     DEPTNO    CUM_SAL
---------- ---------- -------- ---------- ---------- ----------
      7782 CLARK      81/06/09       2450         10       2450
      7839 KING       81/11/17       5000         10       7450
      7934 MILLER     82/01/23       1300         10       8750
      7369 SMITH      80/12/17        800         20        800
      7566 JONES      81/04/02       2975         20       3775
      7902 FORD       81/12/03       3000         20       6775
      7788 SCOTT      87/04/19       3000         20       9775
      7876 ADAMS      87/05/23       1100         20      10875
      7499 ALLEN      81/02/20       1600         30       1600
      7521 WARD       81/02/22       1250         30       2850
      7698 BLAKE      81/05/01       2850         30       5700
      7844 TURNER     81/09/08       1500         30       7200
      7654 MARTIN     81/09/28       1250         30       8450
      7900 JAMES      81/12/03        950         30       9400
```

각 사원들의 급여와 해당 사원 및 먼저 입사한 사원 3명의 평균 급여를 출력하는 방법은 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE, SAL,
  2  AVG(SAL) OVER (ORDER BY HIREDATE ROWS 3 PRECEDING) AVG_SAL
  3  FROM EMP;

     EMPNO ENAME      HIREDATE        SAL    AVG_SAL
---------- ---------- -------- ---------- ----------
      7369 SMITH      80/12/17        800        800
      7499 ALLEN      81/02/20       1600       1200
      7521 WARD       81/02/22       1250 1216.66667
      7566 JONES      81/04/02       2975    1656.25 <- (800+1600+1250+2975)/4
      7698 BLAKE      81/05/01       2850    2168.75
      7782 CLARK      81/06/09       2450    2381.25
      7844 TURNER     81/09/08       1500    2443.75
      7654 MARTIN     81/09/28       1250     2012.5
      7839 KING       81/11/17       5000       2550
      7900 JAMES      81/12/03        950       2175
      7902 FORD       81/12/03       3000       2550
      7934 MILLER     82/01/23       1300     2562.5
      7788 SCOTT      87/04/19       3000     2062.5
      7876 ADAMS      87/05/23       1100       2100 <- (3000+1300+3000+1100)/4
```

사원의 급여와 해당 부서의 최대 급여 및 최소 급여를 출력하는 방법은 다음과 같다. FIRST_VALUE는 해당 PARTITION의 첫 번째 행들을 선택하며, LAST_VALUE는 해당 파티션의 마지막 행들을 선택한다. 아래 예제에서는 각각 MIN, MAX를 사용해도 결과는 동일하다.

```
SQL> SELECT EMPNO, ENAME, SAL, DEPTNO,
  2  FIRST_VALUE(SAL) OVER (PARTITION BY DEPTNO ORDER BY SAL
  3     ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) MIN_SAL,
  4  LAST_VALUE(SAL) OVER (PARTITION BY DEPTNO ORDER BY SAL
  5     ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) MAX_SAL
  6  FROM EMP;

     EMPNO ENAME             SAL     DEPTNO    MIN_SAL    MAX_SAL
---------- ---------- ---------- ---------- ---------- ----------
      7934 MILLER           1300         10       1300       5000
      7782 CLARK            2450         10       1300       5000
      7839 KING             5000         10       1300       5000
      7369 SMITH             800         20        800       3000
      7876 ADAMS            1100         20        800       3000
      7566 JONES            2975         20        800       3000
      7788 SCOTT            3000         20        800       3000
      7902 FORD             3000         20        800       3000
      7900 JAMES             950         30        950       2850
      7521 WARD             1250         30        950       2850
      7654 MARTIN           1250         30        950       2850
      7844 TURNER           1500         30        950       2850
      7499 ALLEN            1600         30        950       2850
      7698 BLAKE            2850         30        950       2850
```

각 사원의 급여와 전체 급여에서 각 사원의 급여가 차지하는 비율을 출력하는 방법은 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, SAL,
  2  RATIO_TO_REPORT(SAL) OVER() RATIO
  3  FROM EMP;

     EMPNO ENAME             SAL      RATIO
---------- ---------- ---------- ----------
      7369 SMITH             800 .027562446
      7499 ALLEN            1600 .055124892
      7521 WARD             1250 .043066322
      7566 JONES            2975 .102497847
      7654 MARTIN           1250 .043066322
      7698 BLAKE            2850 .098191214
      7782 CLARK            2450 .084409991
      7788 SCOTT            3000 .103359173
      7839 KING             5000 .172265289
      7844 TURNER           1500 .051679587
      7876 ADAMS            1100 .037898363
      7900 JAMES             950 .032730405
      7902 FORD             3000 .103359173
      7934 MILLER           1300 .044788975
```

각 사원의 급여와 해당 사원보다 직전에 입사한 사원의 급여 및 직후에 입사한 사원의 급여를 출력하는 방법은 다음과 같다.

```
SQL> SELECT EMPNO, ENAME, HIREDATE,
  2  LAG(HIREDATE, 1) OVER (ORDER BY HIREDATE) LAG_HIREDATE,
  3  LEAD(HIREDATE, 1) OVER (ORDER BY HIREDATE) LEAD_HIREDATE
  4  FROM EMP;

     EMPNO ENAME      HIREDATE LAG_HIRE LEAD_HIR
---------- ---------- -------- -------- --------
      7369 SMITH      80/12/17          81/02/20
      7499 ALLEN      81/02/20 80/12/17 81/02/22
      7521 WARD       81/02/22 81/02/20 81/04/02
      7566 JONES      81/04/02 81/02/22 81/05/01
      7698 BLAKE      81/05/01 81/04/02 81/06/09
      7782 CLARK      81/06/09 81/05/01 81/09/08
      7844 TURNER     81/09/08 81/06/09 81/09/28
      7654 MARTIN     81/09/28 81/09/08 81/11/17
      7839 KING       81/11/17 81/09/28 81/12/03
      7900 JAMES      81/12/03 81/11/17 81/12/03
      7902 FORD       81/12/03 81/12/03 82/01/23
      7934 MILLER     82/01/23 81/12/03 87/04/19
      7788 SCOTT      87/04/19 82/01/23 87/05/23
      7876 ADAMS      87/05/23 87/04/19
```

복습

1. 다음과 같이 급여가 높은 순서대로 순위를 출력하는 쿼리를 작성하시오.

EMPNO	ENAME	SAL	순위
7839	KING	5000	1
7788	SCOTT	3000	2
7902	FORD	3000	2
7566	JONES	2975	4
7698	BLAKE	2850	5
7782	CLARK	2450	6
7499	ALLEN	1600	7
7844	TURNER	1500	8
7934	MILLER	1300	9
7521	WARD	1250	10
7654	MARTIN	1250	10
7876	ADAMS	1100	12
7900	JAMES	950	13
7369	SMITH	800	14

2. 다음과 같이 업무별 급여가 높은 순으로 순위를 출력하는 쿼리를 작성하시오.

EMPNO	ENAME	SAL	JOB	업무별순위
7788	SCOTT	3000	ANALYST	1
7902	FORD	3000	ANALYST	1
7934	MILLER	1300	CLERK	1
7876	ADAMS	1100	CLERK	2
7900	JAMES	950	CLERK	3
7369	SMITH	800	CLERK	4
7566	JONES	2975	MANAGER	1
7698	BLAKE	2850	MANAGER	2
7782	CLARK	2450	MANAGER	3
7839	KING	5000	PRESIDENT	1
7499	ALLEN	1600	SALESMAN	1
7844	TURNER	1500	SALESMAN	2
7521	WARD	1250	SALESMAN	3
7654	MARTIN	1250	SALESMAN	3

3. 다음과 같이 부서별 최대 급여가 높은 순서대로 순위를 출력하시오.

DEPTNO	MAX(SAL)	순위
10	5000	1
20	3000	2
30	2850	3

4. 다음과 같이 동일한 연도에 입사한 사원들에 대하여 입사일을 기준으로 순위를 출력하시오.

EMPNO	ENAME	입사	HIREDATE	년도별순위
7369	SMITH	1980	80/12/17	1
7499	ALLEN	1981	81/02/20	1
7521	WARD	1981	81/02/22	2
7566	JONES	1981	81/04/02	3
7698	BLAKE	1981	81/05/01	4
7782	CLARK	1981	81/06/09	5
7844	TURNER	1981	81/09/08	6
7654	MARTIN	1981	81/09/28	7
7839	KING	1981	81/11/17	8
7900	JAMES	1981	81/12/03	9
7902	FORD	1981	81/12/03	9
7934	MILLER	1982	82/01/23	1
7788	SCOTT	1987	87/04/19	1
7876	ADAMS	1987	87/05/23	2

5. 다음과 같이 각 사원의 급여와 업무별 누적 급여를 출력하시오.

EMPNO	ENAME	SAL	JOB	누적급여
7788	SCOTT	3000	ANALYST	3000
7902	FORD	3000	ANALYST	6000
7369	SMITH	800	CLERK	800
7900	JAMES	950	CLERK	1750
7876	ADAMS	1100	CLERK	2850
7934	MILLER	1300	CLERK	4150
7782	CLARK	2450	MANAGER	2450
7698	BLAKE	2850	MANAGER	5300
7566	JONES	2975	MANAGER	8275
7839	KING	5000	PRESIDENT	5000
7521	WARD	1250	SALESMAN	1250
7654	MARTIN	1250	SALESMAN	2500
7844	TURNER	1500	SALESMAN	4000
7499	ALLEN	1600	SALESMAN	5600

Chapter 22. Oracle 10g에서 향상된 SQL

이번 장에서는 Oracle Database 10g에서 향상된 SQL 문장에 대해서 알아본다. 새로운 조건 및 확장 구문이 추가된 MERGE 문장과 파티션 OUTER JOIN(Partitioned Outer Join)에 대해서 설명하고, 분석 기능을 위한 행간 연산(Inter-row Calculation)에 대하여 알아본다. 또한, DROP TABLE ... PURGE, 플래시백 쿼리에 대해서도 설명한다. 단, 이번 장의 모든 예제는 반드시 Oracle 10g에서 수행하여야 한다.

향상된 MERGE 문장

과거 버전에 비하여 MERGE 문장은 다음과 같은 기능이 향상되었다.

- 표준 MERGE 명령에 새로운 조건 및 확장 구문을 추가하여, 문장을 쉽고 빠르게 실행 할 수 있다.
- MERGE ... UPDATE 문장에 DELETE 구문을 포함 시킬 수도 있다.

Oracle Database 10g에서는 MERGE 문장의 UPDATE 및 INSERT 구문에 WHERE 절을 추가하여, 특정 조건에 만족하지 않는 행들은 INSERT 및 UPDATE가 수행되지 않도록 할 수 있다. 다음과 같은 예제를 실행해보자. 먼저, 실습용 테이블 EMP20을 생성하고, EMP 테이블에서 20번 부서에 소속되어 있는 사원의 사번, 이름, 직급, 급여를 EMP20 테이블에 입력한다.

```
SQL> CREATE TABLE EMP20
  2  (EMPNO NUMBER(4),
  3  ENAME VARCHAR2(10),
  4  JOB VARCHAR2(10),
  5  SAL NUMBER(7,2));

SQL> INSERT INTO EMP20(EMPNO, ENAME, JOB, SAL)
  2  SELECT EMPNO, ENAME, JOB, 0
  3  FROM EMP
  4  WHERE DEPTNO = 20;

SQL> SELECT * FROM EMP20;

     EMPNO ENAME      JOB                SAL
---------- ---------- ---------- ----------
      7369 SMITH      CLERK               0
      7566 JONES      MANAGER             0
      7788 SCOTT      ANALYST             0
      7876 ADAMS      CLERK               0
      7902 FORD       ANALYST             0
```

MERGE 문장을 이용하여 EMP20 테이블의 데이터를 EMP 테이블의 데이터로 갱신해보자. 즉, EMP 테이블에 저장된 행이 EMP20 테이블에 존재하면 EMP20 테이블의 해당 행을 EMP 테이블에 저장된 행으로 UPDATE하고, 존재하지 않으면 EMP20 테이블에 해당 행을 INSERT하는 것이다. 단, EMP 테이블에서 직급이 'CLERK'인 경우에만 EMP20 테이블에 INSERT 또는 UPDATE 한다.

```
SQL> MERGE INTO EMP20 N
  2     USING EMP O
  3     ON (N.EMPNO = O.EMPNO)
  4     WHEN MATCHED THEN
  5       UPDATE SET
  6       N.ENAME = O.ENAME,N.JOB = O.JOB, N.SAL = O.SAL
  7       WHERE O.JOB='CLERK'
  8     WHEN NOT MATCHED THEN
  9       INSERT (N.EMPNO, N.ENAME, N.JOB, N.SAL)
 10       VALUES (O.EMPNO, O.ENAME, O.JOB, O.SAL)
 11       WHERE O.JOB='CLERK';

4 행이 병합되었습니다.

SQL> SELECT * FROM EMP20;

     EMPNO ENAME      JOB              SAL
---------- ---------- ---------- ----------
      7369 SMITH      CLERK            800
      7566 JONES      MANAGER            0
      7788 SCOTT      ANALYST            0
      7876 ADAMS      CLERK           1100
      7902 FORD       ANALYST            0
      7900 JAMES      CLERK            950
      7934 MILLER     CLERK           1300
```

MERGE ... UPDATE 문장에 의해서 행을 UPDATE 하는 경우, DELETE 옵션을 사용 할 수가 있다. DELETE 옵션에 의해 삭제되는 행들은 MERGE 작업에 의해 변경이 수행된 대상 테이블의 행이다. 즉, DELETE WHERE 조건은 변경 후의 데이터를 평가하며, 절대 UPDATE ... SET 구문의 원본 데이터를 평가하는지 않는다. 또한, 대상 테이블의 행이 DELETE 조건에 만족하더라도 ON에 기술된 조인 조건에 포함되지 않으면, 해당 행은 절대 삭제되지 않는다. 실습을 위해 앞서 작성했던 EMP20 테이블의 행을 삭제하고, 다시 행을 입력한다.

```
SQL> DELETE FROM EMP20;

SQL> INSERT INTO EMP20(EMPNO, ENAME, JOB, SAL)
  2  SELECT EMPNO, ENAME, JOB, 0
  3  FROM EMP
  4  WHERE DEPTNO = 20;

SQL> SELECT * FROM EMP20;

     EMPNO ENAME      JOB              SAL
---------- ---------- ---------- ----------
      7369 SMITH      CLERK              0
      7566 JONES      MANAGER            0
      7788 SCOTT      ANALYST            0
      7876 ADAMS      CLERK              0
      7902 FORD       ANALYST            0
```

앞선 예제와 같이, MERGE 문장을 이용하여 EMP20 테이블의 데이터를 EMP 테이블의 데이터로 갱신해보자. 즉, EMP 테이블에 저장된 행이 EMP20 테이블에 존재하면 EMP20 테

이블의 해당 행을 EMP 테이블에 저장된 행으로 UPDATE하고, 존재하지 않으면 EMP20 테이블에 해당 행을 INSERT하는 것이다. 단, UPDATE 후의 직급이 'ANALYST'인 경우에는 해당 행을 DELETE 한다.

```
SQL>  MERGE INTO EMP20 N
  2         USING EMP O
  3         ON (N.EMPNO = O.EMPNO)
  4         WHEN MATCHED THEN
  5           UPDATE SET
  6           N.ENAME = O.ENAME,N.JOB = O.JOB, N.SAL = O.SAL
  7           DELETE WHERE (N.JOB='ANALYST')
  8         WHEN NOT MATCHED THEN
  9           INSERT (N.EMPNO, N.ENAME, N.JOB, N.SAL)
 10           VALUES (O.EMPNO, O.ENAME, O.JOB, O.SAL);

14 행이 병합되었습니다.

SQL> SELECT * FROM EMP20;

     EMPNO ENAME      JOB              SAL
---------- ---------- ---------- ----------
      7369 SMITH      CLERK             800
      7566 JONES      MANAGER          2975
      7876 ADAMS      CLERK            1100
      7499 ALLEN      SALESMAN         1600
      7521 WARD       SALESMAN         1250
      7654 MARTIN     SALESMAN         1250
      7698 BLAKE      MANAGER          2850
      7782 CLARK      MANAGER          2450
      7839 KING       PRESIDENT        5000
      7844 TURNER     SALESMAN         1500
      7900 JAMES      CLERK             950
      7934 MILLER     CLERK            1300
```

파티션 OUTER JOIN(Partitioned Outer Join)

데이터는 일반적으로 희소(Sparse) 형태로 저장된다. 예를 들어, 특정 상점의 주문 내역을 살펴보면 1년 365일 모든 일자에서 반드시 매출이 발생하지는 않으므로, 매출이 발생하지 않는 날짜에 대한 데이터는 존재하지 않게 되고, 이러한 형태의 데이터를 희소(Sparse)하다고 표현한다. 반면, 1년 365일 모든 일자에서 매출이 발생하였다면 이러한 형태의 데이터를 밀집(Dense)하다고 표현 할 수가 있다. 일반적으로, 희소 형태의 데이터를 이용하여 밀집 형태의 결과물을 작성할 필요가 종종 발생하게 되며, 밀집 형태의 데이터에는 LAG()와 LEAD()와 같은 SQL 분석 함수를 적용하는 작업도 수월해 진다. 이를 위해, Oracle 10g는 파티션 OUTER JOIN 문장을 새롭게 추가하였다.

희소 데이터와 관련된 문제를 해결하기 위해서는 데이터의 간격(Gap)을 채우기 위해 파티션 OUTER JOIN을 사용해야 한다. 이러한 조인은 쿼리의 정의에 따라 논리적인 파티션을 구성하고 기존의 OUTER JOIN을 적용시키는 것으로서 기존의 고전적인 OUTER JOIN을 확장한 것이다. 오라클 데이터베이스는 쿼리의 PARTITION BY 구문에 지정된 표현식을 기

반으로 행들을 파티션으로 분류한 뒤, 각 파티션에 OUTER JOIN을 적용한 결과를 UNION 연산하여 표시하게 된다.

파티션 OUTER JOIN의 구문은 다음과 같다.

```
SELECT ...
FROM ...
       PARTITION BY (expr [, expr ] ...)
       RIGHT OUTER JOIN table_reference

SELECT ...
FROM ...
       LEFT OUTER JOIN table_reference
       PARTITION BY (expr [, expr] ...)
```

파티션 OUTER JOIN은 데이터가 존재하지 않으면 NULL 값을 리턴하게 되지만, NULL 값 대신에 가장 최근의 NULL이 아닌 값을 리턴 할 수도 있다. 또한, 해당 컬럼을 상향식으로 스캔하여, 검색되는 최초의 NULL이 아닌 값으로 NULL 값을 대체 할 수도 있다. Oracle 10g에서 IGNORE NULLS 키워드에 LAST_VALUE 및 FIRST_VALUE 함수를 사용하면 이와 같은 작업을 간단하게 수행 할 수 있다.

실습을 위하여 다음과 같은 예제 테이블을 작성해보자. TIME_DATA 테이블은 2001년 1월부터 12월의 날짜 데이터를 가지고 있다.

```
CREATE TABLE TIME_DATA
(TYEAR NUMBER(4),
 TMONTH NUMBER(2));

INSERT INTO TIME_DATA VALUES(2001, 1);
INSERT INTO TIME_DATA VALUES(2001, 2);
INSERT INTO TIME_DATA VALUES(2001, 3);
INSERT INTO TIME_DATA VALUES(2001, 4);
INSERT INTO TIME_DATA VALUES(2001, 5);
INSERT INTO TIME_DATA VALUES(2001, 6);
INSERT INTO TIME_DATA VALUES(2001, 7);
INSERT INTO TIME_DATA VALUES(2001, 8);
INSERT INTO TIME_DATA VALUES(2001, 9);
INSERT INTO TIME_DATA VALUES(2001, 10);
INSERT INTO TIME_DATA VALUES(2001, 11);
INSERT INTO TIME_DATA VALUES(2001, 12);
```

다음의 SALES 테이블이 실제 테이블로서, 제품명, 판매년도, 판매월, 판매일, 수량 정보가 저장되어 있다.

```
CREATE TABLE SALES
(PNAME VARCHAR2(10),
 SYEAR NUMBER(4),
 SMONTH NUMBER(2),
 SDAY NUMBER(2),
 QUANTITY NUMBER(3));

INSERT INTO SALES VALUES('APPLE', 2001, 1, 1, 10); -> APPLE이 2001/1/1에 10개 판매
INSERT INTO SALES VALUES('APPLE', 2001, 1, 2, 20);
INSERT INTO SALES VALUES('APPLE', 2001, 1, 3, 30);
INSERT INTO SALES VALUES('APPLE', 2001, 5, 1, 20);
INSERT INTO SALES VALUES('APPLE', 2001, 5, 2, 30);
INSERT INTO SALES VALUES('APPLE', 2001, 10, 1, 10);
INSERT INTO SALES VALUES('APPLE', 2001, 10, 2, 40);
INSERT INTO SALES VALUES('ORANGE', 2001, 2, 1, 10);
INSERT INTO SALES VALUES('ORANGE', 2001, 2, 2, 20);
INSERT INTO SALES VALUES('ORANGE', 2001, 2, 3, 30);
INSERT INTO SALES VALUES('ORANGE', 2001, 6, 1, 20);
INSERT INTO SALES VALUES('ORANGE', 2001, 6, 2, 30);
INSERT INTO SALES VALUES('ORANGE', 2001, 11, 1, 10);
INSERT INTO SALES VALUES('ORANGE', 2001, 11, 2, 40);
```

SALES 테이블로부터 월별 판매량 보고서를 작성한다.

```
SQL> SELECT S.PNAME, S.SYEAR, S.SMONTH, SUM(S.QUANTITY)
  2  FROM SALES S
  3  GROUP BY S.PNAME, S.SYEAR, S.SMONTH
  4  ORDER BY S.PNAME, S.SYEAR, S.SMONTH;

PNAME           TYEAR     TMONTH SUM(S.QUANTITY)
---------- ---------- ---------- ---------------
APPLE            2001          1              60
APPLE            2001          5              50
APPLE            2001         10              50
ORANGE           2001          2              60
ORANGE           2001          6              50
ORANGE           2001         11              50
```

결과를 살펴보면 'APPLE'에 대하여 판매 실적이 없는 2001년 2, 3, 4, 6, 7, 8, 9, 11, 12월과 'ORANGE'에 대하여 2001에 1, 3, 4, 5, 7, 8, 9, 10, 12월의 계산 결과는 당연히 존재하지 않는다. 이런 경우에 TIME_DATA 테이블과 SALES 테이블을 OUTER JOIN하여 결과를 표시해보자.

```
SQL> SELECT S.PNAME, T.TYEAR, T.TMONTH, SUM(S.QUANTITY)
  2  FROM TIME_DATA T LEFT OUTER JOIN SALES S
  3     ON T.TYEAR=S.SYEAR AND T.TMONTH=S.SMONTH
  4  GROUP BY S.PNAME, T.TYEAR, T.TMONTH
  5  ORDER BY S.PNAME, T.TYEAR, T.TMONTH;

PNAME           TYEAR     TMONTH SUM(S.QUANTITY)
---------- ---------- ---------- ---------------
APPLE            2001          1              60
APPLE            2001          5              50
APPLE            2001         10              50
ORANGE           2001          2              60
ORANGE           2001          6              50
ORANGE           2001         11              50
                 2001          3
                 2001          4
                 2001          7
                 2001          8
                 2001          9
                 2001         12
```

결과를 살펴보면, 'APPLE'과 'ORANGE'의 판매실적이 모두 없는 경우만 추가되어 출력된다. 이번에는 파티션 OUTER JOIN을 수행해본다. 제품별 판매실적이 없는 기간도 결과가 출력됨을 확인 할 수 있다.

```
SQL> SELECT S.PNAME, T.TYEAR, T.TMONTH, SUM(S.QUANTITY)
  2  FROM TIME_DATA T LEFT OUTER JOIN SALES S PARTITION BY (PNAME)
  3     ON T.TYEAR=S.SYEAR AND T.TMONTH=S.SMONTH
  4  GROUP BY S.PNAME, T.TYEAR, T.TMONTH
  5  ORDER BY S.PNAME, T.TYEAR, T.TMONTH;

PNAME           TYEAR     TMONTH SUM(S.QUANTITY)
---------- ---------- ---------- ---------------
APPLE            2001          1              60
APPLE            2001          2
APPLE            2001          3
APPLE            2001          4
APPLE            2001          5              50
APPLE            2001          6
APPLE            2001          7
APPLE            2001          8
APPLE            2001          9
APPLE            2001         10              50
APPLE            2001         11
APPLE            2001         12
ORANGE           2001          1
ORANGE           2001          2              60
ORANGE           2001          3
ORANGE           2001          4
ORANGE           2001          5
ORANGE           2001          6              50
ORANGE           2001          7
ORANGE           2001          8
ORANGE           2001          9
ORANGE           2001         10
ORANGE           2001         11              50
ORANGE           2001         12
```

SQL에서 행간(Inter-row) 연산

쿼리와 서브쿼리에서 스프레드시트와 유사한 배열 연산을 수행 할 수 있는 새로운 구문이 추가 되었다. 이러한 연산에 의해 관계형 테이블을 여러 번의 조인 및 집합 연산 없이 N-차원의 배열처럼 처리 할 수 있게 되었다. 이러한 연산 방식은 관계형 테이블 및 Oracle OLAP 분석 작업에서 사용 할 수 있다.

SQL MODEL 구문은 SQL 문장 내에서 행간 산술 연산을 수행 할 수 있도록 해주고, 선택된 행을 다차원 배열처럼 처리해서 해당 배열 내의 특정 셀(Cell)을 자유롭게 접근 할 수 있게 해준다. 즉, MODEL 구문을 사용하여 규칙(Rule)이라고 부르는 일련의 셀 할당 작업을 정의하고, 개별 셀 및 지정된 범위의 셀들에 대하여 연산을 수행한다. 이러한 규칙은 쿼리의 수행 결과에 대하여 적용되며 절대 데이터베이스 테이블의 데이터를 변경하지는 않는다. SQL MODEL 구문은 SQL 언어 내에서 스프레드시트에서 수행하였던 방식과 같은 직접적인 연산을 수행 할 수 있다. MODEL 구문으로 수행 할 수 있는 작업은 다음과 같다.

- 심볼을 이용한 셀 참조
- 심볼을 이용한 배열 연산
- 규칙 별 UPSERT/UPDATE 수행
- 규칙 별 연산 우선 순위 지정

MODEL 구문은 쿼리의 컬럼들을 세 개의 그룹 즉, 파티션, 측정값, 차원으로 맵핑하여 다차원 배열을 정의한다. 각각의 요소들은 다음과 같은 작업을 수행한다.

- 파티션은 분석 함수의 파티션과 유사한 방식으로, 쿼리의 결과 집합을 논리적인 블록으로 정의한다.
- 측정값은 스타(Star) 스키마에서 팩트(Fact) 테이블 내의 데이터와 유사하다. 일반적으로 판매량 또는 가격과 같은 수치 값을 포함하고 있다. 각 셀은 파티션 내에서 차원의 완전한 조합을 지정함으로써 접근 할 수 있다.
- 차원은 파티션 내에서 각 측정값을 식별한다. 이 컬럼은 날짜, 지역, 제품명과 같은 특성을 식별한다.

이러한 다차원 배열에서 규칙을 작성하려면, 차원으로 표현되는 산술 규칙으로 정의해야 한다. 규칙은 유연하고 간결하게 작성 할 수 있으며 와일드 카드 및 FOR 반복문을 사용 할 수도 있다.

다음 그림은 SALES 테이블을 이용하여 행간 연산의 개념을 보여준다. 먼저, 다음 그림과 같이 SALES 테이블의 컬럼들을 각각 파티션, 차원, 측정값으로 맵핑하였다.

COUNTRY	PRODUCT	YEAR	QTY
파티션	차원	차원	측정값

그림 22-1. SALES 테이블에서 컬럼의 맵핑

여기서, 규칙을 다음과 같이 정의하였다

```
QTY['APPLE',2008] = QTY['APPLE',2006] + QTY['APPLE',2007]
QTY['ORANGE',2008] = QTY['ORANGE',2006] + QTY['ORANGE',2007]
```

그 결과는 다음과 같이 출력 될 것이다.

COUNTRY 파티션	PRODUCT 차원	YEAR 차원	QTY 측정값	
KOR	**APPLE**	**2006**	**200**	
KOR	**APPLE**	**2007**	**100**	
KOR	**ORANGE**	**2006**	**150**	
KOR	**ORANGE**	**2007**	**170**	
USA	**APPLE**	**2006**	**210**	
USA	**APPLE**	**2007**	**120**	
USA	**ORANGE**	**2006**	**110**	
USA	**ORANGE**	**2007**	**180**	
KOR	*APPLE*	*2008*	*300*	*<- 200 + 100*
KOR	*ORANGE*	*2008*	*320*	*<- 150 + 170*
USA	*APPLE*	*2008*	*330*	*<- 210 + 120*
USA	*ORANGE*	*2008*	*290*	*<- 110 + 180*

결과에서 굵은 글씨체로 표시되는 부분은 쿼리에 의해서 검색되는 결과이며, 이탤릭체로 표시되는 결과는 규칙에 의해 연산된 결과 행이다. 규칙은 파티션으로 지정된 컬럼에 대해서 각각 수행되므로 COUNTRY 컬럼에 'KOR'와 'USA'가 있다면, 검색 결과를 먼저 'KOR'인 파티션과 'USA'인 파티션으로 분류하고 각 파티션에 대하여 각각의 규칙이 적용된다.

첫 번째 규칙은 검색 결과에서 각 COUNTRY 별로 2006년도 'APPLE'의 판매량과 2007년도 'APPLE'의 판매량을 합산하여 2008년도 'APPLE'의 판매량을 계산하는 것이고, 두 번째 규칙은 이와 유사하게 각 COUNTRY 별로 2008년도 'ORANGE'의 판매량을 계산한 것이다. 위의 규칙을 적용한 결과는 원본 테이블에 절대 반영되지 않는다. 만약, 이 결과 값을 테이블에 적용하려면 INSERT, UPDATE, MERGE 문장의 서브쿼리에 포함시켜야 한다.

위 SALES 테이블에 대하여 MODEL 구문이 다음과 같을 때, 연산 결과가 어떻게 될 것인지 생각해보자.

```
MODEL
PARTITION BY (COUNTRY)
DIMENSION BY (PRODUCT, YEAR)
MEASURES (QTY)
RULES UPSERT
(QTY[ANY, 2007]=QTY[CV(PRODUCT), CV(YEAR)-1]*2,
 QTY['APPLE', 2008]=QTY['APPLE',2007]+QTY['APPLE',2006],
 QTY['ORANGE', 2008]=AVG(QTY)[CV(PRODUCT), YEAR < 2008])
```

일단, COUNTRY가 'KOR'인 파티션에 대해서만 연산을 수행해본다. 원본 데이터는 다음과 같은 형태가 될 것이다.

COUNTRY 파티션	PRODUCT 차원	YEAR 차원	QTY 측정값
KOR	APPLE	2006	200
KOR	APPLE	2007	100
KOR	ORANGE	2006	150
KOR	ORANGE	2007	170

위와 같은 형태를 2차원 배열로 구성하면 다음과 같다.

	2006	2007
APPLE	200	100
ORANGE	150	170

첫 번째 규칙을 적용한다. CV() 함수는 좌변의 값을 우변에서 복사해서 사용하는 것으로서 좌변의 해당 컬럼을 참조하는 개념이다. 그러므로, 2007년도 각 제품별 판매량을 전년도 판매량의 2배로 변경한다.

	2006	2007
APPLE	200	**400**
ORANGE	150	**300**

두 번째 규칙은 2008년도 'APPLE'의 판매량을 2006년도 및 2007년도 'APPLE'의 판매량을 합산한 결과로 추가하게 된다.

	2006	2007	2008
APPLE	200	400	**600**
ORANGE	150	300	

세 번째 규칙은 2008년도 'ORANGE'의 판매량을 2008년 이전의 'ORANGE' 판매량의 평균으로 추가하게 된다.

	2006	2007	2008
APPLE	200	400	600
ORANGE	150	300	**225.5**

최종적인 출력 형태는 다음과 같다.

COUNTRY 파티션	PRODUCT 차원	YEAR 차원	QTY 측정값
KOR	APPLE	2006	200
KOR	APPLE	2007	400
KOR	APPLE	2008	600
KOR	ORANGE	2006	150
KOR	ORANGE	2007	300
KOR	ORANGE	2008	225.5

기본적으로 MODEL 구문의 RULES 뒤에는 UPSERT 옵션이 디폴트로 생략된 것으로, 규칙 적용시 좌변의 셀이 존재하면 해당 셀을 변경하고, 셀이 존재하지 않으면 해당 셀을 추가하

라는 의미이다. 그러나, RULES 키워드 뒤에 UPDATE를 지정하여 디폴트 설정을 무시 할 수도 있다. 또한, 다음과 같이 각 규칙 별로 UPDATE 또는 UPSERT를 개별적으로 지정할 수도 있다.

```
MODEL
PARTITION BY (COUNTRY)
DIMENSION BY (PRODUCT, YEAR)
MEASURES (QTY)
RULES
(UPDATE QTY[ANY, 2007]=QTY[CV(PRODUCT), CV(YEAR)-1]*2,
 UPSERT QTY['APPLE', 2008]=QTY['APPLE',2007]+QTY['APPLE',2006],
 UPSERT QTY['ORANGE', 2008]=AVG(QTY)[CV(PRODUCT), YEAR < 2008])
```

실제로 MODEL 구문을 이용하여 다음과 같은 예제를 실행해보자.

```
SQL> CREATE TABLE SALES
  2  (COUNTRY VARCHAR2(5),
  3  PRODUCT VARCHAR2(10),
  4  YEAR NUMBER(4),
  5  QTY NUMBER(4));

SQL> INSERT INTO SALES VALUES ('KOR', 'APPLE', 2006, 200);
SQL> INSERT INTO SALES VALUES ('KOR', 'APPLE', 2007,    100);
SQL> INSERT INTO SALES VALUES ('KOR', 'ORANGE', 2006, 150);
SQL> INSERT INTO SALES VALUES ('KOR', 'ORANGE', 2007, 170);
SQL> INSERT INTO SALES VALUES ('USA', 'APPLE', 2006, 210);
SQL> INSERT INTO SALES VALUES ('USA', 'APPLE', 2007, 120);
SQL> INSERT INTO SALES VALUES ('USA', 'ORANGE', 2006, 110);
SQL> INSERT INTO SALES VALUES ('USA', 'ORANGE', 2007, 180);

SQL> SELECT * FROM SALES;

COUNT PRODUCT          YEAR        QTY
----- ---------- ---------- ----------
KOR   APPLE            2006        200
KOR   APPLE            2007        100
KOR   ORANGE           2006        150
KOR   ORANGE           2007        170
USA   APPLE            2006        210
USA   APPLE            2007        120
USA   ORANGE           2006        110
USA   ORANGE           2007        180
```

MODEL 구문을 이용하여 보고서를 작성해본다.

```
SQL> SELECT COUNTRY, PRODUCT, YEAR, QTY
  2  FROM SALES
  3  WHERE COUNTRY='KOR'
  4     MODEL
  5     PARTITION BY (COUNTRY)
  6     DIMENSION BY (PRODUCT, YEAR)
  7     MEASURES (QTY)
  8     RULES (
  9             QTY[ANY, 2007]=QTY[CV(PRODUCT), CV(YEAR)-1]*2,
 10             QTY['APPLE', 2008]=QTY['APPLE', 2007]+QTY['APPLE', 2006],
 11             QTY['ORANGE', 2008]=AVG(QTY)[CV(PRODUCT), YEAR < 2008])
 12  ORDER BY COUNTRY, PRODUCT, YEAR, QTY;

COUNT PRODUCT          YEAR        QTY
----- ---------- ---------- ----------
KOR   APPLE            2006        200
KOR   APPLE            2007        400
KOR   APPLE            2008        600
KOR   ORANGE           2006        150
KOR   ORANGE           2007        300
KOR   ORANGE           2008        225
```

MODEL 구문 뒤에 RETURN UPDATED ROWS를 추가하면 변경 및 추가된 셀만 출력된다.

```
SQL> SELECT COUNTRY, PRODUCT, YEAR, QTY
  2  FROM SALES
  3  WHERE COUNTRY='KOR'
  4     MODEL RETURN UPDATED ROWS
  5     PARTITION BY (COUNTRY)
  6     DIMENSION BY (PRODUCT, YEAR)
  7     MEASURES (QTY)
  8     RULES (
  9             QTY[ANY, 2007]=QTY[CV(PRODUCT), CV(YEAR)-1]*2,
 10             QTY['APPLE', 2008]=QTY['APPLE', 2007]+QTY['APPLE', 2006],
 11             QTY['ORANGE', 2008]=AVG(QTY)[CV(PRODUCT), YEAR < 2008])
 12  ORDER BY COUNTRY, PRODUCT, YEAR, QTY;

COUNT PRODUCT          YEAR        QTY
----- ---------- ---------- ----------
KOR   APPLE            2007        400
KOR   APPLE            2008        600
KOR   ORANGE           2007        300
KOR   ORANGE           2008        225
```

이번에는 다음과 같은 MODEL 구문을 생각해보자.

```
SQL> SELECT COUNTRY, PRODUCT, YEAR, QTY
  2  FROM SALES
  3  WHERE COUNTRY='KOR'
  4     MODEL
  5     PARTITION BY (COUNTRY)
  6     DIMENSION BY (PRODUCT, YEAR)
  7     MEASURES (QTY)
  8     RULES (
  9             QTY['APPLE', 2008]=QTY['APPLE' ,2007]+QTY['APPLE', 2006],
 10             QTY['ORANGE', 2008]=QTY['ORANGE' ,2007],
 11             QTY['FRUITS', 2008]=QTY['APPLE', 2008]+QTY['ORANGE', 2008])
 12  ORDER BY COUNTRY, PRODUCT, YEAR, QTY;

COUNT PRODUCT          YEAR        QTY
----- ---------- ---------- ----------
KOR   APPLE            2006        200
KOR   APPLE            2007        100
KOR   APPLE            2008        300
KOR   FRUITS           2008        470
KOR   ORANGE           2006        150
KOR   ORANGE           2007        170
KOR   ORANGE           2008        170
```

RULES 구문 뒤의 세 번째 규칙은 반드시 첫 번째 및 두 번째 규칙이 먼저 수행되어 셀이 만들어져야 수행 가능하다는 것을 알 수가 있다. RULES 구문 뒤에 키워드가 생략되면 SEQUENTIAL ORDER가 지정 된 것으로 규칙을 순차적으로 수행하게 된다. 그러나, 규칙의 순서를 다음과 같이 변경하게 되면 첫 번째 규칙에 의한 수행 결과는 출력되지 않는다.

```
SQL> SELECT COUNTRY, PRODUCT, YEAR, QTY
  2  FROM SALES
  3  WHERE COUNTRY='KOR'
  4     MODEL
  5     PARTITION BY (COUNTRY)
  6     DIMENSION BY (PRODUCT, YEAR)
  7     MEASURES (QTY)
  8     RULES (
  9             QTY['FRUITS', 2008]=QTY['APPLE', 2008]+QTY['ORANGE', 2008],
 10             QTY['APPLE', 2008]=QTY['APPLE' ,2007]+QTY['APPLE', 2006],
 11             QTY['ORANGE', 2008]=QTY['ORANGE' ,2007])
 12  ORDER BY COUNTRY, PRODUCT, YEAR, QTY;

COUNT PRODUCT          YEAR        QTY
----- ---------- ---------- ----------
KOR   APPLE            2006        200
KOR   APPLE            2007        100
KOR   APPLE            2008        300
KOR   FRUITS           2008
KOR   ORANGE           2006        150
KOR   ORANGE           2007        170
KOR   ORANGE           2008        170
```

이런 경우, RULES 뒤에 명시적으로 AUTOMATIC ORDER 구문을 기술해주면, 오라클 데이터베이스가 규칙들 중에 우선적으로 참조해야 할 규칙을 인식하여 먼저 수행하게 된다.

```
SQL> SELECT COUNTRY, PRODUCT, YEAR, QTY
  2  FROM SALES
  3  WHERE COUNTRY='KOR'
  4     MODEL
  5     PARTITION BY (COUNTRY)
  6     DIMENSION BY (PRODUCT, YEAR)
  7     MEASURES (QTY)
  8     RULES AUTOMATIC ORDER(
  9             QTY['FRUITS', 2008]=QTY['APPLE', 2008]+QTY['ORANGE', 2008],
 10             QTY['APPLE', 2008]=QTY['APPLE' ,2007]+QTY['APPLE', 2006],
 11             QTY['ORANGE', 2008]=QTY['ORANGE' ,2007])
 12  ORDER BY COUNTRY, PRODUCT, YEAR, QTY;

COUNT PRODUCT          YEAR        QTY
----- ---------- ---------- ----------
KOR   APPLE            2006        200
KOR   APPLE            2007        100
KOR   APPLE            2008        300
KOR   FRUITS           2008        470
KOR   ORANGE           2006        150
KOR   ORANGE           2007        170
KOR   ORANGE           2008        170
```

DROP TABLE ... PURGE

Oracle Database 10g에서는 테이블 삭제와 관련하여 새로운 기능을 추가하였다. 테이블을 삭제하면 해당 테이블과 연관된 저장 공간은 즉시 해제 되지 않으며, 해당 테이블의 이름이 변경되어 휴지통(Recycle bin)에 저장된다. 이후, 삭제된 테이블을 복구하고자 하는 경우에는 FLASHBACK TABLE 명령을 사용하면 된다. 그러나, 테이블을 삭제함과 동시에 관련 저장 공간을 즉시 해제하고자 하는 경우에는 DROP TABLE 명령에 PURGE를 추가하면 된다.

즉, PURGE를 지정하면 한 번의 명령으로 해당 테이블의 삭제 및 관련 저장 공간을 즉시 해제하게 되며, 휴지통에 해당 테이블과 관련된 객체를 저장하지 않는다.

이 구문을 사용하는 것은 테이블을 삭제하고, 휴지통에서 해당 테이블을 제거하는 것과 동일한 작업이 되므로 한 번의 명령으로 이와 같은 절차를 완료 할 수 있다. 또한, 중요한 정보가 저장 되어 있는 테이블을 삭제하는 경우에는 반드시 PURGE 구문을 추가하여 휴지통에 저장되지 않도록하는 편이 바람직하다.

다음과 같이 테이블을 삭제한 다음, 다시 복구해보자.

```
SQL> DROP TABLE EMP;

테이블이 삭제되었습니다.

SQL> FLASHBACK TABLE EMP TO BEFORE DROP;

플래시백이 완료되었습니다.

SQL> SELECT COUNT(*) FROM EMP;

  COUNT(*)
----------
        14

SQL> CREATE TABLE EMP2
  2  AS SELECT * FROM EMP;

테이블이 생성되었습니다.

SQL> DROP TABLE EMP2 PURGE;

테이블이 삭제되었습니다.

SQL> FLASHBACK TABLE EMP2 TO BEFORE DROP;
FLASHBACK TABLE EMP2 TO BEFORE DROP
*
1행에 오류:
ORA-38305: 객체가 RECYCLE BIN에 없음
```

FLASHBACK TABLE

Oracle Database 10g에서는 FLASHBACK TABLE이라는 새로운 DDL 명령을 추가하였다. 이 명령은 해당 테이블이 삭제되거나 변경되었을 때, 해당 테이블을 변경 전의 과거 시점으로 복원시켜준다. 즉, FLASHBACK TABLE 명령은 자체 서비스 복구 도구로서 테이블과 인덱스 또는 뷰와 같은 해당 테이블 관련 속성들의 데이터를 복원 할 수 있게 해주며, 이 작업은 데이터베이스가 열려 있는 상태에서 해당 테이블에 발생한 변경사항을 반대로 적용(Rollback)하는 원리로 동작한다. 전통적인 복구 메커니즘과 비교 할 때, 이 기능은 용이성, 가용성, 신속성 측면에서 큰 장점을 제공할뿐더러 DBA의 부담을 크게 줄여주게 된다. 단, 이 기능은 디스크의 물리적 오류로 인하여 발생한 손실은 복원 할 수 없다.

FLASHBACK TABLE의 구문은 다음과 같다.

```
FLASHBACK TABLE[schema.]table[, [schema.]table]...
TO {TIMESTAMP | SCN} expr
```

FLASHBACK TABLE 명령은 1개 이상의 테이블에 대하여 수행 할 수 있으며 서로 다른 스키마 내의 테이블에도 실행 할 수 있다. 사용자가 복원하고자 원하는 시간을 지정 할 수 있으며, 복원되는 테이블에 설정된 트리거는 모두 비활성화 되지만, ENABLE TRIGGER 구문을 지정하면 디폴트 설정을 변경 할 수도 있다.

다음은 테이블을 삭제하고, FLASHBACK TABLE 명령으로 복구하는 방법이다.

```
SQL> DROP TABLE EMP;

테이블이 삭제되었습니다.

SQL> SELECT ORIGINAL_NAME, OPERATION, DROPTIME
  2  FROM RECYCLEBIN;

ORIGINAL_NAME                  OPERATION DROPTIME
------------------------------ --------- -------------------
PK_EMP                         DROP      2007-05-23:14:52:11
EMP                            DROP      2007-05-23:14:52:14

SQL> FLASHBACK TABLE EMP TO BEFORE DROP;

플래시백이 완료되었습니다.

SQL> SELECT COUNT(*) FROM EMP;

  COUNT(*)
----------
        14
```

휴지통(Recycle bin)은 삭제된 객체에 대한 정보를 포함하는 실제 데이터 딕셔너리 테이블이며, 실제로 삭제된 테이블과 인덱스, 제약조건, 중첩된 테이블 등의 연관 객체는 데이터베이스에서 제거되지 않고 여전히 저장 공간을 점유하고 있게 된다. 이러한 객체들은 휴지통에서 삭제되지 않는 한, 사용자의 할당량 계산에도 포함된다.

각 사용자는 자유롭게 휴지통을 검색 할 수 있지만, SYSDBA 권한을 부여 받지 않는 한, 자신이 소유한 객체에 대해서만 접근 할 수 있다. 사용자는 다음과 같은 문장으로 휴지통에서 자신의 객체를 검색 할 수 있다.

```
SELECT * FROM RECYCLEBIN;
```

사용자가 삭제되는 경우에는 해당 사용자가 소유하고 있는 모든 객체가 휴지통으로 이동하지 않으며, 휴지통에 있던 어떤 객체도 삭제되지 않는다. 만약, 휴지통에 저장된 객체를 영구히 삭제하고자 하는 경우에 다음과 같은 명령을 실행하면 된다.

```
PURGE RECYCLEBIN;
```

FLASHBACK VERSION QUERY

테이블의 데이터가 잘못 변경된 경우, 플래시백 쿼리를 사용하면 이러한 변경 사항을 쉽게 검색 할 수 있다. 즉, 플래시백 쿼리를 사용하면 과거 특정 시점에서 테이블의 행에 적용된 모든 변경사항을 검색 할 수 있다. 이 기능을 사용하려면 SELECT 문장 뒤에 VERSIONS 구문을 추가하고 검색하고자 하는 특정 과거 시점에 대한 SCN 또는 시간을 지정하면 된다.

그러면, 쿼리는 해당 데이터뿐만 아니라 해당 변경을 수행했던 트랜잭션에 대한 메타 데이터도 같이 리턴하게 된다.

참고로 SCN(System Change Number)이란 각 커밋된 트랜잭션에 대한 리두 레코드를 식별하기 위한 용도로 오라클 서버가 할당하는 번호이다.

잘못된 변경사항을 수행한 트랜잭션을 식별한 후에는 플래시백 트랜잭션 쿼리를 이용하여 해당 트랜잭션에 의해 발생한 모든 변경 사항을 검색 할 수도 있으며, 플래시백 테이블 기능을 이용하면 해당 트랜잭션에 의한 변경사항이 발생하기 전 상태로 테이블을 복원 할 수도 있다.

테이블을 검색하는 쿼리에 VERSIONS 구문을 추가하면 해당 테이블에 대하여 현재 시간으로부터 UNDO_RETENTION 파라메터에 지정된 시간(초)까지 발생했던 모든 변경 사항을 표시 할 수가 있다. VERSIONS 구문이 추가된 쿼리를 플래시백 버전 쿼리라고 부르며, 해당 구문은 WHERE 구문처럼 동작하게 된다.

예를 들어, 다음과 같이 7934번 사원의 급여가 1300이었으나, 해당 사원의 급여를 30% 증가시키고 커밋을 수행하였다. 이런 경우, 플래시백 버전 쿼리를 이용하여 해당 사원의 변경 전 및 변경 후의 급여를 표시 할 수 있다.

```
SQL> SELECT SAL FROM EMP
  2  WHERE EMPNO=7934;

       SAL
----------
      1300

SQL> UPDATE EMP
  2  SET SAL=SAL*1.30
  3  WHERE EMPNO=7934;

1 행이 갱신되었습니다.

SQL> COMMIT;

커밋이 완료되었습니다.

SQL> SELECT SAL FROM EMP
  2  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
  3  WHERE EMPNO=7934;

       SAL
----------
      1690
      1300
```

VERSIONS 구문은 쿼리의 실행 계획을 변경시키지 않는다. 예를 들어, 인덱스를 이용하는 쿼리를 실행하는 경우, VERSIONS 구문이 추가되더라도 실행 계획은 변경되지 않으므로 해당 쿼리는 여전히 인덱스를 이용하여 검색을 수행하게 된다. 플래시백 버전 쿼리에 의해 리

턴되는 행들의 버전은 모든 트랜잭션에 걸쳐 변경된 버전이며, VERSIONS 구문이 추가된 쿼리는 현재 진행 중인 트랜잭션의 특성에 어떠한 영향도 미치지 않는다.

플래시백 버전 쿼리의 디폴트 VERSIONS 구문은 VERSIONS BETWEEN {SCN|TIMESTAMP} MINVALUE AND MAXVALUE이며, 쿼리에만 사용 할 수 있지만, DML 및 DDL의 서브쿼리에는 VERSIONS 구문을 사용 할 수도 있다. 플래시백 버전 쿼리는 선택된 행에 대하여 커밋이 완료된 버전만을 리턴하며, 현재 진행중인 트랜잭션에 의해 발생한 변경사항은 리턴하지 않는다. 플래시백 버전 쿼리는 해당 행의 모든 구체화(Incarnation) 번호를 리턴하며, 이것은 모든 리턴되는 버전에는 삭제된 행 및 새롭게 입력된 행들의 모든 버전이 포함된다는 뜻이다.

플래시백 버전 쿼리에 의한 행 접근 방식은 다음 두 종류 중의 하나로 정의 된다.
- ROWID 기반 행 접근 : 이 방식의 경우, 특정 ROWID의 모든 버전이 해당 행의 내용과 상관 없이 리턴된다. 즉, ROWID에 의해 참조되는 블록에 위치한 슬롯 내의 모든 버전이 리턴됨을 의미한다.
- 모든 행 접근 : 이 방식의 경우, 해당 행의 모든 버전이 리턴된다.

VERSIONS BETWEEN 구문을 사용하면 쿼리가 실행된 시간과 특정 과거 시점 사이에서 테이블에 현재 존재하는 행 및 현재 존재하지 않는 행들에 대한 모든 버전이 리턴된다. 만약, UNDO_RETENTION 파라메터에 설정된 값이 BETWEEN 구문에 지정된 최소 시간 또는 SCN 보다 작다면, 해당 UNDO_RETENTION 파라메터에 설정된 값까지의 버전들만 리턴 된다. 여기서, BETWEEN 구문의 범위는 SCN 또는 시간으로 지정 할 수 있다.

아래 예제는 7934번 사원의 급여 변경 사항을 리턴한다. END_DATE 컬럼의 NULL 값은 쿼리를 실행하는 현재 시점에 존재하는 행을 의미하며, START_DATE 컬럼의 NULL 값은 UNDO_RETENTION 파라메터에 설정된 값 이전에 생성된 버전임을 나타낸다.

```
SQL> UPDATE EMP
  2  SET SAL=SAL*1.30
  3  WHERE EMPNO=7934;

1 행이 갱신되었습니다.

SQL> COMMIT;

SQL> SELECT VERSIONS_STARTTIME "START_DATE", VERSIONS_ENDTIME "END_DATE", SAL
  2  FROM EMP
  3  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
  4  WHERE EMPNO=7934;

START_DATE                    END_DATE                           SAL
----------------------------- ------------------------------ ----------
07/05/23 16:23:48                                                2197
                              07/05/23 16:23:48                  1690
```

복습

다음 문장을 실행하여 실습에 사용할 테이블을 작성한다.

```
SQL> CREATE TABLE DEPT_JOB
  2  AS
  3  SELECT DISTINCT JOB FROM EMP;
```

1. 사원 테이블에서 부서별 직급별 급여 합계를 다음과 같이 출력하시오.

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

2. DEPT_JOB 테이블과 사원 테이블을 파티션 OUTER 조인하여 다음과 같이 출력하시오.

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

3. MODEL 구문을 이용하여 부서별 직급별 급여 합계에 각 부서별 'CLERK'의 평균을 계산하여 40번 'CLERK'의 결과로 추가하라

```
    DEPTNO JOB          SUM_SAL
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        20 ANALYST         6000
        20 CLERK           1900
        20 MANAGER         2975
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600
        40 CLERK     1383.33333
```

4. MODEL 구문을 이용하여 부서별 직급별 급여 합계에 각 부서별 급여 합계의 평균을 추가하라.

```
    DEPTNO JOB          SUM_SAL
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        10 평균      2916.66667
        20 ANALYST         6000
        20 CLERK           1900
        20 MANAGER         2975
        20 평균            3625
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600
        30 평균      3133.33333
```

5. 다음과 같이 DROP TABLE 명령으로 EMP 테이블을 삭제하였다. EMP 테이블을 복구하는 명령을 작성하시오.

```
SQL> DROP TABLE EMP;

테이블이 삭제되었습니다.
```

Chapter 23. Oracle 10g의 정규 표현식

이번 장에서는 Oracle Database 10g에서 새롭게 추가된 정규 표현식에 대해서 설명한다.

개요

Oracle Database 10g에서는 정규 표현식을 새롭게 지원한다. 오라클에 의해서 구현된 정규 표현식은 ASCII 데이터의 검색에 있어서 IEEE에 의해 주관되는 POSIX(Operating System for UNIX) 표준과 호환된다. 뿐만 아니라, 오라클의 다국어 기능으로 인하여 POSIX 표준 이상으로 문자열 매칭 기능을 확장시킬 수 있다. 즉, 정규 표현식은 문자열의 검색 및 조작에 있어서 단순 및 복잡한 패턴을 모두 사용 할 수 있는 방법을 제공해준다.

문자열 검색과 조작은 웹 기반 애플리케이션 내의 많은 로직에서 활용되며, 지정된 문자열에서 “San Francisco”와 같은 간단한 구문을 검색하는 단순한 기능부터 URL을 추출하는 복잡한 검색도 수행 할 수 있다. 또는, 두 번째 글자가 모음인 모든 단어를 검색 할 수도 있다.

이러한 정규 표현식을 기존 SQL과 결합하면 오라클 데이터베이스에 저장된 데이터의 검색 및 조작 작업을 매우 강력하게 수행 할 수 있으므로, 기존의 복잡하게 구현된 프로그램에서 수행되던 작업을 간단하게 SQL에서 처리 할 수가 있다.

메타 문자의 사용

메타 문자는 와일드카드 문자, 반복 문자, 비일치 문자, 범위 문자와 같이 특별한 의미를 갖는 문자이며, 이러한 사전에 정의된 문자를 이용하여 패턴 매칭을 수행 할 수 있다. 다음과 같은 경우에 대하여 검색을 수행해야 한다고 가정하자.

Case 1 : 문자열이 'abc'인 문자열을 검색하라
· 메타 문자열 : 'abc'
· 일치 문자열 : 'abc'
· 일치 되지 않는 문자열 : 'def'

첫 번째의 경우에서는 단일 일치 작업이 수행되었다.

Case 2 : 문자열에서 'a' 문자 다음에 어떠한 문자라도 상관없이 한 개의 문자가 오고, 그 다음에 'c' 문자가 오는 문자열을 검색하라.
· 메타 문자열 : 'a.c'
· 일치 문자열 : 'abc'
'adc'
'alc'

'a&c'
· 일치 되지 않는 문자열 : 'abb'

두 번째의 경우에서는 '.'으로 정의되는 ANY 메타 문자가 사용되었다.

Case 3 : 문자열에서 'a' 문자가 반드시 한 번 이상 반복되는 문자열을 검색하라.
· 메타 문자열 : 'a+'
· 일치 문자열 : 'a'
'aa'
· 일치 되지 않는 문자열 : 'bbb'

세 번째의 경우에서 '+' 메타 문자는 기호 앞의 문자가 반드시 1회 이상 반복되어야 함을 의미한다.

또한, 비일치 문자를 사용하여 검색을 수행 할 수도 있는데, 비일치 문자란 해당 문자가 지정된 문자열에서 발견되지 말아야 한다. 예를 들어, 문자열이 'a', 'b', 'c'의 문자로만 구성되지 않는 문자열을 검색하려면 다음과 같이 비일치 문자인 '^'를 사용하면 된다.
· 메타 문자열 : '[^abc]'
· 일치 문자열 : 'abcdef'
'ghi'
· 일치 되지 않는 문자열 : 'abc'

'a'에서 'i'까지의 어떠한 문자라도 구성되지 않은 문자열을 검색하는 경우에는 다음과 같이 비일치 문자를 사용한다.
· 메타 문자열 : '[^a-i]'
· 일치 문자열 : 'hijk'
'lmn'
· 일치 되지 않는 문자열 : 'abcdefghi'

다음은 정규 표현식에서 사용 가능한 메타 문자이다.

표 23-1. 메타 문자

메타 문자	연산자 이름	설명
.	임의 문자 1개	임의 문자 1개와 일치해야 한다.
+	한 개 이상의 임의 문자	메타 문자 앞의 문자가 한 개 이상 일치해야 한다.
?	0개 또는 1개의 임의 문자	메타 문자 앞의 문자가 0개 또는 1개 일치해야 한다.
*	0개 이상의 임의 문자	메타 문자 앞의 문자가 0개 이상 일치해야 한다.
{m} {m,} {m,n}	반복되는 문자의 개수	메타 문자 앞의 문자가 m개 반복되어야 한다. 최소 m개 반복되어야 한다. 최소 m개에서 최대 n개 반복되어야 한다.
[...]	일치 문자 목록	문자가 목록에 있는 문자와 일치해야 한다.
[^...]	비일치 문자 목록	문자가 목록에 있는 문자와 일치하지 말아야 한다.
|	OR	'a|b'는 'a' 또는 'b'가 일치해야 함을 의미한다.
(...)	그룹 문자	괄호 안의 문자열을 하나의 문자로 처리한다.
₩n	역 참조	n은 1~9까지의 정수이며, n번째 앞의 문자를 참조한다.
₩	에스케이프(Escape)	다음 문자를 리터럴로 처리한다.
^	라인의 선두 문자	다음 문자가 반드시 라인의 선두에 있어야 한다.
$	라인의 마지막 문자	앞 문자가 반드시 라인의 마지막에 있어야 한다.
[:*class*:]	POSIX 문자 클래스	문자가 *class*에 포함된 문자이어야 한다.

정규 표현식 함수

Oracle Database 10g는 정규 표현식을 사용하여 문자열을 검색 및 조작 할 수 있는 SQL 함수를 제공하며, CHAR, NCHAR, NCLOB, NVARCHAR2, VARCHAR2와 같은 문자 데이터 타입에 사용 할 수 있다. 정규 표현식은 ' '로 싸주어야 SQL 함수에 의해 전체 표현식으로 판단하게 되며, 코드의 가독성도 향상된다.

- REGEXP_LIKE : 이 함수는 문자 컬럼에서 주어진 패턴에 일치하는 문자열을 검색한다. 쿼리의 WHERE 구문에서 이 함수를 사용하면 지정된 정규 표현식에 일치하는 행을 리턴한다.
- REGEXP_REPLACE : 이 함수는 문자 컬럼에서 주어진 패턴에 일치하는 문자를 검색하고, 해당 패턴에 일치하는 문자가 발견 될 때마다 지정된 패턴으로 교체한다.
- REGEXP_INSTR : 이 함수는 주어진 정규 표현식에 일치하는 문자열을 검색하고, 해당 문자의 위치를 정수값으로 리턴한다.
- REGEXP_SUBSTR : 이 함수는 지정한 정규 표현식에 일치하는 일부 문자열을 리턴한다.

REGEXP 함수의 사용 방법

REGEXP 함수의 사용 방법은 다음과 같다.

```
REGEXP_LIKE(srcstr, pattern [, match_option])
REGEXP_INSTR(srcstr, pattern [, position [, occurrence
                [, return_option [, match_option]]]])
REGEXP_SUBSTR(srcstr, pattern [, position [,occurrence [, match_option]]])
REGEXP_REPLACE(srcstr, pattern [, replacestr [, position
                [, occurrence [, match_option]]]])
```

여기서,
· *srcstr* : 검색 할 문자열
· *pattern* : 정규 표현식
· *occurrence* : 지정된 패턴이 반복되는 경우, 몇 번째에 일치 되는지 지정
· *position* : 검색을 시작할 문자의 위치
· *return_option* : 지정된 패턴이 반복되는 경우, 검색이 시작되는 위치 지정
· *replacestr* : 패턴을 교체 할 문자열
· *match_option* : 일치 방식
 - 'c' : 대소문자 구별 (디폴트)
 - 'I' : 대소문자 구별하지 않음
 - 'n' : 임의 메타 문자와 일치
 - 'm' : 원본 문자열을 여러 줄의 문자열로 처리

기본 검색의 수행

다음은 REG_TEST 테이블에서 이름이 Steven 또는 Stephen인 행을 검색하는 쿼리이다.

```
SQL> CREATE TABLE REG_TEST
  2  (NO NUMBER(2),
  3  NAME VARCHAR2(20));

테이블이 생성되었습니다.

SQL> INSERT INTO REG_TEST VALUES(1, 'Steven');
SQL> INSERT INTO REG_TEST VALUES(2, 'Stephen');
SQL> COMMIT;

SQL> SELECT * FROM REG_TEST
  2  WHERE REGEXP_LIKE(NAME, '^Ste(v|ph)en$');

        NO NAME
---------- --------------------
         1 Steven
         2 Stephen
```

여기서, ^는 구문의 시작, $는 구문의 마지막, |는 OR을 의미한다.

패턴의 존재 여부 검색

아래 예제는 이름 컬럼에서 첫 번째로 검색되는 알파벳이 아닌 문자의 위치를 리턴한다.

```
SQL> INSERT INTO REG_TEST VALUES(3, 'Daechi 2 dong');
SQL> INSERT INTO REG_TEST VALUES(4, 'Unma APT 1-1403');
SQL> COMMIT;

SQL> SELECT NAME,
  2  REGEXP_INSTR(NAME, '[^[:alpha:]]')
  3  FROM REG_TEST
  4  WHERE REGEXP_INSTR(NAME, '[^[:alpha:]]')>1;

NAME                 REGEXP_INSTR(NAME,'[^[:ALPHA:]]')
-------------------- ---------------------------------
Daechi 2 dong                                        7
Unma APT 1-1403                                      5
```

여기서, [^[:alpha:]]는 다음과 같은 의미를 갖는다.

- [: 표현식의 시작
- ^ : NOT
- [:alpha:] : 알파벳
-] : 표현식의 끝

부분 문자열 추출 예제

아래 예제는 이름 컬럼에서 첫 번째 공백이 나타나기 전의 문자열을 리턴한다.

```
SQL> SELECT REGEXP_SUBSTR(NAME, '[^ ]+')
  2  FROM REG_TEST;

REGEXP_SUBSTR(NAME,'[^]+')
----------------------------------------
Steven
Stephen
Daechi
Unma
```

여기서, [^]+ 는 다음과 같은 의미를 갖는다.

- [: 표현식의 시작
- ^ : NOT
-   : 공백
-] : 표현식의 끝
- +: 1개 이상 반복

패턴 교체

아래 예제는 이름 컬럼의 각 문자 뒤에 공백을 추가한다.

```
SQL> SELECT REGEXP_REPLACE(NAME, '(.)', '₩1 ')
  2  FROM REG_TEST;

REGEXP_REPLACE(NAME,'(.)','₩1')
------------------------------------------------
S t e v e n
S t e p h e n
D a e c h i  2  d o n g
U n m a  A P T  1 - 1 4 0 3
```

정규 표현식과 CHECK 제약조건

CHECK 제약조건에 정규 표현식을 사용 할 수도 있다. 다음은 이메일을 저장 할 컬럼을 추가하고, CHECK 제약조건을 정의하는 예제이다.

```
SQL> ALTER TABLE REG_TEST
  2  ADD (EMAIL VARCHAR2(20));

테이블이 변경되었습니다.

SQL> ALTER TABLE REG_TEST
  2  ADD CONSTRAINT REG_TEST_EMAIL_CK
  3  CHECK (REGEXP_LIKE(EMAIL, '@')) NOVALIDATE;

테이블이 변경되었습니다.

SQL> INSERT INTO REG_TEST VALUES(5, 'Kim', 'kim@xyz.net');

1 개의 행이 만들어졌습니다.

SQL> INSERT INTO REG_TEST VALUES(6, 'Park', 'park.xyz.net');
INSERT INTO REG_TEST VALUES(6, 'Park', 'park.xyz.net')
*
1행에 오류:
ORA-02290: 체크 제약조건(SCOTT.REG_TEST_EMAIL_CK)이 위배되었습니다
```

복습

1. 사원 테이블에서 사원명에 'A' 또는 'I'가 포함된 사원의 이름을 출력하시오.

```
ENAME
----------
SMITH
ALLEN
WARD
MARTIN
BLAKE
CLARK
KING
ADAMS
JAMES
MILLER
```

2. 사원 테이블에서 사원명에 'L'이 한번 이상 포함된 사원의 이름을 출력하시오.

```
ENAME
----------
ALLEN
MILLER
```

3. 사원 테이블에서 사원명이 'S'로 시작되거나 끝나는 사원의 이름을 출력하시오.

```
ENAME
----------
SMITH
JONES
SCOTT
ADAMS
JAMES
```

부록. 정답

1장

1. 개체, 속성, 관계
2. 요구형성 및 분석, 개념적 설계, 논리적 설계, 물리적 설계
3. 개체, 속성, 관계를 도식적으로 표현한 것
4. 기본키는 테이블내에서 각각의 행들을 유일하게 식별할 수 있는 컬럼이며, 외래키는 다른 테이블의 기본키를 참조하는 컬럼이다.

2장

```
1. SELECT * FROM TAB;
2. DESC
3. SELECT EMPNO, ENAME, SAL*12 "연봉"
   FROM EMP;
4. b, d
5. SELECT ENAME||'의 업무는 '||JOB||'이고 급여는 '||SAL||'만원입니다'
   FROM EMP;
```

3장

```
1. SELECT EMPNO, ENAME, HIREDATE, JOB, SAL
   FROM EMP
   WHERE HIREDATE BETWEEN '81/01/01' AND '81/12/31';
2. SELECT EMPNO, ENAME, HIREDATE, JOB, SAL
   FROM EMP
   WHERE HIREDATE BETWEEN '81/01/01' AND '81/12/31'
   AND JOB <> 'SALESMAN';
3. SELECT EMPNO, ENAME, HIREDATE, JOB, SAL
   FROM EMP
   ORDER BY SAL DESC, HIREDATE;
4. SELECT EMPNO, ENAME
   FROM EMP
   WHERE ENAME LIKE '___N%';
5. SELECT EMPNO, ENAME, SAL*12 "연봉"
   FROM EMP
   WHERE SAL*12 >= 35000;
```

4장

```
1. SELECT SUBSTR(ENAME, 2, 3)
   FROM EMP;
2. SELECT EMPNO, ENAME, HIREDATE
   FROM EMP
   WHERE TO_CHAR(HIREDATE, 'MM') = '12';
3. SELECT EMPNO, ENAME, LPAD(SAL, 10, '*') "급여"
   FROM EMP;
4. SELECT EMPNO, ENAME, TO_CHAR(HIREDATE, 'YYYY-MM-DD') "입사일"
   FROM EMP;
5. SELECT EMPNO, ENAME,
   CASE WHEN SAL >= 0 AND SAL <=1000 THEN 'E'
         WHEN SAL > 1000 AND SAL <=2000 THEN 'D'
         WHEN SAL > 2000 AND SAL <=3000 THEN 'C'
         WHEN SAL > 3000 AND SAL <=4000 THEN 'B'
         WHEN SAL > 4000 AND SAL <=5000 THEN 'A'
   END "등급"
   FROM EMP;

SELECT EMPNO, ENAME, SAL,
        DECODE(SIGN(SAL), -1, NULL,
```

```
        DECODE(SIGN(SAL-1001), -1, 'E',
        DECODE(SIGN(SAL-2001), -1, 'D',
        DECODE(SIGN(SAL-3001), -1, 'C',
        DECODE(SIGN(SAL-4001), -1, 'B',
        DECODE(SIGN(SAL-5001), -1, 'A')))))) "등급"
FROM EMP;
```

5장

```
1. SELECT EMP.EMPNO, EMP.ENAME, EMP.DEPTNO, DEPT.DNAME
   FROM DEPT, EMP
   WHERE DEPT.DEPTNO = EMP.DEPTNO
   ORDER BY EMP.ENAME;
2. SELECT EMP.EMPNO, EMP.ENAME, EMP.SAL, DEPT.DNAME
   FROM DEPT, EMP
   WHERE DEPT.DEPTNO = EMP.DEPTNO
   AND EMP.SAL >= 2000
   ORDER BY SAL DESC;
3. SELECT EMP.EMPNO, EMP.ENAME, EMP.JOB, EMP.SAL, DEPT.DNAME
   FROM DEPT, EMP
   WHERE DEPT.DEPTNO = EMP.DEPTNO
   AND EMP.JOB = 'MANAGER'
   AND EMP.SAL >= 2500
   ORDER BY EMP.EMPNO;
4. SELECT EMP.EMPNO, EMP.ENAME, EMP.SAL, SALGRADE.GRADE
   FROM EMP, SALGRADE
   WHERE EMP.SAL BETWEEN SALGRADE.LOSAL AND SALGRADE.HISAL
   AND SALGRADE.GRADE = 4
   ORDER BY EMP.SAL DESC;
5. SELECT EMP.EMPNO, EMP.ENAME, DEPT.DNAME, EMP.SAL, SALGRADE.GRADE
   FROM DEPT, EMP, SALGRADE
   WHERE DEPT.DEPTNO = EMP.DEPTNO
   AND EMP.SAL BETWEEN SALGRADE.LOSAL AND SALGRADE.HISAL
   ORDER BY SALGRADE.GRADE DESC;
6. SELECT W.ENAME, M.ENAME
   FROM EMP W, EMP M
   WHERE W.MGR=M.EMPNO;
7. SELECT W.ENAME, M1.ENAME, M2.ENAME
   FROM EMP W, EMP M1, EMP M2
   WHERE W.MGR = M1.EMPNO
   AND M1.MGR = M2.EMPNO;
8. SELECT W.ENAME, M1.ENAME, M2.ENAME
   FROM EMP W, EMP M1, EMP M2
   WHERE W.MGR = M1.EMPNO(+)
   AND M1.MGR = M2.EMPNO(+);
```

6장

```
1. SELECT MAX(SAL), MIN(SAL), SUM(SAL), AVG(SAL)
   FROM EMP;
2. SELECT DEPTNO, COUNT(*)
   FROM EMP
   GROUP BY DEPTNO;
3. SELECT DEPTNO
   FROM EMP
   GROUP BY DEPTNO
   HAVING COUNT(*) >= 6;
4. SELECT T1.ENAME, COUNT(T2.ENAME)+1 "등수"
   FROM EMP T1, EMP T2
   WHERE T1.SAL < T2.SAL(+)
   GROUP BY T1.ENAME
```

```
   ORDER BY 2;
5. SELECT DEPT.DNAME,
          SUM(DECODE(JOB, 'CLERK', SAL)) CLERK,
          SUM(DECODE(JOB, 'MANAGER', SAL)) MANAGER,
          SUM(DECODE(JOB, 'PRESIDENT', SAL)) PRESIDENT,
          SUM(DECODE(JOB, 'ANALYST', SAL)) ANALYST,
          SUM(DECODE(JOB, 'SALESMAN', SAL)) SALESMAN
   FROM DEPT, EMP
   WHERE DEPT.DEPTNO = EMP.DEPTNO
   GROUP BY DEPT.DNAME;

   SELECT DEPT.DNAME,
            T1.CLERK,
            T1.MANAGER,
            T1.PRESIDENT,
            T1.ANALYST,
            T1.SALESMAN
   FROM DEPT, (SELECT DEPTNO,
                          SUM(DECODE(JOB, 'CLERK', SAL)) CLERK,
                          SUM(DECODE(JOB, 'MANAGER', SAL)) MANAGER,
                          SUM(DECODE(JOB, 'PRESIDENT', SAL)) PRESIDENT,
                          SUM(DECODE(JOB, 'ANALYST', SAL)) ANALYST,
                          SUM(DECODE(JOB, 'SALESMAN', SAL)) SALESMAN
                FROM EMP
                GROUP BY DEPTNO) T1
   WHERE DEPT.DEPTNO = T1.DEPTNO;
```

7장

```
1. SELECT EMPNO, ENAME, SAL
   FROM EMP
   WHERE SAL > (SELECT SAL
                  FROM EMP
                  WHERE ENAME = 'BLAKE');
2. SELECT EMPNO, ENAME, HIREDATE
   FROM EMP
   WHERE HIREDATE > (SELECT HIREDATE
                         FROM EMP
                         WHERE ENAME = 'MILLER');
3. SELECT EMPNO, ENAME, SAL
   FROM EMP
   WHERE SAL > (SELECT AVG(SAL)
                  FROM EMP);
4. SELECT EMPNO, ENAME, SAL
   FROM EMP
   WHERE JOB = (SELECT JOB
                  FROM EMP
                  WHERE ENAME = 'CLARK')
   AND SAL > (SELECT SAL
                FROM EMP
                WHERE EMPNO = 7698);
5. SELECT EMPNO, ENAME, DEPTNO, SAL
   FROM EMP
   WHERE SAL IN (SELECT MAX(SAL)
                   FROM EMP
                   GROUP BY DEPTNO)
```

8장

```
SET FEEDBACK OFF
TTITLE '사원|보고서'
```

```
BTITLE '대외비'
BREAK ON job
COLUMN job HEADING '업무'
COLUMN ename HEADING '사원명' FORMAT L99,999.99
COLUMN sal HEADING '급여' FORMAT L99,999.99
SELECT ENAME, JOB, SAL
FROM EMP
ORDER BY JOB
/
SET FEEDBACK ON
COLUMN job CLEAR
COLUMN ename CLEAR
COLUMN sal CLEAR
CLEAR BREAK
```

9장

```
1. INSERT INTO DEPT VALUES (99, '관리과', '대구');
2. UPDATE DEPT SET ENAME = '회계과' WHERE EMPNO = 99;
3. DELETE FROM DEPT WHERE EMPNO = 99;
4. a, b, d
5. ROLLBACK, COMMIT
```

10장

```
1. CREATE TABLE JUSO
  (NO NUMBER(3),
   NAME VARCHAR2(10),
   ADDR VARCHAR2(20),
   EMAIL VARCHAR2(5));
2. ALTER TABLE JUSO
   ADD (PHONE VARCHAR2(10));
3. ALTER TABLE JUSO
   MODIFY EMAIL VARCHAR2(20);
4. ALTER TABLE JUSO
   DROP COLUMN ADDR;
5. DROP TABLE JUSO;
```

11장

```
CREATE TABLE CUSTOMERS
(CNO VARCHAR2(5) CONSTRAINT CUSTOMERS_CNO_PK PRIMARY KEY,
CNAME VARCHAR2(10),
ADDRESS VARCHAR2(50),
EMAIL VARCHAR2(20),
PHONE VARCHAR2(20));

CREATE TABLE ORDERS
(ORDERNO VARCHAR2(10) CONSTRAINT ORDERS_ORDERNO_PK PRIMARY KEY,
ADDRESS VARCHAR2(50),
PHONE VARCHAR2(20),
STATUS VARCHAR2(5),
CNO VARCHAR2(5) CONSTRAINT ORDERS_CNO_FK
        REFERENCES CUSTOMERS(CNO));

CREATE TABLE PRODUCTS
(PNO VARCHAR2(5) CONSTRAINT PRODUCTS_PNO_PK PRIMARY KEY,
PNAME VARCHAR2(20),
COST NUMBER(8),
STOCK NUMBER(5));
```

```
CREATE TABLE ORDERDETAIL
(ORDERNO VARCHAR2(5) CONSTRAINT ORDERDETAIL_ORDERNO_FK
        REFERENCES ORDERS(ORDERNO),
PNO VARCHAR2(5) CONSTRAINT ORDERDETAIL_PNO_FK
        REFERENCES PRODUCTS(PNO),
QTY NUMBER(5),
COST NUMBER(8),
CONSTRAINT ORDERDETAIL_PK PRIMARY KEY (ORDERNO, PNO));
```

12장

```
1. CREATE VIEW DNAME_ENAME_VU
   AS SELECT D.DNAME, E.ENAME
   FROM DEPT D, EMP E
   WHERE D.DEPTNO=E.DEPTNO;
2. CREATE VIEW WORKER_MANAGER_VU
   AS SELECT W.ENAME MNAME, M.ENAME ENAME
   FROM EMP W, EMP M
   WHERE W.MGR=M.EMPNO;
3. SELECT EMPNO, ENAME, HIREDATE FROM EMP
   ORDER BY HIREDATE DESC;
4. SELECT ROWNUM, EMPNO, ENAME, HIREDATE
   FROM (SELECT EMPNO, ENAME, HIREDATE FROM EMP
   ORDER BY HIREDATE DESC)
   WHERE ROWNUM <= 5;
5. SELECT RN, EMPNO, ENAME, HIREDATE
   FROM (SELECT ROWNUM RN, EMPNO, ENAME, HIREDATE
   FROM (SELECT EMPNO, ENAME, HIREDATE FROM EMP
   ORDER BY HIREDATE DESC))
   WHERE RN BETWEEN 6 AND 10;
```

13장

```
1. CREATE SEQUENCE TEST_SEQ
   INCREMENT BY 1
   START WITH 1
   MINVALUE 1
   MAXVALUE 9999
   NOCYCLE
   CACHE 10;
2. CREATE INDEX EMP_DEPTNO_IDX ON EMP(DEPTNO);
3. CREATE INDEX EMP_IDX ON EMP(ENAME, JOB);
4. CREATE SYNONYM D FOR DEPT;
5. CONNECT / AS SYSDBA
   CREATE PUBLIC SYNONYM E FOR SCOTT.EMP;
```

14장

```
1. CREATE USER KIM
   IDENTIFIED BY LION;
2. GRANT CREATE SESSION, CREATE TABLE TO KIM;
3. GRANT SELECT ON SCOTT.DEPT TO KIM;
   GRANT SELECT ON SCOTT.EMP TO KIM;
4. GRANT UPDATE (SAL, COMM) ON SCOTT.EMP TO KIM;
5. REVOKE UPDATE ON SCOTT.EMP FROM KIM;
```

15장

```
1. B
2. D
3. C
```

16장

```
1. SELECT TZ_OFFSET('Canada/Yukon') FROM DUAL;
2. SELECT SESSIONTIMEZONE FROM DUAL;
3. SELECT ENAME, EXTRACT(YEAR FROM HIREDATE) 년도,
   EXTRACT(MONTH FROM HIREDATE) 월,
   EXTRACT(DAY FROM HIREDATE) 일
   FROM EMP
   WHERE ENAME = 'SMITH';
4. ALTER SESSION
   SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';
5. SELECT CURRENT_TIMESTAMP FROM DUAL;
```

17장

```
1. (a, b), (a), ()
2. (a, b), (a), (b), ()
3. (a, b, c, d, e), (a, b, c, d), (a), ()
4. (a, b, c, d, e), (a, b, c, d), (a, b, e), (c, d, e), (a, b), (c, d), (e), ()
5. SELECT JOB, DEPTNO, AVG(SAL)
   FROM EMP
   GROUP BY ROLLUP(JOB, DEPTNO);
```

18장

```
1. SELECT E.ENAME
   FROM EMP E
   WHERE E.DEPTNO = (SELECT D.DEPTNO
                          FROM DEPT D
                          WHERE D.DEPTNO = E.DEPTNO);
2. SELECT E.ENAME, E.JOB, E.SAL
   FROM EMP E
   WHERE E.SAL > (SELECT AVG(SAL)
                      FROM EMP
                      WHERE JOB = E.JOB);
3. SELECT E.ENAME, E.DEPTNO, E.SAL
   FROM EMP E
   WHERE E.SAL = (SELECT MAX(SAL)
                      FROM EMP
                      WHERE DEPTNO = E.DEPTNO);
4. UPDATE MASTER M
   SET M.NAME = (SELECT T.NAME
                      FROM TEMP T
                      WHERE T.ID = M.ID)
   WHERE EXISTS (SELECT 'X'
                      FROM TEMP T
                      WHERE T.ID = M.ID
                      AND T.NAME IS NOT NULL);
5. INSERT INTO MASTER
   SELECT T.ID, T.NAME
   FROM MASTER M, TEMP T
   WHERE M.ID(+) = T.ID
   AND M.ID IS NULL;
6. DELETE FROM MASTER M
   WHERE M.ID = (SELECT T.ID
                     FROM TEMP T
                     WHERE T.ID = M.ID
                     AND T.NAME IS NULL);
```

19장

```
1. SELECT LPAD(ENAME, LENGTH(ENAME)+(2*LEVEL)-2, ' ') NAME
```

```
   FROM EMP
   START WITH ENAME='JONES'
   CONNECT BY PRIOR EMPNO = MGR;
2. SELECT LPAD(ENAME, LENGTH(ENAME)+(2*LEVEL)-2, ' ') NAME
   FROM EMP
   START WITH ENAME='ADAMS'
   CONNECT BY PRIOR MGR = EMPNO;
3. SELECT LPAD(ENAME, LENGTH(ENAME)+(2*LEVEL)-2, ' ') NAME
   FROM EMP
   START WITH ENAME='KING'
   CONNECT BY PRIOR EMPNO = MGR
   AND ENAME != 'BLAKE';
```

20장

```
1. INSERT ALL
   WHEN JOB='ANALYST' THEN INTO ANALYST
   WHEN JOB='CLERK' THEN INTO CLERK
   WHEN JOB='MANAGER' THEN INTO MANAGER
   WHEN JOB='PRESIDENT' THEN INTO PRESIDENT
   WHEN JOB='SALESMAN' THEN INTO SALESMAN
   SELECT * FROM EMP;
2. INSERT ALL
   WHEN DEPTNO=10 AND SAL>2000 THEN INTO T10
   WHEN DEPTNO=20 AND SAL<1000 THEN INTO T20
   WHEN DEPTNO=30 AND SAL>2000 AND SAL<3000 THEN INTO T30
   SELECT * FROM EMP;
```

21장

```
1. SELECT EMPNO, ENAME, SAL,
   RANK() OVER (ORDER BY SAL DESC) 순위
   FROM EMP;
2. SELECT EMPNO, ENAME, SAL, JOB,
   RANK() OVER (PARTITION BY JOB ORDER BY SAL DESC) 업무별순위
   FROM EMP;
3. SELECT DEPTNO, MAX(SAL),
   RANK() OVER (ORDER BY MAX(SAL) DESC) 순위
   FROM EMP
   GROUP BY DEPTNO;
4. SELECT EMPNO, ENAME, TO_CHAR(HIREDATE, 'YYYY') 입사년도,
   HIREDATE,
   RANK() OVER (PARTITION BY TO_CHAR(HIREDATE, 'YYYY')
   ORDER BY HIREDATE) 년도별순위
   FROM EMP;
5. SELECT EMPNO, ENAME, SAL, JOB,
   SUM(SAL) OVER (PARTITION BY JOB
   ORDER BY SAL ROWS UNBOUNDED PRECEDING) 누적급여
   FROM EMP;
```

22장

```
1. SELECT DEPTNO, JOB, SUM(SAL)
   FROM EMP
   GROUP BY DEPTNO, JOB
   ORDER BY DEPTNO, JOB;
2. SELECT DEPT_JOB.DEPTNO, DEPT_JOB.JOB, SUM(EMP.SAL)
   FROM DEPT_JOB LEFT OUTER JOIN EMP PARTITION BY (DEPTNO)
   ON DEPT_JOB.DEPTNO=EMP.DEPTNO
   AND DEPT_JOB.JOB=EMP.JOB
   GROUP BY DEPT_JOB.DEPTNO, DEPT_JOB.JOB
```

```
   ORDER BY DEPT_JOB.DEPTNO, DEPT_JOB.JOB;
3. SELECT *
   FROM (SELECT DEPTNO, JOB, SUM(SAL) SUM_SAL
         FROM EMP
         GROUP BY DEPTNO, JOB) T
   MODEL
   DIMENSION BY (DEPTNO, JOB)
   MEASURES (SUM_SAL)
   RULES(SUM_SAL[40, 'CLERK']=AVG(SUM_SAL)[ANY, 'CLERK'])
   ORDER BY DEPTNO, JOB;
4. SELECT *
   FROM (SELECT DEPTNO, JOB, SUM(SAL) SUM_SAL
         FROM EMP
         GROUP BY DEPTNO, JOB) T
   MODEL
   DIMENSION BY (DEPTNO, JOB)
   MEASURES (SUM_SAL)
   RULES(SUM_SAL[10, '평균']=AVG(SUM_SAL)[10, ANY],
         SUM_SAL[20, '평균']=AVG(SUM_SAL)[20, ANY],
         SUM_SAL[30, '평균']=AVG(SUM_SAL)[30, ANY])
   ORDER BY DEPTNO, JOB;
5. FLASHBACK TABLE EMP TO BEFORE DROP;
```

23장

```
1. SELECT ENAME
   FROM EMP
   WHERE REGEXP_LIKE(ENAME, 'A|I');
2. SELECT ENAME
   FROM EMP
   WHERE REGEXP_LIKE(ENAME, 'L{2,}');
3. SELECT ENAME
   FROM EMP
   WHERE REGEXP_LIKE(ENAME, '^S|S$');
```